On the CRON Conjecture

Tom J. Ameloot [*] and Jan Van den Bussche

Hasselt University & Transnational University of Limburg, Hasselt, 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 can show that positive Dedalus programs can tolerate non-causal messages, in the sense that they compute the correct answer 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 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

[*] PhD. Fellow of the Fund for Scientific Research – Flanders (FWO)

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 [21], 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. Preliminaries on databases and Dedalus are given in Sect. 2. In Sect. 3 we give an intuitive operational semantics for Dedalus. The formalization of non-causality, the CRON conjecture, and the related results are all in Sect. 4. We conclude in Sect. 5.

2 Preliminaries

2.1 Databases and Network

A *database schema* $\mathcal{D}$ is a nonempty 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 also write (R, k) as $R^{(k)}$.

We assume a countably infinite universe **dom** of atomic data values that includes the set $\mathbb{N}$ of natural numbers. A fact $\boldsymbol{f}$ with *predicate* R is of the form $R(a_1, \ldots, a_k)$ with $a_i \in \mathbf{dom}$ for each $i = 1, \ldots, k$. We say that a fact $R(a_1, \ldots, a_k)$ is over a database schema $\mathcal{D}$ if $R^{(k)} \in \mathcal{D}$. A *database instance* I over $\mathcal{D}$ is a set of facts over $\mathcal{D}$.

A *network* $\mathcal{N}$ is a nonempty finite subset of **dom**. Intuitively, $\mathcal{N}$ represents a set of identifiers of compute *nodes* involved in a distributed system. This model is general enough to represent distributed computing on any network topology, because we can restrict attention to programs where nodes only send messages to nodes to which they are explicitly linked, as expressed by input relations. Now, a *distributed database instance H over $\mathcal{N}$ and a database schema $\mathcal{D}$* is a total function mapping every node of $\mathcal{N}$ to a finite (normal) database instance over $\mathcal{D}$.

2.2 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$^\neg$ to represent updateable memory for the nodes of a network and to provide a mechanism for communication between these nodes. Here, we present Dedalus as Datalog$^\neg$ extended with annotations, which simplify the presentation.[1]

Let $\mathcal{D}$ be a database schema. Below, we write $\mathbf{B}\{\bar{\mathtt{w}}\}$, where $\bar{\mathtt{w}}$ is a tuple of variables, to denote any sequence β of atoms and negated atoms over database schema $\mathcal{D}$, such that the variables in β are precisely those in the tuple $\bar{\mathtt{w}}$. Also, let R be a relation name in $\mathcal{D}$. 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

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

So, inductive and asynchronous rules are basically normal Datalog$^\neg$ rules with respectively head-annotations "$\bullet$" and "$\mid \mathtt{y}$", where $\mathtt{y}$ is a variable. For asynchronous rules, the notation "$\mid \mathtt{y}$" means that the derived head facts are transferred ("piped") to the node represented by $\mathtt{y}$. Intuitively, deductive, inductive and asynchronous rules express respectively local computation, updateable memory, and message sending (cf. Sect. 3). We will only consider *safe* rules: all variables of these rules occur in at least one positive body atom. Moreover, because constants can always be represented by unary input relations, we will assume that no values of **dom** occur in the rules. For technical simplicity, we also assume that rule-bodies contain at least one positive 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 $\mathcal{D}$:

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

Definition 1. *A Dedalus* program over a schema $\mathcal{D}$ *is a set of deductive, inductive and asynchronous Dedalus rules over* $\mathcal{D}$*, such that the set of deductive rules is syntactically stratifiable.*

Let $\mathcal{P}$ be a Dedalus program. We write $sch(\mathcal{P})$ to denote the schema that $\mathcal{P}$ is over. We define $idb(\mathcal{P}) \subseteq sch(\mathcal{P})$ to be the relations that occur in rule-heads of $\mathcal{P}$. We abbreviate $edb(\mathcal{P}) = sch(\mathcal{P}) \setminus idb(\mathcal{P})$. An *input* for $\mathcal{P}$ is a distributed database instance H over some network $\mathcal{N}$ and the schema $edb(\mathcal{P})$.

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

3 Operational Semantics

Let $\mathcal{P}$ be a Dedalus program. Let H be an input distributed database instance for $\mathcal{P}$, over a network $\mathcal{N}$. We give an operational semantics for Dedalus, which respects causality. This operational semantics is in line with earlier formal work on declarative networking [10, 19, 13, 6, 1]. In Sect. 4, we will contrast the operational semantics with a non-causal semantics, to formalize the CRON conjecture.

In this section we will sketch the most important concepts of the operational semantics. The interested reader can consult the formal details in the appendix. To represent a possible execution of $\mathcal{P}$ on input H, we use a *run*. A run consists of *configurations* and *transitions*. A configuration describes for each node of $\mathcal{N}$ the facts that it has stored locally (state), and also what messages are in flight on the network. At the beginning, the *start configuration* assigns to each node only its local input fragment in H, and there are no messages. Now, a transition transforms one configuration into another: it selects one *active node* $x \in \mathcal{N}$ to receive some messages addressed to x and to do a local computation. Specifically, the active node x reads its old state together with the received messages. The node then executes the deductive rules to "complete" these facts, using the stratified semantics. We consider the resulting set D of deductive facts as being "all" facts that x locally has during the transition. Next, the inductive rules are given input D, and the derived facts are stored in the next state of x, always together with the local input fragment of x in H (which is preserved). Similarly, the asynchronous rules are also given input D, and the derived facts are considered messages that are sent around the network. The first component in these facts represents the addressee. The resulting configuration reflects all these actions taken by x. Then, a run $\mathcal{R}$ is an infinite sequence of such transitions, initially departing from the start configuration. Natural fairness conditions are imposed: we consider only runs in which each node is made active an infinite number of times and every sent message is eventually delivered. The operational semantics closely corresponds to that of the language Webdamlog [1].

This operational semantics is highly nondeterministic because in each transition we can choose which node is made active and also what messages it receives (from those that are in flight).

Assume a subset $\mathit{out}(\mathcal{P}) \subseteq \mathit{idb}(\mathcal{P})$, called the *output schema*, is selected: the relation names in this schema 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 $\boldsymbol{f}$ over schema $\mathit{out}(\mathcal{P})$ is *ultimate* at some node x if there is some transition after which $\boldsymbol{f}$ is present at x during every subsequent transition of x once the deductive rules are executed. Thus, this is a fact that will eventually always be present at x. The output of $\mathcal{R}$, denoted $\mathit{output}(\mathcal{R})$, is the union of all ultimate facts over all nodes. In this definition we ignore what node is responsible for what piece of the output, which follows the intuition of cloud computing. Since the operational semantics is nondeterministic, there can be different runs producing a different output. Program $\mathcal{P}$ is called *consistent* if individually for every input distributed database instance H, every run produces the *same* output, which we denote as $\mathit{outInst}(\mathcal{P}, H)$. This is an instance over *out*($\mathcal{P}$). Guaranteeing or deciding consistency in special cases is an important research topic [1, 18, 7].

As some additional terminology, in a run, for each transition t, we define the *timestamp* of the active node x during t to be the number of transitions of x that come strictly before t. This can be thought of as the local (zero-based) clock of x during t. 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.

4 CRON 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, then 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". We introduce such a formalism in Sect. 4.1, and in Sect. 4.2 we look at our formalizations of the CRON conjecture and the associated results.

4.1 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 [12]. There, we described a *causality transform* that converts a Dedalus program to a pure Datalog$^\neg$ program containing extra rules, called the *causality rules*. When the stable model semantics is applied to this pure Datalog$^\neg$ program, these rules enforce causality on message sending. For the current work, we will remove the causality rules, and now stable models can represent non-causal message sending.

Let $\mathcal{P}$ be a Dedalus program. Below, we present the *SZ-transformation* that transforms $\mathcal{P}$ into $\mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$, which is a pure Datalog$^\neg$ program that models the distributed computation in a holistic fashion: the distributed data of a network across all nodes and their local timestamps is modeled as facts of the form $R(x, s, \bar{a})$, representing that the fact $R(\bar{a})$ is present at node x at its timestamp s. In $\mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$, for asynchronous rules, we also 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].

In $\mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$, we will use relations of the following database schema:

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

We may assume that these relations are not in $sch(\mathcal{P})$, which can be solved with namespaces if needed. The relation '$\neq$' will be written in infix notation. We consider only the following instance over $\mathcal{D}_{\mathrm{time}}$:

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

This instance provides timestamps, together with a non-equality relation. Next, we will specify $pure_{\mathrm{SZ}}(\mathcal{P})$ incrementally. Let $\mathtt{x}$, $\mathtt{s}$, $\mathtt{t}$ and $\mathtt{t}'$ be variables not yet used in $\mathcal{P}$. For any sequence L of atoms and negated atoms, let $L^{\Uparrow \mathtt{x},\mathtt{s}}$ denote the sequence obtained by adding $\mathtt{x}$ and $\mathtt{s}$ as first and second components to each atom in L (negated atoms stay negated).

For each *deductive* rule '$R(\bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}$' in $\mathcal{P}$, we add to $pure_{\mathrm{SZ}}(\mathcal{P})$ the following rule:

$$R(\mathtt{x}, \mathtt{s}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}^{\Uparrow \mathtt{x},\mathtt{s}}. \tag{1}$$

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

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

$$R(\mathtt{x}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}\}^{\Uparrow \mathtt{x},\mathtt{s}},\, \mathtt{tsucc}(\mathtt{s}, \mathtt{t}). \tag{2}$$

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

We will also assume that the following relation names are not in $sch(\mathcal{P})$: name $\mathtt{All}$, and the names $\mathtt{cand}_R$, $\mathtt{chosen}_R$ and $\mathtt{other}_R$ for each name R in $idb(\mathcal{P})$. Now, for each asynchronous rule '$R(\bar{\mathtt{u}}) \mid \mathtt{y} \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}$' in $\mathcal{P}$, we add to $pure_{\mathrm{SZ}}(\mathcal{P})$ the following rules, for which the intuition is given below:

$$\mathtt{cand}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathbf{B}\{\bar{\mathtt{u}}, \bar{\mathtt{v}}, \mathtt{y}\}^{\Uparrow \mathtt{x},\mathtt{s}},\, \mathtt{All}(\mathtt{y}),\, \mathtt{time}(\mathtt{t}). \tag{3}$$

$$\mathtt{chosen}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathtt{cand}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}),\, \neg\mathtt{other}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}). \tag{4}$$

$$\mathtt{other}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathtt{cand}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}),\, \mathtt{chosen}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}', \bar{\mathtt{u}}),\, \mathtt{t} \neq \mathtt{t}'. \tag{5}$$

$$R(\mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}) \leftarrow \mathtt{chosen}_R(\mathtt{x}, \mathtt{s}, \mathtt{y}, \mathtt{t}, \bar{\mathtt{u}}). \tag{6}$$

A fact of the form $\mathtt{All}(x)$ means that x is a node of the network. Rule (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 derived facts, we include the sender's location and send-timestamp, the addressee's location and arrival-timestamp, and the actual transmitted data. Next, rules (4) and (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 (6) models the actual arrival of messages, where the sender-information is projected away, and the data-tuple in the message becomes part of the addressee's state for relation R. We repeat the above transformation for all asynchronous rules in $\mathcal{P}$, and $pure_{\mathrm{SZ}}(\mathcal{P})$ is now completed. Remark: multiple asynchronous rules in $\mathcal{P}$ can have the same head predicate R, and after the above transformation, there can be multiple rules with head predicates $\mathtt{cand}_R$, $\mathtt{chosen}_R$, $\mathtt{other}_R$ and R.

Let H be an input for $\mathcal{P}$, over a network $\mathcal{N}$. We define

$$\mathit{input}_{\mathrm{SZ}}(H) = \bigcup_{x\in\mathcal{N}} \bigcup_{s\in\mathbb{N}} \{R(x,s,\bar{a}) \mid R(\bar{a}) \in H(x)\} \cup \{\mathtt{All}(x) \mid x \in \mathcal{N}\} \cup I_{\mathtt{time}} \enspace .$$

Intuitively, in $\mathit{input}_{\mathrm{SZ}}(H)$, for each node its input facts are available at each of its local timestamps; relation `All` represents the network; and all timestamps are provided, together with a non-equality relation. Now, we call any stable model M of $\mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$ on input $\mathit{input}_{\mathrm{SZ}}(H)$ an *SZ-model* of $\mathcal{P}$ on input H. Program $\mathit{pure}_{\mathrm{SZ}}(\mathcal{P})$ does not enforce causality on the messages in M since the arrival timestamps can be chosen arbitrarily, even into the past.

Similar to [3], we only consider "fair" models, defined as follows. We say that an SZ-model M is *fair* if for each pair $(y,t) \in \mathcal{N} \times \mathbb{N}$ there are only a finite number of facts in M of the form $\mathtt{chosen}_R(x,s,y,t,\bar{a})$. This expresses that every node receives only a finite number of messages at any given timestamp. We focus on fair models because in reality a node always processes a finite number of messages at each computation step.

We define the *output* of an SZ-model M, denoted *output*(M), as

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

Thus, we use the intuition of ultimate facts, as was used in the operational semantics (cf. Sect. 3). Now, a consistent program $\mathcal{P}$ is called *SZ-consistent* if individually for every input distributed database instance H, every SZ-model M yields the output $\mathit{outInst}(\mathcal{P}, H)$. Intuitively, if a consistent program is SZ-consistent, then it also computes the same result when messages can be sent into the past.

4.2 Results

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 $\mathcal{D}_2$. A Dedalus program $\mathcal{P}$ can compute a query as follows: we say that $\mathcal{P}$ *(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 partition H of I over $\mathcal{N}$, we have $\mathit{outInst}(\mathcal{P}, H) = \mathcal{Q}(I)$. To compute non-monotone queries, every node needs its own identifier and the identifiers of the other nodes, or equivalent information [6]. Therefore, we restrict attention to Dedalus programs $\mathcal{P}$ for which $\{\mathtt{Id}^{(1)}, \mathtt{Node}^{(1)}\} \subseteq \mathit{edb}(\mathcal{P})$, where relation `Id` is initialized to contain on every node the identifier of that node, and relation `Node` is initialized to contain the identifiers of all nodes (including the local node). These node identifiers are not considered part of the query input. In this context, we have looked at the following formalization of the CRON conjecture:

> 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. First, for the if-direction, we give a Dedalus program that computes 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):

$$\begin{array}{ll} \mathtt{empty}(x) \mid y \leftarrow \neg S(\,),\, \mathtt{Id}(x),\, \mathtt{Node}(y). & \mathtt{notDone}(\,) \leftarrow \mathtt{Node}(y),\, \neg\mathtt{empty}(y). \\ \mathtt{empty}(y)\bullet \leftarrow \mathtt{empty}(y). & T(\,) \leftarrow \mathtt{Id}(x),\, \neg\mathtt{notDone}(\,). \end{array}$$

Here, 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 rules on the right let a node output $T(\,)$ starting at the moment that it has all identifiers (including its own). This program is consistent. There are no causal message dependencies, so it does not really matter at what time a node receives some identifier: in every SZ-model, after a while this node will still have received and stored the identifier. Thus every SZ-model yields the output $T(\,)$ iff all nodes have an empty relation S. The program is SZ-consistent.

Second, for the only-if direction, we give a (contrived) Dedalus program that computes 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):

$$\begin{array}{ll} A(\,) \mid x \leftarrow S(\,),\, \mathtt{Id}(x). & B(\,) \mid x \leftarrow A(\,),\, \neg\mathtt{sent}_B(\,),\, \mathtt{Id}(x). \\ A(\,)\bullet \leftarrow A(\,). & T(\,) \leftarrow A(\,),\, B(\,). \\ \mathtt{sent}_B(\,)\bullet \leftarrow A(\,). & T(\,)\bullet \leftarrow T(\,). \end{array}$$

Here, when a node has a nonempty relation S, it sends $A(\,)$ to itself continuously. On receipt of $A(\,)$, it stores this fact, and it 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 indefinitely. The program is consistent, but is however 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 $\mathtt{sent}_B(\,)$ 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, and to let $B(\,)$ 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 is not SZ-consistent.

So, contrary to the CALM conjecture [15, 6, 22], a formalization of the CRON conjecture that is situated purely on the semantical level does not seem to give any results. A Dedalus program without negation is called *positive*. Our main result now is that the following does hold:

Theorem 1. *Every positive, consistent Dedalus program is SZ-consistent.*

Note that the converse direction of Theorem 1, to the effect that every SZ-consistent Dedalus program is equivalent to a positive program, cannot hold by our above counterexample for the if-direction. We sketch the proof of Theorem 1. Let $\mathcal{P}$ be a positive, *consistent* Dedalus program, and let H be an input for $\mathcal{P}$. Let M be an SZ-model of $\mathcal{P}$ on H. To show $\mathit{output}(M) \subseteq \mathit{outInst}(\mathcal{P}, H)$, we show $\mathit{output}(M) \subseteq \mathit{output}(\mathcal{R})$ where $\mathcal{R}$ is the run of $\mathcal{P}$ on H that operates in rounds: in every round all nodes empty their entire buffer, and this run saturates towards the derivation of all "possible" deductive facts per node. To show $\mathit{outInst}(\mathcal{P}, H) \subseteq \mathit{output}(M)$, we convert M to a run $\mathcal{R}$ in which we create no more opportunities for messages to "join" in comparison to M. Concretely, we make sure that in $\mathcal{R}$ we only send messages that are sent an infinite number of times in M, which, by the "fairness" assumption on M, allows us to pick an arrival time that is *still* represented by M. Hence, $\mathit{output}(\mathcal{R}) \subseteq \mathit{output}(M)$.

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.

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

References

[1] Abiteboul, S., Bienvenu, M., Galland, A., et al.: A rule-based language for Web data management. In: Proceedings 30th ACM Symposium on Principles of Database Systems, pp. 293–304. ACM Press (2011)

[2] Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley (1995)

[3] Alvaro, P., Ameloot, T.J., Hellerstein, J.M., Marczak, W., Van den Bussche, J.: A declarative semantics for dedalus. Technical Report UCB/EECS-2011-120, EECS Department, University of California, Berkeley (November 2011)

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

[5] Alvaro, P., Marczak, W.R., Conway, N., Hellerstein, J.M., Maier, D., Sears, R.: Dedalus: Datalog in Time and Space. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A. (eds.) Datalog 2010. LNCS, vol. 6702, pp. 262–281. Springer, Heidelberg (2011)

[6] Ameloot, T.J., Neven, F., Van den Bussche, J.: Relational transducers for declarative networking. In: Proceedings 30th ACM Symposium on Principles of Database Systems, pp. 283–292. ACM Press (2011)

[7] Ameloot, T.J., Van den Bussche, J.: Deciding eventual consistency for a simple class of relational transducers. In: Proceedings of the 15th International Conference on Database Theory, pp. 86–98. ACM Press (2012)

[8] Apt, K.R., Francez, N., Katz, S.: Appraising fairness in languages for distributed programming. Distributed Computing 2, 226–241 (1988)

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

[10] Deutsch, A., Sui, L., Vianu, V., Zhou, D.: Verification of communicating data-driven Web services. In: Proceedings 25th ACM Symposium on Principles of Database Systems, pp. 90–99. ACM Press (2006)

[11] Francez, N.: Fairness. Springer-Verlag New York, Inc., New York (1986)

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

[13] Grumbach, S., Wang, F.: Netlog, a Rule-Based Language for Distributed Programming. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 88–103. Springer, Heidelberg (2010)

[14] Hellerstein, J.M.: Datalog redux: experience and conjecture. Video available (under the title The Declarative Imperative) (2010), PODS 2010 keynote `http://db.cs.berkeley.edu/jmh/`

[15] Hellerstein, J.M.: The declarative imperative: experiences and conjectures in distributed logic. SIGMOD Record 39(1), 5–19 (2010)

[16] Lamport, L.: Fairness and hyperfairness. Distributed Computing 13, 239–245 (2000)

[17] Loo, B.T.: et al. Declarative networking. Communications of the ACM 52(11), 87–95 (2009)

[18] Marczak, W., Alvaro, P., Conway, N., Hellerstein, J.M., Maier, D.: Confluence analysis for distributed programs: A model-theoretic approach. Technical Report UCB/EECS-2011-154, EECS Department, University of California, Berkeley (December 2011)

[19] Navarro, J.A., Rybalchenko, A.: Operational Semantics for Declarative Networking. In: Gill, A., Swift, T. (eds.) PADL 2009. LNCS, vol. 5418, pp. 76–90. Springer, Heidelberg (2008)

[20] Saccà, D., Zaniolo, C.: Stable models and non-determinism in logic programs with negation. In: Proceedings of the Ninth ACM Symposium on Principles of Database Systems, pp. 205–217. ACM Press (1990)

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

[22] Zinn, D., Green, T.J., Ludaescher, B.: Win-move is coordination-free. In: Proceedings of the 15th International Conference on Database Theory, pp. 99–113. ACM Press (2012)

Appendix

A Operational Semantics

Let $\mathcal{P}$ be a Dedalus program. Let H be an input distributed database instance for $\mathcal{P}$, over a network $\mathcal{N}$. We define formally an operational semantics for Dedalus.

A.1 Subprograms

We split the program $\mathcal{P}$ into three subprograms, that contain respectively the deductive, inductive and asynchronous rules. First, we define $\mathit{deduc}_{\mathcal{P}}$ to be the Datalog$^{\neg}$ program consisting of precisely all deductive rules of $\mathcal{P}$. Secondly, we define $\mathit{induc}_{\mathcal{P}}$ to be the Datalog$^{\neg}$ program consisting of all inductive rules of $\mathcal{P}$ after the annotation "$\bullet$" in their head is removed. Thirdly, we define $\mathit{async}_{\mathcal{P}}$ to be the Datalog$^{\neg}$ program consisting of precisely 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 $\mathcal{P}$. The first component in the rules of $\mathit{async}_{\mathcal{P}}$ will represent the addressee of messages. The programs $\mathit{deduc}_{\mathcal{P}}$, $\mathit{induc}_{\mathcal{P}}$ and $\mathit{async}_{\mathcal{P}}$ are just Datalog$^{\neg}$ programs over the schema $\mathit{sch}(\mathcal{P})$, or a sub-schema thereof. Moreover, $\mathit{deduc}_{\mathcal{P}}$ is syntactically stratifiable because the deductive rules in $\mathcal{P}$ must be syntactically stratifiable. The semantics of these subprograms is given below.

Let I be an instance over $\mathit{sch}(\mathcal{P})$. We define the *output of* $\mathit{deduc}_{\mathcal{P}}$ *on input* I, denoted as $\mathit{deduc}_{\mathcal{P}}(I)$, to be given by the stratified semantics [2]. This implies $I \subseteq \mathit{deduc}_{\mathcal{P}}(I)$. We define the *output of* $\mathit{induc}_{\mathcal{P}}$ *on input* I to be the set of facts derived by the rules of $\mathit{induc}_{\mathcal{P}}$ for all possible satisfying valuations in I, in just one derivation step. This output is denoted as $\mathit{induc}_{\mathcal{P}}\langle I \rangle$. The *output of* $\mathit{async}_{\mathcal{P}}$ *on input* I is defined in the same way as for $\mathit{induc}_{\mathcal{P}}$, except that we now use the rules of $\mathit{async}_{\mathcal{P}}$ instead of $\mathit{induc}_{\mathcal{P}}$. This output is denoted as $\mathit{async}_{\mathcal{P}}\langle I \rangle$.

A.2 Configurations

A *configuration* ρ of $\mathcal{P}$ on input H is a pair $(\mathrm{st}^{\rho}, \mathrm{bf}^{\rho})$ where st^{ρ} is a function that maps each node of $\mathcal{N}$ to a set of facts over $\mathit{sch}(\mathcal{P})$, and bf^{ρ} is a function that maps each node of $\mathcal{N}$ to a set of pairs of the form $\langle i, \boldsymbol{f} \rangle$, where $i \in \mathbb{N}$ and $\boldsymbol{f}$ is a fact over $\mathit{idb}(\mathcal{P})$. The set st^{ρ} represents the *state* of each node. The set bf^{ρ}, called *(message) buffer*, represents for each node all messages addressed to that node but that are not yet received. The reason for having numbers i, called *send-tags*, attached to facts in the image of bf^{ρ} is to differentiate between multiple instances of the same message being sent at different moments (to the same addressee), and these tags are not visible to the Dedalus program. The *start configuration* of $\mathcal{P}$ on input H, denoted $\mathit{start}(\mathcal{P}, H)$, is the configuration ρ defined for each $x \in \mathcal{N}$ as $\mathrm{st}^{\rho}(x) = H(x)$ and $\mathrm{bf}^{\rho}(x) = \emptyset$.

A.3 Transitions and Runs

To transform one configuration ρ_a into another configuration ρ_b, we describe *transitions* in each of which one active node does a local computation and possibly sends messages around the network. Such transitions can be chained to form a *run* that describes a full execution of the Dedalus program on the given input. As a small notational aid, for a set m of pairs of the form $\langle i, \boldsymbol{f}\rangle$, we define $\mathit{untag}(m) = \{\boldsymbol{f} \mid \exists i \in \mathbb{N} : \langle i, \boldsymbol{f}\rangle \in m\}$. Now, a *transition with send-tag* $i \in \mathbb{N}$ is a five-tuple $(\rho_a, x, m, i, \rho_b)$ such that ρ_a and ρ_b are configurations of $\mathcal{P}$ on input H, $x \in \mathcal{N}$, $m \subseteq \mathrm{bf}^{\rho_a}(x)$, and, letting

$$\begin{aligned} I = {}& \mathrm{st}^{\rho_a}(x) \cup \mathit{untag}(m), \qquad D = \mathit{deduc}_{\mathcal{P}}(I), \\ \delta^{i \to y} = {}& \{\langle i,\, R(\bar{a})\rangle \mid R(y, \bar{a}) \in \mathit{async}_{\mathcal{P}}\langle D\rangle\} \text{ for each } y \in \mathcal{N}, \end{aligned}$$

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

$$\begin{aligned} \mathrm{st}^{\rho_b}(x) &= H(x) \cup \mathit{induc}_{\mathcal{P}}\langle D\rangle, & \mathrm{st}^{\rho_b}(y) &= \mathrm{st}^{\rho_a}(y), \\ \mathrm{bf}^{\rho_b}(x) &= (\mathrm{bf}^{\rho_a}(x) \setminus m) \cup \delta^{i \to x}, & \mathrm{bf}^{\rho_b}(y) &= \mathrm{bf}^{\rho_a}(y) \cup \delta^{i \to y} \quad . \end{aligned}$$

We say that this transition is *of* the *active* node x. The transition models that the active node x reads its old state $\mathrm{st}^{\rho_a}(x)$ together with the received facts in $\mathit{untag}(m)$ (thus without the tags), and then completes this information with subprogram $\mathit{deduc}_{\mathcal{P}}$. Next, the state of x is changed to $\mathrm{st}^{\rho_b}(x)$, which always contains the input facts of x, over schema $\mathit{edb}(\mathcal{P})$, and it also includes all facts derived by subprogram $\mathit{induc}_{\mathcal{P}}$, which is applied to the deductive fixpoint. This represents that input facts are never lost, and that the facts over $\mathit{idb}(\mathcal{P})$ that are explicitly derived by $\mathit{induc}_{\mathcal{P}}$ are remembered. Only the state of x changes. The facts generated by $\mathit{async}_{\mathcal{P}}$ are called *messages*. By the syntax of $\mathit{async}_{\mathcal{P}}$, these facts have an additional first component to indicate the addressee. For each $y \in \mathcal{N}$, the set $\delta^{i \to y}$ contains all messages addressed to y: we drop the addressee-component because it is now redundant, and we attach the send-tag i. The set $\delta^{i \to y}$ is added to the buffer of y. We ignore messages with an addressee outside $\mathcal{N}$.

A *run* $\mathcal{R}$ of $\mathcal{P}$ on input H is an infinite sequence of transitions, such that *(i)* the very first configuration is $\mathit{start}(\mathcal{P}, H)$, *(ii)* the output configuration of each transition is the input configuration for the next transition, and *(iii)* the transition at ordinal i of the sequence uses send-tag i. The transition system is highly non-deterministic because in each transition we can choose the active node and also what messages to deliver. Note that messages with a valid addressee are never lost.

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.