

On the CRON Conjecture

Peer-reviewed author version

AMELOOT, Tom & VAN DEN BUSSCHE, Jan (2012) On the CRON Conjecture. In: Barceló, Pablo; Pichler, Reinhard (Ed.). Datalog in Academia and Industria, p. 44-55.

DOI: 10.1007/978-3-642-32925-8_6

Handle: <http://hdl.handle.net/1942/14567>

On the CRON Conjecture

Tom J. Ameloot ^{*} Jan Van den Bussche

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 can 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.

1 Introduction

In declarative networking, distributed computations and networking protocols are modeled and programmed using formalisms based on Datalog [17]. Hellerstein has made a number of intriguing conjectures concerning the expressiveness of declarative networking [14, 15]. 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 applications of the CRON conjecture, one of which is crash recovery. During crash recovery, a program can read an old checkpointed state and a log of received messages, which is disjoint from that state. These messages could appear to come from the “future” when put side-by-side with the old state because according to the old state, those messages have yet to be sent. Then, it is not always clear how the program should combine the old state and the message log, certainly if negation and more generally non-monotone operations are involved. One can understand the CRON conjecture as saying that during recovery, for non-monotone operations, messages from the log

^{*}PhD. Fellow of the Fund for Scientific Research – Flanders (FWO)

should be read in causal order, like the order in which they are received, and they should not be exposed all at once. From the other direction, if you know that only monotone operations are involved, the recovery could perhaps become more efficient by reading the messages all at once. Distributed computations happen often in large clusters of compute nodes, where failure of nodes is not uncommon [22], and indeed distributed computing software should be robust against failures [9]. We want to avoid restarting entire computations when only a few nodes fail, and therefore it seems natural to use some lightweight crash recovery facility for individual nodes that can still make the computation succeed, although perhaps some partial results might have to be recomputed. The CRON conjecture could help us better understand how such recovery facilities can be designed.

In this paper we formally investigate the CRON conjecture in the setting of the language Dedalus, which is a Datalog-variant for distributed programming [4, 5, 15]. It turns out that stable models [12] 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, which is perhaps not entirely surprising as we can do special tricks with negation. 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 paper is organized as follows. Section 2 gives preliminaries on databases, Datalog, and Dedalus. Next, Section 3 states the CRON conjecture and gives the formalization of non-causality. Section 4 contains the results. We conclude in Section 5.

Acknowledgment We thank Joseph M. Hellerstein for his thoughtful comments on an earlier draft of this paper.

2 Preliminaries

2.1 Databases and Facts

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* \mathbf{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 $R(a_1, \dots, a_k)$ 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 $\text{adom}(I)$ to denote the set of values occurring in facts of I .

2.2 Datalog with Negation

We recall Datalog with negation [2], abbreviated Datalog[¬]. Let **var** be a universe of *variables*, disjoint from **dom**. An *atom* is of the form $R(u_1, \dots, u_k)$ where R is a relation name and $u_i \in \mathbf{var} \cup \mathbf{dom}$ for $i = 1, \dots, k$. 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[¬] 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[¬] 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 $head_\varphi = T(u, v)$, $pos_\varphi = \{R(u, v)\}$ and $neg_\varphi = \{S(v)\}$, with $u, v \in \mathbf{var}$, then we can write φ as

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

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 $R(u_1, \dots, u_k) \in \{head_\varphi\} \cup pos_\varphi \cup neg_\varphi$ we have $R^{(k)} \in \mathcal{D}$. A Datalog[¬] program P over \mathcal{D} is a set of safe Datalog[¬] 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 $edb(P) = sch(P) \setminus idb(P)$.¹

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 2.5. Let $\varphi \in P$. A *valuation for* φ is a total function $V : vars(\varphi) \rightarrow \mathbf{dom}$. The *application* of V to an atom $R(u_1, \dots, u_k)$ of φ , denoted $V(R(u_1, \dots, u_k))$, results in the *fact* $R(a_1, \dots, a_k)$ where for each $i \in \{1, \dots, k\}$ 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 $V(head_\varphi)$.

¹The abbreviation “idb” stands for “intensional database schema” and “edb” stands for “extensional database schema” [2].

2.2.1 Positive and Semi-positive

Let P be a Datalog[−] 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[−] program P [2]. 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 . Note, $adom(J') \subseteq adom(J)$.

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 (if P is considered constant [21]).

2.2.2 Stratified Semantics

We now recall the stratified semantics for a Datalog[−] program P [2]. 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 $\sigma : idb(P) \rightarrow \{1, \dots, |idb(P)|\}$ 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[−] programs P_1, \dots, P_k with $k \leq |idb(P)|$ such that for each $i = 1, \dots, k$, 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 $P_k(P_{k-1}(\dots P_1(I) \dots))$. 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[−] programs are syntactically stratifiable.

2.2.3 Stable Model Semantics

We now recall the stable model semantics for a Datalog[−] program P [12, 20]. Let I be an instance over $sch(P)$. Let $\varphi \in P$. Let V be a valuation for φ whose image is contained in $adom(I)$. 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 $u \in \text{vars}(\varphi)$ with $V(u)$. We call ψ a *ground rule* of φ with respect to I . Let $\text{ground}(\varphi, I)$ denote the set of all ground rules of φ with respect to I . The *ground program* of P on I , denoted $\text{ground}(P, I)$, is defined as $\bigcup_{\varphi \in P} \text{ground}(\varphi, I)$.

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

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

Note, $\text{ground}_M(P, I)$ is a positive program. We say that M is a *stable model* of P on input I if M is the output of $\text{ground}_M(P, I)$ on input I . If so, the semantics of positive Datalog⁻ programs implies $I \subseteq M$ and $\text{adom}(M) \subseteq \text{adom}(I)$. Not all Datalog⁻ programs have stable models on every input.

2.3 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 [17, 15], 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.

2.4 Dedalus Programs

We now recall the language Dedalus, that can be used to describe distributed computations [4, 5, 15]. Essentially, Dedalus is an extension of Datalog⁻ 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⁻ extended with annotations.²

Let \mathcal{D} be a database schema. We write $\mathbf{B}\{\bar{v}\}$, where \bar{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{v} . Let $R(\bar{u})$ denote any atom over \mathcal{D} . There are three types of Dedalus rules over \mathcal{D} :

²These annotations correspond to syntactic sugar in the previous presentations of Dedalus.

- A *deductive* rule is a normal Datalog[−] rule over \mathcal{D} .
- An *inductive* rule is of the form

$$R(\bar{u})\bullet \leftarrow \mathbf{B}\{\bar{u}, \bar{v}\}.$$

- An *asynchronous* rule is of the form

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

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 ‘ $\mid y$ ’ with $y \in \mathbf{var}$ means that the derived head facts are transferred (“piped”) to the node represented by y . Deductive, inductive and asynchronous rules will express respectively local computation, updatable memory, and message sending (cf. Section 2.5). Like in Section 2.2, a Dedalus rule is called *safe* if all its variables occur in at least one positive body atom.

To illustrate, 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} :

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

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

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

Now consider the following definition:

Definition 2.1. 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.

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

2.5 Operational Semantics

In this section we give an operational semantics for Dedalus. We describe how a network executes a Dedalus program \mathcal{P} when an input distributed database instance H is given. This operational semantics is in line with earlier formal work on declarative networking [10, 19, 13, 6, 1].

The essence of the operational semantics is as follows. Let \mathcal{N} be the network that H is over. Every node of \mathcal{N} runs the *same* Dedalus program, 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[−] 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.

2.5.1 Configurations

Let \mathcal{P} , H , and \mathcal{N} be like above. A configuration describes the network at a certain point in its evolution. We define a *configuration* of \mathcal{P} on H to be 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 $\langle i, \mathbf{f} \rangle$, where $i \in \mathbb{N}$ and \mathbf{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.

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 $x \in \mathcal{N}$.

2.5.2 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 2.5.3.

First, we define $deduc_{\mathcal{P}}$ to be the Datalog[−] program consisting of all deductive rules of \mathcal{P} . Secondly, we define $induc_{\mathcal{P}}$ to be the Datalog[−] program consisting of all inductive rules of \mathcal{P} after removing the annotation ‘ \bullet ’. Thirdly, we define $async_{\mathcal{P}}$ to be the Datalog[−] program consisting of all rules

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

where

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

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[−]

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 $induc_{\mathcal{P}}\langle I \rangle$, 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 $async_{\mathcal{P}}\langle I \rangle$, is defined as for $induc_{\mathcal{P}}$, but now using $async_{\mathcal{P}}$ instead of $induc_{\mathcal{P}}$.

Regarding data complexity [21], the output of each subprogram can be computed in PTIME with respect to the size of its input.

2.5.3 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 $\langle i, \mathbf{f} \rangle$, we define $untag(m) = \{\mathbf{f} \mid \exists i \in \mathbb{N} : \langle i, \mathbf{f} \rangle \in m\}$.

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

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

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

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

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. Specifically, for each $y \in \mathcal{N}$, the set $\delta^{i \rightarrow y}$ contains all messages addressed to y ; there we drop the addressee-component because y is known. We also attach the send-tag i . Messages with an addressee outside the network are ignored. This way of defining local computation closely corresponds to that of the language Webdamlog [1]. If $m = \emptyset$, we call it a *heartbeat* transition.

A *run* \mathcal{R} of \mathcal{P} on input H is an infinite sequence of transitions, such that (i) the very first configuration 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 . The resulting transition system is highly non-deterministic because in each transition we can choose the active node and also what messages to deliver. An infinite number of transitions is always possible because the set of delivered messages may be empty.

It is natural to require certain “fairness” conditions on the execution of a system [11, 8, 16]. A run \mathcal{R} of \mathcal{P} on H is called *fair* if (i) every node does an infinite number of transitions, and (ii) every sent message is eventually delivered. We only consider fair runs.

2.5.4 Output and Consistency

We formalize the output of a run. 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. [18], we define this output based on *ultimate* facts. In a run \mathcal{R} , we say that a fact \mathbf{f} over schema $out(\mathcal{P})$ is *ultimate* at some node x if there is some transition of \mathcal{R} after which \mathbf{f} is output by $deduc_{\mathcal{P}}$ during every transition of x . Thus, \mathbf{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, following the intuition of cloud computing.

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 [1, 18, 7].

2.5.5 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.

3 CRON Conjecture and Non-Causality

3.1 Conjecture

Conjecture 1. Causality Required Only for Non-monotonicity (CRON) [15]:

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. For this reason, we want a new formalism to reason about non-causality, which entails sending messages into the “past”. In Section 3.2 we introduce such a formalism. This is used in Section 4 to formally investigate the CRON conjecture.

3.2 Modeling Non-Causality

In a previous work [3], we have shown that the operational semantics of Dedalus is equivalent to a declarative semantics based on stable models. There we described a *causality transform* that converts a Dedalus program to a pure Datalog[⊥] program containing extra rules, called the *causality rules*, that enforce causality on message sending in every stable model of the pure Datalog[⊥] program. In the current work, we remove the causality rules and explain how stable models can now represent non-causal message sending.

3.2.1 Transformation

Let \mathcal{P} be a Dedalus program. Below, we present the *SZ-transformation* that transforms \mathcal{P} into $\text{pure}_{\text{SZ}}(\mathcal{P})$, which is a pure Datalog[⊥] program that models the distributed computation in a holistic fashion: the data across all nodes and their local timestamps is modeled as facts of the form $R(x, s, \bar{a})$, representing that fact $R(\bar{a})$ is present at node x during local timestamp s . For asynchronous rules, to select an arrival timestamp for every sent message, we use a rewriting technique inspired by the work of Saccà and Zaniolo, who show how to express dynamic choice under the stable model semantics [20].

For technical convenience, we assume that the relation names presented below do not yet occur in $\text{sch}(\mathcal{P})$. We will also assume that rules of \mathcal{P} contain at least one positive body atom; this assumption allows for a more elegant way to enforce the safety condition on rules of $\text{pure}_{\text{SZ}}(\mathcal{P})$, and is not fundamental.

First, timestamps in $\text{pure}_{\text{SZ}}(\mathcal{P})$ will be represented by the following database schema:

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

Relation ‘ \neq ’ will be written in infix notation. We assume $\mathbb{N} \subseteq \mathbf{dom}$ and consider only the following instance over $\mathcal{D}_{\text{time}}$:

$$I_{\text{time}} = \{\text{time}(s), \text{tsucc}(s, s+1) \mid s \in \mathbb{N}\} \cup \{(s \neq t) \mid s, t \in \mathbb{N} : s \neq t\}.$$

We now specify $\text{pure}_{\text{SZ}}(\mathcal{P})$ by transforming the rules of \mathcal{P} . Let $\mathbf{x}, \mathbf{s}, \mathbf{t}$ and \mathbf{t}' be variables not yet used in \mathcal{P} . For any sequence L of literals, let $L^{\uparrow \mathbf{x}, \mathbf{s}}$ denote the sequence obtained by adding \mathbf{x} and \mathbf{s} as first and second components to each atom in L (negated atoms stay negated).

First, for each *deductive* rule ‘ $R(\bar{\mathbf{u}}) \leftarrow \mathbf{B}\{\bar{\mathbf{u}}, \bar{\mathbf{v}}\}$ ’ in \mathcal{P} , we add to $\text{pure}_{\text{SZ}}(\mathcal{P})$ the following rule:

$$R(\mathbf{x}, \mathbf{s}, \bar{\mathbf{u}}) \leftarrow \mathbf{B}\{\bar{\mathbf{u}}, \bar{\mathbf{v}}\}^{\uparrow \mathbf{x}, \mathbf{s}}. \quad (3.1)$$

This expresses that deductively derived facts are directly visible within the same step (of the same node) in which they were derived.

Next, for each *inductive* rule ' $R(\bar{u}) \bullet \leftarrow \mathbf{B}\{\bar{u}, \bar{v}\}$ ' in \mathcal{P} , we add to $pure_{SZ}(\mathcal{P})$ the following rule:

$$R(x, t, \bar{u}) \leftarrow \mathbf{B}\{\bar{u}, \bar{v}\}^{\uparrow x, s}, \text{tsucc}(s, t). \quad (3.2)$$

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

Lastly, for each asynchronous rule ' $R(\bar{u}) \mid y \leftarrow \mathbf{B}\{\bar{u}, \bar{v}, y\}$ ' in \mathcal{P} , letting \bar{w} be a tuple of new and distinct variables with $|\bar{w}| = |\bar{u}|$, we add to $pure_{SZ}(\mathcal{P})$ the following rules, for which the intuition is given below:

$$\text{cand}_R(x, s, y, t, \bar{u}) \leftarrow \mathbf{B}\{\bar{u}, \bar{v}, y\}^{\uparrow x, s}, \text{all}(y), \text{time}(t). \quad (3.3)$$

$$\text{chosen}_R(x, s, y, t, \bar{w}) \leftarrow \text{cand}_R(x, s, y, t, \bar{w}), \neg \text{other}_R(x, s, y, t, \bar{w}). \quad (3.4)$$

$$\text{other}_R(x, s, y, t, \bar{w}) \leftarrow \text{cand}_R(x, s, y, t, \bar{w}), \text{chosen}_R(x, s, y, t', \bar{w}), t \neq t'. \quad (3.5)$$

$$R(y, t, \bar{w}) \leftarrow \text{chosen}_R(x, s, y, t, \bar{w}). \quad (3.6)$$

A fact of the form $\text{all}(x)$ means that x is a node of the network. Rule (3.3) represents message sending: it derives messages by evaluating the original asynchronous rule, verifies that the addressee of each message is in the network, and it considers for each message all possible candidate arrival timestamps at the addressee. In the cand_R -facts, we include the sender's location and send-timestamp, the addressee's location and arrival-timestamp, and the actual transmitted data. Next, rules (3.4) and (3.5) together enforce under the stable model semantics that precisely one arrival timestamp will be chosen for every sent message, using the technique of [20]. Rule (3.6) models the actual arrival of messages, where the sender-information is projected away, and the transmitted data is placed in the addressee's relation R . Note, if multiple asynchronous rules in \mathcal{P} have the same head predicate R , only new cand_R -rules have to be added because the rules (3.4)–(3.6) are sufficiently general.

This concludes the specification of program $pure_{SZ}(\mathcal{P})$.

3.2.2 Semantics

Let H be an input for \mathcal{P} , over a network \mathcal{N} . To represent H , we give $pure_{SZ}(\mathcal{P})$ the following input:

$$\begin{aligned} \text{input}_{SZ}(H) = & \{R(x, s, \bar{a}) \mid x \in \mathcal{N}, R(\bar{a}) \in H(x), s \in \mathbb{N}\} \\ & \cup \{\text{all}(x) \mid x \in \mathcal{N}\} \cup I_{\text{time}}. \end{aligned}$$

Intuitively, for each node its input facts are made available at each of its local timestamps; relation all represents the network; and, all timestamps are provided together with comparison relations.

Now, we call any stable model M of $pure_{SZ}(\mathcal{P})$ on $\text{input}_{SZ}(H)$ an *SZ-model* of \mathcal{P} on input H . Program $pure_{SZ}(\mathcal{P})$ does not enforce causality on the messages in M because the arrival timestamps can be chosen arbitrarily, even into the

past. 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: across all transitions one unites the outputs of $\text{deduc}_{\mathcal{P}}$ after adorning those facts with the location and timestamp of their creation, and all message sending and arrival events are encoded with cand_R -, chosen_R - and other_R -facts.

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

Similar to [3], we focus on SZ-models M that contain for each $(y, t) \in \mathcal{N} \times \mathbb{N}$ only finitely many facts of the form $\text{chosen}_R(x, s, y, t, \bar{a})$. This expresses that every node y receives only a finite number of messages on every timestamp t . This is a natural constraint, because in real system a node always processes a finite number of messages during each computation step. This constraint could also be directly enforced with additional rules of $\text{pure}_{\text{SZ}}(\mathcal{P})$, but we have omitted the technical details for easier presentation.

3.2.3 Output and SZ-consistency

The *output* of an SZ-model M , denoted $\text{output}(M)$, is defined with ultimate facts like in the operational semantics (Section 2.5.4):

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

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

4 Results

We have considered a semantical and syntactical interpretation of the CRON conjecture, for which we present the results below.

4.1 Semantical Interpretation

We have first formalized the CRON conjecture purely on the semantical level, by relating causality to the monotonicity of the queries computed by Dedalus programs.

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 *monotone* if for each I and J over \mathcal{D}_1 , $I \subseteq J$ implies $\mathcal{Q}(I) \subseteq \mathcal{Q}(J)$. Relating to the distributed setting, an instance I over a database schema \mathcal{D} can be *partitioned* over a network \mathcal{N} by

Algorithm 1 Program for emptiness query

```
empty(x) | y ← ¬S(), Id(x), Node(y).  
empty(y)• ← empty(y).  
missing() ← Node(y), ¬empty(y).  
T() ← Id(x), ¬missing().
```

putting each fact of I on at least one node, resulting in a distributed database instance over \mathcal{N} and \mathcal{D} . Now, we say that a Dedalus program \mathcal{P} (*distributedly*) *computes* query Q if \mathcal{P} is consistent and for every input instance I for Q , for every network \mathcal{N} , for every partition H of I over \mathcal{N} , we have $outInst(\mathcal{P}, H) = Q(I)$. To compute non-monotone queries, every node needs its own identifier and the identifiers of the other nodes, or equivalent information [6]. Therefore, we restrict attention to Dedalus programs \mathcal{P} for which $\{Id^{(1)}, Node^{(1)}\} \subseteq edb(\mathcal{P})$, and each input H , over a network \mathcal{N} , includes for each $x \in \mathcal{N}$ the facts $\{Id(x)\} \cup \{Node(y) \mid y \in \mathcal{N}\}$, which are treated just like any other *edb*-fact.

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

Conjecture 2. Semantical CRON: A Dedalus program computes a monotone query if and only if it is SZ-consistent.

Both directions of this conjecture can be refuted by counterexamples, as we do in the following two subsections. So, contrary to the CALM conjecture [15, 6, 23], a formalization of the CRON conjecture that is situated purely on the semantical level does not seem promising.

4.1.1 If Direction

For the if-direction of Conjecture 2, we give an SZ-consistent Dedalus program computing a *non-monotone* query.

Algorithm 1 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 (at all nodes). The 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).³ The program is consistent.

Now we consider SZ-consistency. 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 still has a timestamp after which it has all node identifiers. Thus every SZ-model

³The atom $Id(x)$ is just to have at least one positive body atom (cf. Section 3.2.1).

Algorithm 2 Program for non-emptiness query

$$\begin{aligned} A() \mid \mathbf{x} &\leftarrow S(), \text{Id}(\mathbf{x}). \\ A() \bullet &\leftarrow A(). \\ B() \mid \mathbf{x} &\leftarrow A(), \neg \text{sent}_B(), \text{Id}(\mathbf{x}). \\ \text{sent}_B() \bullet &\leftarrow A(). \\ T() &\leftarrow A(), B(). \\ T() \bullet &\leftarrow T(). \end{aligned}$$

yields the output $T()$ if and only if all nodes have an empty relation S . So, the program is SZ-consistent. A formal proof can be found in Appendix B.1.

4.1.2 Only-If Direction

For the only-if direction of Conjecture 2, we give a Dedalus program computing a monotone query and that is *not* SZ-consistent.

Algorithm 2 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 the inductive rule. The program is consistent.

However, the program is not SZ-consistent, 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 $\text{sent}_B()$ 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 $\text{output}(M) = \emptyset$, and the program is not SZ-consistent. Formal details can be found in Appendix B.2.

4.2 Syntactical Interpretation

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

Theorem 4.1. Every positive consistent Dedalus program is SZ-consistent.

The converse direction of Theorem 4.1, to the effect that every SZ-consistent Dedalus program is equivalent to a positive program, cannot hold by our counterexample for the if-direction of Conjecture 2 (Section 4.1.1).

The following subsections show Theorem 4.1. 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 $outInst(\mathcal{P}, H)$ and (ii) at most $outInst(\mathcal{P}, H)$, respectively shown in Sections 4.2.1 and 4.2.2.

We remark for completeness that a positive program is not automatically consistent; Appendix D gives a simple example.

4.2.1 First Direction

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 have to show $outInst(\mathcal{P}, H) \subseteq output(M)$. We construct a fair run \mathcal{R} of \mathcal{P} on H such that $output(\mathcal{R}) \subseteq output(M)$. Then, since $output(\mathcal{R}) = outInst(\mathcal{P}, H)$ by consistency of \mathcal{P} , we have $outInst(\mathcal{P}, H) \subseteq output(M)$, as desired.

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

For each $(x, s) \in \mathcal{N} \times \mathbb{N}$, let $rcv_M(x, s)$ be the set of all facts $R(\bar{a})$ for which there is some y and t such that $chosen_R(y, t, x, s, \bar{a}) \in M$, i.e., the set of all messages arriving at (x, s) in M . Note, $rcv_M(x, s) \subseteq all_M(x, s)$ by rules of the form (3.6) in $pure_{SZ}(\mathcal{P})$.

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

We define $sndFin_M(x) \subseteq snd_M(x)$ to be the subset of pairs $(y, R(\bar{a}))$ for which there are only a finite number of times s such that $chosen_R(x, s, y, t, \bar{a}) \in M$ 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 $start_M(x) = 0$ if $sndFin_M(x) = \emptyset$ and otherwise we define $start_M(x)$ to be 1 plus the largest timestamp on which x sends a pair of $sndFin_M(x)$ in M . Intuitively, $start_M(x)$ is the first local timestamp of x at which x no longer sends messages in $sndFin_M(x)$, so the messages that x sends starting from $start_M(x)$ are sent infinitely often.

Main Idea We inductively define the transitions of \mathcal{R} . More specifically, for each $i = 0, 1, \dots$, we define the (partial) *arrival function* $\alpha_{\mathcal{R}}^{(i)}$ that contains for each transition $j \leq i$ 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 $j < k$).⁴ The arrival function is merely a technical aid; it

⁴To satisfy fairness (Section 2.5.3), all messages sent in transitions $j \leq i$ will get a mapping in $\alpha_{\mathcal{R}}^{(i)}$.

helps us make explicit how messages are delivered. For easier notation, we also write a mapping $(j, y, R(\bar{a})) \mapsto k$ simply as $(j, y, R(\bar{a}), k)$.

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 the total order \leq on $\mathcal{N} \times \mathbb{N}$. We define the active node in transition i of \mathcal{R} to be the unique $x \in \mathcal{N}$ satisfying $ord(x, s) = i$ for some $s \in \mathbb{N}$. 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 induction properties to be satisfied, for which the intuition is provided below:

$$D_i \subseteq all_M(x_i, s_i) \tag{4.1}$$

$$\forall (j, y, \mathbf{f}, k) \in \alpha_{\mathcal{R}}^{(i)} : \mathbf{f} \in rcv_M(x_k, s_k) \tag{4.2}$$

$$\forall (j, y, \mathbf{f}, k) \in \alpha_{\mathcal{R}}^{(i)} : s_k \geq start_M(y) \tag{4.3}$$

Property (4.1) ensures all ultimate facts of \mathcal{R} are ultimate facts of M , resulting in $output(\mathcal{R}) \subseteq output(M)$, as desired. Property (4.2) ensures we do not have more opportunities in \mathcal{R} for messages to arrive “together” when compared to M , so that induction property (4.1) can be satisfied. To explain property (4.3), 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 (4.2). Now, induction property (4.3) 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 (4.2).

Inductive construction For uniformity, we start with $i = -1$, and define $\alpha_{\mathcal{R}}^{(-1)} = \emptyset$ and $D_{-1} = \emptyset$. So, properties (4.1) through (4.3) are trivially satisfied for $i = -1$. For the induction hypothesis, assume \mathcal{R} has been partially constructed up to and including transition $i - 1$, where $i \geq 0$, and assume the properties hold for all transitions $j = -1, 0, \dots, i - 1$. For the inductive step, we show that property (4.1) is satisfied for i , and we show how to extend $\alpha_{\mathcal{R}}^{(i-1)}$ to $\alpha_{\mathcal{R}}^{(i)}$ such that properties (4.2) and (4.3) are satisfied. The set m_i of (tagged) messages to be delivered in transition i consists of all pairs $\langle j, \mathbf{f} \rangle$ for which $\alpha_{\mathcal{R}}^{(i-1)}$ contains (j, y, \mathbf{f}, i) .⁵ Helper claims are in Appendix C.

Property (4.1) We have to show $D_i \subseteq all_M(x_i, s_i)$. Using the definition $D_i = deduc_{\mathcal{P}}(st_i(x_i) \cup untag(m_i))$ with $\rho_i = (st_i, bf_i)$ the source-configuration of transition i , by Claim C.1 it suffices to show $st_i(x_i) \cup untag(m_i) \subseteq all_M(x_i, s_i)$.

⁵This implies $y = x_i$.

First, by applying the induction hypothesis for property (4.2) to $\alpha_{\mathcal{R}}^{(i-1)}$, we know $\text{untag}(m_i) \subseteq \text{rcv}_M(x_i, s_i) \subseteq \text{all}_M(x_i, s_i)$.

We are left to show $st_i(x_i) \subseteq \text{all}_M(x_i, s_i)$. We have $st_i(x_i)|_{\text{edb}(\mathcal{P})} \subseteq \text{all}_M(x_i, s_i)$ because $st_i(x_i)|_{\text{edb}(\mathcal{P})} = H(x_i)$ by the operational semantics and $H(x_i)^{\uparrow x_i, s_i} \subseteq \text{input}_{\text{SZ}}(H) \subseteq M$ by definition of M . Next, if i is the first transition of x_i , we have $st_i(x_i)|_{\text{idb}(\mathcal{P})} = \emptyset \subseteq \text{all}_M(x_i, s_i)$. Otherwise, we consider the last transition j before i in which x_i was also the active node. By the operational semantics, $st_i(x_i)|_{\text{idb}(\mathcal{P})} = \text{induc}_{\mathcal{P}}\langle D_j \rangle$. Because $D_j \subseteq \text{all}_M(x_i, s_j)$ by the induction hypothesis for property (4.1), Claim C.2 gives $\text{induc}_{\mathcal{P}}\langle D_j \rangle \subseteq \text{all}_M(x_i, s_j + 1) = \text{all}_M(x_i, s_i)$, as desired.

Properties (4.2) and (4.3) We have to extend $\alpha_{\mathcal{R}}^{(i-1)}$ to $\alpha_{\mathcal{R}}^{(i)}$ so that properties (4.2) and (4.3) are satisfied. Suppose 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 $s_i < \text{start}_M(x_i)$. The induction hypothesis for property (4.3) 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 C.3, node x_i sends $R(\bar{a})$ to y on an infinite number of timestamps in M .

Now suppose $s_i \geq \text{start}_M(x_i)$. Using $D_i \subseteq \text{all}_M(x_i, s_i)$ (shown above), $R(y, \bar{a}) \in \text{async}_{\mathcal{P}}\langle D_i \rangle$, and $y \in \mathcal{N}$, by Claim C.4 there is a local timestamp t of y for which $\text{chosen}_R(x_i, s_i, y, t, \bar{a}) \in M$. So, in M , node x_i sends $R(\bar{a})$ to y on a timestamp at least $\text{start}_M(x_i)$, which by definition of $\text{start}_M(x_i)$ implies that node x_i sends $R(\bar{a})$ to y on an infinite number of timestamps in M .

Now, because x_i sends $R(\bar{a})$ to y on an infinite number of timestamps in M , and y receives only a finite number of messages on each timestamp (Section 3.2.2), there must be an infinite number of timestamps $t \in \mathbb{N}$ on which y receives $R(\bar{a})$ in M . Among these, we can surely choose some arrival timestamp $t \in \mathbb{N}$ for which $\text{ord}(y, t) > i$ and $t \geq \text{start}_M(y)$. Then we extend $\alpha_{\mathcal{R}}^{(i-1)}$ by adding the mapping $(i, y, R(\bar{a}), k)$ where $k = \text{ord}(y, t)$. Note, this mapping satisfies properties (4.2) and (4.3).

4.2.2 Second Direction

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 have to show $\text{output}(M) \subseteq \text{outInst}(\mathcal{P}, H)$. We construct a fair run \mathcal{R} such that $\text{output}(M) \subseteq \text{output}(\mathcal{R})$. Then, using $\text{output}(\mathcal{R}) = \text{outInst}(\mathcal{P}, H)$ by consistency of \mathcal{P} , we get $\text{output}(M) \subseteq \text{outInst}(\mathcal{P}, H)$, 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 sets of deductively derived facts monotonically increase on each 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 $G_M(\mathcal{P}) = ground_M(\mathcal{P}', I)$ where $\mathcal{P}' = pure_{SZ}(\mathcal{P})$ and $I = input_{SZ}(H)$. Because $M = G_M(\mathcal{P})(I)$ by definition of stable model, we can consider the infinite sequence M_0, M_1, M_2, \dots , 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., $pos_{\psi} \subseteq M_{l-1}$ and $head_{\psi} = R(x, s, \bar{a})$. Rule ψ must have one of the following three forms: the deductive form (3.1), the inductive form (3.2), or the delivery form (3.6). We handle each case in turn.

Deductive Let $\varphi \in pure_{SZ}(\mathcal{P})$ be the rule corresponding to ψ , so φ is of the form (3.1). 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, $\varphi \in deduc_{\mathcal{P}}$. By the syntactical correspondence between φ and φ' , we can apply V to φ' . Now, it suffices to show $V(pos_{\varphi'}) \subseteq D_i$ for some transition i of x in \mathcal{R} , resulting in $V(head_{\varphi'}) = R(\bar{a}) \in D_i$ by the fixpoint semantics of $deduc_{\mathcal{P}}$, as desired.

Let $S(\bar{b}) \in V(pos_{\varphi'})$. By the syntactical correspondence between φ' and φ , we have $S(x, s, \bar{b}) \in V(pos_{\varphi}) = pos_{\psi}$. Using $pos_{\psi} \subseteq M_{l-1}$ 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 $S(\bar{b}) \in V(pos_{\varphi'})$.

Inductive Let φ and V be like in the deductive case, but now φ is of the form (3.2). Let $\varphi' \in induc_{\mathcal{P}}$ be the rule corresponding to φ . Again, we can apply V to φ' . Now, it suffices to show $V(pos_{\varphi'}) \subseteq D_i$ for some transition i of x in \mathcal{R} , causing $V(head_{\varphi'}) = R(\bar{a})$ to be stored in the next state of x . Then, with j being the first transition of x after i , we get $R(\bar{a}) \in D_j$ by the operational semantics, as desired. The existence of i is established similarly to the deductive case.

Delivery Rule ψ is of the form (3.6), with body fact $\mathbf{chosen}_R(y, t, x, s, \bar{a}) \in M_{l-1}$. 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, $\mathbf{chosen}_R(y, t, x, s, \bar{a}) \in M_{l-1}$ implies $\mathbf{cand}_R(y, t, x, s, \bar{a}) \in M_{l-1}$. There is some $k \in \mathbb{N}$ with $0 < k < l - 1$ such that $\mathbf{cand}_R(y, t, x, s, \bar{a}) \in M_k \setminus M_{k-1}$. Let $\psi' \in G_M(\mathcal{P})$ be a rule responsible for deriving the \mathbf{cand}_R -fact. Let $\varphi' \in \mathbf{pure}_{\text{SZ}}(\mathcal{P})$ be the rule corresponding to ψ' , and let V' be the valuation for φ' giving rise to ψ' . In turn, let $\varphi'' \in \mathbf{async}_{\mathcal{P}}$ be the rule corresponding to φ' . By the syntactical correspondence between φ' and φ'' , we can apply V' to φ'' . Note, $V'(\text{head}_{\varphi''}) = R(x, \bar{a})$. To make y send $R(\bar{a})$ to x in some transition i , we need $V'(\text{pos}_{\varphi''}) \subseteq D_i$. The existence of transition i is again established like in the deductive case.

5 Discussion

In future work, we may want to understand better the spectrum of causality. We have seen that for positive programs no causality at all is required, and perhaps richer classes of programs can tolerate some relaxations of causality as well. We would also like to investigate how the CRON conjecture can be concretely linked to crash recovery applications, and the design of recovery mechanisms. It might also be interesting to look at other local operational semantics for Dedalus, besides the stratified semantics used here.

References

- [1] 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*, pages 293–304. ACM Press, 2011.
- [2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
- [3] P. Alvaro, T.J. Ameloot, J.M. Hellerstein, W. Marczak, and J. Van den Bussche. A declarative semantics for dedalus. Technical Report UCB/EECS-2011-120, EECS Department, University of California, Berkeley, Nov 2011.
- [4] P. Alvaro, W. Marczak, et al. Dedalus: Datalog in time and space. Technical Report EECS-2009-173, University of California, Berkeley, 2009.
- [5] P. Alvaro, W.R. Marczak, et al. Dedalus: Datalog in time and space. In O. de Moor, G. Gottlob, T. Furche, and A. Sellers, editors, *Datalog Reloaded: First International Workshop, Datalog 2010*, volume 6702 of *Lecture Notes in Computer Science*, pages 262–281, 2011.

- [6] T.J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. In *Proceedings 30th ACM Symposium on Principles of Database Systems*, pages 283–292. ACM Press, 2011.
- [7] T.J. Ameloot and 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*, pages 86–98. ACM Press, 2012.
- [8] K.R. Apt, N. Francez, and S. Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2:226–241, 1988.
- [9] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley, 2004.
- [10] A. Deutsch, L. Sui, V. Vianu, and D. Zhou. Verification of communicating data-driven Web services. In *Proceedings 25th ACM Symposium on Principles of Database Systems*, pages 90–99. ACM Press, 2006.
- [11] N. Francez. *Fairness*. Springer-Verlag New York, Inc., New York, NY, USA, 1986.
- [12] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
- [13] S. Grumbach and F. Wang. Netlog, a rule-based language for distributed programming. In M. Carro and R. Peña, editors, *Proceedings 12th International Symposium on Practical Aspects of Declarative Languages*, volume 5937 of *Lecture Notes in Computer Science*, pages 88–103, 2010.
- [14] J.M. Hellerstein. Datalog redux: experience and conjecture. Video available (under the title “The Declarative Imperative”) from <http://db.cs.berkeley.edu/jmh/>, 2010. PODS 2010 keynote.
- [15] J.M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *SIGMOD Record*, 39(1):5–19, 2010.
- [16] L. Lamport. Fairness and hyperfairness. *Distributed Computing*, 13:239–245, November 2000.
- [17] B.T. Loo et al. Declarative networking. *Communications of the ACM*, 52(11):87–95, 2009.
- [18] W. Marczak, P. Alvaro, N. Conway, J.M. Hellerstein, and D. Maier. Confluence analysis for distributed programs: A model-theoretic approach. Technical Report UCB/EECS-2011-154, EECS Department, University of California, Berkeley, Dec 2011.

- [19] J.A. Navarro and A. Rybalchenko. Operational semantics for declarative networking. In A. Gill and T. Swift, editors, *Proceedings 11th International Symposium on Practical Aspects of Declarative Languages*, volume 5419 of *Lecture Notes in Computer Science*, pages 76–90, 2009.
- [20] D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, pages 205–217. ACM Press, 1990.
- [21] M. Vardi. The complexity of relational query languages. In *Proceedings 14th ACM Symposium on the Theory of Computing*, pages 137–146, 1982.
- [22] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1:7–18, 2010.
- [23] D. Zinn, T.J. Green, and B. Ludaescher. Win-move is coordination-free. In *Proceedings of the 15th International Conference on Database Theory*, pages 99–113. ACM Press, 2012.

Appendix

A Common Lemmas

Lemma A.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 $\mathbf{cand}_R(x, s, y, t, \bar{a}) \in M$ there is a value $t' \in \mathbb{N}$ such that $\mathbf{chosen}_R(x, s, y, t', \bar{a}) \in M$.

Proof. Abbreviate $G_M(\mathcal{P}) = \mathit{ground}_M(\mathcal{P}', I)$ where $\mathcal{P}' = \mathit{pure}_{\text{SZ}}(\mathcal{P})$ and $I = \mathit{input}_{\text{SZ}}(H)$. Towards a proof by contradiction, suppose there is no such timestamp t' . Consider the following ground rule of the form (3.4), after removing the negative body literal:

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

This rule can not be in $G_M(\mathcal{P})$ because otherwise $\mathbf{cand}_R(x, s, y, t, \bar{a}) \in M$ implies $\mathbf{chosen}_R(x, s, y, t, \bar{a}) \in M$, which is assumed to be false. The absence of the above ground rule from $G_M(\mathcal{P})$ implies $\mathbf{other}_R(x, s, y, t, \bar{a}) \in M$. This \mathbf{other}_R -fact must be derived by a ground rule of the form (3.5):

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

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

B Semantical CRON

B.1 If Direction

Let \mathcal{Q} and \mathcal{P} be respectively the emptiness query and the Dedalus program from Section 4.1.1. We show that \mathcal{P} is SZ-consistent.

B.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 $\mathcal{P}' = pure_{SZ}(\mathcal{P})$ and $I = input_{SZ}(H)$.

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 $empty(y, t, x) \in M$. Let $x \in \mathcal{N}$. We show that x at every local timestamp u sends $empty(x)$ to y . We have $S(x, u) \notin input_{SZ}(H)$ 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 (3.3) to the asynchronous rule of \mathcal{P} :

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

Here, $v \in \mathbb{N}$ is arbitrary. The body facts of this rule are in M by definition of $input_{SZ}(H)$. Hence, $cand_{empty}(x, u, y, v, x) \in M$ because M is stable. Then, by Lemma A.1, there is a timestamp $w \in \mathbb{N}$ such that $chosen_{empty}(x, u, y, w, x) \in M$. Then a ground rule of the form (3.6) derives $empty(y, w, x) \in M$, and inductive ground rules for relation $empty$ will derive $empty(y, w', x) \in M$ for all $w' \geq w$. So, there is a timestamp s on which $empty(y, s, x) \in M$ for each $x \in \mathcal{N}$.

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

$$T(y, t) \leftarrow Id(y, t, y).$$

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

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

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

B.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, $outInst(\mathcal{P}, H) = \emptyset$. Let M be an SZ-model of \mathcal{P} on H . We have to show

$output(M) = \emptyset$. Abbreviate $G_M(\mathcal{P}) = ground_M(\mathcal{P}', I)$ where $\mathcal{P}' = pure_{SZ}(\mathcal{P})$ and $I = input_{SZ}(H)$.

Towards a proof by contradiction, suppose $output(M) \neq \emptyset$, i.e., $T() \in output(M)$ since T is the only output relation. We will show z has an empty relation S , which is the desired contradiction. First, $T() \in output(M)$ 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 $empty(x, s, z) \in M$. The following ground rule must be in $G_M(\mathcal{P})$ to derive $T(x, s) \in M$:

$$T(x, s) \leftarrow Id(x, s, x).$$

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

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

But then $missing(x, s) \in M$ since $Node(x, s, z) \in input_{SZ}(H)$, which is false. So, $empty(x, s, z) \in M$.

Now we show relation S is empty at z . The fact $empty(x, s, z) \in M$ 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 (3.6) to the asynchronous rule of \mathcal{P} and the second one is based on the inductive rule of \mathcal{P} :

$$empty(x, s, z) \leftarrow chosen_{empty}(y, t, x, s, z).$$

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

Intuitively, the second form is like a chain we can follow backwards in time. So we must eventually use the first form: there is a value $u \in \mathbb{N}$ such that $empty(x, u, z) \in M$ and $chosen_{empty}(y, t, x, u, z) \in M$ 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 $chosen_{empty}(z, t, x, u, z)$ was derived by a ground rule in $G_M(\mathcal{P})$ of the form (3.4):

$$chosen_{empty}(z, t, x, u, z) \leftarrow cand_{empty}(z, t, x, u, z).$$

Hence, $cand_{empty}(z, t, x, u, z) \in M$. This $cand_{empty}$ -fact is derived by a ground rule in $G_M(\mathcal{P})$ obtained by applying transformation (3.3) to the asynchronous rule of \mathcal{P} :

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

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

B.2 Only-if Direction

Let \mathcal{P} be the program in Algorithm 2. Let H be the input over singleton network $\mathcal{N} = \{z\}$ that assigns $S()$ to z . So, $outInst(\mathcal{P}, H) = \{T()\}$. We define an SZ-model M of \mathcal{P} on H such that $output(M) = \emptyset$, making \mathcal{P} not SZ-consistent.

In a fair run of \mathcal{P} on H , message $B()$ always arrives *after* message $A()$ at z , and because $A()$ itself is persisted, this makes the program \mathcal{P} consistent. 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

$$M = \text{input}_{\text{SZ}}(H) \cup M_A^{\text{snd}} \cup M_A^{\text{rcv}} \cup M_B^{\text{snd}} \cup M_B^{\text{rcv}},$$

where

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

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

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

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

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. The inductive rule for relation A has the same effect as 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.

Using straightforward arguments, one can verify that M is a stable model of $\text{pure}_{\text{SZ}}(\mathcal{P})$ on $\text{input}_{\text{SZ}}(H)$. These details are omitted. Note, in M we deliver only a finite number of messages on each timestamp of z (cf. Section 3.2.2). Lastly, we have $\text{output}(M) = \emptyset$ as desired, because M contains no T -facts.

C Syntactical CRON

C.1 First Direction

These claims are in the context of Section 4.2.1.

Claim C.1. Let $(x, s) \in \mathcal{N} \times \mathbb{N}$ and let I be an instance over $\text{sch}(\mathcal{P})$. Suppose $I \subseteq \text{all}_M(x, s)$. Then $\text{deduc}_{\mathcal{P}}(I) \subseteq \text{all}_M(x, s)$.

Proof. We proceed by induction on the fixpoint computation of $\text{deduc}_{\mathcal{P}}$. So, $\text{deduc}_{\mathcal{P}}(I) = \bigcup_j D^j$ where $D^0 = I$ and for each $j \geq 1$ set D^j is obtained by applying the immediate consequence operator of $\text{deduc}_{\mathcal{P}}$ to D^{j-1} . For the base case, we have $D^0 = I \subseteq \text{all}_M(x, s)$ by the given assumption. For the induction hypothesis, we assume $D^{j-1} \subseteq \text{all}_M(x, s)$ 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 $\text{head}_{\psi} = R(x, s, \bar{a})$ in the ground program $G_M(\mathcal{P}) = \text{ground}_M(\mathcal{P}', J)$ where $\mathcal{P}' = \text{pure}_{\text{SZ}}(\mathcal{P})$ and $J = \text{input}_{\text{SZ}}(H)$. Let $\varphi \in \text{deduc}_{\mathcal{P}}$ and V be a rule and valuation that have derived $R(\bar{a}) \in D^j$. Let $\varphi' \in \text{pure}_{\text{SZ}}(\mathcal{P})$ be obtained by applying transformation (3.1) 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, $\text{head}_{\psi} = R(x, s, \bar{a})$, and $\psi \in G_M(\mathcal{P})$ because ψ is positive.

Lastly, we show $\text{pos}_{\psi} \subseteq M$. Then $R(x, s, \bar{a}) \in M$ by using $M = G_M(\mathcal{P})(J)$ (definition of stable model). Since $V(\text{pos}_{\varphi}) \subseteq D^{j-1} \subseteq \text{all}_M(x, s)$ by the induction hypothesis, we have $\text{pos}_{\psi} = V(\text{pos}_{\varphi})^{\uparrow x, s} \subseteq \text{all}_M(x, s)^{\uparrow x, s} \subseteq M$. \square

Claim C.2. Let $(x, s) \in \mathcal{N} \times \mathbb{N}$ and let D be an instance over $\text{sch}(\mathcal{P})$. Assume $D \subseteq \text{all}_M(x, s)$. Then $\text{induc}_{\mathcal{P}}(D) \subseteq \text{all}_M(x, s+1)$.

Proof. This is similar to the proof of Claim C.1. Let $R(\bar{a}) \in \text{induc}_{\mathcal{P}}(D)$. We show $R(x, s+1, \bar{a}) \in M$. Let φ and V be a rule and valuation deriving $R(\bar{a}) \in \text{induc}_{\mathcal{P}}(D)$. Let $\varphi' \in \text{pure}_{\text{SZ}}(\mathcal{P})$ be obtained by applying transformation (3.2) 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, $\text{head}_{\psi} = R(x, s+1, \bar{a})$. Abbreviate $G_M(\mathcal{P}) = \text{ground}_M(\mathcal{P}', J)$ where $\mathcal{P}' = \text{pure}_{\text{SZ}}(\mathcal{P})$ and $J = \text{input}_{\text{SZ}}(H)$. We have $\psi \in G_M(\mathcal{P})$ because ψ is positive. We are left to show $\text{pos}_{\psi} \subseteq M$. Since $V(\text{pos}_{\varphi}) \subseteq D$ and $D \subseteq \text{all}_M(x, s)$ by the given assumption, we have $\text{pos}_{\psi} = V(\text{pos}_{\varphi})^{\uparrow x, s} \cup \{\text{tsucc}(s, s+1)\} \subseteq M$. \square

Claim C.3. Let S be the set of transition ordinals 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 in 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.

Proof. Necessarily, $R(y, \bar{a}) \in \text{async}_{\mathcal{P}}(D_i)$. Suppose we would know $D_i \subseteq \text{all}_M(x_i, t)$ for each $t \geq s_i$. Then Claim C.4 would imply that for each $t \geq s_i$ there is a value u such that $\text{chosen}_R(x_i, t, y, u, \bar{a}) \in M$, as desired.

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

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

For each $j \in S$, let $\rho_j = (st_j, bf_j)$ denote the source configuration of transition j . Since $D_j = \text{deduc}_{\mathcal{P}}(st_j(x_j) \cup \text{untag}(m_j))$ by the operational semantics, Claim C.1 implies it is sufficient to show for each $j \in S$ that

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

Algorithm 3 Positive but not consistent

$$\begin{aligned}
 A() &| \mathbf{x} \leftarrow \text{Id}(\mathbf{x}). \\
 B() &| \mathbf{x} \leftarrow \text{Id}(\mathbf{x}). \\
 T() &\leftarrow A(), B(). \\
 T() \bullet &\leftarrow T().
 \end{aligned}$$

As additional simplification, $st_j(x_j) \cup \text{untag}(m_j) = st_j(x_j)$ because j is a heart-beat transition. For the base case $j = \min(S)$, we have $st_j(x_j) = H(x_j)$ by the operational semantics. Then $\text{input}_{\text{SZ}}(H) \subseteq M$ implies $st_j(x_j) \subseteq \text{all}_M(x_j, t)$ for all $t \geq s_j$. For the induction hypothesis, let $j \in S$ with $j > \min(S)$; we assume for all $k \in S$ with $k < j$ that

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

For the inductive step, we show $st_j(x_j) \subseteq \text{all}_M(x_j, t)$ 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, $st_j(x_j) = \text{induc}_{\mathcal{P}}\langle D_k \rangle$. Now, the induction hypothesis on k gives $st_k(x_k) \subseteq \text{all}_M(x_k, u)$ for all $u \geq s_k$. Hence, by Claim C.1 we have $D_k \subseteq \text{all}_M(x_k, u)$ for all $u \geq s_k$. Then Claim C.2 gives $\text{induc}_{\mathcal{P}}\langle D_k \rangle \subseteq \text{all}_M(x_k, u+1)$ for all $u \geq s_k$. Using $st_j(x_j) = \text{induc}_{\mathcal{P}}\langle D_k \rangle$, $x_j = x_k$, and $s_j = s_k + 1$, we can equivalently say $st_j(x_j) \subseteq \text{all}_M(x_j, t)$ for all $t \geq s_j$. \square

Claim C.4. Let $(x, s) \in \mathcal{N} \times \mathbb{N}$ and let D be an instance over $\text{sch}(\mathcal{P})$. Suppose $D \subseteq \text{all}_M(x, s)$. For each $R(y, \bar{a}) \in \text{async}_{\mathcal{P}}\langle D \rangle$ with $y \in \mathcal{N}$ there exists a value t such that $\text{chosen}_R(x, s, y, t, \bar{a}) \in M$.

Proof. Let $R(y, \bar{a}) \in \text{async}_{\mathcal{P}}\langle D \rangle$ with $y \in \mathcal{N}$, derived by a rule φ and valuation V . By Lemma A.1, it suffices to show $\text{cand}_R(x, s, y, u, \bar{a}) \in M$ for some $u \in \mathbb{N}$.

Let $\varphi' \in \mathcal{P}$ be the original rule on which φ is based. Let $\varphi'' \in \text{pure}_{\text{SZ}}(\mathcal{P})$ be the result of applying transformation (3.3) 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 $\text{head}_{\psi} = \text{cand}_R(x, s, y, u, \bar{a})$. Because ψ is positive, we have $\psi \in \text{ground}_M(\mathcal{P}', I)$ where $\mathcal{P}' = \text{pure}_{\text{SZ}}(\mathcal{P})$ and $I = \text{input}_{\text{SZ}}(H)$. It remains to be shown that $\text{pos}_{\psi} \subseteq M$, so that $\text{head}_{\psi} \in M$. Transformation (3.3) implies $\text{pos}_{\psi} = V(\text{pos}_{\varphi})^{\uparrow x, s} \cup \{\text{all}(y), \text{time}(u)\}$. First, note $\{\text{all}(y), \text{time}(u)\} \subseteq \text{input}_{\text{SZ}}(H) \subseteq M$. Second, $V(\text{pos}_{\varphi})^{\uparrow x, s} \subseteq D^{\uparrow x, s} \subseteq \text{all}_M(x, s)^{\uparrow x, s} \subseteq M$. \square

D Positive and Not Consistent

Algorithm 3 gives a Dedalus program \mathcal{P} that is positive but not consistent.⁶ This example is inspired by the work of Marczak et al. [18]. Let H be an input distributed database instance for \mathcal{P} with at least one node x . In any fair run, x will send $A()$ and $B()$ to itself during every transition. But $T()$ is only created when we deliver $A()$ and $B()$ simultaneously. Some fair runs never do this. Hence, different fair runs can produce different outputs.

Remark: in the language Webdamlog, every positive program *is* consistent (also called *convergent*) [1]. This apparent paradox can be explained by the semantics of Webdamlog runs, which differ from the runs as defined here in that in Webdamlog, messages (called *delegations*) are always promptly delivered.

⁶Relation $\text{Id}^{(1)}$ is from Section 4.1.