DBSP: A Language for Expressing Incremental View Maintenance for Rich Query Languages — full version

Mihai Budiu
VMware Research

Frank McSherry
Materialize Inc.

Leonid Ryzhyk
VMware Research

Val Tannen
University of Pennsylvania

March 28, 2022

Abstract

Incremental view maintenance has been for a long time a central problem of database theory. Many solutions have been proposed for restricted classes of database languages, such as the relational algebra, or Datalog. These techniques do not naturally generalize to richer languages. In this paper we give a general solution to this problem in 3 steps: (1) we describe a simple but expressive language called DBSP for describing computations over data streams; (2) we give a general algorithm for solving the incremental view maintenance problem for arbitrary DBSP programs, and (3) we show how to model several classes of database query languages (including relational queries, grouping and aggregation, and recursion) using DBSP. DBSP can be used to model other interesting query languages, including streaming aggregation queries, non-monotonic recursion, and queries on nested relations. As a consequence, we obtain efficient incremental view maintenance techniques for all these rich languages.

This document is work in progress. It contains a formal specification of the DBSP language, proofs of the theoretical results, and the specification of the implementation of several query languages in DBSP.

Contents

1 Introduction

In this paper we present a simple mathematical theory for modeling streaming and incremental computations. This model has immediate practical applications in the design and implementation of streaming databases and incremental view maintenance. Our model is based on mathematical formalisms used in discrete digital signal processing (DSP) [36], but we apply it to database computations. Thus, we have called it "DBSP".

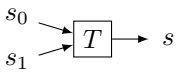


Figure 1: A simple streaming circuit that consumes two streams and produces one stream.

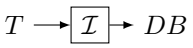
We borrow the following concepts from DSP: discrete signal streams, the delay operator z^{-1}, time-invariance (also called shift-invariance), causality, strictness, z-transforms, integration, and differentiation. The concept of linear transformation is also central to DSP.

DBSP is also inspired from Differential Dataflow [32] (DD), and started as an attempt to provide a simpler formalization of DD than the one of Abadi et al. [2] (as discussed in Section 2), but has evolved behind that purpose.

The core concept of DBSP is the *stream*: a stream s with type $\mathcal{S}_A$ maps "time" moments $t \in \mathbb{N}$ to values $s[t]$ of type A; think of it as an "infinite vector". A streaming computation is a function that consumes one or more streams and produces another stream. We depict streaming computations with typical DSP box-and-arrow diagrams (also called "circuits"), where boxes are computations and streams are arrows, as in the diagram from Figure 1, which shows a stream operator T consuming two input streams s_0 and s_1 and producing one output stream s.

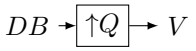
We generally think of streams as sequences of *small* values, and we will use them in this way. However, we make a leap of imagination and also treat a *whole database* as a stream value. What is a stream of databases? It is a *sequence of database snapshots*. We model the time-evolution of a database DB as a stream $DB \in \mathcal{S}_{SCH}$, where SCH is the database schema. Time is not the wall-clock time, but essentially a counter of the *sequence of transactions* applied to the database. Since transactions are linearizable, they have a total order, which defines a linear time t dimension: the value of the stream $DB[t]$ is the snapshot of the database contents after t transactions have been applied. We assume that $DB[0] = 0$, i.e., the database starts empty.

Database transactions also form a stream T, a stream of *changes*, or *deltas* that are applied to our database. The database snapshot at time t is the cumulative result of applying all transactions in the sequence up to t: $DB[t] = \sum_{i \leq t} T[i]$. The operation of adding up all changes is *stream integration*. The following diagram expresses this relationship using the $\mathcal{I}$ operator for stream integration:

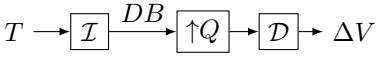


Conversely, we can say that transactions are the *changes* of a database, and write $T = \mathcal{D}(DB)$. Stream differentiation, denoted by $\mathcal{D}$, is an operation that computes the changes of a stream, and is the inverse of stream integration. Section 3 precisely defines streams, integration and differentiation, and analyzes their properties.

Let us apply these concepts to view maintenance. Consider a database DB and a query Q defining a view V as a function of a database snapshot $V = Q(DB)$. Corresponding to the stream of database snapshots DB we have a *stream of view snapshots*: $V[t]$ is the view's contents after the t-th transaction has been applied. We show this relationship using the following diagram:



The symbol $\uparrow Q$ (the "lifting" of Q) shows that the query Q is applied pointwise to every element of the stream of database snapshots DB. We say that $\uparrow Q$ is a "streaming query" since it operates on a stream of values. The incremental view maintenance problem requires an algorithm to compute the stream ΔV of *changes* of the view V, i.e., $\mathcal{D}(V)$, as a function of the stream T. By chaining these definitions together we get the following **fundamental equation** of the view maintenance problem: $\Delta V = \mathcal{D}(\uparrow Q(DB)) = \mathcal{D}(\uparrow Q(\mathcal{I}(T)))$, graphically shown as:



This definition is the subject of Section 4. The notion can be generalized to more general streaming queries $S : \mathcal{S}_A \to \mathcal{S}_B$ that are richer than lifted pointwise queries Q. The incremental version of streaming query S is denoted by S^Δ and is defined according to the above equation, which can also be written as: $S^\Delta = \mathcal{D} \circ S \circ \mathcal{I}$.

It is generally assumed that the changes to a dataset are much smaller than the dataset itself; thus, computing on streams of changes may produce significant performance benefits.

Applying the query *incrementalization* operator $S \mapsto S^\Delta$ constructs a query that computes directly on changes; however, the resulting query is no more efficient than a query that computes on the entire dataset, because it uses an integration operator to reconstitute the full dataset. Section 4 shows algebraic properties of the $\cdot^\Delta$ operator that can used to optimize the implementation of S^Δ:

(1) The first property is that many classes of primitive operations have very efficient incremental versions. In particular, linear queries have the property $Q = Q^\Delta$. Almost all relational and Datalog queries are based on linear operators. Thus, the incremental version of such queries can be computed in time proportional to the size of the changes. Bilinear operators (such as joins) have a more complex implementation, which nevertheless still performs work proportional to the size of the changes, but require storing an amount of data proportional to the size of the relations.

(2) The second key property is the chain rule: $(S_1 \circ S_2)^\Delta = {S_1}^\Delta \circ {S_2}^\Delta$. This rule gives the incremental version of a complex streaming query as a composition of incremental versions of its components. It follows that we

can implement any incremental query as a composition of primitive incremental queries, *all of which perform work proportional to the size of the changes*.

While this machinery is sufficient to express the whole relational algebra and recursive queries, it is not sufficient to incrementalize recursive queries. So in Section 5.2 we extend DBSP to operate on nested streams.

First, in Section 5.1 we introduce two additional operators: δ_0 creates a stream from a scalar value, and $\int$ creates a scalar value from a stream. These operators can be used to implement computations with `while` loops. So, in addition to modelling changing inputs and database, we also use streams as a model for sequences of *consecutive values of loop iteration variables*. This model will enable us to implement recursive queries.

In Section 5.2 we use DBSP to model computations on nested streams, where each value of a stream is another stream. With this extension DBSP becomes rich enough to model incremental streaming recursive queries.

Armed with (a) a language that can express many classes of queries and (b) a general theory of incremental computation, we proceed in a sequence of sections to apply these techniques to richer and richer query languages.

In Section 6 we show how to model relational queries in DBSP. This immediately gives us a general algorithm to compute the incremental version of any relational query. These results are well-known, but they are cleanly modeled by DBSP.

Section 7 shows how recursive queries can be implemented in DBSP. This allows us to define in Section 7.2 *incremental streaming computations for recursive programs*. As a consequence we derive a universal algorithm for incrementalizing arbitrary streaming Datalog programs.

Section 8 shows how other classes of powerful query languages can be modeled in DBSP: streaming window queries, queries on nested relations (such as grouping), non-monotone recursive queries, and while-relational queries.

Interestingly, the incrementalization algorithm can applied to all DBSP programs.

DBSP is a **simple** language: the basic DBSP streaming model is built essentially from two elementary mathematical operators: lifting $\uparrow$ and delay z^{-1}. The nested streams model adds two additional operators, δ_0 for stream construction and $\int$ for destruction.

Despite its simplicity, DBSP is also **expressive**, and we show how many useful query languages can be modeled in DBSP.

DBSP is **mathematically abstract**: the two core concepts of (1) streaming computations, and (2) incremental computations are completely abstract, since they work over any algebraic group structure. By instantiating the group structure with the real numbers $\mathbb{R}$, we get traditional signal processing systems. By substituting for the group structure $\mathbb{Z}$-sets we obtain streaming incremental computations over databases.

1.1 Notations used in this paper

The following tables summarize the mathematical notations used in the rest of this paper.

General notations	
$\mathbb{Z}$	The ring of integer numbers
$\mathbb{N}$	The set of natural numbers $0, 1, 2, \ldots$
$\mathbb{B}$	The set of Boolean values
$[n]$	The natural numbers between 0 and $n-1$
id	The identity function over some domain $id : A \to A$, $id(x) = x$
$[\![Q]\!]$	Semantics of query (function) Q
$\langle a, b \rangle$	The pair containing values a and b
$\mathrm{fst}(p)$	The operator that returns the first value of a pair p
$\mathrm{snd}(p)$	The operator that returns the second value of a pair p
$a \mapsto b$	The function that maps a to b and everything else to 0
$\lambda x.M$	An anonymous function with argument x and body M
$\mathrm{fix}\, x.f$	The (unique) solution (fixed point) of the equation $f(x) = x$

Streams	
$\mathcal{S}_A$	The set of streams with elements from a group A; $\mathcal{S}_A = \{f \mid f : \mathbb{N} \to A\}$
$\overline{\mathcal{S}_A}$	Streams with elements from a group A that are 0 almost everywhere
$s[t]$	The t-th element of a stream; $s[t] = s(t)$
$\uparrow f$	An operator applied to a function $f : A \to B$ to produce a function $\uparrow f : \mathcal{S}_A \to \mathcal{S}_B$ operating pointwise
zpp(f)	zpp(f) iff $f(0) = 0$ for $f : A \to B$ for A, B groups
z^{-1}	The stream delay operator $z^{-1} : \mathcal{S}_A \to \mathcal{S}_A$, that outputs a 0 followed by the input stream
$\mathcal{I}$	The stream integration operator $\mathcal{I} : \mathcal{S}_A \to \mathcal{S}_A$
$\mathcal{D}$	The stream differentiation operator $\mathcal{D} : \mathcal{S}_A \to \mathcal{S}_A$
Q^Δ	The incremental version of an operator $Q^\Delta = \mathcal{D} \circ Q \circ \mathcal{I}$
$s\|_{\leq t}$	A stream that has the same prefix as s up to t, then it is all 0s
$s\|_{< t}$	A stream that has the same prefix as s up to $t-1$, then it is all 0s
$\cong$	Symbol that indicates that two circuits compute the same function
δ_0	A function that produces a stream from a scalar: scalar, followed by zeros
$\int$	A function that produces a scalar by adding all elements of a stream
E	$E = \mathcal{I} \circ \delta_o$
X	$X = \int \circ \mathcal{D}$
$\mathbb{Z}$-sets	
$\mathbb{Z}[A]$	$\mathbb{Z}$-sets: finite functions from $A \to \mathbb{Z}$
$\|\|s\|\|$	Size of $\mathbb{Z}$-set s
isset	A function isset : $\mathbb{Z}[A] \to \mathbb{B}$ that determines whether its argument is a set
distinct	A function *distinct* : $\mathbb{Z}[A] \to \mathbb{Z}[A]$ that always returns a set
ispositive	A function ispositive : $\mathbb{Z}[A] \to \mathbb{B}$ that determines whether all elements of a $\mathbb{Z}$-set have positive weights
toszet	Function converting a set to a $\mathbb{Z}$-set
toset	Function converting a $\mathbb{Z}$-set into a set

2 Related work

2.1 Incremental View Maintenance

DBSP using non-nested streams is a simplified instance of a Kahn network [22]. Johnson [21] studies a very similar computational model without nested streams and its expressiveness. The implementation of such streaming models of computation and their relationship to dataflow machines has been studied by Lee [27]. Lee [26] also introduced streams of streams and the $\uparrow z^{-1}$ operator.

In Section 8 we discuss the connections with window and stream database

queries [7, 1].

Incremental view maintenance (e.g. [17]) is surveyed in [16]; a large bibliography is present in [33]. Its most formal aspect is propagating "deltas" through algebraic expressions: $Q(R+\Delta R) = Q(R) + \Delta Q(R, \Delta R)$. This work eventually crystallized in [24]. DBSP incrementalization is both more modular and more fine-grain since it deals with streams of updates. Both [23] and [14] use $\mathbb{Z}$-sets to uniformly model insertions/deletions.

Picallo et al. [5] provide a general solution to IVM for rich languages. DBSP requires a group structure on the values operated on; this assumption has two major practical benefits: it simplifies the mathematics considerably (e.g., Picallo uses monoid actions to model changes), and it provides a general, simple algorithm (6.13) for incrementalizing arbitrary programs. The downside of DBSP is that one has to find a suitable group structure (e.g., $\mathbb{Z}$-sets for sets) to "embed" the computation. Picallo's notion of "derivative" is not unique: they need creativity to choose the right derivative definition, we need creativity to find the right group structure.

Many heuristic algorithms were published for Datalog-like languages, e.g., counting based approaches [11, 34] that maintain the number of derivations, DRed [18] and its variants [9, 38, 37, 25, 29, 6], the backward-forward algorithm and variants [34, 19, 33]. DBSP is more general than these approaches. Interestingly, the $\mathbb{Z}$-sets multiplicities in our relational implementation are related to the counting-number-of-derivation approaches.

DBSP is tightly related to Differential Dataflow (DD) [32, 35] and its theoretical foundations [2] (and recently [31, 10]). All DBSP operators are based on DD operators. DD's computational model is more powerful than DBSP, since it allows past values in a stream to be "updated". In contrast, our model assumes that the inputs of a computation arrive in the time order while allowing for nested time domains via the modular lifting transformer. However, DBSP can express both incremental and non-incremental computations; in essence DBSP is "deconstructing" DD into simple component building blocks; the core Proposition 4.2 and the Algorithm based on it 6.13 are new contributions.

2.2 Stream computation models

[12] surveys the connection between synchronous digital circuits and functional programs. Our circuits are nothing but higher order functions computing on streams (functions themselves). The paper's main focus are circuits processing numeric data, whereas, taking advantage of our circuits' ability to compute on arbitrary groups, we use circuits to implement incremental view maintenance for relational databases.

DB Toaster [4] and its associated theory [23, 24] provide a formal treatment of incremental view maintenance in relational query languages.

Reconcilable differences [14]

Provenance semi-rings [15]

Mamouras [30]

2.3 Connection to synchronous circuits

There is a vast literature on **synchronous circuits**, which are well-defined models for hardware circuits e.g. [12]. These circuits also compute over infinite streams of values, usually of Booleans $\mathcal{S}_{\mathbb{B}}$. In a **combinational circuit** the output values depend only on the current input values. These are pure lifted streaming computations. A **sequential circuit** can have outputs that depend on past input values. These are always causal circuits. Sequential synchronous circuits use latches or flip-flops to store state; the latches are controlled by a global clock signal. These correspond to the z^{-1} operator. In a well-formed sequential circuit all back-edges must go through some latch — this corresponds to our circuit construction rule that requires a delay element on each back-edge.

Languages such as Verilog or VHDL can be used to specify such circuits. (However, both Verilog and VHDL are strictly more powerful, and can express richer classes of circuits than just synchronous sequential circuits.)

There is a rich literature on synchronous circuits, and some of these results are directly applicable to the circuits we discuss. Here are a few examples.

Retiming [28] is an optimization that "moves" around delay elements while preserving the circuit semantics. Retiming is used traditionally to reduce the clock cycle by minimizing the signal propagation delay between any pair of latches. In our case it could be used for minimizing the amount of internal circuit state.

In a synchronous circuit the *state* is entirely stored in the latches. Saving and restoring the contents of the latches enables such circuits to take a snapshot of their state and resume computation.[1]

Fault tolerance of synchronous circuits is provided by replicating the state elements, to prevent accidental state changes caused by e.g., cosmic rays. We can borrow this idea for building redundant distributed computations.

Pipelining digital circuits is an effective technique for increasing throughput through parallelization, by inserting additional latches and allowing different pipeline stages to compute concurrently on distinct stream values, at the expense of increased latency between the inputs and the corresponding outputs.

Digital circuit latches depend on a special "reset" signal to initialize their state to a pre-established value; this corresponds to the special 0 value in our value domain.

Our nested streams are related to the notion of delta-cycles in the definition of VHDL [8].

[1] In Boolean synchronous circuits this is achieved by connecting all latches into a *scan chain* which can be read and written sequentially after stopping the circuit clock.

Part I
Streaming and incremental computations

3 Streams

3.1 Streams and stream operators

$\mathbb{N}$ is the set of natural numbers, $\mathbb{B}$ is the set of Booleans, and $\mathbb{Z}$ is the set of integers. $[n] = \{0, 1, 2, \ldots n-1\}$ is the set of natural numbers less than n.

Definition 3.1 (stream)**.** Given a set A, a **stream** *of values from* A, or an *A-stream*, is a function $\mathbb{N} \to A$. We denote by $\mathcal{S}_A \stackrel{\text{def}}{=} \{s \mid s : \mathbb{N} \to A\}$ the set of all A-streams.

When $s \in \mathcal{S}_A$ and $t \in \mathbb{N}$ we write $s[t]$ for the t-th element of the stream s instead of the usual $s(t)$ to distinguish it from other function applications.

We usually think of the index $t \in \mathbb{N}$ as (discrete) time and of $s[t] \in A$ as the value of the the stream s "at time" t.

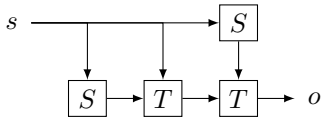
For example, the stream of natural numbers given by $id[t] = t$ is the sequence of values [0 1 2 3 4 $\cdots$].

Definition 3.2 (stream operator)**.** A (typed) **stream operator** with n inputs is a function $T : \mathcal{S}_{A_0} \times \cdots \times \mathcal{S}_{A_{n-1}} \to \mathcal{S}_B$.

In general we will use "operator" for functions on streams, and "function" for computations on "scalar" values.

We are using an extension of the simply-typed lambda calculus to write DBSP programs; we will introduce its elements gradually. However, we find it more readable to also use signal-processing-like circuit diagrams to depict DBSP programs, as in Figure 1.

Stream operator *composition* (function composition) is shown as chained circuits. The composition of a binary operator $T : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_A$ with the unary operator $S : \mathcal{S}_A \to \mathcal{S}_B$ into the computation $\lambda s.T(T(s, S(s)), S(s)) : \mathcal{S}_A \to \mathcal{S}_A$ is given by the following circuit:



Arrows with a single start and multiple ends denote a stream that is reused multiple times, e.g., s in the above diagram is used 3 times. Diagrams, however, do obscure the ordering of the inputs of an operator; in the above examples we have to indicate which ones are the first and respectively second inputs of T if T is not commutative. Most of our binary operators are commutative.

3.1.1 Stream operators by lifting

One way of building stream operators is by (pointwise) **lifting** functions operating on the stream values. For example, given a (scalar) $f : A \to B$ we can define the stream operator $\uparrow f : \mathcal{S}_A \to \mathcal{S}_B$ by $(\uparrow f)(s) = f \circ s$, or, pointwise, $(\uparrow f)(s)[t] \stackrel{\text{def}}{=} f(s[t])$.

This extends straightforwardly to functions of multiple arguments, e.g., given $T : A \times B \to C$, we can define $\uparrow T : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_C$ as $((\uparrow T)(s_0, s_1))[t] \stackrel{\text{def}}{=} T(s_0[t], s_1[t])$.

We call such stream operators **lifted**.

VAL: I hate to be picky about this but we might want to use a different notation for set-theoretical pairing of elements, e.g., in functions of multiple arguments, and category-theoretical pairing of functions. I don't think the latter is captured properly below by $\uparrow\langle ., . \rangle$.

For example, applying the lifted operator $\lambda x.(2x)$ to the stream $id : \mathbb{N} \to \mathbb{N}$ gives as result a stream containing all even values:
$(\uparrow(\lambda x.(2x)))(id) = [\ 0 \quad 2 \quad 4 \quad 6 \quad 8 \quad \cdots\ ]$.

Proposition 3.3 (distributivity)**.** Lifting distributes over function composition: $\uparrow(f \circ g) = (\uparrow f) \circ (\uparrow g)$.

Proof. This is easily proved by using associativity of function composition: $\forall s.(\uparrow(f \circ g))(s) = (f \circ g) \circ s = f \circ (g \circ s) = f \circ (\uparrow g)(s) = (\uparrow f)((\uparrow g)(s)) = (\uparrow f \circ \uparrow g)(s)$. □

We say that two circuits are **equivalent** if they compute the same input-output function on streams. We use the symbol $\cong$ to indicate that two circuits are equivalent. For example, Proposition 3.3 states the following circuit equivalence:

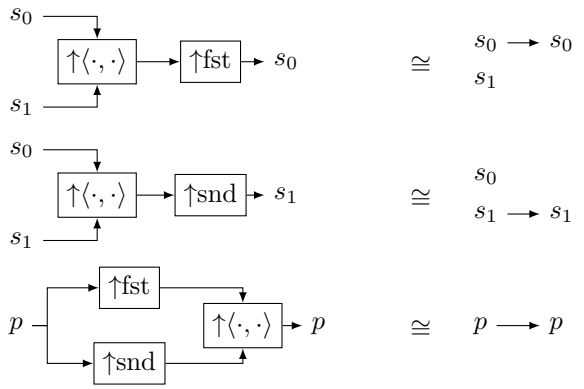
Two (or more) streams can be combined (**paired**) into a single stream of pairs (tuples) by lifting the scalar pairing operator $\langle \cdot, \cdot \rangle : A \to B \to (A \times B)$, obtaining the stream pair operator: $\uparrow\langle \cdot, \cdot \rangle : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_{A \times B}$, defined as mapping $a \in \mathcal{S}_A$ and $b \in \mathcal{S}_B$ to $\langle a, b \rangle \in \mathcal{S}_{A \times B}$ by pairing elements pointwise $\uparrow\langle a, b \rangle[t] = \langle a[t], b[t] \rangle \in A \times B$.

For example, the stream $\langle id, id \rangle$ is the sequence of pairs
$[\ \langle 0, 0 \rangle \quad \langle 1, 1 \rangle \quad \langle 2, 2 \rangle \quad \langle 3, 3 \rangle \quad \langle 4, 4 \rangle \quad \cdots\ ]$.

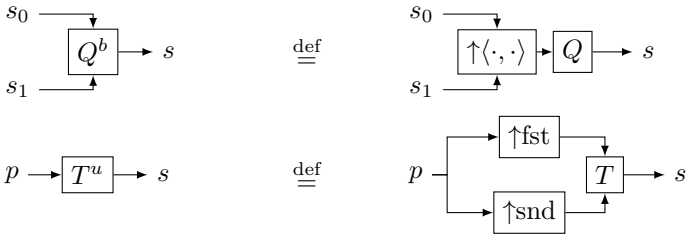
Let us also denote by $\text{fst} : A \times B \to A$ the projection that obtains the first element of a pair $\text{fst}(\langle a, b \rangle) = a$, and by $\text{snd} : A \times B \to B$ the projection that obtains the second element of a pair. We obtain useful stream operators by lifting $\uparrow\text{fst}$ and $\uparrow\text{snd}$.

3.1.2 Basic stream operator equivalences

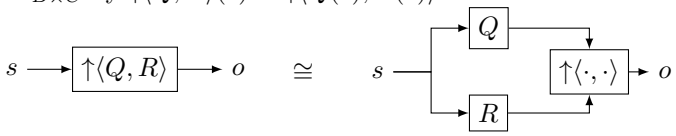
From type theory (or category theory) we recall the standard equalities that pairing and projections satisfy: $\text{fst}(\langle s_0, s_1 \rangle) = s_0$, $\text{snd}(\langle s_0, s_1 \rangle) = s_1$, and $\langle \text{fst}(p), \text{snd}(p) \rangle = p$. By lifting the functions on both left and right we obtain some similar **equivalences of circuits**. In some of the circuits below some inputs are not used.



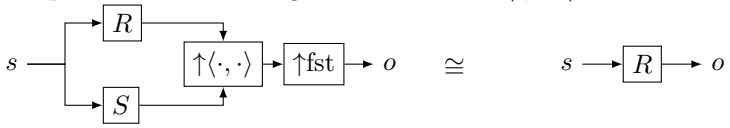
Pairing and projections allow for switching between pairs of streams and streams of pairs, whichever is more convenient. For example, instead of a binary operator $T : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_C$ we can work with a unary operator $T^u : \mathcal{S}_{A\times B} \to \mathcal{S}_C$ where $T^u(p) = T(\uparrow\text{fst}(p), \uparrow\text{snd}(p))$ and instead of a unary operator $Q : \mathcal{S}_{A\times B} \to \mathcal{S}_C$ we can work with a binary operator $Q^b : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_C$ where $Q^b(s_0, s_1) = Q(\uparrow\langle s_0, s_1\rangle)$.



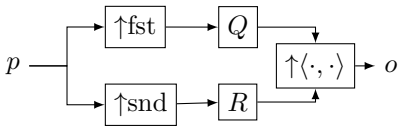
Given two operators $Q : \mathcal{S}_A \to \mathcal{S}_B$ and $R : \mathcal{S}_A \to \mathcal{S}_C$ we define $\uparrow\langle Q, R\rangle : \mathcal{S}_A \to \mathcal{S}_{B\times C}$ by $\uparrow\langle Q, R\rangle(s) = \uparrow\langle Q(s), R(s)\rangle$. In terms of circuit diagrams:



We have standard equalities (from category theory) for this construct such as $\uparrow\text{fst} \circ \uparrow\langle Q, R\rangle = Q$, similarly for snd, and $\uparrow\langle\uparrow\text{fst} \circ W, \uparrow\text{snd} \circ W\rangle = W$. These correspond to equivalences of circuits that follow from the simpler ones above. For example, after substituting the definition of $\langle Q, R\rangle$ we have



Another useful operator expression notation takes $Q : \mathcal{S}_A \to \mathcal{S}_B$ and $R : \mathcal{S}_D \to \mathcal{S}_C$ and combines them into $Q \times R : \mathcal{S}_{A\times D} \to \mathcal{S}_{B\times C}$ where $(Q \times R)(p) = \langle Q(\uparrow\text{fst}(p)), R(\uparrow\text{snd}(p))\rangle$. This corresponds to the following circuit:



We also have the following circuit equivalence:

Lifting functions on values and composing stream operators results in a very simple, yet limited, programming language on streams. We next introduce operators that "shift" streams in time. These will be instrumental for enriching the language.

VAL: Should this one be here? It is unrelated to products. Let em email you a proposal for organizing these equivalences and definitions

3.2 Streams over abelian groups

For the rest of the technical development we will require the set of values $(A, +, 0, -)$ for any stream $\mathcal{S}_A$ to form a commutative group.

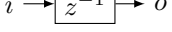
We denote by $0_{\mathcal{S}_A}$ (or simply 0 when the type is clear) the stream that consist of the special value 0 at each time moment: $0_{\mathcal{S}_A} \in \mathcal{S}_A$, $\forall t \in \mathbb{N}. 0_{\mathcal{S}_A}[t] \stackrel{\text{def}}{=} 0_A$.

3.2.1 Delays and time-invariance

Definition 3.4 (Delay)**.** The **delay operator**[2] emits an output stream that is the input stream delayed by one element: $z_A^{-1} : \mathcal{S}_A \to \mathcal{S}_A$ defined by:

$$z_A^{-1}(s)[t] \stackrel{\text{def}}{=} \begin{cases} s[t-1] & \text{when } t \geq 1 \\ 0_A & \text{when } t = 0 \end{cases}$$

We often omit the type parameter A, and write just z^{-1}. We denote by z^{-k} the composition of z^{-1} with itself k times (delay by k time units).



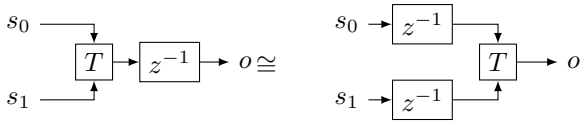
For example, the delay of the id stream is $z^{-1}(id)$, containing the sequence of values [0 0 1 2 3 $\cdots$].

The following definition applies to stream operators of any number of arguments but to keep the notation simpler we formulate it only for binary operators.

Definition 3.5 (Time invariance)**.** A stream operator $T : \mathcal{S}_{A_0} \times \mathcal{S}_{A_1} \to \mathcal{S}_B$ is **time-invariant** if it commutes with the delay operator z^{-1}, that is, $T(z^{-1}(s_0), z^{-1}(s_1)) = z^{-1}(T(s_0, s_1))$ for any $s_0 \in \mathcal{S}_{A_0}, s_1 \in \mathcal{S}_{A_1}$.

[2]The name z^{-1} comes form the DSP literature, and it is related to the z-transform.

In other words, T is time-invariant if and only if the following two circuits are equivalent:



It is straightforward to check that the composition of any number of time-invariant operators of any number of arguments is time invariant. Similarly, the delay operators z^{-k} as well as the pairing and projection operators are time-invariant. In this framework we only deal with time-invariant operators.

Definition 3.6. We say that a function between groups $f : A \to B$ has the **zero-preservation property** iff $f(0_A) = 0_B$. We write $\text{zpp}(f)$. This property generalizes to functions with multiple inputs: e.g., $g : A \times B \to C$ where A, B, C are groups. $\text{zpp}(g)$ iff $g(0_A, 0_B) = 0_C$.

Proposition 3.7. A lifted operator $\uparrow f$ is time-invariant iff $\text{zpp}(f)$.

Notice that it is easy to construct operators that are not time-invariant. Consider the "constant 1" function: $c_1 : \mathbb{N} \to \mathbb{N}$, defined by $c_1(x) = 1, \forall x \in \mathbb{N}$. The stream operator defined by $\uparrow c_1 : \mathcal{S}_N \to \mathcal{S}_N$ produces a stream containing only the constant value 1: $c_1(s)[t] = 1, \forall s \in \mathcal{S}_N, t \in \mathbb{N}$. The stream operator $\uparrow c_1$ is *not* time-invariant, since it does not have the zero preservation property.

3.3 Causal and strict operators

For notation simplicity we again give the next definition only for unary operators; it extends naturally to binary operators through the use of pairing as shown above.

Definition 3.8 (Causality)**.** A stream operator, $S : \mathcal{S}_A \to \mathcal{S}_B$, is **causal** when for any $s, s' \in \mathcal{S}_A$, and all times t we have

$$(\forall i \leq t \; s[i] = s'[i]) \;\; \text{implies} \;\; S(s)[t] = S(s')[t]$$

Note that all operators produced by lifting scalar functions are causal. z^{-1} is causal. All DBSP operators are causal.

Definition 3.9 (Cutting)**. Cutting** the stream $s \in \mathcal{S}_A$ at time $t \in \mathbb{N}$ produces a stream

$$(s|_{\leq t})[i] \stackrel{\text{def}}{=} \begin{cases} s[i] & \text{if } i \leq t \\ 0_A & \text{if } i > t \end{cases}$$

For example, cutting the stream id at time 2 gives the stream $id|_{\leq 2}$ composed of the sequence [0 1 2 0 0 $\cdots$].

Note that $s|_{\leq t_1}|_{\leq t_2} = s|_{\leq \min(t_1, t_2)}$. It follows that $s|_{\leq t_1}|_{\leq t_2} = s|_{\leq t_2}|_{\leq t_1}$ (cutting is commutative) and $s|_{\leq t}|_{\leq t} = s|_{\leq t}$, (cutting at time t is idempotent).

Cutting, however, is **not** time-invariant. Cutting is not used as a DBSP operator, it is just a mathematical tool that we will use to reason about the behavior of the circuits we build.

Lemma 3.10. The following are equivalent for a binary stream operator T

(i) T is causal

(ii) $\forall s_1, s_2$ and t we have $T(s_1, s_2)|_{\leq t} = T(s_1|_{\leq t}, s_2|_{\leq t})|_{\leq t}$.

Using part (ii) it follows immediately that the composition of any number of causal operators of any number of arguments is causal. Moreover, using also the commutativity and idempotence of cutting, it follows that for any t the operator $\lambda s.s|_{\leq t}$ is itself causal.

Definition 3.11 (zero almost everywhere)**.** We say that a stream s is **zero almost-everywhere** if there exists a time $t_0 \in \mathbb{N}$ s.t. $s|_{\leq t_0} = s$.

We denote the set of streams over A that are zero almost everywhere by $\overline{\mathcal{S}_A}$.

Definition 3.12 (Strictness)**.** A stream operator, $F : \mathcal{S}_A \to \mathcal{S}_B$ is **strictly causal** (abbreviated **strict**) if for any $s, s' \in \mathcal{S}_A$ and all times t we have

$$(\forall i < t. s[i] = s'[i]) \text{ implies } F(s)[t] = F(s')[t]$$

In particular, $F(s)[0] = 0_B$ is the same for all inputs $s \in \mathcal{S}_A$. Strict operators are of course causal. Note that lifted stream operators, while causal, in general are *not* strict.

It can be immediately checked that the operator z^{-1} (in fact, z^{-k} for any positive integer k) is strict. In this text z^{-1} is the only primitive strict operator used.

Definition 3.13 (Strict cutting)**. Strictly cutting** the stream $s \in \mathcal{S}_A$ at time $t \in \mathbb{N}$ produces the stream

$$(s|_{<t})[i] \stackrel{\text{def}}{=} \begin{cases} s[i] & \text{if } i < t \\ 0_A & \text{if } i \geq t \end{cases}$$

$s|_{<0}$ is the stream $0_{\mathcal{S}_A}$ that is 0_A at all times. Note also that $s|_{<t+1} = s|_{\leq t}$.

Analogously to Lemma 3.10 an operator $F : \mathcal{S}_A \to \mathcal{S}_B$ is strict iff for any s and t we have

$$F(s)|_{\leq t} = F(s|_{<t})|_{\leq t}$$

In particular, $F(s)[0] = F(0_{\mathcal{S}_A})[0]$ and $F(s)[t+1] = F(s|_{\leq t})[t+1]$. Note the different zeros in $F(0_{\mathcal{S}_A})[0]$: it features both the stream $0_{\mathcal{S}_A} \in \mathcal{S}_A$, consisting of the group element 0_A at each time moment, and the time moment 0.

The next proposition shows the importance of strict operators.

Proposition 3.14. For any strict operator $F : \mathcal{S}_A \to \mathcal{S}_A$ the equation $\alpha = F(\alpha)$ has a unique solution $\alpha \in \mathcal{S}_A$. In other words, every strict operator has a unique fixed point, which we denote by $\operatorname{fix} \alpha.F(\alpha)$.

Proof. Define the solution (the fixed point) by recurrence:

$$\begin{aligned} \alpha[0] &= F(\alpha)[0] = F(0_{\mathcal{S}_A})[0] \\ \alpha[t+1] &= F(\alpha)[t+1] = F(\alpha|_{\leq t})[t+1] \end{aligned}$$

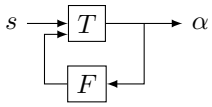
The second equality defines $\alpha[t+1]$ in terms of $\alpha[0], \ldots, \alpha[t]$. Uniqueness follows by strong induction.

□

We will apply the previous proposition to operators obtained by composing strict and causal ones.

Lemma 3.15. Let $k \geq 2$. If F is strict and the k-ary T operator is causal, then for any fixed $s_0, \ldots, s_{k-2}$ the operator $\lambda\alpha.T(s_0, \ldots, s_{k-2}, F(\alpha))$ is strict.

Proof. We show the case $k = 2$. This operator is described by the following diagram with a "feedback loop":



$$\begin{aligned} T(s, F(\alpha))|_{\leq t} &= T(s|_{\leq t}, F(\alpha)|_{\leq t})|_{\leq t} \\ &= T(s|_{\leq t}, F(\alpha|_{<t})|_{\leq t})|_{\leq t} \\ &= T(s, F(\alpha|_{<t}))|_{\leq t} \end{aligned}$$

□

Corollary 3.16. For strict F we have

$$(\operatorname{fix}\alpha.F(\alpha))|_{\leq t} = \operatorname{fix}\alpha.(F(\alpha)|_{\leq t})$$

Proof. Since cutting itself is a causal operator, it follows from Lemma 3.15 that $\lambda\alpha.(F(\alpha)|_{\leq t})$ is strict so Proposition 3.14 applies and $\operatorname{fix}\alpha.(F(\alpha)|_{\leq t})$ is well-defined.

If we let α be the solution of $\alpha = F(\alpha)$ then, by uniqueness, it suffices to show that $\beta = \alpha|_{\leq t}$ satisfies the equation $\beta = F(\beta)|_{\leq t}$. Indeed, since F is in particular causal

$$\alpha|_{\leq t} = F(\alpha)|_{\leq t} = F(\alpha|_{\leq t})|_{\leq t}$$

□

Corollary 3.17. Let $k \geq 1$. If $F : \mathcal{S}_A \to \mathcal{S}_A$ is strict and $(k+1)$-ary T is causal then the k-ary operator $Q(s_0, \ldots, s_{k-1}) = \operatorname{fix}\alpha.T(s_0, \ldots, s_{k-1}, F(\alpha))$ is well-defined and causal. If, moreover, F and T are time-invariant then so is Q.

Proof. We show the case $k = 1$. The well-definedness of Q follows by applying, for each s, Proposition 3.14 to the operator $\lambda\alpha.T(s, F(\alpha))$ which is strict by Lemma 3.15. For future reference it might be useful to state the defining recurrence for a stream α produced by this operator, that is, $\alpha = Q(s)$:

$$\begin{aligned} \alpha[0] &= T(s, F(0_{\mathcal{S}_A}))[0] \\ \alpha[t+1] &= T(s, F(\alpha|_{\leq t}))[t+1] \end{aligned}$$

To prove that Q is causal we could use this recurrence and induction. Instead we use the causality of T and the idempotence of cutting in conjunction with Corollary 3.16 as follows:

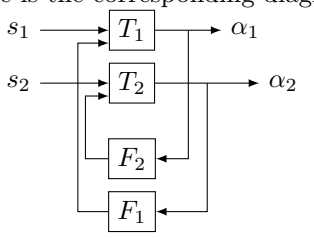
$$\begin{aligned} Q(s)|_{\leq t} &= (\mathrm{fix}\,\alpha.T(s, F(\alpha)))|_{\leq t} \\ &= \mathrm{fix}\,\alpha.(T(s, F(\alpha))|_{\leq t}) && \text{(Corollary 3.16)} \\ &= \mathrm{fix}\,\alpha.(T(s|_{\leq t}, F(\alpha)|_{\leq t})|_{\leq t}) && \text{(Causality of } T\text{)} \\ &= \mathrm{fix}\,\alpha.(T(s|_{\leq t}|_{\leq t}, F(\alpha)|_{\leq t})|_{\leq t}) && \text{(Idempotence of cutting)} \\ &= \mathrm{fix}\,\alpha.(T(s|_{\leq t}, F(\alpha))|_{\leq t}) && \text{(Causality of } T\text{)} \\ &= (\mathrm{fix}\,\alpha.T(s|_{\leq t}, F(\alpha)))|_{\leq t} && \text{(Corollary 3.16)} \\ &= Q(s|_{\leq t})|_{\leq t} \end{aligned}$$

For time-invariance we observe that if $\alpha = T(s, F(\alpha))$ then $z^{-1}(\alpha) = z^{-1}(T(s, F(\alpha))) = T(z^{-1}(s), z^{-1}(F(\alpha)) = T(z^{-1}(s), F(z^{-1}(\alpha)))$. It follows that $\beta = z^{-1}(\alpha)$ satisfies the equation $\beta = T(z^{-1}(s), F(\beta))$ so, by uniqueness of the fixed point $Q(z^{-1}(s)) = z^{-1}(Q(s))$. □

Ostensibly this covers a form of straightforward recursion but how about mutual recursion? For instance, assuming T_1, T_2 are causal and F_1, F_2 are strict we wish to claim that the following is well defined: $Q_1(s_1, s_2) = \alpha_1$ and $Q_2(s_1, s_2) = \alpha_2$ where:

$$\begin{aligned} \alpha_1 &= T_1(s_1, F_1(\alpha_2)) \quad \text{and} \\ \alpha_2 &= T_2(s_2, F_2(\alpha_1)) \end{aligned}$$

Here is the corresponding diagram:



In section 3.1.2 we defined constructions on pairs of streams/streams of pairs. In particular, for binary $T_1 : \mathcal{S}_{A_1} \times \mathcal{S}_{B_1} \to \mathcal{S}_{C_1}$ and binary $T_2 : \mathcal{S}_{A_2} \times \mathcal{S}_{B_2} \to \mathcal{S}_{C_2}$ let binary $T_1 \times T_2 : \mathcal{S}_{A_1 \times A_2} \times \mathcal{S}_{B_1 \times B_2} \to \mathcal{S}_{C_1 \times C_2}$

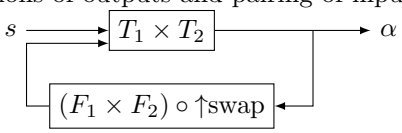
$$[T_1 \times T_2](p, q) = \langle T_1(\uparrow\mathrm{fst} \circ p, \uparrow\mathrm{fst} \circ q), T_2(\uparrow\mathrm{snd} \circ p, \uparrow\mathrm{snd} \circ q)\rangle$$

VAL: For binary operators we should have defined × this way in section 3.1.2 ... Also, in the def of $T_1 \times T_2$ note that I did not put a lift in front of the stream pairing operator on the RHS.

In addition, we use $F_1 \times F_2 : \mathcal{S}_{C_1 \times C_2} \to \mathcal{S}_{B_1 \times B_2}$ as defined in section 3.1.2. Let also swap : $C_1 \times C_2 \to C_2 \times C_1$ be the operator that swaps the components of a pairs (obtained by pairing the second projection with the third.

Proposition 3.18. If T_1 and T_2 are causal then $T_1 \circ T_2$ is causal. If F_1 and F_2 are strict then $(F_1 \times F_2) \circ \uparrow\text{swap}$ is strict.

The circuit above is equivalent to the following (when composed with projections of outputs and pairing of inputs:



In other words, we can apply Corollary 3.17 to the causal operator $T_1 \times T_2$ and the strict operator $F_1 \times F_2$ and obtain Q_1 and Q_2 from

$$\text{fix}\,\alpha.[T_1 \times T_2](\langle s_1, s_2\rangle, [F_1 \times F_2](\text{swap}(\alpha)))$$

by further projecting, where α is a variable of type pair of streams and $\overline{\alpha}$ swaps the two components.

VAL: We should state this a some kind of corollary to the corollary.

MIHAI: This becomes complicated for n-way mutual recursion, you have a quadratic number of edges. Maybe it's simpler to assume that all of them use all of the alphas, and the projection to a subset is part of T if needed.

3.4 Streams as an abelian group

Remember that we require the elements of a stream to come from an Abelian group: $(A, +, 0, -)$. This structure also lifts to streams:

Proposition 3.19. $(\mathcal{S}_A, +, 0_{\mathcal{S}_A}, -)$ (with the operations lifted pointwise in time) is also an abelian group. Moreover, lifting a group homomorphism produces a stream operator that is itself a group homomorphism. In addition, when A and B are abelian groups there is a standard abelian group structure on $A \times B$, with the zero $0_{A\times B}$ the pair $\langle 0_A, 0_B\rangle$.

Definition 3.20 (linear)**.** If A and B are abelian groups, we call a function $f : A \to B$ **linear** if it is a group homomorphism, that is, $f(a+b) = f(a)+f(b)$ (and therefore $f(0_A) = 0_B$ and $f(-a) = -f(a)$); thus zpp(f).

We use the abbreviation LTI for a stream operator that is linear and time-invariant.

Lifting a linear function $f : A \to B$ produces a stream operator $\uparrow f$ that is causal and LTI. It follows that stream addition and negation are causal and LTI. z^{-1} is LTI, (and so is z^{-k} for all k).

Definition 3.21 (multilinear, bilinear)**.** We define **multilinear** (in particular, **bilinear**) functions as functions (between groups) of multiple arguments that are linear separately in each argument (that is, if we fix all but one argument, the resulting function is linear in that argument. In other words, the function distributes over addition): e.g., for $g : A_0 \times A_1 \to B, \forall a, b \in A_0, c, d \in A_1.g(a + b, c) = g(a, c) + g(b, c)$, and $g(a, c + d) = g(a, c) + g(c, d)$.

Multiplication over $\mathbb{Z}$ is a bilinear function. For a bilinear function g we have $\text{zpp}(g)$.

This definition extends to stream operators. Lifting any bilinear function $g : A \times B \to C$ produces a bilinear stream operator $\uparrow g$. An example bi-linear operator over $\mathcal{S}_\mathbb{Z}$ is the lifted integer multiplication: $T : \mathcal{S}_\mathbb{Z} \times \mathcal{S}_\mathbb{Z} \to \mathcal{S}_\mathbb{Z}, T(a, b)[t] = a[t] \cdot b[t]$.

The composition of multilinear operators with linear operators is multilinear (since homomorphisms compose). Since linear and bilinear functions have the zero-preservation property, lifted linear and bilinear functions operators are all time-invariant.

Proposition 3.22. The composition of a bilinear operator followed by a linear operator is a bilinear operator.

Proof. Consider $T : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_C$ a bilinear operator, and $S : \mathcal{S}_C \to \mathcal{S}_C$, a linear operator. Let us compute $S(T(a + b, c)) = S(T(a, c) + T(b, c)) = S(T(a, c)) + S(T(b, c))$. Thus $S \circ T$ is bilinear. □

Lifting a multilinear operator $A_1 \times \cdots \times A_n \to B$ produces a multilinear, time-invariant stream operator. Although combining pairs (tuples) of streams into stream of pairs (tuples) can be useful we must note a distinction (well understood in algebra): we have seen that we can use instead of a binary stream operator $T : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_C$ a unary version that acts on streams of pairs $T^u : \mathcal{S}_{A\times B} \to \mathcal{S}_C$ where $T^u\langle a, b\rangle = T(a, b)$. However, in contrast to causality, there is, in general, no relation between the linearity.

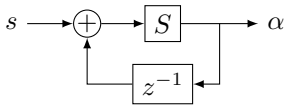
In traditional signal processing most operators are LTI but in our development we will use some important non-linear ones.

The "feedback-loop" operators defined by recurrence in Corollary 3.17, e.g., $\lambda s.\text{fix}\,\alpha.T(s, F(\alpha))$ are, in general, not (multi)linear. However, multilinearity holds in important particular cases, as shown in the following proposition:

VAL: Counterexample even for bilinear S and $\lambda s_1.\lambda s_2.\text{fix}\,\alpha.S(s_1, s_2 + z^{-1}(\alpha))$? Possibly S is join NOT followed by distinct?

MIHAI: Notice that this is a very nice kind of recursion, a tail-recursion.

Proposition 3.23. Let S be a unary causal, LTI operator. Then, the operator $Q(s) = \text{fix}\,\alpha.S(s + z^{-1}(\alpha))$ is well-defined and LTI.



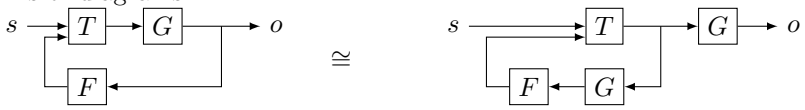
Proof. Since S and the addition operator are causal and z^{-1} is strict, Proposition 3.14 applies, for each s, to the operator $\lambda\alpha.S(s + z^{-1}(\alpha))$ which is strict by Lemma 3.15. Thus Q is well-defined.

Fix streams s_0 and s_1. Let α_0 be the unique solution of $\alpha_0 = S(s_0+z^{-1}(\alpha_0))$ and α_1 be the unique solution of $\alpha_1 = S(s_1 + z^{-1}(\alpha_1))$. Then $\alpha = \alpha_0 + \alpha_1$ is the unique solution of $\alpha = Q(s_0 + s_1 + z^{-1}(\alpha))$. This is justified by adding the equations and using the linearity of S and the linearity of z^{-1}. □

Proposition 3.24. If T is binary causal, G is unary causal, and F is unary strict then:

$$\operatorname{fix}\alpha.G(T(s,F(\alpha))) = G(\operatorname{fix}\beta.T(s,F(G(\beta))))$$

In terms of diagrams:



Proof. For the second fixpoint to exist we need to show that $F \circ G$ is strict. Once we do that, by uniqueness of solutions, it remains to show that if β is a solution to $\beta = T(s, F(G(\beta))$ then $\alpha = G(\beta)$ is a solution to $\alpha = G(T(s, F(\alpha))$ which is immediate by applying G to the first equation. So let's prove that $F \circ G$ is strict. For $t > 0$ we have

$$\begin{aligned} F(G(s))|_{\leq t} &= F(G(s)|_{<t})|_{\leq t} && (F \text{ strict}) \\ &= F(G(s)|_{\leq t-1})|_{\leq t} && (t \geq 1) \\ &= F(G(s|_{\leq t-1}))|_{\leq t} && (G \text{ causal}) \\ &= F(G(s|_{<t}))|_{\leq t} \end{aligned}$$

For $t = 0$ just observe that $F(G(s))[0]$ is the same for any $G(s)$ therefore for any s. Note that if G is not causal then $F \circ G$ is not strict and the fixed point may not exist. □

3.5 Differentiation and Integration

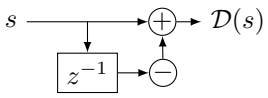
Definition 3.25 (Differentiation)**.** The **differentiation operator** $\mathcal{D}_{\mathcal{S}_A} : \mathcal{S}_A \to \mathcal{S}_A$ is defined by:

$$\mathcal{D}_{\mathcal{S}_A}(s) \stackrel{\text{def}}{=} s - z^{-1}(s)$$

We generally omit the type, and write just $\mathcal{D}$ when the type can be inferred from the context.

The value of $\mathcal{D}(s)$ (at time t) is the difference between the current (time t) value of s and the previous (time $t-1$) value of s.

As an example, applying $\mathcal{D}$ to the stream id gives a result a stream $\mathcal{D}(id)$ containing the values [0 1 1 1 1 ⋯].



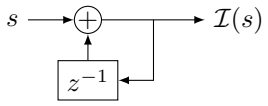
Proposition 3.26. The differentiation operator $\mathcal{D}$ is causal and LTI.

Proof. Follow from definition using the properties of subtraction and delay. □

The integration operator "reconstitutes" a stream from its changes:

Definition 3.27 (Integration)**.** The **integration operator** $\mathcal{I}_{\mathcal{S}_A} : \mathcal{S}_A \to \mathcal{S}_A$ is defined by $\mathcal{I}_{\mathcal{S}_A}(s) \stackrel{\text{def}}{=} \lambda s.\mathrm{fix}\,\alpha.(s + z^{-1}(\alpha))$.

We also generally omit the type, and write just $\mathcal{I}$. This is the construction from Proposition 3.23 using the identity function for S.



As an example, applying $\mathcal{I}$ to the *id* stream gives as result a stream $\mathcal{I}(id)$ composed of the values [0 1 3 6 10 $\cdots$].

Proposition 3.28. $\mathcal{I}(s)$ is the discrete (indefinite) integral applied to the stream s: $\mathcal{I}(s)[t] = \sum_{i \le t} s[i]$.

Proof. The recurrence from Corollary 3.17 specializes to

$$\begin{aligned} \alpha[0] &= s[0] \\ \alpha[t+1] &= \alpha[t] + s[t+1] \end{aligned}$$

and it's straightforward to check that $\alpha[t] = \sum_{i \le t} s[i]$ satisfies it. □

Proposition 3.29 (Properties of $\mathcal{I}$)**.** The integration operator $\mathcal{I}$ is causal and LTI.

Proof. By Proposition 3.28 these properties follow from Corollary 3.17 and Proposition 3.23. They also be checked directly using the definition by summation. □

Theorem 3.30 (Inversion)**.** The integration and differentiation operators are inverse to each other. Equivalently, for any streams α and s we have $\alpha = \mathcal{I}(s)$ iff $\mathcal{D}(\alpha) = s$.

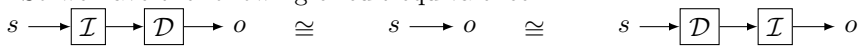
Proof. This can be shown directly from the definitions, for example

$$\begin{aligned} \mathcal{D}(\mathcal{I}(s))[t] &= (\mathcal{I}(s) - z^{-1}(\mathcal{I}(s)))[t] && \text{definition of } \mathcal{D} \\ &= \sum_{k \le t} s[k] - z^{-1}(\sum_{k \le t} s[k])[t] && \text{Property 3.29} \\ &= \sum_{k \le t} s[k] - \sum_{k \le t-1} s[k] && \text{definition of } z^{-1} \\ &= s[t] \end{aligned}$$

(and similarly we can show that $\mathcal{I}(\mathcal{D}(s'))[t] = s'[t]$).

Alternatively, the equivalent form of the theorem follows from Proposition 3.28 by observing that $\mathcal{D}(\alpha) = s$ iff $\alpha = s + z^{-1}(\alpha)$ which is the equation on streams that defines $\alpha = \mathcal{I}(s)$. □

So we have the following circuit equivalence:



Since $\mathcal{I}$ and $\mathcal{D}$ are inverse to each other they are both bijections on streams.

If we define addition of pairs as adding the pair elements pointwise, we also have the following identities: $\mathcal{I}(\langle s,t\rangle) = \langle \mathcal{I}(s), \mathcal{I}(t)\rangle$ and $\mathcal{D}(\langle s,t\rangle) = \langle \mathcal{D}(s), \mathcal{D}(t)\rangle$.

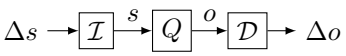
Observation It is a standard algebraic fact that the inverse of a homomorphism is also a homomorphism. Thus, $\mathcal{I}$ is linear iff $\mathcal{D}$ is linear. Another immediate consequence of this theorem is that $\mathcal{I}$ is time-invariant iff $\mathcal{D}$ is time-invariant. Moreover $\mathcal{I}$ is causal iff $\mathcal{D}$ is causal. Therefore by the Inversion Theorem we could have stated and proved only one of Proposition 3.29 or Proposition 3.26.

Observation In digital signal processing, $\mathcal{I}$ is a IIR, an infinite-impulse response filter: given a cut stream it can produce an unbounded stream. $\mathcal{D}$ is a FIR, a finite-impulse response filter: from a cut stream it always produces a cut stream.

4 Incremental computation

Definition 4.1. Given a unary stream operator $Q : \mathcal{S}_A \to \mathcal{S}_B$ we define the **incremental version** of Q as $Q^\Delta \stackrel{\text{def}}{=} \mathcal{D} \circ Q \circ \mathcal{I}$. Q^Δ has the same "type" as Q: $Q^\Delta : \mathcal{S}_A \to \mathcal{S}_B$. For an operator with multiple inputs we define the incremental version by applying $\mathcal{I}$ to each input independently: e.g., if $T : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_C$ then $T^\Delta : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_C$ and $T^\Delta(a,b) \stackrel{\text{def}}{=} \mathcal{D}(T(\mathcal{I}(a), \mathcal{I}(b)))$.

The following diagram illustrates the intuition behind this definition:



If $Q(s) = o$ is a computation, then Q^Δ performs the "same" computation as Q, but between streams of changes Δs and Δo. This is the diagram from the introduction, substituting Δs for the transaction stream T, and o for the stream of view versions V.

Notice that our definition of incremental computation is meaningful only for *streaming* computations; this is in contrast to classic definitions, e.g. [16] which consider only one change. Generalizing the definition to operate on streams gives us additional power, especially when operating with recursive queries.

The following proposition is one of our central results.

Proposition 4.2. (Properties of the incremental version):
For computations of appropriate types, the following hold:

inversion: $Q \mapsto Q^\Delta$ is bijective; its inverse is $Q \mapsto \mathcal{I} \circ Q \circ \mathcal{D}$.

invariance: $+^\Delta = +, (z^{-1})^\Delta = z^{-1}, -^\Delta = -, \mathcal{I}^\Delta = \mathcal{I}, \mathcal{D}^\Delta = \mathcal{D}$

push/pull: $Q \circ \mathcal{I} = \mathcal{I} \circ Q^{\Delta}$; $\mathcal{D} \circ Q = Q^{\Delta} \circ \mathcal{D}$

chain: $(Q_1 \circ Q_2)^{\Delta} = {Q_1}^{\Delta} \circ {Q_2}^{\Delta}$ (This generalizes to operators with multiple inputs.)

add: $(Q_1 + Q_2)^{\Delta} = {Q_1}^{\Delta} + {Q_2}^{\Delta}$

cycle: $(\lambda s.\mathrm{fix}\,\alpha.T(s, z^{-1}(\alpha)))^{\Delta} = \lambda s.\mathrm{fix}\,\alpha.T^{\Delta}(s, z^{-1}(\alpha))$

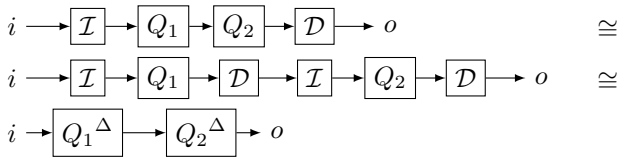
Proof. The inversion and push-pull properties follow straightforwardly from the fact that $\mathcal{I}$ and $\mathcal{D}$ are inverses of each other.

For proving invariance we have $+^{\Delta}(a, b) \stackrel{\text{def}}{=} \mathcal{D}(\mathcal{I}(a) + \mathcal{I}(b)) = a + b$, due to linearity of $\mathcal{I}$. $-^{\Delta}(a) = \mathcal{D}(-\mathcal{I}(a)) = \mathcal{D}(0 - \mathcal{I}(a)) = \mathcal{D}(\mathcal{I}(0) - \mathcal{I}(a)) = \mathcal{D}(\mathcal{I}(0 - a)) = -a$, also due to linearity of $\mathcal{I}$.

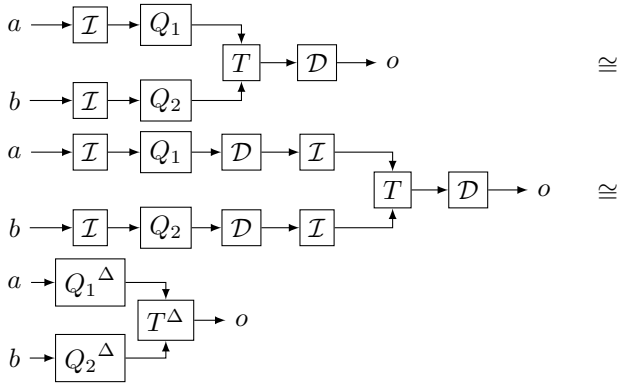
The chain rule follows from push-pull. Indeed,

$$\mathcal{I} \circ {Q_1}^{\Delta} \circ {Q_2}^{\Delta} = Q_1 \circ \mathcal{I} \circ {Q_2}^{\Delta} = Q_1 \circ Q_2 \circ \mathcal{I}$$

I.e., we have the following sequence of equivalent circuits:



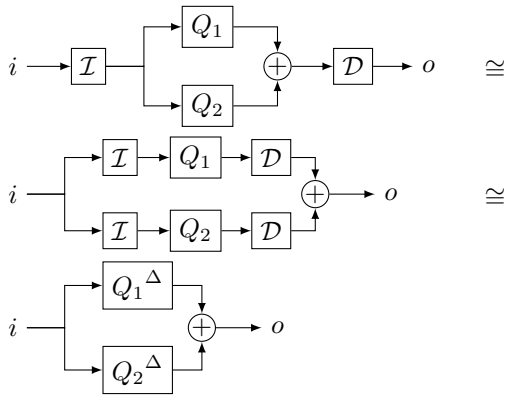
Here is a version of the chain rule with a binary operator:



The add rule follows from push/pull and the linearity of $\mathcal{I}$ (or $\mathcal{D}$). Indeed,

$$\mathcal{I} \circ ({Q_1}^{\Delta} + {Q_2}^{\Delta}) = \mathcal{I} \circ {Q_1}^{\Delta} + \mathcal{I} \circ {Q_2}^{\Delta} = Q_1 \circ \mathcal{I} + Q_2 \circ \mathcal{I} = (Q_1 + Q_2) \circ \mathcal{I}$$

I.e., the following diagrams are equivalent:



The cycle rule is most interesting. First, observe that if T is causal then so is T^{Δ} thus both sides of the equality are well-defined. Next, we can use again push/pull to show the equality if we can check that

$$\mathcal{I} \circ (\lambda s.\mathrm{fix}\,\alpha.T^{\Delta}(s, z^{-1}(\alpha)) = (\lambda s.\mathrm{fix}\,\alpha.T(s, z^{-1}(\alpha)) \circ \mathcal{I}$$

that is, for any s,

$$\mathcal{I}(\mathrm{fix}\,\alpha.T^{\Delta}(s, z^{-1}(\alpha)) = \mathrm{fix}\,\alpha.T(\mathcal{I}(s), z^{-1}(\alpha))$$

This follows from the following lemma. □

Lemma 4.3. If the parameters a and b are related by $b = \mathcal{D}(a)$ (equivalently $a = \mathcal{I}(b)$) then the unique solutions of the fixed point equations

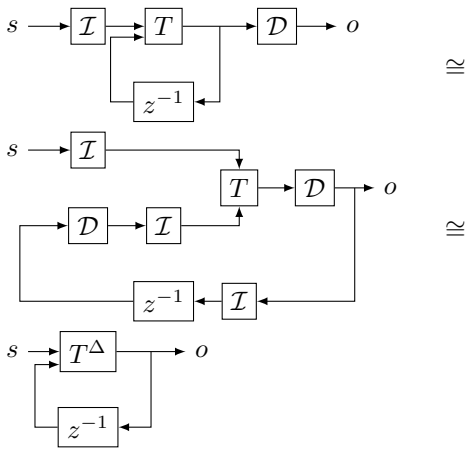
$$\alpha \;=\; T(a, z^{-1}(\alpha)) \quad \text{and} \quad \beta \;=\; T^{\Delta}(b, z^{-1}(\beta))$$

are related by $\alpha = \mathcal{I}(\beta)$ (equivalently $\beta = \mathcal{D}(\alpha)$).

Proof. (Of Lemma 4.3) Let β be the unique solution of $\beta \;=\; T^{\Delta}(\mathcal{D}(a), z^{-1}(\beta))$. We verify that $\alpha = \mathcal{I}(\beta)$ satisfies the equation $\alpha \;=\; T(a, z^{-1}(\alpha))$. Indeed, using the fact that $\mathcal{I}$ and $\mathcal{D}$ are inverses as well as the time-invariance of $\mathcal{I}$ we have

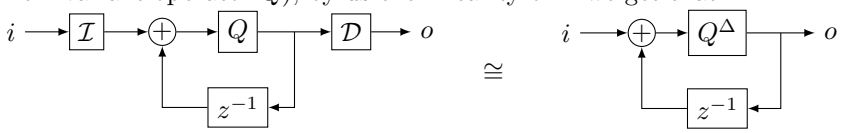
$$\begin{aligned}
\mathcal{I}(\beta) &= \mathcal{I}(T^{\Delta}(\mathcal{D}(a), z^{-1}(\beta))) \\
&= \mathcal{I}(\mathcal{D}(T(\mathcal{I}(\mathcal{D}(a)), \mathcal{I}(z^{-1}(\beta))))) \\
&= T(a, \mathcal{I}(z^{-1}(\beta)) \\
&= T(a, z^{-1}(\mathcal{I}(\beta))
\end{aligned}$$

I.e., starting from this diagram we apply a sequence of term-rewriting semantics-preserving transformations:



□

If we specialize the above formula for the case $T(a, b) = Q(a + b)$ (for some time-invariant operator Q), by us the linearity of $\mathcal{I}$ we get that:



Theorem 4.4 (Linear)**.** For any LTI operator Q we have $Q^{\Delta} = Q$.

Proof. By the push/pull rule from Proposition 4.2 it suffices to show that Q commutes with differentiation:

$$\begin{aligned}
\mathcal{D}(Q(s)) &= Q(s) - z^{-1}(Q(s)) && \text{by definition of } \mathcal{D} \\
&= Q(s) - Q(z^{-1}(s)) && \text{by time-invariance of } Q \\
&= Q(s - z^{-1}(s)) && \text{by linearity of } Q \\
&= Q(\mathcal{D}(s)) && \text{by definition of } \mathcal{D}.
\end{aligned}$$

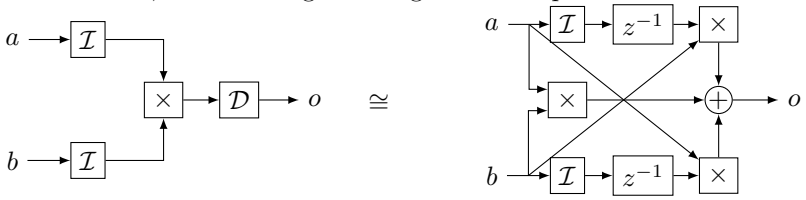
□

As we have shown, the incremental version of a linear unary operator equals the operator itself. However, this is not true, in general, for multilinear operators. Nonetheless, there is a useful relationship between the incremental version of a multilinear operator and the operator itself. We illustrate with bilinear operators.

Theorem 4.5 (Bilinear)**.** For any bilinear time-invariant operator $\times$ we have $(a \times b)^{\Delta} = a \times b + \mathcal{I}(z^{-1}(a)) \times b + a \times \mathcal{I}(z^{-1}(b))$.

By rewriting this statement using Δa for the stream of changes to a we get the familiar formula for incremental joins: $\Delta(a \times b) = \Delta a \times \Delta b + a \times (\Delta b) + (\Delta a) \times b$.

In other words, the following two diagrams are equivalent:



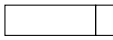
Proof.

$$
\begin{aligned}
(a \times b)^{\Delta} &= \mathcal{D}(\mathcal{I}(a) \times \mathcal{I}(b)) && \text{def of } \cdot^{\Delta} \\
&= (\mathcal{I}(a) \times \mathcal{I}(b)) \; - \; z^{-1}(\mathcal{I}(a) \times \mathcal{I}(b)) && \text{def of } \mathcal{D} \\
&= \mathcal{I}(a) \times \mathcal{I}(b) \; - \; z^{-1}(\mathcal{I}(a)) \times z^{-1}(\mathcal{I}(b)) && \times \text{ time inv.} \\
&= \mathcal{I}(a) \times \mathcal{I}(b) \; - \; z^{-1}(\mathcal{I}(a)) \times \mathcal{I}(b) && \text{add and sub a term} \\
&\qquad + \; z^{-1}(\mathcal{I}(a)) \times \mathcal{I}(b) \; - \; z^{-1}(\mathcal{I}(a)) \times z^{-1}(\mathcal{I}(b))) \\
&= \mathcal{D}(\mathcal{I}(a)) \times \mathcal{I}(b) \; + \; z^{-1}(\mathcal{I}(a)) \times \mathcal{D}(\mathcal{I}(b))) && \text{bilinearity, def of } \mathcal{D} \\
&= a \times \mathcal{I}(b) \; + \; z^{-1}(\mathcal{I}(a)) \times b && \mathcal{D} \circ \mathcal{I} = id \\
&= a \times \mathcal{I}(b) \; - \; a \times z^{-1}(\mathcal{I}(b)) && \text{add and sub a term} \\
&\qquad + \; a \times z^{-1}(\mathcal{I}(b)) \; + \; z^{-1}(\mathcal{I}(a)) \times b \\
&= a \times \mathcal{D}(\mathcal{I}(b)) \; + \; a \times z^{-1}(\mathcal{I}(b)) \; + \; z^{-1}(\mathcal{I}(a)) \times b && \text{bilinearity, def of } \mathcal{D} \\
&= a \times b \; + \; z^{-1}(\mathcal{I}(a)) \times b \; + \; a \times z^{-1}(\mathcal{I}(b))
\end{aligned}
$$

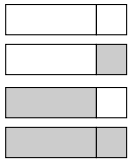
□

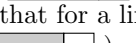
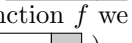
4.1 Vector representations

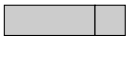
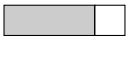
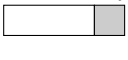
Some formulas are easier to read as mathematical expressions over "vectors". We will use the following representation to depict a fragment of a stream s:

 0

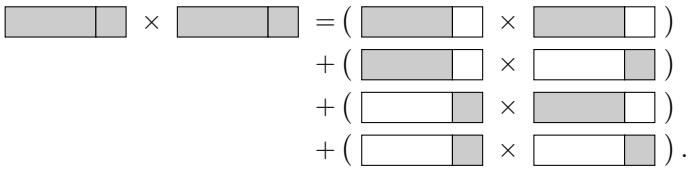
This "vector" representation fixes a time t and shows the integral of the stream prefix up to $\mathcal{I}(s)[t-1]$ as a rectangle, and $s[t]$ as a square. We will use a grey rectangle to show which part of a stream participates in a computation. For example, we have the following notations:



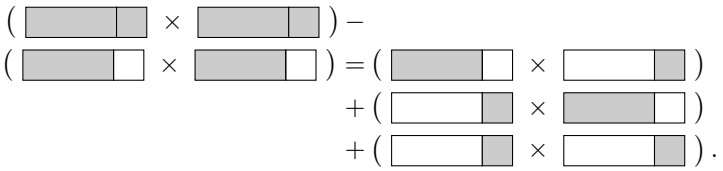
Using this notation, the incremental version of a function f is defined as $f^{\Delta}($  $1) = f($  $1) - f($  $0)$.

Theorem 4.4 states that for a linear function f we have:
$f($  $) - f($  $) = f($  $)$.

The statement of Theorem 4.5, for a bilinear stream operation $\times$ can be written as:



By definition $\times^{\Delta}$ is:



5 Nested streams

5.1 Creating and destroying streams

We introduce two new stream operators that are instrumental in expressing recursive query evaluation. These operators allow us to build circuits implementing looping constructs, which are used to iterate computations until a fixed-point is reached.

5.1.1 Generalizing box-and-arrow diagrams

From now on our circuits will mix computations on scalars and streams. We will use the same graphical representation for functions that compute on scalars: boxes with input and output arrows. The values on the the connecting arrows will be scalars instead of streams; otherwise the interpretation of boxes as function application is unchanged.

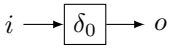
When connecting boxes the types of the arrows must match. E.g., the output of a box producing a stream cannot be connected to the input of a box consuming a scalar.

5.1.2 Stream introduction

Definition 5.1 (Dirac delta)**.** The delta function (named from the Dirac delta function) $\delta_0 : A \to \mathcal{S}_A$ produces a stream from a scalar value. The output stream is produced as follows from the input scalar:

$$\delta_0(v)[t] \stackrel{\text{def}}{=} \begin{cases} v & \text{if } t = 0 \\ 0_A & \text{otherwise} \end{cases}$$

Here is a diagram showing a δ_0 operator; note that the input is a scalar value, while the output is a stream:



For example, $\delta_0(5)$ is the stream [5 0 0 0 0 $\cdots$].

5.1.3 Stream elimination

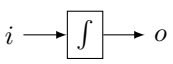
Recall that $\overline{\mathcal{S}_A}$ was defined in Definition 3.11 to be the set of A-streams over a group A that are zero almost everywhere.

Definition 5.2 (indefinite integral)**.** We define a function $\int : \overline{\mathcal{S}_A} \to A$ as $\int(s) \stackrel{\text{def}}{=} \sum_{t \geq 0} s[t]$.

$\int$ is closely related to $\mathcal{I}$; if $\mathcal{I}$ is the indefinite integral, $\int$ is the definite integral on the interval $0 - \infty$. Unlike $\mathcal{I}$ $\int$ produces a scalar value, the "last" distinct value that would appear in the stream produced by $\mathcal{I}$. For example $\int(id|_{\leq 4}) = 0 + 1 + 2 + 3 = 6$, because $\mathcal{I}(id|_{\leq 4}) = [\ 0 \quad 1 \quad 3 \quad 6 \quad 6 \quad \cdots\ ]$.

An alternative definition for $\int$ for all streams $\mathcal{S}_A$ would extend the set A with an "infinite": $\overline{A} \stackrel{\text{def}}{=} A \cup \{\top\}$, and define $\int s \stackrel{\text{def}}{=} \top$ for streams that are not zero a.e., $s \in \mathcal{S}_A \setminus \overline{\mathcal{S}_A}$.

Here is a diagrams showing the $\int$ operator; note that the result it produces is a scalar, and not a stream:



δ_0 is the left inverse of $\int$, i.e., the following equation holds: $\int \circ \delta_0 = id_A$.

5.1.4 The E and X operators

The composition $\mathcal{I} \circ \delta_0$ is frequently used, so we will give it a name, denoting it by $E : A \to \mathcal{S}_A$, $E \stackrel{\text{def}}{=} \mathcal{I} \circ \delta_0$.

Notice that the output of the E operator is a constant infinite stream, consisting the scalar value at the input. $E(5) = [\ 5 \quad 5 \quad 5 \quad 5 \quad 5 \quad \cdots\ ]$.

Similarly, we denote by $X : \mathcal{S}_A \to A$ the combination $X \stackrel{\text{def}}{=} \int \circ \mathcal{D}$.

Proposition 5.3. For a monotone stream $o \in \mathcal{S}_A$ we have $X(o) = \lim_{n \to \infty} o[n]$, if the limit exists.

Proof. $X(o|_{\leq n}) = (\int \circ \mathcal{D})(o|_{\leq n}) = o[0] + (o[1] - o[0]) + \ldots + (o[n] - o[n-1]) = o(n)$. The result follows by taking limits on both sides. □

Clearly, E is the left-inverse of X.

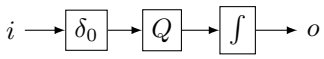
MIHAI: This looks almost right, but it is not.

Proposition 5.4. δ_0, $\int$, E, and X are LTI.

Proof. The proof is easy using simple algebraic manipulation of the definitions of these operators. □

5.1.5 Time domains

So far we had the tacit assumption that "time" is common for all streams in a program. For example, when we add two streams, we assume that they use the same "clock" for the time dimension. However, the δ_0 operator creates a streams with a "new", independent time dimension. In Section 9 we will define some well-formed circuit construction rules that will ensure that such time domains are always "insulated", by requiring each diagram that starts with a δ_0 operator to end with a corresponding $\int$ operator:



Proposition 5.5. If Q is time-invariant, the circuit above has the zero-preservation property: $\text{zpp}(\int \circ Q \circ \delta_o)$.

Proof. This follows from the fact that all three operators preserve zeros, and thus so does their composition. □

5.2 Streams of streams

5.2.1 Defining nested streams

Since all streams we work with are defined over abelian groups and streams themselves form an abelian group, as pointed in Section 3.4, it follows that we can naturally define streams of streams. $\mathcal{S}_{\mathcal{S}_A} = \mathbb{N} \to (\mathbb{N} \to A)$. This construction can be iterated, but our applications do not require more than two-level nesting. Box-and-arrow diagrams can be used equally to depict functions computing on nested streams; in this case an arrow represent a stream where each value is another stream.

Equivalently, a nested stream in $\mathcal{S}_{\mathcal{S}_A}$ is a value in $\mathbb{N} \times \mathbb{N} \to A$, i.e., a "matrix" with an infinite number of rows, where each row is a stream. For example, we can depict the nested stream $i \in \mathcal{S}_{\mathcal{S}_\mathbb{N}}$ defined by $i[t_0][t_1] = t_0 + 2t_1$ as:

$$i = \begin{bmatrix} [& 0 & 1 & 2 & 3 & \cdots &] \\ [& 2 & 3 & 4 & 5 & \cdots &] \\ [& 4 & 5 & 6 & 7 & \cdots &] \\ [& 6 & 7 & 8 & 9 & \cdots &] \\ & & & \vdots & & & \end{bmatrix}$$

(t_0 is the column index, and t_1 is the row index).

5.2.2 Lifting stream operators

We have originally defined lifting (Section 3.1.1) for scalar functions. We can generalize lifting to apply to stream operators as well. Consider a stream operator $S : \mathcal{S}_A \to \mathcal{S}_B$. We define $\uparrow S : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_B}$ as: $(\uparrow S(s))[t_0][t_1] \stackrel{\text{def}}{=} S(s[t_0])[t_1], \forall t_0, t_1 \in \mathbb{N}$. Alternatively, we can write $(\uparrow S)(s) = S \circ s$.

In particular, a scalar function $f : A \to B$ can be can lifted twice to produce an operator between streams of streams: $\uparrow\uparrow f : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_B}$.

Lifting twice a scalar function computes on elements of the matrix pointwise:

$$(\uparrow\uparrow(x \mapsto x \bmod 2))(i) = \begin{bmatrix} [& 0 & 1 & 0 & 1 & \cdots &] \\ [& 0 & 1 & 0 & 1 & \cdots &] \\ [& 0 & 1 & 0 & 1 & \cdots &] \\ [& 0 & 1 & 0 & 1 & \cdots &] \\ & & & \vdots & & & \end{bmatrix}$$

z^{-1} delays the rows of the matrix:

$$z^{-1}(i) = \begin{bmatrix} [& 0 & 0 & 0 & 0 & \cdots &] \\ [& 0 & 1 & 2 & 3 & \cdots &] \\ [& 2 & 3 & 4 & 5 & \cdots &] \\ [& 4 & 5 & 6 & 7 & \cdots &] \\ & & & \vdots & & & \end{bmatrix}$$

while its lifted counterpart delays each column of the matrix:

$$(\uparrow z^{-1})(i) = \begin{bmatrix} [& 0 & 0 & 1 & 2 & \cdots &] \\ [& 0 & 2 & 3 & 4 & \cdots &] \\ [& 0 & 4 & 5 & 6 & \cdots &] \\ [& 0 & 6 & 7 & 8 & \cdots &] \\ & & & \vdots & & & \end{bmatrix}$$

We can also apply both operators, and they commute:

$$(\uparrow z^{-1})(z^{-1}(i)) = z^{-1}((\uparrow z^{-1})(i)) = \begin{bmatrix} [& 0 & 0 & 0 & 0 & \cdots &] \\ [& 0 & 0 & 1 & 2 & \cdots &] \\ [& 0 & 2 & 3 & 4 & \cdots &] \\ [& 0 & 4 & 5 & 6 & \cdots &] \\ & & & \vdots & & & \end{bmatrix}$$

Similarly, we can apply $\mathcal{D}$ to nested streams $\mathcal{D} : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_A}$, computing on rows of the matrix:

$$\mathcal{D}(i) = \begin{bmatrix} [& 0 & 1 & 2 & 3 & \cdots &] \\ [& 2 & 2 & 2 & 2 & \cdots &] \\ [& 2 & 2 & 2 & 2 & \cdots &] \\ [& 2 & 2 & 2 & 2 & \cdots &] \\ & & & \vdots & & & \end{bmatrix}$$

while $\uparrow\mathcal{D} : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_A}$ computes on the columns:

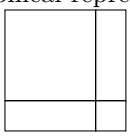
$$(\uparrow\mathcal{D})(i) = \begin{bmatrix} [& 0 & 1 & 1 & 1 & \cdots &] \\ [& 2 & 1 & 1 & 1 & \cdots &] \\ [& 4 & 1 & 1 & 1 & \cdots &] \\ [& 6 & 1 & 1 & 1 & \cdots &] \\ & & & \vdots & & & \end{bmatrix}$$

Similarly, we can apply both differentiation operators in sequence:

$$(\mathcal{D}(\uparrow\mathcal{D}))(i) = \begin{bmatrix} [& 0 & 1 & 1 & 1 & \cdots &] \\ [& 2 & 0 & 0 & 0 & \cdots &] \\ [& 2 & 0 & 0 & 0 & \cdots &] \\ [& 2 & 0 & 0 & 0 & \cdots &] \\ & & & \vdots & & & \end{bmatrix}$$

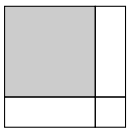
5.2.3 Matrix representations of nested streams

Similar to the vector graphical representations from Section 4.1, we can use a graphical representation of a nested stream s:

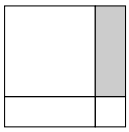


The four rectangles correspond to the following expressions:

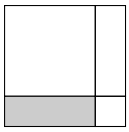
$$\mathcal{I}(\uparrow\mathcal{I}(s))[t_0-1][t_1-1] = \mathcal{I}(z^{-1}(\uparrow\mathcal{I}(\uparrow z^{-1})))[t_0][t_1] =$$



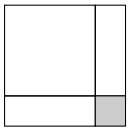
$$\mathcal{I}(s)[t_0][t_1-1] = \mathcal{I}(z^{-1}(s))[t_0][t_1] =$$



$$\uparrow\mathcal{I}(s)[t_0-1][t_1] = \uparrow\mathcal{I}(\uparrow z^{-1})(s)[t_0][t_1] =$$



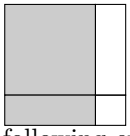
$$s[t_0][t_1] =$$



.

Lifting a vector we obtain a matrix, e.g.: $\uparrow$

$=$

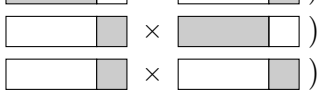


.

Consider a bilinear scalar operator $\times$. Let us expand the following expression: $(\uparrow((\uparrow(a \times b))^{\Delta}))^{\Delta}$.

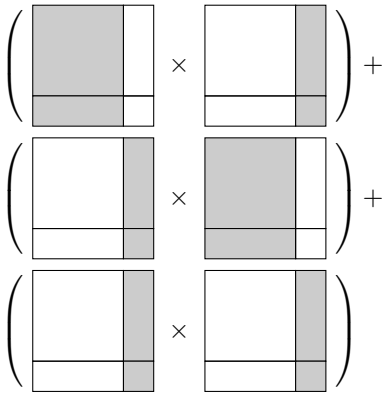
In Section 4.1 we expanded the inner term

$$(\uparrow a \times b)^{\Delta} =$$



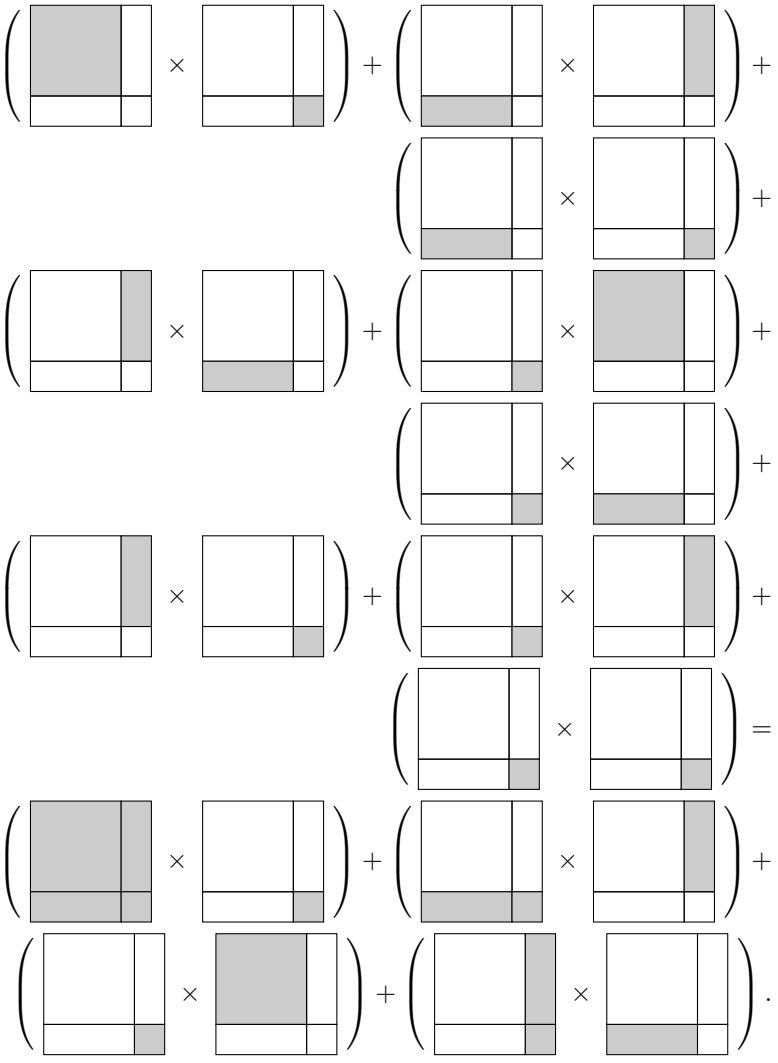
Lifting this term again we get:

$$\uparrow((\uparrow(a \times b))^{\Delta}) =$$



And now we apply incrementalize this again, by expanding each term into 3 other terms and regrouping due to distributivity of $\times$ over addition (bilinearity):

$$(\uparrow((\uparrow(a \times b))^{\Delta}))^{\Delta} =$$



5.2.4 Strict operators on nested streams

In order to show that operators defined using feedback are well-defined on nested streams we need to extend the notion of strict operators from Section 3.3.

We define a partial order over timestamps: $(i_0, i_1) \leq (t_0, t_1)$ iff $i_0 \leq t_0$ and $i_1 \leq t_1$. We extend the definition of strictness for operators over nested streams: a stream operator $F : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_B}$ is strict if for any $s, s' \in \mathcal{S}_{\mathcal{S}_A}$ and

all times $t, i \in \mathbb{N} \times \mathbb{N}$ we have $\forall i < t, s[i] = s'[i]$ implies $F(s)[t] = F(s')[t]$. Proposition 3.14 holds for this notion of strictness, i.e., the fixed point operator $\operatorname{fix} \alpha.F(\alpha)$ is well defined for a strict operator F.

MIHAI: Should write down this proof.

Proposition 5.6. The operator $\uparrow z^{-1} : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_A}$ is strict.

The $\mathcal{I}$ operator on $\mathcal{S}_{\mathcal{S}_A}$ is well-defined: it operates on rows of the matrix, treating each row as a single value:

$$\mathcal{I}(i) = \begin{bmatrix} [& 0 & 1 & 2 & 3 & \cdots &] \\ [& 2 & 4 & 6 & 8 & \cdots &] \\ [& 6 & 9 & 12 & 15 & \cdots &] \\ [& 12 & 16 & 20 & 24 & \cdots &] \\ & & & \vdots & & & \end{bmatrix}$$

With this extended notion of strictness we have that the lifted integration operator is also well-defined: $\uparrow\mathcal{I} : \mathcal{S}_{\mathcal{S}_A} \to \mathcal{S}_{\mathcal{S}_A}$. This operator integrates each column of the stream matrix:

$$(\uparrow\mathcal{I})(i) = \begin{bmatrix} [& 0 & 1 & 3 & 6 & \cdots &] \\ [& 2 & 5 & 9 & 14 & \cdots &] \\ [& 4 & 9 & 15 & 22 & \cdots &] \\ [& 6 & 13 & 21 & 30 & \cdots &] \\ & & & \vdots & & & \end{bmatrix}$$

Notice the following commutativity properties for integration and differentiation on nested streams: $\mathcal{I} \circ (\uparrow\mathcal{I}) = (\uparrow\mathcal{I}) \circ \mathcal{I}$ and $\mathcal{D} \circ (\uparrow\mathcal{D}) = (\uparrow\mathcal{D}) \circ \mathcal{D}$.

5.2.5 Lifted cycles

Proposition 5.7 (Lifting cycles)**.** For a binary, causal T we have:

$$\uparrow(\lambda s.\operatorname{fix} \alpha.T(s, z^{-1}(\alpha))) = \lambda s.\operatorname{fix} \alpha.(\uparrow T)(s, (\uparrow z^{-1})(\alpha))$$

i.e., lifting a circuit containing a "cycle" can be accomplished by lifting all operators independently.

Proof. Consider a stream of streams $a = [a_0, a_1, a_2, \cdots] \in \mathcal{S}_{\mathcal{S}_A}$ (where each $a_i \in \mathcal{S}_A$). The statement to prove becomes:

$$\uparrow(\lambda s.\operatorname{fix} \alpha.T(s, z^{-1}(\alpha)))(a) = \operatorname{fix} \alpha.(\uparrow T)(a, (\uparrow z^{-1})(\alpha))$$

This follows if we show that the value defined as:

$$\beta = \uparrow(\lambda s.\operatorname{fix} \alpha.T(s, z^{-1}(\alpha)))(a)$$

satisfies

$$\beta = (\uparrow T)(a, (\uparrow z^{-1})(\beta))$$

Now, expanding the definition of lifting a function:

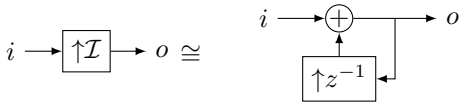
$$\begin{aligned}\uparrow(\lambda s.\mathrm{fix}\,\alpha.T(s, z^{-1}(\alpha)))(a) &\stackrel{\text{def}}{=} \uparrow(\lambda s.\mathrm{fix}\,\alpha.T(x, z^{-1}(\alpha)))([a_0, a_1, \cdots]) \\ &\stackrel{\text{def}}{=} [\mathrm{fix}\,\alpha.T(a_0, z^{-1}(\alpha)), \mathrm{fix}\,\alpha.T(a_1, z^{-1}(\alpha)), \cdots] \\ &= [\alpha_0, \alpha_1, \alpha_2, \cdots]\end{aligned}$$

where, $\forall i.\alpha_i$ is the unique solution of the equation $\alpha_i = T(a_i, z^{-1}(\alpha_i))$. Finally, for $\beta = [\alpha_0, \alpha_1, \cdots]$ we have

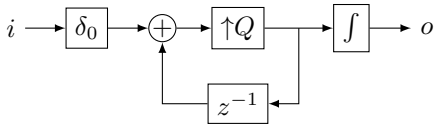
$$(\uparrow T)([a_0, a_1, \ldots], (\uparrow z^{-1})(\beta)) = [T(a_0, z^{-1}(\alpha_0)), T(a_1, z^{-1}(\alpha_1)), \ldots]$$

which finishes the proof. □

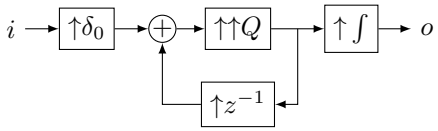
Proposition 5.7 gives us the tool to lift whole circuits. For example, we have:



As another example, consider the following circuit $T : A \to B$ that represents a *scalar* function:

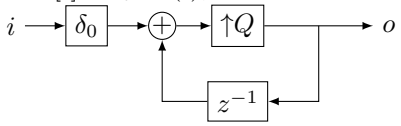


Since this circuit represents a scalar function, it can be lifted like any other scalar function to create a stream computation[3]:



5.3 Fixed-point computations

Theorem 5.8. Given a scalar operator $Q : A \to A$, the output stream o computed by the following diagram (using the lifted version of Q) is given by $\forall t \in \mathbb{N}.o[t] = Q^{t+1}(i)$, where k is the composition of Q with itself k times.



[3]Notice that + is not shown lifted in this circuit, since there is in fact a + operator for any type, and we have that $\uparrow(+_A) = +_{\mathcal{S}_A}$

Proof. Let us compute $o[t]$. We have that $o[0] = Q(\delta_0(i)[0] + 0) = Q(i)$. $o[1] = Q(o[0] + \delta_0(i)[1]) = Q(Q(i) + 0) = Q^2(i)$. We can prove by induction over t that $o[t] = Q^{t+1}(i)$. □

Observation: in this circuit the "plus" operator behaves as an `if`, selecting between the base case and the inductive case in a recursive definition. This is because the left input contains a single non-zero element (i), in the first position, while the bottom input starts with a 0.

Part II
Incremental view maintenance in DBSP

6 Relational algebra in DBSP

Results in Section 3 and Section 4 apply to streams of arbitrary group values. In this section we turn our attention to using these results in the context of relational view maintenance. As explained in the introduction, we want to efficiently compute the incremental version of any relational query Q that updates a database view.

However, we face a technical problem: the $\mathcal{I}$ and $\mathcal{D}$ operators were defined on abelian groups, and relational databases in general are not abelian groups, since they operate on sets. Fortunately, there is a well-known tool in the database literature which converts set operations into group operations by using $\mathbb{Z}$-sets (also called z-relations [15]) instead of sets.

We start by defining the $\mathbb{Z}$-sets group, and then we review how relational queries are converted into DBSP circuits over $\mathbb{Z}$-sets. What makes this translation incrementalizable is the fact that many basic relational queries can be expressed using LTI $\mathbb{Z}$-set operators.

6.1 $\mathbb{Z}$-sets as an abelian group

Given a set A we define **$\mathbb{Z}$-sets**[4] over A as functions with *finite support* from A to $\mathbb{Z}$ (i.e., which are 0 almost everywhere). These are functions $f : A \to \mathbb{Z}$ where $f(x) \neq 0$ for at most a finite number of values $x \in A$. We also write $\mathbb{Z}[A]$ for the type of $\mathbb{Z}$-sets with elements from A. The values in $\mathbb{Z}[A]$ can also be thought as being key-value maps with keys in A and values in $\mathbb{Z}$, justifying the array indexing notation.

Since $\mathbb{Z}$ is an abelian group, $\mathbb{Z}[A]$ is also an abelian group. This group $(\mathbb{Z}[A], +_{\mathbb{Z}[A]}, 0_{\mathbb{Z}[A]}, -_{\mathbb{Z}A})$ has addition and subtraction defined pointwise:

$$(f +_{\mathbb{Z}[A]} g)(x) = f(x) + g(x). \forall x \in A.$$

[4] Also called $\mathbb{Z}$-relations elsewhere [14].

The 0 element of $\mathbb{Z}[A]$ is the function $0_{\mathbb{Z}[A]}$ defined by $0_{\mathbb{Z}[A]}(x) = 0.\forall x \in A$. (In fact, since $\mathbb{Z}$ is a ring, $\mathbb{Z}[A]$ is also ring, endowed with a multiplication operation, also defined pointwise.)

A particular $\mathbb{Z}$-set $m \in \mathbb{Z}[A]$ can be denoted by enumerating the inputs that map to non-zero values and their multiplicities: $m = \{x_1 \mapsto w_1, \ldots, x_n \mapsto w_n\}$. We call $w_i \in Z$ the **multiplicity** (or weight) of $x_i \in A$. Multiplicities can be negative. We write that $x \in m$ for $x \in A$, iff $m[x] \neq 0$.

For example, let's consider a concrete $\mathbb{Z}$-set $R \in \mathbb{Z}[\texttt{string}]$, defined by $R = \{\texttt{joe} \mapsto 1, \texttt{anne} \mapsto -1\}$. R has two elements in its domain, `joe` with a multiplicity of 1 (so $R[\texttt{joe}] = 1$), and `anne` with a multiplicity of -1. We say $\texttt{joe} \in R$ and $\texttt{anne} \in R$.

Given a $\mathbb{Z}$-set $m \in \mathbb{Z}[A]$ and a value $v \in A$, we overload the array index notation $m[v]$ to denote the multiplicity of the element v in m. Thus we write $R[\texttt{anne}] = -1$. When $c \in \mathbb{Z}$, and $v \in A$ we also write $c \cdot v$ for the **singleton** $\mathbb{Z}$-set $\{v \mapsto c\}$. In other words, $3 \cdot \texttt{frank} = \{\texttt{frank} \mapsto 3\}$. We extend scalar multiplication to operate on $\mathbb{Z}$-sets: for $c \in Z, m \in \mathbb{Z}[A]$, $c \cdot m \stackrel{\text{def}}{=} \sum_{x \in m}(c \cdot m[x]) \cdot x$. We then have $2 \cdot R = \{\texttt{joe} \mapsto 2, \texttt{anne} \mapsto -2\}$: multiplying each row weight by 2.

We define the **size** of a $\mathbb{Z}$-set as the size of its support set, and we use the modulus symbol to represent the size: $|m| \stackrel{\text{def}}{=} \sum_{x \in m} 1$. So $|R| = 2$.

6.2 Sets, bags, and $\mathbb{Z}$-sets

$\mathbb{Z}$-sets generalize sets and bags. Given a set with elements from A, it can be represented as a $\mathbb{Z}$-set $\mathbb{Z}[A]$ by associating a weight of 1 with each set element. The function tozset : $2^A \to \mathbb{Z}[A]$, defined as $\text{tozset}(s) = \sum_{x \in s} 1 \cdot x$, converts a set to a $\mathbb{Z}$-set by associating a multiplicity of 1 with each set element. Thus $\text{tozset}(\{\texttt{joe}, \texttt{anne}\}) = \{\texttt{joe} \mapsto 1, \texttt{anne} \mapsto 1\}$.

Definition 6.1. We say that a $\mathbb{Z}$-set represents a **set** if the multiplicity of every element is one. We define a function to check this property $\text{isset} : \mathbb{Z}[A] \to \mathbb{B}$, given by:

$$\text{isset}(m) \stackrel{\text{def}}{=} \begin{cases} \text{true} & \text{if } m[x] = 1, \forall x \in m \\ \text{false} & \text{otherwise} \end{cases}$$

For our example $\text{isset}(R) = \text{false}$, since $R[\texttt{anne}] = -1$. $\text{isset}(\text{tozset}(m)) = \text{true}$ for any set $m \in 2^A$.

Definition 6.2. We say that a $\mathbb{Z}$-set is **positive** (or a **bag**) if the multiplicity of every element is positive. We define a function to check this property $\text{ispositive} : \mathbb{Z}[A] \to \mathbb{B}$, given by

$$\text{ispositive}(m) \stackrel{\text{def}}{=} \begin{cases} \text{true} & \text{if } m[x] \geq 0, \forall x \in A \\ \text{false} & \text{otherwise} \end{cases}$$

For our example $\text{ispositive}(R) = \text{false}$, since $R[\texttt{anne}] = -1$, but $\text{isset}(m) \Rightarrow \text{ispositive}(m).\forall m \in \mathbb{Z}[A]$.

We also write $m \geq 0$ when m is positive. For positive m, n we write $m \geq n$ for $m, n \in \mathbb{Z}[A]$ iff $m - n \geq 0$. The relation $\geq$ is a partial order.

Definition 6.3. The function $\mathit{distinct} : \mathbb{Z}[A] \to \mathbb{Z}[A]$ projects a $\mathbb{Z}$-set into an underlying set (but *the result is still a $\mathbb{Z}$-set*). The definition is $\forall x \in A$

$$\mathit{distinct}(m)[x] \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } m[x] > 0 \\ 0 & \text{otherwise} \end{cases}$$

$\mathit{distinct}(R) = \{\texttt{joe} \mapsto 1\}$.

distinct "removes" elements with negative multiplicities. zpp(*distinct*).

Circuits derived from relational program will only operate with positive $\mathbb{Z}$-sets; non-positive values will be only used to represent *changes* to $\mathbb{Z}$-sets (a change with negative weights will remove elements from a $\mathbb{Z}$-set).

Proposition 6.4. *distinct* is idempotent: $\mathit{distinct} = \mathit{distinct} \circ \mathit{distinct}$.

Proposition 6.5. For any $m \in \mathbb{Z}[A]$ we have: isset(*distinct*(m)) and ispositive(*distinct*(m)).

We call a function $f : \mathbb{Z}[I] \to \mathbb{Z}[O]$ **positive** if $\forall x \in \mathbb{Z}[I], x \geq 0_{\mathbb{Z}[I]} \Rightarrow f(x) \geq 0_{\mathbb{Z}[0]}$. We extend the notation used for $\mathbb{Z}$-sets for functions as well: ispositive(f).

Correctness of the DBSP implementations The function toset : $\mathbb{Z}[A] \to 2^A$, defined as $\text{toset}(m) = \cup_{x \in \mathit{distinct}(m)}\{x\}$, converts a $\mathbb{Z}$-set into a set.

A relational query f that transforms a set V into a set U will be implemented by a DBSP computation f' on $\mathbb{Z}$-sets. The correctness of the implementation requires that the following diagram commutes:

$$\begin{array}{ccc} V & \xrightarrow{\quad f \quad} & U \\ {\scriptstyle \text{tozset}}\downarrow & & \uparrow{\scriptstyle \text{toset}} \\ VZ & \xrightarrow{\quad f' \quad} & UZ \end{array}$$

Remark: We can generalize the notion of $\mathbb{Z}$-sets to functions $m : A \to \mathbf{G}$ for a ring $\mathbf{G}$ other than $\mathbb{Z}$. The properties we need from the ring structure are the following: the ring must be a commutative group (needed for defining $\mathcal{I}$, $\mathcal{D}$, and z^{-1}), the multiplication operation must distribute over addition (needed to define Cartesian products), and there must be a notion of positive values, needed to define the *distinct* function. Rings such as $\mathbb{Q}$ or $\mathbb{R}$ would work perfectly.

6.3 Streams over $\mathbb{Z}$-sets

Since all the results from Section 3 are true for streams over an arbitrary abelian group, they extend to streams where the elements are $\mathbb{Z}$-sets. In the rest of this text we only consider streams of the form $\mathcal{S}_{\mathbb{Z}[A]}$, for some element type A.

An example of a stream of $\mathbb{Z}$-sets is
$s = [\ 0 \quad R \quad -1 \cdot R \quad 2 \cdot R \quad -2 \cdot R \quad \cdots\ ]$. We have $s[2] = -R = \{\texttt{joe} \mapsto -1, \texttt{anne} \mapsto 1\}$.

Definition 6.6. A stream $s \in \mathcal{S}_{\mathbb{Z}[A]}$ is **positive** if every value of the stream is positive: $s[t] \geq 0. \forall t \in \mathbb{N}$.

Definition 6.7. A stream $s \in \mathcal{S}_{\mathbb{Z}[A]}$ is **monotone** if $s[t] \geq s[t-1], \forall t \in \mathbb{N}$.

Lemma 6.8. Given a positive stream $s \in \mathcal{S}_{\mathbb{Z}[A]}$ the stream $\mathcal{I}(s)$ is monotone.

Proof. Let us compute $\mathcal{I}(s)[t{+}1] - \mathcal{I}(s)[t] = \sum_{i \leq t+1} s[i] - \sum_{i \leq t} s[i] = s[t{+}1] \geq 0$, by commutativity and positivity of s. □

Lemma 6.9. Given a monotone stream $s \in \mathcal{S}_{\mathbb{Z}[A]}$, the elements of the stream $\mathcal{D}(s)$ are positive.

Proof. By the definition of monotonicity $s[t+1] \geq s[t]$. By definition of $\mathcal{D}$ we have $\mathcal{D}(s)[t+1] = s[t+1] - s[t] \geq 0$. □

6.4 Implementing the relational algebra

The fact that the relational algebra can be implemented by computations on $\mathbb{Z}$-sets has been shown before, e.g. [15]. The translation of the core relational operators is summarized in Table 1 and discussed below.

The translation is fairly straightforward, but many operators require the application of a *distinct* to produce sets. The correctness of this implementation is predicated on the global circuit inputs being sets as well.

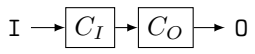
6.4.1 Query composition

A composite query is translated by compiling each sub-query separately into a circuit and composing the respective circuits.

For example, consider the following SQL query:

```
SELECT ... FROM (SELECT ... FROM ...)
```

given circuits C_O implementing the outer query and C_I implementing the inner query, the translation of the composite query is:



We have $\text{ispositive}(C_I) \wedge \text{ispositive}(C_0) \Rightarrow \text{ispositive}(C_O \circ C_I)$ and $\text{zpp}(C_I) \wedge \text{zpp}(C_O) \Rightarrow \text{zpp}(C_O \circ C_I)$.

6.4.2 Set union

Consider the following SQL query:

```
(SELECT * FROM I1) UNION (SELECT * FROM I2)
```

The following circuit implements the union program:

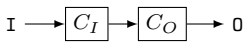
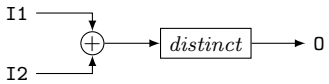
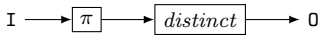
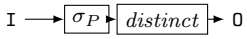
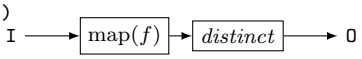
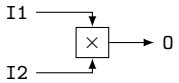
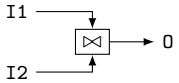
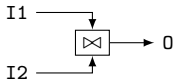
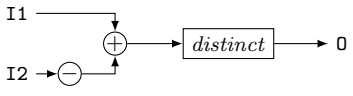
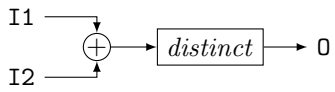
Operation	SQL example	DBSP circuit
Composition	`SELECT DISTINCT ... FROM (SELECT ... FROM ...)`	
Union	`(SELECT * FROM I1) UNION (SELECT * FROM I2)`	
Projection	`SELECT DISTINCT I.c FROM I`	
Filtering	`SELECT * FROM I WHERE p(I.c)`	
Selection	`SELECT DISTINCT f(I.c, ...) FROM I`	
Cartesian product	`SELECT I1.*, I2.* FROM I1, I2`	
Join	`SELECT I1.*, I2.* FROM I1 JOIN I2 ON I1.c1 = I2.c2`	
Intersection	`(SELECT * FROM I1) INTERSECT (SELECT * FROM I2)`	
Difference	`SELECT * FROM I1 EXCEPT SELECT * FROM I2`	

Table 1: Implementation of SQL relational set operators in DBSP. Each query assumes that inputs `I`, `I1`, `I2`, are sets and it produces output sets.



Given $\mathbb{Z}$-sets $a, b \in \mathbb{Z}[I]$ s.t. isset(a) and isset(b), their *set union* can be computed as: $\cup : \mathbb{Z}[I] \times \mathbb{Z}[I] \to \mathbb{Z}[I]$,

$$a \cup b \stackrel{\text{def}}{=} distinct(a +_{\mathbb{Z}[I]} b).$$

The *distinct* application is necessary to provide the set semantics.

6.4.3 Projection

Consider a query such as:

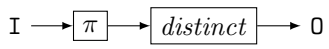
```
SELECT I.c FROM I
```

We can assume without loss of generality that table `I` has two columns, and that a single column is preserved in the projection. Hence the type of `I` is $\mathbb{Z}[A_0 \times A_1]$ while the result has type is $\mathbb{Z}[A_0]$. In terms of $\mathbb{Z}$-sets, the projection of a $\mathbb{Z}$-set i on A_0 is defined as:

$$\pi(i)[y] = \sum_{x \in i, x|_0 = y} i[x]$$

where $x|_0$ is first component of the tuple x. In other words, the multiplicity of a tuple in the result is the sum of the multiplicities of all input tuples that project to it.

The circuit for a projection query is:



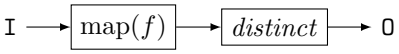
The *distinct* is necessary to convert the result to a set.
π is linear; ispositive(π), zpp(π).

6.4.4 Selection

We generalize the SQL selection operator to allow it to apply an arbitrary function to each row of the selected set. Given a function $f : A \to B$, the mathematical **map** operator "lifts" the function f to operate on $\mathbb{Z}$-sets: map(f) : $\mathbb{Z}[A] \to \mathbb{Z}[B]$. A map operator appears in SQL due to the use of expressions in the SELECT clause, as in the following example:

```
SELECT f(I.c) FROM I
```

The circuit implementation of this query is:



For any function f we have the following properties: map(f) is linear, ispositive(map(f)), andzpp(map(f)).

6.4.5 Filtering

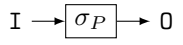
Filtering occurs in SQL through a WHERE clause, as in the following example:

```
SELECT * FROM I WHERE p(I.c)
```

Let us assume that we are filtering with a predicate $P : A \to \mathbb{B}$. We define the following function $\sigma_P : \mathbb{Z}[A] \to \mathbb{Z}[A]$ as:

$$\sigma_P(m)[t] = \begin{cases} m[t] \cdot t & \text{if } P(t) \\ 0 & \text{otherwise} \end{cases}$$

The circuit for filtering with a predicate P is:



For any predicate P we have $\text{isset}(i) \Rightarrow \text{isset}(\sigma_P(i))$ and $\text{ispositive}(\sigma_P)$. Thus a *distinct* is not needed. σ_P is linear and $\text{zpp}(\sigma_P)$.

6.4.6 Cartesian products

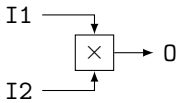
Consider this SQL query performing a Cartesian product between sets `I1` and `I2`:

```
SELECT I1.*, I2.* FROM I1, I2
```

We first define a product operation on $\mathbb{Z}$-sets. For $a \in \mathbb{Z}[A]$ and $b \in \mathbb{Z}[B]$ we define $a \times b \in \mathbb{Z}[A \times B]$ by

$$(a \times b)((x, y)) \stackrel{\text{def}}{=} a[x] \times b[y]. \forall x \in a, y \in b.$$

The weight of a pair in the result is the product of the weights of the elements in the sources. The circuit for the query is:



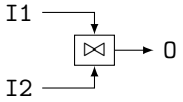
$\text{isset}(x) \wedge \text{isset}(y) \Rightarrow \text{isset}(x \times y)$. $\times$ is bilinear, $\text{ispositive}(\times), \text{zpp}(\times)$.

6.4.7 Joins

As is well-known, joins can be modeled as Cartesian products followed by filtering. Since a join is a composition of a bilinear and a linear operator, it is also a bilinear operator. $\text{ispositive}(\bowtie), \text{zpp}(\bowtie)$.

In practice joins are very important computationally, and they are implemented by a built-in scalar function on $\mathbb{Z}$-sets:

$$(a \bowtie b)((x, y)) \stackrel{\text{def}}{=} a[x] \times b[y] \text{ if } x|_{c1} = y|_{c2}.$$



6.4.8 Set intersection

Set intersection is a special case of join. It follows that set intersection is bilinear, and has the zero-preservation property.

6.4.9 Set difference

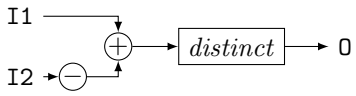
Consider the following query:

```
SELECT * FROM I1 EXCEPT SELECT * FROM I2
```

We define the set difference on $\mathbb{Z}$-sets as follows: $\setminus : \mathbb{Z}[I] \times \mathbb{Z}[I] \to \mathbb{Z}[I]$, where

$$i_1 \setminus i_2 = \mathit{distinct}(i_1 - i_2).$$

Note that we have $\forall i_1, i_2, \text{ispositive}(i_1 \setminus i_2)$ due to the *distinct* operator. The circuit computing the above query is:

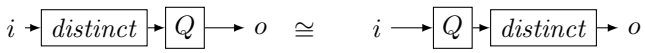


6.5 Optimizing relational circuits

6.5.1 Optimizing *distinct*

All standard algebraic properties of the relational operators can be used to optimize circuits. In addition, a few optimizations are related to the *distinct* operator, which is not linear, and thus expensive to incrementalize:

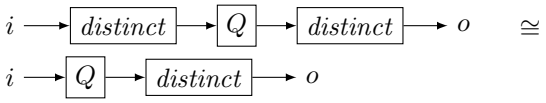
Proposition 6.10. Let Q be one of the following $\mathbb{Z}$-sets operators: filtering σ, join $\bowtie$, or Cartesian product $\times$. Then we have $\forall i \in \mathbb{Z}[I], \text{ispositive}(i) \Rightarrow Q(\mathit{distinct}(i)) = \mathit{distinct}(Q(i))$.



This rule allows us to delay the application of *distinct*.

Proposition 6.11. Let Q be one of the following $\mathbb{Z}$-sets operators: filtering σ, projection π, selection (map(f)), addition $+$, join $\bowtie$, or Cartesian product $\times$. Then we have $\forall i \in \mathbb{Z}[I], \text{ispositive}(i) \Rightarrow \mathit{distinct}(Q(\mathit{distinct}(i))) = \mathit{distinct}(Q(i))$.

This is Proposition 6.13 in [14].



These properties allow us to "consolidate" distinct operators by performing one distinct at the end of a chain of computations.

Proposition 6.12. The following circuit implements $(\uparrow distinct)^{\Delta}$:

where $H : \mathbb{Z}[A] \times \mathbb{Z}[A] \to \mathbb{Z}[A]$ is defined as:

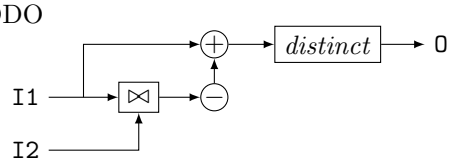
$$H(i,d)[x] \stackrel{\text{def}}{=} \begin{cases} -1 & \text{if } i[x] > 0 \text{ and } (i+d)[x] \leq 0 \\ 1 & \text{if } i[x] \leq 0 \text{ and } (i+d)[x] > 0 \\ 0 & \text{otherwise} \end{cases}$$

The function H detects whether the multiplicity of an element in the input set i is changing from negative to positive or vice-versa.

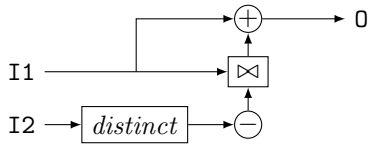
6.5.2 Anti-joins

TODO

MIHAI: justify this.



This can be optimized as follows:



6.6 Incremental relational queries

Let us consider a relational query Q defining a view. To create a circuit that maintains incrementally the view defined by Q we apply the following mechanical steps:

Algorithm 6.13 (incremental view maintenance)**.**

1. Translate Q into a circuit using the rules in Table 1.
2. Apply optimization rules, including *distinct* consolidation.
3. Lift the whole circuit, by applying Proposition 3.3, converting it to a circuit operating on streams.
4. Incrementalize the whole circuit "surrounding" it with $\mathcal{I}$ and $\mathcal{D}$.
5. Apply the chain rule and other properties of the $\cdot^{\Delta}$ operator from Proposition 4.2 to optimize the incremental implementation.

Step (3) yields a circuit that consumes a stream of complete database snapshots and outputs a stream of complete view snapshots. Step (4) yields a circuit that consumes a stream of changes to the database and outputs a stream of view changes; however, the internal operation of the circuit is non-incremental, as it computes on the complete state of the database reconstructed by the integration operator. Step (5) incrementalizes the internals of the circuit by rewriting it to compute on changes, avoiding integration when possible (see Section 4).

6.6.1 Examples

TODO.

MIHAI: Add there a moderately complicated SQL query and show its incremental circuit.

6.6.2 Computational Complexity

Incremental circuits are efficient. The work (and memory) performed by a circuit is the sum of the work performed (and memory used) by its operators. We argue that each operator in the incremental version of a circuit is efficient.

For incrementalized circuits the input stream of each operator contains *changes* in its input relations. Denote $C[t] \stackrel{\text{def}}{=} \|s[t]\|$ the **size** of the value of stream s of changes at time t, and $R[t] \stackrel{\text{def}}{=} \|\mathcal{I}(s)[t]\|$ the size of the relation produced by integrating all changes in s. An unoptimized incremental operator $Q^\Delta = \mathcal{D} \circ Q \circ \mathcal{I}$ evaluates query Q on the integration of its input streams; hence its time complexity is the same as that of the non-incremental operator, a function of $R[t]$. In addition, because of the $\mathcal{I}$ and $\mathcal{D}$ operators, it uses $O(R[t])$ memory.

The optimizations described in Section 4 reduce the reduce the time complexity of an operator to be a function of $C[t]$. Assuming $C[t] \ll R[t]$, this translates to major performance improvements in practice. For example, Theorem 4.4, allows evaluating T^Δ, where T is a linear operator, in time $O(C[t])$. While the *distinct* operator is not linear, $(\uparrow distinct)^\Delta$ can also be evaluated in $O(C[t])$ according to Proposition 6.12. Bilinear operators, including join, can be evaluated in time proportional to the product of the sizes of their input changes $O(C[t]^2)$ (Theorem 4.5).

The space complexity of linear operators is 0 (zero), since they store no data persistently. The space complexity of $(\uparrow distinct)^\Delta$ and join is $O(R[t])$.

7 Recursive queries in DBSP

Recursive queries are very useful in a many applications. For example, many graph algorithms (such as graph reachability or transitive closure) are naturally expressed using recursive queries.

We illustrate the implementation of recursive queries in DBSP for stratified Datalog.

7.1 Implementing recursive queries

For warm-up we start with a single recursive queries, and then we discuss the case of mutually recursive queries.

7.1.1 Recursive rules

In Datalog a recursive rule appears when a relation that appears in the head of a rule is also used in a positive term in the rule's body. (Stratification disallows the use of the same relation negated in the rule's body). The Datalog semantics of recursive rules is to compute a fixedpoint.

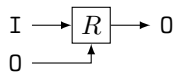
Consider a Datalog program of the form:

```
O(v) :- C(v).   // base case
O(v) :- ..., O(x), I(z), ... .   // recursive case
```

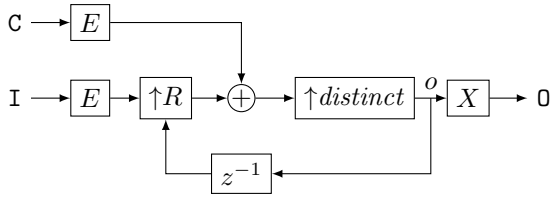
Note that relation `O` is recursively defined. Let us assume wlog that the `O` relation depends on two other relations (i.e., in the rule bodies defining `O` the two other relations appear) : a "base case" relation `C` (which appears in a non-recursive rule), and a relation `I` which appears in the recursive rule, but does not itself depend on `O`.

To implement the computation of `O` as a circuit we perform the following algorithm:

1. Implement a circuit R for the recursive rule body by treating the relation `O` in the body as an input relation. This produces a circuit with two inputs: $R(\mathtt{I}, \mathtt{O})$.



2. Lift this circuit and connect it as follows:



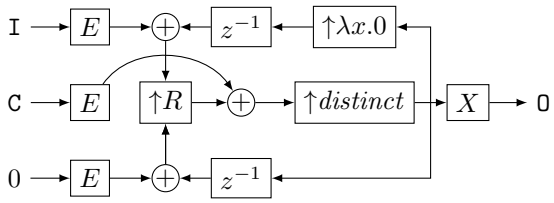
   Notice that the recursive input is connected from the output of the *distinct* node through a z^{-1} operator. In addition, the `C` input is connected through an addition operator. The combination + followed by *distinct* is in fact just the set union implementation from Section 11.2.3.

This circuit as drawn is not a well-formed circuit.

It can, however, be modified into an equivalent well-formed circuit by adding two constant zero value streams:

VAL: Why not well-formed? As far as I can see its semantics follows from Corollary 3.17.

MIHAI: The WFC rules require any circuit bracketed by $\delta_0 - \int$ to have no other input or output edges. It also requires back-edges to go through a plus only. It is more strict than the stream computation rules.



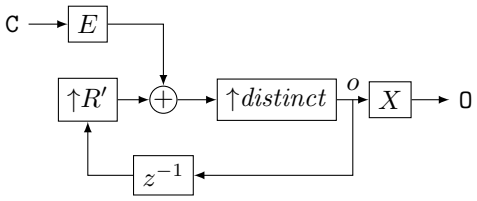
Theorem 7.1. If isset(I) and isset(C), the output of the circuit above is the relation O as defined by the Datalog semantics of recursive relations as a function of the input relations I and C.

Proof. The proof is by structural induction on the structure of the circuit. As a basis for induction we assume that the circuit R correctly implements the semantics of the recursive rule body when treating O as an independent input. We need to prove that the output of circuit encompassing R produces the correct value of the O relation, as defined by the recursive Datalog equation.

Let us compute the contents of the o stream, produced at the output of the *distinct* operator. We will show that this stream is composed of increasing approximations of the value of O, and in fact $\mathtt{O} = \lim_{t\to\infty} o[t]$ if the limit exists.

We define the following one-argument function: $R'(x) = \lambda x.R(\mathtt{I}, x)$. Notice that the left input of the $\uparrow R$ block is a constant stream with the value I. Due to the stratified nature of the language, we must have ispositive(R'), so $\forall x.R'(x) \geq x$. Also $\uparrow R'$ is time-invariant, so $R'(0) = 0$.

With this notation for R' the previous circuit has the output as the following simpler circuit:



We use the following notation: $x \cup y = distinct(x + y)$. As discussed in Section 11.2.3 the $\cup$ operation computes the same result as set union when x and y are sets. With this notation let us compute the values of the o stream:

$$\begin{aligned} o[0] &= distinct(\mathtt{C} + R'(0)) = \mathtt{C} \cup R'(0) = \mathtt{C} \\ o[1] &= distinct(\mathtt{C} + R'(o[0])) = \mathtt{C} \cup R'(\mathtt{C}) \\ o[t] &= distinct(\mathtt{C} + R(o[t-1])) = \mathtt{C} \cup R'(o[t-1]) \end{aligned}$$

Defining a new helper function $S(x) = \mathtt{C} \cup R'(x)$, the previous system of equations becomes:

$$\begin{aligned} o[0] &= S(0) \\ o[1] &= S(S(0)) \\ o[t] &= S(o[t-1]) \end{aligned}$$

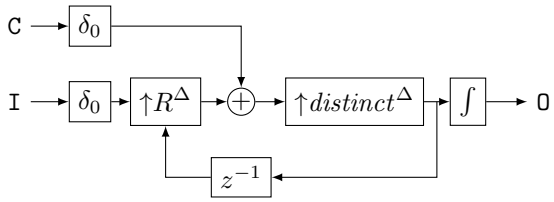
So, by induction $o[t] = S^t(0)$, where by S^t we mean $\underbrace{S \circ S \circ \ldots \circ S}_{t}$. S is monotone because R' is monotone; thus, if there is a time k such that $S^k(0) = S^{k+1}(0)$, we have $\forall j \in \mathbb{N}. S^{k+j}(0) = S^k(0)$.

$\mathtt{O}$ is computed by the X operator as the limit of stream o: $\mathtt{O} = X(o) = \lim_{n\to\infty} o[n]$. If this limit exists (i.e., a fixed-point is reached), the circuit computes the fixed point $\mathrm{fix}\, x.S(x)$. This is exactly the definition of the Datalog semantics of a recursive relation definition: $\mathtt{O} = \mathrm{fix}\, x.\mathtt{C} \cup R(\mathtt{I}, x)$. □

Note that the use of unbounded domains (like integers with arithmetic operations) does not guarantee convergence for all programs.

Our circuit implementation is in fact computing the value of relation $\mathtt{O}$ using the standard **naïve evaluation** algorithm (e.g., see Algorithm 1 from [13]).

Observe that the "inner" part of the circuit is the incremental form of another circuit, since is "sandwiched" between $\mathcal{I}$ and $\mathcal{D}$ operators. According to Proposition 4.2, part 7, the circuit can be rewritten as:

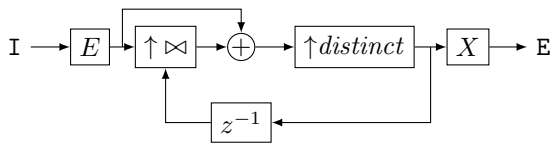


This form of the circuit is effectively implementing the **semi-naïve evaluation** of the same relation (Algorithm 2 from [13]). So the correctness of semi-naïve evaluation is an immediate consequence of the cycle rule from Proposition 4.2.

Example: transitive closure Consider the Datalog program below computing the transitive closure of a relation `I`:

```
E(x, y) :- I(x, y).
E(x, y) :- I(x, z), E(z, y).
```

According to the rules above, the following circuit is an implementation of relation `E` as a function of the input relation `I`:



7.1.2 Mutually recursive rules

Given a stratified Datalog program we can compute a graph where relations are nodes and dependences between relations are edges. We then compute the

strongly connected components of this graph. All relations from a strongly-connected component are mutually recursive.

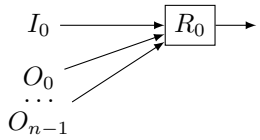
Let us consider the implementation of a single strongly-connected component defining n relations $O_i, i \in [n]$. We can assume wlog that the definition of O_i has the following structure:

$$\begin{array}{lll} O_i(v) & \texttt{:-} & C_i(v)\,. \\ O_i(v) & \texttt{:-} & I_i(x), O_0(v_0), O_1(v_1), \ldots, O_{n-1}(v_{n-1}), v = \ldots\,. \end{array}$$

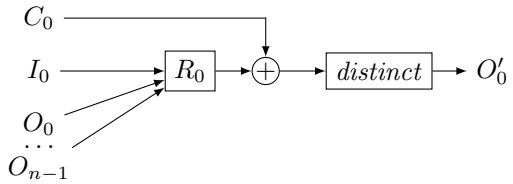
There are exactly n base cases, one defining each O_i. Also, we assume that each O_i relation depends on an external relation I_i, which does not itself depend on any O_k.

To compile these into circuits we generalize the algorithm from Section 7.1.1:

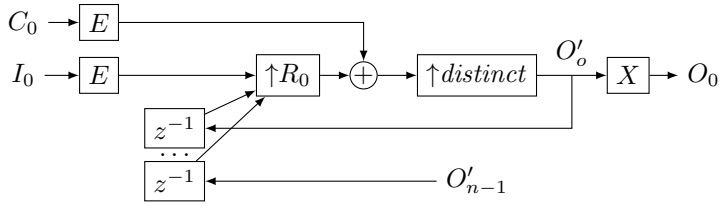
1. For each recursive rule for O_i implement a circuit R_i that treats all $O_j.j \in [n]$ and I_i in the rule body as inputs. Here is the circuit R_0 for relation O_0:



2. Embed each circuit R_i as part of a "widget" as follows:



3. Finally, lift each such widget and connect them to each other via z^{-1} operators to the corresponding recursive inputs. The following is the shape of the circuit computing O_0; the O'_j sources correspond to the widget outputs of the other recursive circuits:



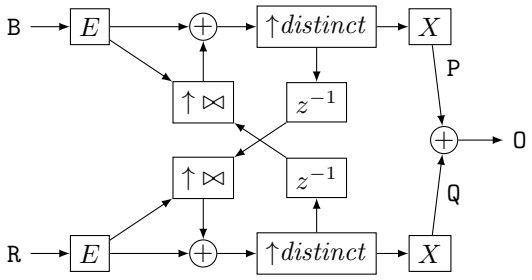
Theorem 7.2. The program defined by the previous circuit computes the relations O_i as a function of the input relations I_j and C_i.

Proof. TODO. □

Example: mutually recursive relations Consider the Datalog program below computing the transitive closure of a graph having two kinds of edges, blue (B) and red (R):

```
P(x,y) :- B(x,y).
Q(x,y) :- R(x,y).
P(x,y) :- B(x,z), Q(z,y).
Q(z,y) :- R(x,z), P(z,y).
O(x,y) :- P(x,y).
O(x,y) :- Q(x,y).
```

The program defined by the following circuit computes the relation O as a function of the input relations R, B:

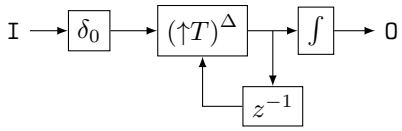


VAL: I will add in Section 3.3 a consequence of Corollary 3.17 to justify the well-definedness of this.

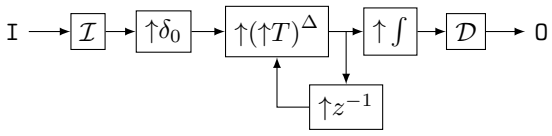
7.2 Incremental recursive queries

In Section 3–6 we showed how to incrementalize a relational query by compiling it into a circuit, lifting the circuit to compute on streams, and applying the $\cdot^\Delta$ operator to the lifted circuit. In Section 11 we showed how to compile a recursive query into a circuit that employs incremental computation internally to compute the fixed point. Here we combine these results to construct a circuit that evaluates a *recursive query incrementally*. The circuit receives a stream of updates to input relations, and for every update recomputes the fixed point. To do this incrementally, it preserves the stream of changes to recursive relations produced by the iterative fixed point computation, and adjusts this stream to account for the modified inputs. Thus, every element of the input stream yields a stream of adjustments to the fixed point computation, using *nested streams*.

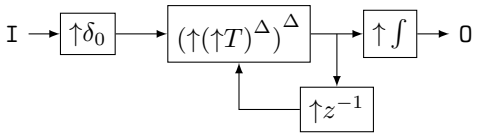
This proposition gives the ability to lift entire circuits, including circuits computing on streams and having feedback edges, which are well-defined, due to Proposition 5.6. With this machinery we can now apply Algorithm 6.13 to arbitrary circuits, even circuits built for recursively-defined relations. Consider the "semi-naive" circuit from Section 7: and denote $\mathit{distinct} \circ R$ with T:



Lift the entire circuit using Proposition 5.7 and incrementalize it:



Now apply the chain rule to this circuit, and use the linearity of δ_0 and $\int$:



(1)

This is the incremental version of an arbitrary recursive query.

7.2.1 Computational complexity

Time complexity The time complexity of an incremental recursive query can be estimated as a product of the number of fixed point iterations and the complexity of each iteration. The incrementalized circuit (1) performs the same number of iterations as the non-incremental circuit (**??**) in the worst case: once the non-incremental circuit reaches the fixed point, its output is constant and so is its derivative computed by the incrementalized circuit.

Consider a nested stream of changes $s \in \mathcal{S}_{\mathcal{S}_A}, s[t_1][t_2]$, where t_1 is the input timestamp and t_2 is the fixed point iteration number. The unoptimized loop body $(\uparrow(\uparrow T)^\Delta)^\Delta = \mathcal{D} \circ \uparrow\mathcal{D} \circ \uparrow\uparrow T \circ \uparrow\mathcal{I} \circ \mathcal{I}$ has the same time complexity as T applied to the aggregated input of size $R(s)[t_1][t_2] \stackrel{\text{def}}{=} \|(\uparrow\mathcal{I} \circ \mathcal{I})(s)[t_1][t_2]\| = \|\sum_{(i_1,i_2)\leq(t_1,t_2)} s[i_1][i_2]\|$. As before, an optimized circuit can be significantly more efficient. For instance, by applying Theorem 4.5 twice, to $\bowtie$ and $\uparrow\bowtie$, we obtain a circuit for nested incremental join $s_1(\uparrow(\uparrow\bowtie)^\Delta)^\Delta s_2$ that runs in $O(\|\uparrow\mathcal{I}(s1)[t1][t2]\| \times \|\mathcal{I}(s2)[t1][t2]\|) \ll O(R(s_1) \times R(s_2))$ (because each term is correspondingly smaller).

Space complexity Integration ($\mathcal{I}$) and differentiation ($\mathcal{D}$) of a stream $s \in \mathcal{S}_{\mathcal{S}_A}$ uses memory proportional to $\sum_{t_2} \|\sum_{t_1} s[t_1][t_2]\|$, i.e., the total size of changes aggregated over columns of the matrix. The unoptimized circuit integrates and differentiates respectively inputs and outputs of the recursive program fragment. As we move $\mathcal{I}$ and $\mathcal{D}$ inside the circuit using the chain rule, we additionally store changes to intermediate streams. Effectively we cache results of fixed point iterations from earlier timestamps to update them efficiently as new input changes arrive. Notice that space is proportional to the number of iterations of the inner while loop.

8 Additional query languages

In this section we describe several query models that go behind stratified Datalog and show how they can be implemented in DBSP.

8.1 Nested relations

8.1.1 Indexed partitions

Let $A[K]$ be the set of functions with finite support from K to A. Consider a group A, an arbitrary set of **key values** K, and a *partitioning function* $k : A \to A[K]$ with the property that $\forall a \in A.a = \sum k(a)$. We call elements of $A[K]$ *indexed* values of A — indexed by a key value.

Notice that $A[K]$ also has a group structure, and k itself is a linear function (homomorphism). As an example, if $A = \mathbb{Z}[B_0 \times B_1]$, we can use for k the first projection $k : A \to \mathbb{Z}[A][B_0]$, where $k(a)[b] = \sum_{t \in a, t|0=b} a[t] \cdot t$. In other words, k projects the elements in $\mathbb{Z}[B_0 \times B_1]$ on their first component. This enables *incremental computations on nested relations.* This is how operators such as group-by are implemented: the result of group-by is an indexed $\mathbb{Z}$-set, where each element is indexed by the key of the group it belongs to. Since indexing is linear, its incremental version is very efficient. Notice that the structure $\mathbb{Z}[A][K]$ represents a form of *nested relation.*

8.1.2 Grouping

We model the SQL `GROUP BY` operator in DBSP. Consider a partitioning function $p : A \to K$, where K is an arbitrary set of "key values". We define the grouping function $G_p : \mathbb{Z}[A] \to (K \to \mathbb{Z}[A])$ as $G_p(a)(k) \stackrel{\text{def}}{=} \sum_{x \in a.p(x)=k} a[x] \cdot x$; when applied to a $\mathbb{Z}$-set a this function returns a collection of groupings[5]: for each key k a grouping is a $\mathbb{Z}$-set containing all elements of a that map to k (as in SQL, groupings are multisets). The structure $K \to \mathbb{Z}[A] = \mathbb{Z}[A][K]$ is also an abelian group, and the function G_p is linear; thus, it is efficiently incrementalizable.

8.1.3 Aggregation

Grouping in SQL is followed by aggregation: given a function $f : \mathbb{Z}[A] \to B$, the aggregation operator $Agg_f : \mathbb{Z}[A][K] \to \mathbb{Z}[B]$ applies f to the contents of each grouping and adds the results, producing another multiset: $Agg_f(g) \stackrel{\text{def}}{=} \sum_{k \in K.g(k) \neq 0} 1 \cdot f(g(k))$. If f is linear (like `SUM` and `COUNT`), Agg_f is also linear, and hence $\uparrow(Agg_f)^{\Delta}$ can be efficiently incrementalized.

8.1.4 Flatmap

Formally, given a function from scalars to $\mathbb{Z}$-sets, $f : A \to \mathbb{Z}[B]$, we define the following operator between $\mathbb{Z}$-sets: flatmap$(f) : \mathbb{Z}[A] \to \mathbb{Z}[B]$ as flatmap$(f)(i) \stackrel{\text{def}}{=} \sum_{x \in i} i[x] \cdot f(x)$.

Proposition 8.1. The flatmap(f) operator is linear for any function f:

[5] We use "group" for the algebraic structure and "grouping" for the result of `GROUP BY`.

Proof.

$$
\begin{aligned}
\text{flatmap}(f)(a+b) &= \sum_{x \in a+b} (a+b)[x] \cdot f(x) && \text{definition of flatmap} \\
&= \sum_{x \in a+b} (a[x] + b[x]) \cdot f(x) && \text{commutativity of plus} \\
&= \sum_{x \in a+b} a[x] \cdot f(x) + \sum_{x \in a+b} b[x] \cdot f(x) && \text{commutativity} \\
&= \sum_{x \in a} a[x] \cdot f(x) + \sum_{x \in b} b[x] \cdot f(x) && \text{ignoring 0 terms} \\
&= \text{flatmap}(f)(a) + \text{flatmap}(f)(b) && \text{definition of flatmap}
\end{aligned}
$$

□

It immediately follows that $\uparrow\text{flatmap}(f)$ is time-invariant for any function f, since any linear operator has the zero-preservation property. **MIHAI**: continue this

8.2 Streaming joins

Consider a binary query $T(s,t) = \mathcal{I}(s) \uparrow \bowtie t$. This is the *relation-to-stream join* operator supported by streaming databases like ksqlDB [20]. Stream s carries changes to a relation, while t carries arbitrary data, e.g., logs or telemetry data points. T discards values from t after matching them against the accumulated contents of the relation.

8.3 Explicit delay

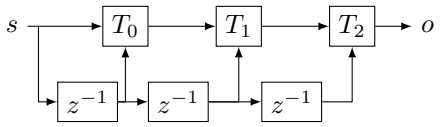
So far the z^{-1} operator was confined to its implicit use in integration or differentiation. However, it can be exposed as a primitive operation that can be applied to streams or collections. This enables programs that can perform time-based window computations over streams, and convolution-like operators.

8.4 Multisets/bags

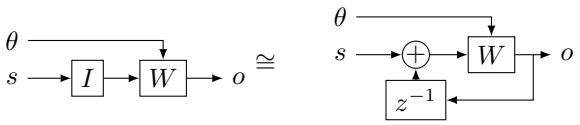
Internally DDlog computes using $\mathbb{Z}$-relations, but the semantics provided is of a classic Datalog computing on sets. We can expand the definition of some DDlog operators to compute on bags/multisets as well (this can be done naturally for all operators except recursion).

8.5 Window aggregates

Streaming databases often organize the contents of streams into windows, which store a subset of data points with a predefined range of timestamps. The circuit below (a convolution filter in DSP) computes a *fixed-size sliding-window aggregate* over the last four timestamps defined by the T_i functions.



In practice, windowing is usually based on physical timestamps attached to stream values rather than logical time. For instance, the CQL [7] query "`SELECT * FROM events [RANGE 1 hour]`" returns all events received within the last hour. The corresponding circuit (on the left) takes input stream $s \in \mathcal{S}_{\mathbb{Z}[A]}$ and an additional input $\theta \in \mathcal{S}_{\mathbb{R}}$ that carries the value of the current time.



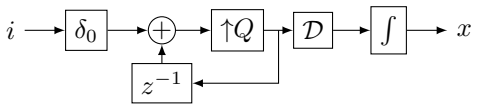
where the *window operator* W prunes input $\mathbb{Z}$-sets, only keeping values with timestamps less than an hour behind $\theta[t]$. Assuming $ts : A \to \mathbb{R}$ returns the physical timestamp of a value, W is defined as $W(v, \theta)[t] \stackrel{\text{def}}{=} \{x \in v[t].ts(x) \geq \theta[t] - 1hr\}$. Assuming θ increases monotonically, W can be moved inside integration, resulting in the circuit on the right, which uses bounded memory to compute a window of an unbounded stream. This circuit is a building block of a large family of window queries, including window joins and aggregation. We conjecture that DBSP can express any CQL query.

8.6 Relational while queries

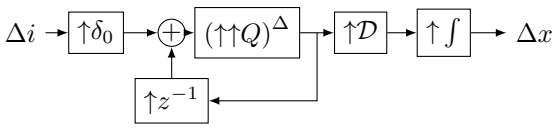
(See also non-monotonic semantics for Datalog$^\neg$ and Datalog$^{\neg\neg}$[3].) To illustrate the power of DBSP we implement the following "while" program, where Q is an arbitrary relational algebra query:

```
x := i;
while (x changes)
    x := Q(x);
```

The DBSP implementation of this program is:



This circuit can be converted to a streaming circuit that computes a stream of values i by lifting it; it can be incrementalized using Algorithm 6.13 to compute on changes of i:



Part III
Implementation

9 Well-formed circuits

In this section we formalize the shape legal computations allowed in our framework. We model computations as circuits. A circuit is a directed graph where each vertex is a primitive computation node (a function) and each edge has a type; at circuit evaluation time each edge will represent one value of that type.

We provide a recursive set of circuit construction rules. We call a circuit **well-formed** (**WFC**) if it can be constructed by a sequence of applications of these rules. For each WFC construction rule of a circuit C from simpler parts we also provide typing derivation rules and a denotational semantics for $[\![C]\!]$ that reduces the meaning of C to its components. The semantics of a circuit expresses each circuit output as a function of the circuit inputs.

9.1 Primitive nodes

We assume a set of base types that represent abelian groups: $A_1, A_2, \ldots,$ $B_1, B_2, \ldots$. (The base types do *not* include stream types; stream types are derived.)

We are given a fixed set of **primitive computation nodes P**; each primitive node has an input arity k. Iin our applications we only use unary and binary primitive nodes, so we will restrict ourselves to such nodes, but these constructions can be generalized to nodes with any arity. A binary node $n \in \mathbf{P}$ has a type of the form $n : A_0 \times A_1 \to A$, where all As are base types. We assume the existence of a total function "meaning" $[\![\cdot]\!]$ that gives the semantics of each node: $[\![n]\!] : [\![A_0]\!] \times [\![A_1]\!] \to [\![A]\!]$.

In Section 3.1 we have seen many generic primitive nodes: the identity function $id : A \to A$, scalar function nodes (where all inputs are base types A_i), sum $\bigoplus : A \times A \to A$, negation $- : A \to A$, pairs $\langle \cdot, \cdot \rangle : A \times B \to \langle A, B \rangle$, fst : $A \times B \to A$ and snd : $A \times B \to B$. Their typing and semantics is standard. We will introduce more domain-specific primitive nodes when discussing specific applications, in Section 11.

9.2 Circuits as graphs

A circuit is a 5-tuple $C = (I, O, V, E, M)$.

- I is an ordered list of input ports. An input port indicates a value produced by the environment. We use letters like i, j for input ports. Each input port has a type $i : A$ for some abelian group A (where A can be a stream type).

- O is an ordered list of output ports. An output port represents a value produced by the circuit. as a function of the values of the input ports. We use letters like o, l for output ports. Each output port has a type $o : A$ for some abelian group A (where A can be a stream type).
- V is a set of internal vertices.
- E is a set of edges; $E \subseteq (V \times V) \cup (I \times V) \cup (V \times O)$. We call an edge of the form (i, v) for $i \in I, v \in V$ an **input edge**, and an edge of the form (v, o) for $v \in V, o \in O$ an **output edge**.
- Each internal vertex v is associated with a primitive computation node or with another circuit; for primitive nodes the **implementation function** $M : V \to \mathbf{P}$ gives the primitive computation associated with a vertex.

Given a circuit C we use the suffix notation to indicate its various components, e.g., $C.O$ is the list of output edges, and $C.V$ is the set of vertexes.

9.3 Circuit semantics

Given a circuit C with k input ports $C.I = (i_j, j \in [k])$ with types $i_j : A_j, j \in [k]$, and m output ports $C.O = (o_j, j \in [m])$ with types $o_j : B_j, j \in [m]$, the semantics of the circuit is given by the semantics of all its output ports: $[\![C]\!] = \prod_{j \in [m]} [\![C.o_j]\!]$. The semantics of output port $C.o_j$ is a total function $[\![C.o_j]\!] : [\![A_0]\!] \times \ldots [\![A_{k-1}]\!] \to [\![B_j]\!]$.

9.4 Circuit construction rules

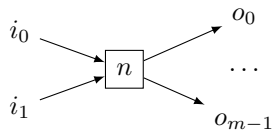
We give a set of inductive rules for constructing WFCs. In the inductive definitions we always combine or WFCs to produce a new WFC.

These rules maintain the following invariants about all constructed circuits, which can proved by structural induction:

- All input and outputs are either all scalars or they are all streams of the same "depth".
- All circuits are time-invariant.
- For circuits operating over streams, all circuit outputs are causal in all of the circuit inputs.

9.4.1 Single node

Given a primitive (unary or binary) node n with type $n : A_0 \times A_1 \to B$, (where each A_i, B is a base type), we can construct a circuit $C(n)$ with a single vertex. The circuit has exactly 2 input ports and any number m of output ports having all the same type B; the output ports will carry all the same value.



Formally:

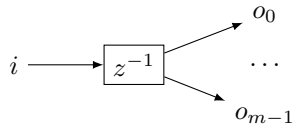
- $C(n).V = \{v\}$.
- $C(n).I = (i_j, j \in [k])$
- $C(n).O = (o_j, j \in [m])$
- $C(n).E = \{(i_j, v), j \in [k]\} \cup \{(v, o_j), j \in [m]\}$
- $C(n).M(v) = n$.

$$\frac{i_0 : A_0, \dots, i_{k-1} : A_{k-1}, n : A_0 \times A_1 \times \dots A_{k-1} \to B}{\forall j \in [m].n.o_j : B}$$

All output edges of the circuit produce the same value. $[\![C(n).o_j]\!] : [\![A_0]\!] \times \dots [\![A_{k_1}]\!] \to [\![A]\!]$ given by $[\![C(n).o_j]\!] = [\![n]\!]$.

9.4.2 Delay node

A similar circuit construction rule is applicable to the z^{-1} node, with the difference that z^{-1} operates on streams. Given a type A we can construct the circuit Cz with a single vertex. The circuit has 1 input port of type $i : \mathcal{S}_A$ and any number m of output ports with the same type $\mathcal{S}_A$.



Formally:

- $Cz.V = \{v\}$.
- $Cz.I = (i)$.
- $Cz.O = (o_j, j \in [m])$.
- $Cz.E = \{(i, v)\} \cup \{(v, o_j), j \in [m]\}$.
- $Cz.M(v) = z^{-1} : \mathcal{S}_A \to \mathcal{S}_A$

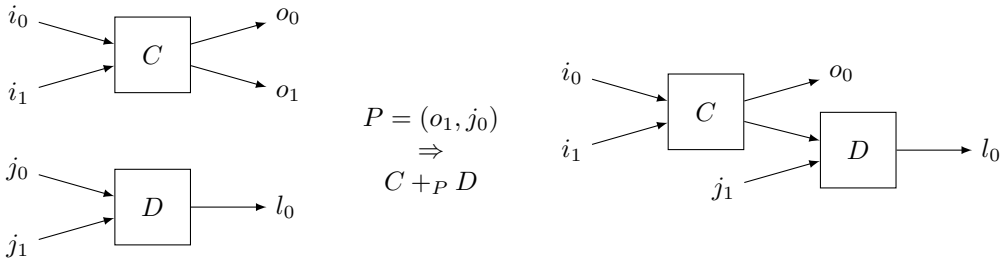
$$\frac{i : \mathcal{S}_A}{Cz.o_j : \mathcal{S}_A}$$

All output edges of the circuit produce the same value. $[\![Cz.o_j]\!] : [\![\mathcal{S}_A]\!] \to [\![\mathcal{S}_A]\!]$ given by $[\![Cz.o_j]\!] = [\![z^{-1}]\!]$.

9.4.3 Sequential composition

Given two WFCs C and D with inputs (and necessarily outputs as well) in the same clock domain, their sequential composition is specified by a set of pairs of ports; each pair has an output port of C and an input port of D: $P = \{(o_j, i_j), j \in [n]\}$, where $o_j \in C.O$ and $i_j \in D.I$ with respectively matching types. The sequential composition $C +_P D$ is a new circuit where each output port of C is "connected" with the corresponding input port of D from P.

To simplify the definition, we can assume WLOG that each of C has exactly 2 inputs and outputs and D has 2 inputs and 1 output, and also that the second output of C is connected to first input of D (i.e., P contains at most one pair or ports). Circuits and connections with more inputs and outputs can be built by "bundling" multiple edges using the pair operator $\langle \cdot, \cdot \rangle$. Sequential composition is given by the following diagram:



Formally the definition is given by:

- $(C +_P D).I = C.I \cup D.I \setminus \{i \mid \exists o.(o, i) \in P\}$.
- $(C +_P D).O = C.O \setminus \{o \mid \exists i.(o, i) \in P\} \cup D.O$.
- $(C +_P D).V = C.V \cup D.V$.
- $(C +_P D).E = C.E \cup D.E \setminus \{(v, o) \mid \exists i.(o, i) \in P, v \in C.V\}$ $\setminus \{(i, v) \mid \exists i.(o, i) \in P, v \in D.V\} \cup \{(u, v) \mid \exists (o, i) \in P, (u, o) \in C.E, (i, v) \in D.E\}$.
- $(C +_P D).M = \{v \mapsto C.M(v) \mid v \in C.V\} \cup \{v \mapsto D.M(v) \mid v \in D.V\}$.

$$\frac{\begin{array}{c} C : A_0 \times A_1 \to B_0 \times B_1 \\ D : B_1 \times A_2 \to B_2 \\ P = \{(C.O_0, D.I_0)\} \end{array}}{D +_P C : A_0 \times A_1 \times A_2 \to B_0 \times B_2}$$

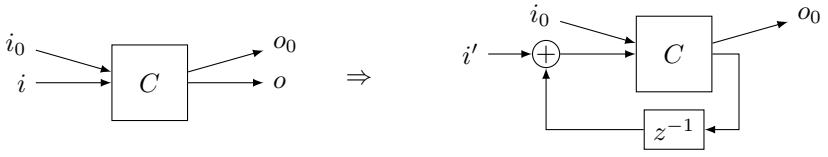
$$[\![(C +_P D).O_0]\!] = [\![C.O_0]\!]$$
$$[\![(C +_P D).O_1]\!] = \lambda i_0, i_1, j_1.[\![D]\!]([\![C]\!](i_0, i_1), j_1)$$

This transformation combines acyclic graphs into an acyclic graph.

MIHAI: We may also need a parallel composition

9.4.4 Adding a back-edge

We can assume WLOG that we are given a circuit C with two inputs and two outputs. Assume that C's output port $o \in C.O$ has a stream type $o : \mathcal{S}_A$ and the input port $i \in C.I$ has the same type $i : \mathcal{S}_A$. We can create a new circuit $C \overset{\frown}{io}$ by adding two nodes: one z node implemented by z^{-1} and one p node implemented by $+_{\mathcal{S}_A}$, and a new input port $i' : \mathcal{S}_A$, connected as in the following diagram:



Formally the definition is given by:

- $(C \overset{\frown}{io}).I = C.I \setminus \{i\} \cup \{i'\}$.
- $(C \overset{\frown}{io}).O = C.O \setminus \{o\}$.
- $(C \overset{\frown}{io}).V = C.V \cup \{z, p\}$.
- $(C \overset{\frown}{io}).E = C.E \cup \{(i', p)\} \setminus \{(u, o) \in C.E \mid u \in C.V\} \cup \{(p, v) \mid \exists i.(i, v) \in C.E\} \cup \{(u, p) \mid \exists (u, o) \in C.E\}$.
- $(C \overset{\frown}{io}).M = C.M \cup \{p \mapsto +_{\mathcal{S}_A}, z \mapsto z^{-1}\}$.

We call the edge connecting z^{-1} to $\bigoplus$ a *back-edge*. One can prove by induction on the structure of C that the graph of C ignoring all back-edges is acyclic.

$$\frac{C : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_A \times \mathcal{S}_C}{C \overset{\frown}{io} : \mathcal{S}_A \times \mathcal{S}_B \to \mathcal{S}_C}$$

Since the source of a back-edge is always z^{-1}, strict operator, and any circuit is causal, the composition has a well-defined semantics, according to Corollary 3.17.

$$[\![C \overset{\frown}{io}]\!] = \lambda i_0, i'.\mathrm{fix}\, i.[\![C.O_0]\!](i_0, i' + [\![z^{-1}]\!](i)).$$

9.4.5 Lifting a circuit

Given a circuit C with scalar inputs and outputs, we can lift the entire circuit to operate on streams. As before, we can assume WLOG that C has a single input and output. If the circuit is a function: $C : A \to B$, the lifted circuit $\uparrow C$ operates time-wise on streams: $\uparrow C : \mathcal{S}_A \to \mathcal{S}_B$.

VAL: According to the model, the two outputs can produce different streams. Algebraically, this means the circuit C is implementing two operators, one for each output, say $T(i_0, i)$ for output o and $T_0(i_0, i)$ for output o_0.

MIHAI: Yes, that is correct. In fact, this is the main difference between circuits and operators: a circuit can have many different outputs, while an operator always has exactly 1.

VAL: For notation T, T_0 please see my comment above. Which one of T or T_0 is $[\![C.O_0]\!]$ in this definition? I think you intend it to be T_0 in order to produce the right output. But then, the well-definedness of the semantics of this construction does not follow from Corollary 3.17 because that result uses T rather than T_0. I am working on the more general result that we need to justify this case.

Formally the definition is given by:

- $(\uparrow C).I = C.I$.
- $(\uparrow C).O = C.O$.
- $(\uparrow C).V = C.V$.
- $(\uparrow C).E = C.E$.
- $(\uparrow C).M = \{\uparrow(C.M(v)) \mid v \in C.V\}$.

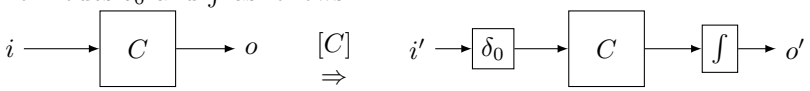
$$\frac{C : A \to B}{\uparrow C : \mathcal{S}_A \to \mathcal{S}_B}$$

If C is a WFC, then $[\![\uparrow C]\!] = \lambda s.[\![C]\!] \circ s$, for $s : \mathbb{N} \to B = \mathcal{S}_B$, as described in Section 3.1.

9.4.6 Bracketing

This construction uses nodes $\delta_0 : A \to \mathcal{S}_A$ and $\int : \mathcal{S}_A \to A$ which "create" and "eliminate" streams, as defined in Section 5.1. These nodes are always used in pairs.

Given a WFC C computing on streams with a single input $i : \mathcal{S}_A$ and a single output of the exact same type $o : \mathcal{S}_A$, we can "bracket" this circuit with a pair of nodes δ_0 and $\int$ as follows:



The types of the resulting input and given by $i' : A$, and $o' : A$.

It is very important for C to have a single input and output; this prevents connections to C from going "around" the bracketing nodes. If multiple input or outputs are needed to C, they can be "bundled" into a single one using a pairing operator $\langle \cdot, \cdot \rangle$.

Formally the definition is given by:

- $[C].I = \{i'\}$.
- $[C].O = \{o'\}$.
- $[C].V = C.V \cup \{d, s\}$.
- $[C].E = C.E \cup \{(i', d), (s, o')\} \setminus \{(i, v) \mid v \in V\} \setminus \{(v, o) \mid v \in V\} \cup \{(d, v) \mid (i, v) \in C.E\} \cup \{(v, s) \mid (v, o) \in C.E\}$.
- $[C].M = C.M \cup \{d \mapsto \delta_0, s \mapsto \int\}$.

$$\frac{C : \mathcal{S}_A \to \mathcal{S}_B}{[C] : A \to B}$$

The semantics of the resulting circuit is just the composition of the three functions: $[\![[C]]\!] = [\![\int]\!] \circ [\![C]\!] \circ [\![\delta_0]\!]$. (However, note that the semantics of $\int$ is only defined for streams that are zero almost everywhere.)

Theorem 9.1. For any WFCs all inputs and outputs are streams of the same "depth".

Proof. The proof proceeds by induction on the structure of the circuit. All construction rules maintain this invariant, assuming that it is true for all primitive nodes. □

10 Implementing WFC as Dataflow Machines

In this section we give a compilation scheme that translates a WFC into a set of cooperating state machines that implement the WFC behavior. Each primitive node is translated into a state machine, each circuit is translated into a control element, and each edge is translated into a communication channel between two state machines, storing at most one value at any one time.

MIHAI: I realize that this would be much simpler if we force the toplevel circuit to have exactly 1 input and output edge. I will work on that.

There are essentially 6 kinds of nodes in our circuits:

- Lifted scalar nodes.
- Delay nodes z^{-1} operating on streams.
- Delay nodes z^{-1} operating on nested streams.
- "Loop entry" nodes, corresponding to δ_0.
- "Loop exit" nodes, corresponding to $\int$.
- Controller nodes, corresponding to circuits.

There are 4 types of events in our implementation:

Reset events: cause a circuit to be initialized. The main effect is to cause z^{-1} nodes to initialize their internal state to 0. The reset events have no effect on any other node.

Latch events: these events cause z^{-1} nodes to emit their internal state as an output. The latch events have no effect on any other node.

Data events: these events signal to a node or circuit that data is present on one of the input channels.

Repeat events: signal that a loop has to perform one more iteration. Only sent between $\int$ and corresponding δ_0 nodes.

Here are the state machines of each of these nodes:

MIHAI: This is pseudocode, but it would probably look better as real code.

Environment state machine The environment feeds data to the input edges of a top-level circuit and retrieves results from the output edges. The environment is expected to operate in epochs, executing the following infinite loop:

- Send a reset event to circuit
- Repeat forever
  - Assign data to all input edges.
  - Wait for all output edges to receive a "data" event.
  - Collect results from all output edges.

Circuit state machine

1. On receipt of a reset event:
   - Send a reset event to all nodes in the circuit.
   - Send a latch event to all nodes in the circuit.
2. On receipt of data on an input edge send a data event to the destinations connected to the input edge. The environment of the circuit should send

Primitive node state machine

1. On receipt of a "data" event check if all inputs have received data. If they have, compute the output, and send it as a "data" event on all output wires.

z^{-1} **node state machine** Stores a value in the internal state.

1. On receipt of "reset" event set internal state to 0.
2. On receipt of "latch" event, send a data event on the output channel with the value that is already present there.
3. On receipt of "data" event, copy internal state to output channel and input to internal state.

z^{-1} **node operating on a nested stream state machine** This node internally stores a potentially unbounded *list* of values as internal state. It also maintains a counter "time" to index within this list.

1. On receipt of a "reset" even set time to 0.
2. On receipt of a "latch" event, send the value in list[time] as a "data" event to the output channel.
3. On receipt of a "data" event

- Increment "time"
- Set the output channel to the list[time] value (zero if the list is not long enough, and grow the list)
- Store data value received in list[time].

Loop entry node state machine

1. On receipt of a "data" event
   - Send a "reset" event to circuit that is connected as output
   - Send a "data" event to the circuit connected as output with the data received
2. On receipt of a "repeat" event send a "data" event with value 0 to the circuit connected as output.

Loop exit node state machine Maintain an internal accumulator.

1. On receipt of a "reset" event set the accumulator to 0.
2. On receipt of a "data" event:
   - if the data value is 0, send a "data" event to the output channel with the current value of the accumulator.
   - otherwise add the input value to the accumulator and send a "repeat" event to the corresponding $delta_0$ node.

11 Semantics of Differential Datalog

This section gives a semantics of DDlog in terms of DBSP circuits. This is a precise specification of the semantics of DDlog.

11.1 Differential Datalog syntax and semantics

We describe the syntax and semantics of a dialect of Datalog called Differential Datalog, or DDlog. We start by ignoring the differential aspects, we will return to these in Section 11.3. In defining the syntax and semantics of Datalog we mostly follow standard definitions, e.g., [3]. In this section we give a syntax-directed translation of Datalog programs into circuits. We model a *core* of the language (ignoring constructs that can be viewed as syntactic sugar), to simplify the description. We argue informally that the resulting circuits implement the standard semantics of Datalog.

Our Datalog is strongly-typed, supports stratified negation and recursion, and is enhanced with additional operators, such as grouping (which we will describe below). Figure 2 shows the EBNF-like grammar of the core DDlog language (omitting types and expressions).

```
1  DatalogProgram := TypeDeclaration*
2                    RelationDeclaration*
3                    Rule*
4  TypeDeclaration := ...
5  Type := ...
6  Expression := ...
7  Id := ... // identifiers
8  RelationDeclaration := ("input"|"output")? "relation"
9                         Id "(" ColumnTypes? ")"
10 ColumnTypes := ColumnType ( "," ColumnType)*
11 ColumnType := Id ":" Type
12 Rule := Head ":-" Body
13 Head := RelationTerm
14 RelationTerm := Id "(" Columns? ")"
15 Columns := Variable ( "," Variable )*
16 Body := RelationTerm ( "," Term )?
17 Term := RelationTerm
18      | Predicate
19      | NegatedTerm
20      | VariableDefinitionTerm
21      | FlatmapTerm
22      | GroupByTerm
23 NegatedTerm := "not" RelationTerm
24 VariableDefinitionTerm := "var" Variable "=" Expression
25 FlatmapTerm := "var" Variable "=" "Flatmap" "(" Id ")"
26 GroupByTerm := "var" Variable "="
27      Expression "." "group_by" "(" VariableList? ")"
28 Variable := Id
29 VariableList := Id (, Id)*
30 Predicate := Expression
```

Figure 2: Datalog rules grammar.

Types Assume that we are given a set of basic types, including `integer`, Booleans, `string`, but also structures (product types), unions (sum types), tuples, vectors, and sets. Each such type must support an operation to compare values for equality. This is the only requirement to store values of a particular type in a set.

We allow arbitrary computations over the base types (e.g., arithmetic) through the use of built-in functions (e.g., addition, subtraction, equality comparison, etc.). A standard language of expressions can be used to combine built-in functions into more complex functions. We require all such functions to be total and deterministic. We treat such computations as uninterpreted functions (black boxes) and no longer concern ourselves with them in this document.

All DDlog programs must be strongly typed, but we don't specify the typing rules in this document (e.g., predicates must produce Boolean results). The typing rules are standard. Only the semantics of well-typed programs is defined.

Relations Datalog programs compute over relations. The inputs and outputs of a Datalog program are relations. The standard Datalog semantics of a relation is a *set* of values from some domain.

DDlog programs continuously interact with their environment. Thus they distinguish relations by their roles. Some relations are `input` relations; their contents is supplied by the environment. Some relations are declared as `output` relations. The contents of these relations is visible to external observers.

A DDlog program must have a declaration for each relation, specifying the type of its elements, as in this example:

```
input relation People(name: string, age: integer)
relation Ages(age: integer)
output relation Names(name: string)
```

This declares three relations. The first relation is an `input` relation, named `People`, and it has 2 columns, `name` of type `string`, and `age` of type `integer`. Its elements are 2-tuples of type (`string`, `integer`).

The value of relation `People` is a *set* of such tuples. As a running example, let us assume that the input relation `People` has the value (supplied by the program's environment) $\{(\texttt{bob}, 10), (\texttt{john}, 20), (\texttt{amy}, 10)\}$, containing three tuples.

The relation `Names` has a single column, and it is an `output` relation. This means that the environment can observe it's contents. The relation `Ages` is neither `input` nor `output`. The Datalog program must contain *rules* that show how `Ages` and `Names` are computed from the `input` relations, i.e., `People`.

Rules Besides type and relation declarations, the most important part of a Datalog program is a set of rules. A **Datalog rule** defines the contents of a relation as a function of other relations. A rule has a **head** and a **body**, as shown in the grammar from Figure 2. In the following rule:

```
Names(n) :- People(n, a).
```

the head is to the left of the turnstyle symbol `:-`, and it is always a relation with variables standing for the tuple fields. This rule defines how `Names` is computed from the contents of `People`. In this example the rule's head is `Names(n)`. You may guess that the value of `Names` is the set $\{\texttt{bob}, \texttt{john}, \texttt{amy}\}$. We will explain how this result is computed. The turnstyle symbol can be read as "if". `n` is a variable of type `string`, standing for the column `name` (the type of `n` is `string` because it stands positionally for the declared column `name` with type `string` of relation `Names`). The values that `n` may take are defined by the body of the rule — each variable in the head must appear in the body of the rule.

The body of a rule consists of one or two **terms** separated by commas; a comma is read as an "and", and such a rule is form of a *conjunctive query*. The valuation computed by the body is defined recursively on the list of terms in the body. Datalog allows an arbitrary number of terms in a rule body, but our grammar allows only two. We argue in the paragraph below that this does not reduce the power of the language, but it simplifies the description.

The grammar of terms is shown in line 16 in Figure 2. We will discuss the semantics of the various terms and their implementation as circuits in the rest of this section.

Valuations The body of the above rule is `People(n, a)`. The body defines two variables, `n` and `a`. The semantics of a rule body is a **valuation**: a set of values that the variables defined by the body may *jointly* take. Our example body defines a valuation for the tuple of variables (`n`, `a`). Since the body is `People(n,a)`, the valuation defines the set of values for (`n`, `a`) to be the contents of the relation `People`, that is $(\texttt{n}, \texttt{a}) \in \{(\texttt{bob}, 10), (\texttt{john}, 20), (\texttt{amy}, 10)\}$.

We can assume without loss of generality that every rule body has at most two terms. A rule with n terms can be decomposed into n - 1 rules with 2 terms each by introducing a temporary relation that stores the entire valuation. For example, consider the following rule:

```
input relation Lives(name:string, country:string)
output relation USAges(age: integer)
USAges(a) :- People(n, a), Lives(n, c), c == "USA".
```

The rule prefix `People(n, a), Lives(n, c)` defines a valuation for variables `n, a, c`. By introducing a temporary relation `Temp(n, a, c)` we can rewrite this rule as two rules, producing the same result for the visible relatin `USAges`:

```
relation Temp(name:string, age:integer, country:string)
Temp(n, a, c) :- People(n, a), Lives(n, c).
USAges(a) :- Temp(n, a, c), country == "USA".
```

11.2 Compiling Datalog programs to circuits

A Datalog program is a set of rules computing over valuations and relations. Both relations and valuations are represented as $\mathbb{Z}$-sets in a translation to circuits, by representing each set by a $\mathbb{Z}$-set with weights 1. (Subsequent optimizations can relax this requirement for internal relations as long as the semantics of output relations is preserved, since only `output` relations are observable from the environment. We expand on this in Section **??**.)

11.2.1 Rule compilation

Each rule is compiled to a circuit with the following properties:

- Each relation and valuation is represented by a $\mathbb{Z}$-set.
- The head of the rule is the output edge of the circuit;
- The relations that appear in the body are inputs of the circuit;
- The circuit structure is defined by the terms that appear in the relation body (as discussed in the rest of this section);

- The turnstyle is compiled into a projection operator (described in Section 11.2.4);
- The *valuation* at a conjunction in the rule body is translated into an edge in the circuit, carrying $\mathbb{Z}$-set values.

This translation is illustrated in Figure 3.

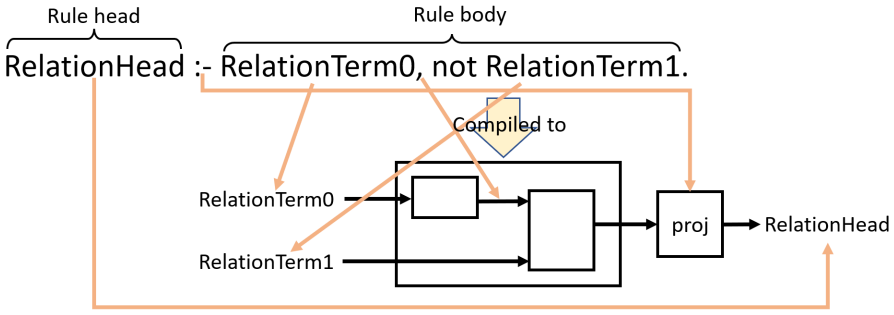


Figure 3: Compilation of a DDlog rule into a circuit.

The compilation of a program containing a set of rules produces a "toplevel" circuit, composed of the circuits for the rules interconnected with each other (as described in Section 11.2.2). For the toplevel circuit the `input` relations correspond to the input edges. (`input` relations cannot appear in the head of any rule). Similarly, the `output` relations will correspond to output edges of the toplevel circuit, (each `output` relation must appear in some rule head).

11.2.2 Relation terms in rule bodies

A `RelationTerm` is a term in a rule body containing a relation with variables substituted for the columns: `People(n, a)` is such an example. This term *defines a valuation* for all variables that appear in the columns; the valuation associates the variables with the contents of the relation itself. In our example the term `People(n, a)` defines the following valuation: $(\texttt{n}, \texttt{a}) \in \{(\texttt{bob}, 10), (\texttt{john}, 20), (\texttt{amy}, 10)\}$. Each Datalog relation is represented by an edge in a circuit carrying a $\mathbb{Z}$-set.

As Figure 3 shows, a relation in the head of a rule is compiled into the output edge of the circuit corresponding to the rule, and that a `RelationTerm` in the body of a rule is compiled into an input edge of the circuit corresponding to the rule.

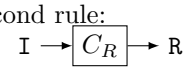
A relation that appears in the body of a rule and in the head of another rule is compiled into an edge connecting the circuits representing the two rules. This is in fact just a form of function composition.

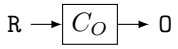
(This rule does not apply directly for recursive rules; the translation for recursive or mutually recursive rules (which define a relation in terms of itself), is described in Section 7.1.1).

For example, for the following Datalog program structure, where `R` is used within a body and within a separate head:

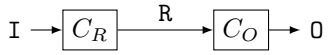
```
R(y) :- I(x), ....
O(y) :- ..., R(y).
```

and, given circuits C_R implementing the first rule and C_O implementing the second rule:





the translation of the program with both rules is:



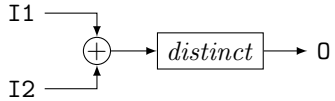
This construction is repeated for all rules, translating a program with n rules into n circuits connected to each other. If the rules are not recursive the resulting circuit is acyclic.

11.2.3 Repeated rule heads (set union)

The same relation may appear in the head of multiple rules. In this case the contents of the head relation is the *set union* of the values assigned by all heads. Consider the following example, where `I1` and `I2` are rule bodies of arbitrary complexity providing a valuation for variable `v`:

```
O(v) :- I1(v).
O(v) :- I2(v).
```

The following circuit implements the Datalog program with both rules:



Given two $\mathbb{Z}$-sets $a \in \mathbb{Z}[I]$ and $b \in \mathbb{Z}[I]$ which are sets (i.e., isset(a) and isset(b)), their *set union* can be computed as: $\cup : \mathbb{Z}[I] \times \mathbb{Z}[I] \to \mathbb{Z}[I]$. $a \cup b \stackrel{\text{def}}{=} \mathit{distinct}(a +_{\mathbb{Z}[I]} b)$. The *distinct* application is necessary to provide the set semantics of Datalog. We have isset(a)$\wedge$isset(b) $\Rightarrow$ isset($a \cup b$) and ispositive(a)$\wedge$ ispositive(b) $\Rightarrow$ ispositive($a \cup b$).

Consider a concrete example for the above program where the value of `I1(v)` is `v` $\in \{$`bob` $\mapsto 1,$ `mike` $\mapsto 1\}$ and the value of `I2(v)` is `v` $\in \{$`bob` $\mapsto 1,$ `john` $\mapsto 1\}$. In terms of $\mathbb{Z}$-sets we are performing the following addition:

v	**W**
bob	1
mike	1

$+$

v	**W**
bob	1
john	1

$=$

v	**W**
bob	2
mike	1
john	1

It is apparent why the *distinct* operator is needed.

11.2.4 Projection

Given a valuation produced by the body of a rule, the head of the rule defines the contents of a relation as *the projection* of the valuation on the variables used in the head. For our example rule `Names(n) :- People(n, a)`, the body defines a valuation for $(\texttt{n}, \texttt{a})$, but the head uses only `n`. The projection of the valuation $(\texttt{n}, \texttt{a}) \in \{(\texttt{bob}, 10), (\texttt{john}, 20), (\texttt{amy}, 10)\}$ on the variable `n` is the valuation $\texttt{n} \in \{\texttt{bob}, \texttt{john}, \texttt{amy}\}$. This defines the contents of the relation in the head: $\texttt{Names} = \{\texttt{bob}, \texttt{john}, \texttt{amy}\}$.

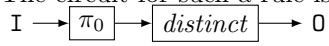
Thus, in Datalog projection is used when some of the bound variables in the body of a rule are not used in the head. We can assume without loss of generality that a single variable is removed in a projection (by bundling multiple variables in a single tuple-valued variable). Let us consider the following example, where `I` stands for a rule body producing a valuation for `(v, v1)`.

```
O(v) :- I(v, v1).
```

Here the type of the implementation of `I` is $\mathbb{Z}[A_0 \times A_1]$ (a $\mathbb{Z}$-set of tuples with two elements), while the type of the implementation of `O` is $\mathbb{Z}[A_0]$. In terms of $\mathbb{Z}$-sets, the projection of a $\mathbb{Z}$-set i on A_0 is defined as: $\pi_0(i)[t] = \sum_{x \in i, x|_0 = t} i[x]$, where $x|_0$ is first component of the tuple x. The multiplicity of a tuple in the result is the sum of the multiplicities of all tuples that project to it.

As a concrete example of projection, consider the $\mathbb{Z}$-set corresponding to the `People` relation and it's projection on the `Age` column. The projection is $\pi_{Age}(\texttt{People}) = \{10 \mapsto 1 + 1, 20 \mapsto 1\}$. Notice that in the projection the weight of 10 is the sum of all weight of the tuples that have age 10, i.e., 2.

The circuit for such a rule is:



Note that $\uparrow\pi$ is time-invariant, since π has the zero-preservation property. We have $\text{isset}(i) \Rightarrow \text{isset}(\pi_A(i))$ and $\text{ispositive}(\pi_0)$.

11.2.5 Flatmap in DDlog

Recall that in DDlog the type of a column in a relation can be any of the types supported by the language, including complex types, such as vectors, sets, or even maps. For example, the following declaration indicates that the values in relation `I` are sets of integers.

```
relation I(set: Set<integer>)
```

Flatmap is an operator that can expand the data in such a collection value stored in a relation into the contents of a relation. Classic Datalog does not support flatmaps. In DDlog `Flatmap` is an explicit keyword. It appears in rules in the form of a `FlatmapTerm` in the grammar in Figure 2. The DDlog type system ensures that the `Flatmap` operator can only be applied to an expression whose type is a collection.

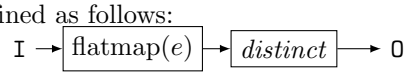
Here is an example of a program using `Flatmap`:

```
relation I(set: Set<integer>)
relation O(integer)
O(v) :- I(set), var v = Flatmap(set).
// O = union of all sets in I
```

Each element in relation `I` is a set of integers. The DDlog `Flatmap` operator implements a restricted form of the functionality of the general mathematical operator from above (unlike the mathematical flatmap, which is parameterized by a function f, the DDlog `Flatmap` uses a hardwired function, essentially the identity function.). The semantics is as follows: the `Flatmap` rule body term extends the existing valuation with a new variable, `v` in this example. `Flatmap`'s argument is an expression that depends on the current valuation (`set` in this example) whose value is a collection. Let us assume that the contents of the `I` relation is: $\{\{1,2\},\{2,3\}\}$.

The valuation produced by the rule `I(set), var v = Flatmap(set)` is the following: $(\texttt{set}, \texttt{v}) \in \{(\{1,2\},1),(\{1,2\},2),(\{2,3\},2),(\{2,3\},3)\}$.

The circuit-based implementation of `Flatmap`, operating on $\mathbb{Z}$-sets, can be defined as follows:



where the function e extends each tuple in a $\mathbb{Z}$-set with the newly introduced variable and each set value with a Cartesian product between the collection and all it's elements. For our example: e : `Set<integer>` $\rightarrow \mathbb{Z}$[`Set<integer>` $\times$ `integer`] defined by: $e(\texttt{set}) = \sum_{x \in \texttt{set}}(\texttt{set}, x) \mapsto 1$.

The `distinct` operator is needed because some collections (e.g., vectors) may contain duplicate values.

Proposition 11.1. ispositive(`Flatmap`).

11.2.6 Map in DDlog

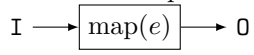
Given a function $f : A \rightarrow B$, the mathematical **map** operator "lifts" the function f to operate on $\mathbb{Z}$-sets: $\text{map}(f) : \mathbb{Z}[A] \rightarrow \mathbb{Z}[B]$. map can be defined in terms of flatmap: $\text{map}(f) \stackrel{\text{def}}{=} \text{flatmap}(x \mapsto 1 \cdot f(x))$.

Classic Datalog does not support map computations, but many practical implementations do. DDlog programs perform map computations when using `VariableDefinitionTerm` in a rule, by using an expression to computing a value for a new variable, that is added to a valuation, as in the following example:

```
O(v) :- I(x), var v = x + 1.
```

The `VariableDefinitionTerm`, `var v = x + 1`, extends the current valuation, which contains just `x`, to include the newly defined variable `v`.

The circuit implementation of the previous rule is:



The function e extends the current valuation tuple with a new column (corresponding to `v` in the example) and evaluates the expression in the term (`x + 1`)

for each row of the valuation to compute the corresponding value for the new column. In our example, $e : \mathbb{Z}[\texttt{integer}] \to \mathbb{Z}[\texttt{integer} \times \texttt{integer}]$, $e(x) = (x, x+1)$.

Note that ispositive(map(f)) for any function f. From the linearity of flatmap it follows that map is linear as well. Moreover, the operator $\uparrow$map(f) is time-invariant for any f.

11.2.7 Filtering

Filtering occurs in Datalog whenever a `TermPredicate` appears in the body of a rule, in the guise of a Boolean expression, as in the following example:

```
relation Minors(n: string, a: integer)
Minors(n, a) :- People(n, a), a < 18.
```

(A predicate may not appear in the first position in the body of a rule.) The predicate must only use variables in the current valuation. The produced valuation contains the same variables as the source valuation. The value of the valuation the set of tuples in the source valuation that satisfy the predicate.

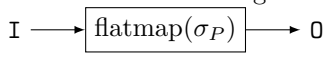
Recall that the valuation of the term `People(n, a)` is $(\texttt{n}, \texttt{a}) \in \{(\texttt{bob}, 10), (\texttt{john}, 20), (\texttt{amy}, 10)\}$. The valuation of the entire rule is the set of tuples $(\texttt{n}, \texttt{a})$ for which the predicate $\texttt{a} < 18$ holds. That valuation is $(\texttt{n}, \texttt{a}) \in \{(\texttt{bob}, 10), (\texttt{amy}, 10)\}$. Thus the contents of relation `Minors` is $\{(\texttt{bob}, 10), (\texttt{amy}, 10)\}$.

To compute on $\mathbb{Z}$-sets, let us assume that we are filtering with a predicate $P : A \to \mathbb{B}$. We define the following function $\sigma_P : A \to \mathbb{Z}[A]$ as:

$$\sigma_P(x) = \begin{cases} 1 \cdot x & \text{if } P(x) \\ 0 & \text{otherwise} \end{cases}$$

The filter of a $\mathbb{Z}$-set is defined as $\text{filter}_P : \mathbb{Z}[A] \to \mathbb{Z}[A]$ by $\text{filter}_P \stackrel{\text{def}}{=} \text{flatmap}(\sigma_P)$. We have $\text{isset}(i) \Rightarrow \text{isset}(\text{filter}_P(i))$ and $\text{ispositive}(\text{filter}_P)$. Thus a *distinct* is not needed. As a consequence of the linearity of flatmap, we have that filtering is also linear. The lifted version of filtering is also time-invariant.

The circuit for filtering with a predicate P can be implemented as:



11.2.8 Grouping

Classic Datalog does not support grouping. In DDlog grouping is the fundamental operator used for aggregation. Grouping is applied to a set and produces a partition of that set into a set of collections. The type of a partition is a built-in type in DDlog, called `Group`. The following example shows an example of grouping in DDlog:

```
output relation O(v: Group<integer, string>)
// Groups with key integer and values Vector<string>
ByAge(g) :- People(n, a), var g = (n).group_by(a).
// Each g is the group of all names that have the same age
```

The general syntax of a `GroupByTerm` in Figure 2 is given by: `var g = (project-expression).group_by(key-expression)`. In DDlog the `key-expression` is restricted to be a tuple of variables in the current valuation.

The semantics of a `GroupByTerm` is given by the following algorithm:

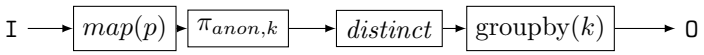
1. We start with some input valuation.
2. The `project-expression` is evaluated for the input valuation V, adding a new (anonymous) variable to the valuation, storing the result of the `project-expression` for each row.
3. The resulting valuation is projected on a tuple of variables containing the new anonymous variable and all variables that appear in `key-expression`. The result of the projection is a new valuation P, a set (with no duplicates). This valuation *only includes the variables that appear in the projection and the key expression*; all other variables are removed. This behavior is unusual — this is the only DDlog operator that removes variables from a valuation.
4. Finally, the data in the valuation is grouped by key, and the result is a valuation that contains for the new variable a group.

As an example, let us evaluate the above DDlog rule according to these steps:

1. The term `Persons(n, a)` provides the following valuation: $(\texttt{n}, \texttt{a}) \in \{(\texttt{bob}, 10), (\texttt{john}, 20), (\texttt{amy}, 10)\}$.
2. In our example `project-expression` is just `n`, whose value will be assigned to the anonymous variable. This creates a new valuation: $(\texttt{n}, \texttt{a}, \texttt{anonymous}) \in \{(\texttt{bob}, 10, \texttt{bob}), (\texttt{john}, 20, \texttt{john}), (\texttt{amy}, 10, \texttt{amy})\}$.
3. The valuation is projected on `a` (the group key) and `anonymous` (the projection key), obtaining: $(\texttt{a}, \texttt{anonymous}) \in \{(10, \texttt{bob}), (20, \texttt{john}), (10, \texttt{amy})\}$. In this case there are no duplicates, but any duplicates would be removed.
4. The values in the valuation are grouped by their `a` value, providing a new group for each value. The variable `g` is added to the resulting valuation. The result is $(\texttt{a}, \texttt{g}) \in \{(10, [\texttt{bob}, \texttt{amy}]), (20, [\texttt{john}])\}$. Notice how the value of `g` in the valuation is a collection, shown with square brackets.

Let us define this computation in terms of $\mathbb{Z}$-sets. Consider an arbitrary type of keys K, and a function that computes a key for a value $k : I \to K$. Then we define $\text{groupby}(k) : \mathbb{Z}[I] \to \mathbb{Z}[I][K]$, as $\text{groupby}(k)(i) = \sum_{x \in i} \{k(x) \mapsto 1 \cdot x\}$. Note that groupby always produces a set of $\mathbb{Z}$-sets. The weight of each group is always 1. Note that $\text{ispositive}(\text{groupby}(k))$. Also, $\uparrow\text{groupby}(k)$ is time-invariant for any function k, since $\text{groupby}(k)$ has the zero-preservation property.

The implementation of the `group_by` operator requires chaining the implementation of the projection and of groupby function just described: the resulting circuit is (using p as the translation of `project-expression` and k as the translation of `key-expression`):



11.2.9 Aggregation

Classic Datalog does not support aggregations but many practical implementations have extended Datalog with a construct equivalent with a composition of groupby-aggregate.

Strictly speaking, DDlog does not support for aggregation – the only aggregate supported is a group. However, since DDlog allows users to apply arbitrary functions to a `Group` object, traditional aggregation can be performed by grouping and then applying a scalar-returning function using a `map`, as described Section 11.2.6. Consider the following example, an extension of the example in Section 11.2.8:

```
output relation NamesByAge(s: string)
NamesByAge(s) :- ByAge(g),
      var s = g.key + ":␣" + g.toString().
```

This example uses built-in functions `g.key` that obtains the key of a group, and `g.toString()`, which converts the group contents to a string.

Formally, given a function a : `Group<`K`,`I`>` $\to O$, aggregation is just map(a) applied to a set of groups. As a consequence lifted aggregation is time-invariant.

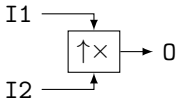
11.2.10 Cartesian products

Cartesian products in Datalog appear from the use of in a rule body of a `TermRelation` where the relation arguments are all new variables (not already defined in the input valuation). The Datalog semantics of Cartesian products is to produce a new valuation that includes all variables, and having as values the Cartesian product of the values of the input valuation and the relation in the term.

The following program shows an example Cartesian product:

```
O(v1, v2) :- I1(v1), I2(v2).
```

A Cartesian product is implemented as a circuit using the product operation on $\mathbb{Z}$-sets. For $i_1 \in \mathbb{Z}[A]$ and $i_2 \in \mathbb{Z}[B]$ we define $i_1 \times i_2 \in \mathbb{Z}[A \times B]$ by $(i_1 \times i_2)(\langle x, y\rangle) \stackrel{\text{def}}{=} i_1[x] \times i_2[y].\forall x \in i_1, y \in i_2$. The circuit computing the Cartesian product is given by:



As an example, let us consider the product of the following two $\mathbb{Z}$-sets:

x	W
bob	1
mike	2

$\times$

y	W
bob	1
john	-1

$=$

(x, y)	W
(bob, bob)	1
(mike, bob)	2
(bob, john)	-1
(mike, john)	-2

Notice that $\text{isset}(x) \wedge \text{isset}(y) \Rightarrow \text{isset}(x \times y)$. The Cartesian product as defined on $\mathbb{Z}$-sets is a bilinear operator. Also, $\uparrow\times$ is time-invariant.

11.2.11 Joins

A join appears in a Datalog program by using a `TermRelation` that has as arguments some variables that are already defined in the current valuation. The following example shows a join: since the second relation reuses variable `v`, which is already bound by the valuation, this is a join, and not a Cartesian product:

```
O(x, y) :- I1(x, v), I2(v, y).
```

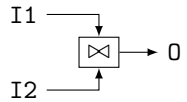
The semantics of a join can be modeled as a cartesian product followed by a sequence of filters. This is achieved by using fresh variable names for the arguments of each `TermRelation`, and adding predicates that require these fresh variables to be equal to the bound variables they replace. For example, the following program is equivalent to the one above.

```
O(x, y) :- I1(x, v), I2(v1, y), v = v1.
```

Since a join is a composition of a bilinear (the Cartesian product) and a linear (filtering) operator, it is also a bilinear operator, and thus its lifted version is time-invariant.

In practice joins are very important computationally, and they are implemented by a custom operator on $\mathbb{Z}$-sets denoted by $\bowtie$.

The circuit computing the join product is given by:



11.2.12 Set intersection

Set intersection in Datalog is just a particular case of join where a `TermRelation` uses *all* the variables defined in the current valuation as arguments. In the following example relation `O` is the intersection of relations `I1` and `I2`:

```
O(v) = I1(v), I2(v).
```

The join implementation using circuits immediately applies to set intersections. It follows that set intersection is a bilinear operator, and thus time-invariant when lifted.

11.2.13 Negation

We only support Datalog programs with stratified negation. See [3] for a precise definition. Negation in a Datalog program is introduced syntactically by a `NegatedTerm` from the grammar in Figure 2. A negated term cannot appear first in a rule body. All variables that appear in the negated term must have been already defined by previous terms in the body.

With these syntactic constraints the are two different meanings to negation:

- If the negated term uses *all* variables already in the valuation, it is modeled as a set difference.
- If the negated term uses a subset of all the variables in the existing valuation, it is modeled as an antijoin.

We describe each of these two cases.

11.2.14 Set difference

If the `NegatedTerm` contains as arguments all variables in the current valuation, the meaning of negation is just set difference: the resulting valuation will *exclude* all tuples from the negated relation.

For example, consider the rule:

```
relation Major(name: string, age: integer)
Major(n, a) :- People(n, a), not Minor(n, a).
```

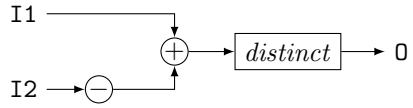
The valuation computed by the rule's body is $\{(\texttt{bob}, 10), (\texttt{john}, 20), (\texttt{amy}, 10)\} \setminus \{(\texttt{bob}, 10), (\texttt{amy}, 10)\} = \{(\texttt{john}, 20)\}$.

In terms of $\mathbb{Z}$-sets, let us consider the following program:

```
O(v) :- I1(v), not I2(v).
```

We define the set difference on $\mathbb{Z}$-sets as follows: $\setminus : \mathbb{Z}[I] \times \mathbb{Z}[I] \to \mathbb{Z}[I]$, where $i_1 \setminus i_2 = \mathit{distinct}(i_1 - i_2)$. Note that we have $\forall i_1, i_2, \text{ispositive}(i_1 \setminus i_2)$ due to the application of the *distinct* operator.

The circuit computing the valuation of the body of this rule is:



This whole circuit is time-invariant, since it composed only of time-invariant operators.

11.2.15 Antijoin

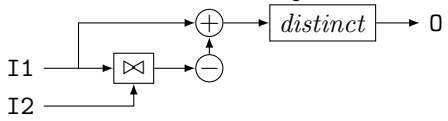
Antijoin is the semantics of a Datalog `NegatedTerm` that uses a relation which does not use some of the variables in the current valuation. Consider the following program:

```
O(v) :- I1(v, z), not I2(v).
```

The semantics of such a rule can be defined in terms of joins and set difference. This rule is equivalent with the following pair of rules:

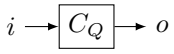
```
C(v, z) :- I1(v, z), I2(v).
O(v) :- I1(v, z), not C(v, z).
```

This transformation reduces an antijoin to a join (using all variables in the current valuation), followed by a set difference. The translation of these rules is covered by Sections 11.2.11 and Section 11.2.14. In terms of circuits we can just build the circuit for the pair of rules:



11.3 Streaming Differential Datalog

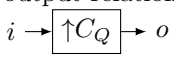
In this section we have shown how Given a Datalog (or SQL) query Q, can be converted into a circuit C_Q that computes the same input-output function as Q.



We can perform two simple transformations to this circuit: we can lift it to convert it into a streaming program, and then we can incrementalize it, to convert it into a differential program.

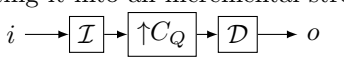
11.3.1 Streaming Datalog

Given the circuit $C_Q : A \to B$, We can lift it to compute on *streams* of relations. $\uparrow C_Q : \mathcal{S}_A \to \mathcal{S}_B$ interacts with its environment in "epochs," corresponding to the time dimension of the streams. In each epoch the circuit receives a new set of values for the inputs relations and it provides the corresponding values for the output relations.



11.3.2 Streaming Differential Datalog

Furthermore, we can apply the $\cdot^\Delta$ operator to the streaming circuit $\uparrow C_Q$, converting it into an incremental streaming circuit: $(\uparrow C_Q)^\Delta : \mathcal{S}_A \to \mathcal{S}_B$.



This is a differential streaming version of the circuit C_Q. This circuit interacts with its environment in "ephochs," corresponding to the time dimension of the streams. In each epoch the circuit receives a new set of *changes* to the inputs relations and it provides the corresponding *change* for the output relations.

This is in essence the service provided by the DDlog compiler: given a query Q it provides a streaming implementation of ${\uparrow C_Q}^\Delta$. However, the DDlog runtime provides some additional services, described in the next section.

12 Additional Implementation Details

12.1 Checkpoint/restore

DDlog programs are stateful streaming systems. Fault-tolerance and migration for such programs requires state migration. We claim that it is sufficient to checkpoint and restore the "contents" of all z^{-1} operator in order to migrate the state of a Ddlog computation.

12.2 Maintaining a database

DDlog is not a database, it is just a streaming view maintenance system. In particular, DDlog will not maintain more state than absolutely necessary to compute the changes to the views. There is no way to find out whether a specific value exists at a specific time moment within a DDlog relation. However, a simple extension to DDlog runtime can be made to provide a *view query* API: essentially all relations that may be queried have to be maintained internally in an integrated form as well. The system can then provide an API to enumerate or query a view about element membership between input updates.

12.3 Materialized views

An incremental view maintenance system is not a database – it only computes changes to views when given changes to tables. However, it can be integrated with a database, by providing capabilities for *querying* both tables and views. An input table is just the integral of all the changes to the table. This makes possible building a system that is both stateful (like a database) and streaming (like an incremental view maintenance system).

12.4 Maintaining input invariants

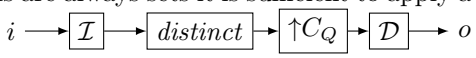
For relational query systems there is however an important caveat: the proofs about the correctness of the C_Q implementing the same semantics as Q all require some preconditions on the circuit inputs. In particular, the semantics of Q is only defined for sets. In order for C_Q to faithfully emulate the behavior of Q we must enforce the invariant that the input relations are in fact sets.

However, the differential streaming version of the circuits accepts an arbitrary stream of changes to the input relations. Not all such streams define input relations that are sets! For example, consider an input stream where the first element removes a tuple from an input relation. The resulting $\mathbb{Z}$-set does not represent a set, and thus the proof of correctness does not hold. This problem has been well understood in the context of the relational algebra: it is the same as the notion of positivity from [14].

We propose three different solutions to this problem, in increasing degrees of complexity.

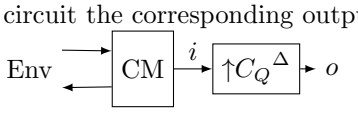
Assume that the environment is well-behaved The simplest solution is to do nothing and assume that at any point in time the integral of the input stream of changes i is a set: $\forall t \in \mathbb{N}.\text{isset}(\mathcal{I}(i)[t])$. This may be a reasonable assumption if the changes come from a controlled medium, e.g., a traditional database, where they represent legal database changes.

Normalize input relations to sets In order to enforce that the input relations are always sets it is sufficient to apply a *distinct* operator after integration.



The semantics of the resulting circuit is identical to the semantics of ${\uparrow C_Q}^{\Delta}$ for well-behaved input streams. For non-well behaved input streams one can give a reasonable definition: a change is applied to the input relations, and then non-relations are normalized into relations. Removing a non-existent element is a no-op, and adding twice an element is the same as adding it once.

Use a "change manager" In this solution we interpose a separate software component between the environment and the circuit. Let us call this a "change manager" (CM). The CM is responsible for accepting commands from the environment that perform updates on the input relations, validating them, and building incrementally an input change, by computing the effect of the commands. The CM needs to maintain enough internal state to validate all commands; this will most likely entail maintaining the full contents of the input tables. Note that the input tables can be computed as the $\mathcal{I}$ of all input deltas ever applied. Once all commands producing a change have been accepted, the environment can apply the produced input change atomically, and obtain from the circuit the corresponding output changes.



Part IV
Appendixes

A Z-transform and stream convolutions

Definition A.1. The **Z-transform** of a stream, as traditionally defined in signal processing, is a function that associates with any stream over a group a formal power series in the indeterminate z: $\mathcal{Z} : \mathcal{S}_A \to A[\![z]\!]$ defined as $\mathcal{Z}(s) \stackrel{\text{def}}{=} \sum_{t \geq 0} s[t] z^{-t}$.

For example, the Z-transform of the *id* stream is the power series $0 + z^{-1} + z^{-2} + z^{-3} + \ldots$.

Definition A.2. Let $(R, +, \cdot, 0, 1)$ be a commutative ring. The **Cauchy product** (also called discrete convolution) of two streams $* : \mathcal{S}_R \times \mathcal{S}_R \to \mathcal{S}_R$ is defined as:

$$(a * b)[t] \;=\; \sum_{i=0}^{t} a[i] \cdot b[t-i]$$

For example, the convolution of the id stream with itself is the stream $id * id$ containing the sequence of values $0, (0 \cdot 1 + 1 \cdot 0), (0 \cdot 2 + 1 \cdot 1 + 2 \cdot 0), \ldots = 0, 0, 1, 4, 8, \ldots$.

Proposition A.3. The structure $(\mathcal{S}_R, +, *, 0, 1)$ is also a commutative ring. This ring is isomorphic to the ring of formal power series in one indeterminate $R[\![z]\!]$ with coefficients from R.

Sometimes it is more convenient to use the formal power series notation. Notice that we have $z^{-1}(s) = z^{-1} * s$, justifying the traditional notation for the delay operator z^{-1}. It follows that the differentiation of a stream s is $\mathcal{D}(s) = (1 - z^{-1}) * s$.

Moreover, the equation that defines the integration of a stream s, $\xi = z^{-1}(\xi) + s$ is equivalent to $\xi = z^{-1} * \xi + s$ and then to $(1 - z^{-1}) * \xi = s$. Since $1 - z^{-1}$ has multiplicative inverse

$$(1 - z^{-1})^{-1} \;=\; 1 + z^{-1} + z^{-2} + z^{-3} + \cdots$$

we can express the integration operator by $\mathcal{I}(s) = (1 - z^{-1})^{-1} * s$. Theorem 3.30 now follows by algebraic manipulations in the ring of formal power series. Similarly for the time-invariance and linearity properties of $\mathcal{D}$ and $\mathcal{I}$. Even causality can be treated algebraically, once we note that, like addition, convolution is causal.

Observation As shown above, there are two proof styles for equations over streams: one is (usually) by induction over the time dimension, and the other one is equational, by operating with polynomials over z. The theory of digital signal processing posits that these two proof styles give the same results.

References

[1] The aurora project. http://cs.brown.edu/research/aurora/, December 2004.

[2] Martín Abadi, Frank McSherry, and Gordon Plotkin. Foundations of differential dataflow. In *Foundations of Software Science and Computation Structures (FoSSaCS)*, 2015. URL: http://homepages.inf.ed.ac.uk/gdp/publications/differentialweb.pdf.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. URL: http://webdam.inria.fr/Alice/.

[4] Yanif Ahmad and Christoph Koch. DBToaster: A SQL compiler for high-performance delta processing in main-memory databases. *Proc. VLDB Endow.*, 2(2):1566–1569, August 2009. URL: https://doi.org/10.14778/1687553.1687592.

[5] Mario Alvarez-Picallo, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. Fixing incremental computation. In *European Symposium on Programming Languages and Systems (ESOP)*, pages 525–552, 2019. URL: https://link.springer.com/chapter/10.1007/978-3-030-17184-1_19.

[6] Krzysztof R. Apt and Jean-Marc Pugin. Maintenance of stratified databases viewed as a belief revision system. In Moshe Y. Vardi, editor, *ACM SIGMOD International conference on Management of data (SIGMOD)*, pages 136–145, San Diego, California, March 23-25 1987. doi:10.1145/28659.28674.

[7] Arvind Arasu, Shivnath Babu, and Jennifer Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical Report 2002-57, Stanford InfoLab, 2002. URL: http://ilpubs.stanford.edu:8090/563/.

[8] W. Baker and A. Newton. The maximal VHDL subset with a cycle-level abstraction. In *Proceedings of the Conference on European Design Automation (DATE)*, pages 470–475, 1996.

[9] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *International Conference of Very Large Data Bases (VLDB)*, pages 577–589, Barcelona, Spain, 1991. URL: http://www.vldb.org/conf/1991/P577.PDF.

[10] Zaheer Chothia, John Liagouris, Frank McSherry, and Timothy Roscoe. Explaining outputs in modern data analytics. *Proc. VLDB Endow.*, 9(12):1137–1148, August 2016. URL: https://doi.org/10.14778/2994509.2994530.

[11] Hasanat M. Dewan, David Ohsie, Salvatore J. Stolfo, Ouri Wolfson, and Sushil Da Silva. Incremental database rule processing in PARADISER. *J. Intell. Inf. Syst.*, 1(2):177–209, 1992. doi:10.1007/BF00962282.

[12] Peter Gammie. Synchronous digital circuits as functional programs. *ACM Comput. Surv.*, 46(2), November 2013. URL: https://doi.org/10.1145/2543581.2543588.

[13] Sergio Greco and Cristian Molinaro. Datalog and logic databases. *Synthesis Lectures on Data Management*, 7(2):1–169, 2015. URL: https://doi.org/10.2200/S00648ED1V01Y201505DTM041.

[14] Todd J Green, Zachary G Ives, and Val Tannen. Reconcilable differences. *Theory of Computing Systems*, 49(2):460–488, 2011. URL: https://web.cs.ucdavis.edu/~green/papers/tocs11_differences.pdf.

[15] Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Symposium on Principles of Database Systems (PODS)*, page 31–40, 2007. URL: https://doi.org/10.1145/1265530.1265535.

[16] Ashish Gupta, Inderpal Singh Mumick, et al. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

[17] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, SIGMOD '93, page 157–166, Washington, D.C., USA, 1993. URL: https://doi.org/10.1145/170035.170066.

[18] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *ACM SIGMOD International conference on Management of data (SIGMOD)*, pages 157–166, Washington, DC, May 26-28 1993. ACM Press. doi:10.1145/170035.170066.

[19] John V. Harrison and Suzanne W. Dietrich. Maintenance of materialized views in a deductive database: An update propagation approach. In Kotagiri Ramamohanarao, James Harland, and Guozhu Dong, editors, *Workshop on Deductive Databases*, volume CITRI/TR-92-65 of *Technical Report*, pages 56–65, Washington, D.C., November 14 1992. Department of Computer Science, University of Melbourne.

[20] Hojjat Jafarpour, Rohan Desai, and Damian Guy. KSQL: Streaming SQL engine for Apache Kafka. In *International Conference on Extending Database Technology (EDBT)*, pages 524–533, Lisbon, Portugal, March 26-29 2019. URL: http://openproceedings.org/2019/conf/edbt/EDBT19_paper_329.pdf.

[21] Steven Dexter Johnson. *Synthesis of Digital Designs from Recursion Equations*. PhD thesis, Indiana University, May 1983. https://help.luddy.indiana.edu/techreports/TRNNN.cgi?trnum=TR141.

[22] Gilles Kahn. The semantics of a simple language for parallel programming. In *IFIP Congress on Information Processing*, 1974. URL: http://www1.cs.columbia.edu/~sedwards/papers/kahn1974semantics.pdf.

[23] Christoph Koch. Incremental query evaluation in a ring of databases. In *Symposium on Principles of Database Systems (PODS)*, page 87–98, 2010. URL: https://doi.org/10.1145/1807085.1807100.

[24] Christoph Koch, Daniel Lupei, and Val Tannen. Incremental view maintenance for collection programming. In *Symposium on Principles of Database Systems (PODS)*, page 75–90, 2016. URL: https://doi.org/10.1145/2902251.2902286.

[25] Jakub Kotowski, François Bry, and Simon Brodt. Reasoning as axioms change - incremental view maintenance reconsidered. In *Web Reasoning and Rule Systems RR*, volume 6902 of *Lecture Notes in Computer Science*, pages 139–154, Galway, Ireland, August 29-30 2011. Springer. doi:10.1007/978-3-642-23580-1_11.

[26] Edward A. Lee. Multidimensional streams rooted in dataflow. In *IFIP Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, Orlando, FL, January 20-22 1993. URL: https://ptolemy.berkeley.edu/publications/papers/93/mdsdf/.

[27] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, pages 773–801, May 1995. URL: https://ptolemy.berkeley.edu/publications/papers/95/processNets/.

[28] Charles E. Leiserson and James B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991. URL: http://www.cs.columbia.edu/~CS6861/handouts/leiserson-algorithmica-88.pdf, doi:https://doi.org/10.1007/BF01759032.

[29] James J. Lu, Guido Moerkotte, Joachim Schü, and V. S. Subrahmanian. Efficient maintenance of materialized mediated views. In *ACM SIGMOD International conference on Management of data (SIGMOD)*, pages 340–351, San Jose, California, May 22-25 1995. doi:10.1145/223784.223850.

[30] Konstantinos Mamouras. Semantic foundations for deterministic dataflow and stream processing. In Peter Müller, editor, *European Symposium on Programming*, pages 394–427, Dublin, Ireland, April 25–30 2020.

[31] Frank McSherry, Andrea Lattuada, Malte Schwarzkopf, and Timothy Roscoe. Shared arrangements: Practical inter-query sharing for streaming dataflows. *Proc. VLDB Endow.*, 13(10):1793–1806, June 2020. URL: https://doi.org/10.14778/3401960.3401974.

[32] Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard. Differential dataflow. In *Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, January 6–9 2013. URL: http://cidrdb.org/cidr2013/Papers/CIDR13_Paper111.pdf.

[33] Boris Motik, Yavor Nenov, Robert Piro, and Ian Horrocks. Maintenance of Datalog materialisations revisited. *Artif. Intell.*, 269:76–136, 2019. URL: https://doi.org/10.1016/j.artint.2018.12.004.

[34] Boris Motik, Yavor Nenov, Robert Edgar Felix Piro, and Ian Horrocks. Incremental update of Datalog¡ materialisation: the backward/-forward algorithm. pages 1560–1568, Austin, Texas, January 25-30 2015. AAAI Press. URL: http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9660.

[35] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *ACM Symposium on Operating Systems Principles (SOSP)*, page 439–455, 2013. doi:10.1145/2517349.2522738.

[36] L. R. Rabiner and B. Gold, editors. *Theory and Application of Digital Signal Processing*. Prentice-Hall, 1975.

[37] Martin Staudt and Matthias Jarke. Incremental maintenance of externally materialized views. In *International Conference of Very Large Data Bases (VLDB)*, pages 75–86, Mumbai (Bombay), India, September 3-6 1996. URL: http://www.vldb.org/conf/1996/P075.PDF.

[38] Ouri Wolfson, Hasanat M. Dewan, Salvatore J. Stolfo, and Yechiam Yemini. Incremental evaluation of rules and its relationship to parallelism. In *ACM SIGMOD International conference on Management of data (SIGMOD)*, pages 78–87, Denver, Colorado, May 29-31 1991. ACM Press. doi:10.1145/115790.115799.