

Foundations of Differential Dataflow

Martín Abadi^{1,2} Frank McSherry Gordon D. Plotkin^{1,3}

¹ Microsoft Research

² University of California at Santa Cruz

³ LFCS, School of Informatics, University of Edinburgh

Abstract. Differential dataflow is a recent approach to incremental computation that relies on a partially ordered set of differences. In the present paper, we aim to develop its foundations. We define a small programming language whose types are abelian groups, with both a standard and a differential denotational semantics. The two semantics coincide in that the differential semantics is the differential of the standard one. Möbius inversion, a well-known idea from combinatorics, permits a systematic treatment of various operators and constructs.

1 Introduction

Differential computation [2] is a recent approach to incremental computation (see, e.g., [1,3]) that relies on a partially ordered collection of versions of data. In its intended implementations, the set of updates required to reconstruct any given version is retained in a data structure indexed by the partial order, rather than consolidated into a “current” version. For example, in an iterative algorithm with two nested loops with counters i and j , differential computation may associate a version with each pair (i, j) (with the product order on such pairs). Then an implementation may re-use work done at all $(i', j') < (i, j)$ to compute the (i, j) -th version. We model such collections of data as functions from the partial order to the possible data versions, and call them *streams*.

Differential dataflow is an instantiation of differential computation in a data-parallel dataflow setting. In such a setting the data used are large collections of records and the fundamental operators are independently applied to disjoint parts of their inputs. Differential computation preserves the sparseness of input differences in the output, as an output can change only if its input has changed. The result can be very concise representations and efficient updates. The Naiad system [4] includes a realization of differential dataflow that supports high-throughput, low-latency computations on large datasets with frequent updates.

Differential dataflow aims to avoid redundant computation by replacing the versions of its collection-valued variables with versions of *dif-*

ferences. These may have negative multiplicities, so that a version A_t of a stream A is the sum of the differences $(\delta A)_s$ at versions $s \leq t$: $A_t = \sum_{s \leq t} \delta A_s$. The formula resembles those used in incremental computation, where $s, t \in \mathbb{N}$, but permits more general orders.

Functions on streams A are replaced by their *differentials*, operating on the corresponding difference streams δA , and responsible for producing corresponding output difference streams. In particular, as established in [2], the product order \mathbb{N}^k enables very efficient nested iterative differential computation, because each nested iteration can selectively re-use some of the previously computed differences, but is not required to use all of them. Efficiently updating the state of an iterative computation is challenging, and is the main feature of differential dataflow.

In the present paper we aim to develop the foundations of differential dataflow. We show that the use of collections allowing negative multiplicities and product partial orders of the natural numbers are special cases of general differential computation on abelian groups and locally finite partial orders. We demonstrate the relevance and usefulness of Möbius inversion, a well-known idea from combinatorics (see, for example, [5,6]), to understanding and verifying properties of function differentials.

Specifically, we consider the question of finding the differential of a computation given by a program in a small programming language that includes nested iteration. To this end, we define both a standard denotational semantics for the language and a differential one. Our main theorem (Theorem 1 below) states that the semantics are consistent in that the differential semantics is the differential of the standard semantics. Möbius inversion is central to these results.

In Section 2 we lay the mathematical foundations for differential computation. We discuss how abelian groups arise naturally when considering collections with negative multiplicities. We explain Möbius inversion for spaces of functions from partial orders to abelian groups. We then define function differentials, giving some examples. In particular, we derive some formulas for such differentials, previously set out without justification [2].

In Section 3 we consider loops. Two policies for loop egress are mentioned in [2]: exit after a fixed number of iterations and exit on a first repetition. We consider only the first of these, as it is the one used in practice and mathematically simpler; the second would require the use of partial functions from partial orders to groups.

In Section 4 we present the language and its two semantics, and establish Theorem 1. As noted above, the semantics are denotational, defining what is computed, rather than how; going further, it may be attractive to

describe an operational semantics in terms of the propagation of differences in a dataflow graph, somewhat closer to Naiad’s implementation.

In Section 5 we discuss the treatment of prioritization, a technique from [2] aimed at supporting nested iterative computations starting from where they last left off for each outer iteration, rather than restarting. The treatment of priorities in [2] via lexicographic products of partial orders does not correctly support more than one nested loop (despite the suggestion there that it should); further, the treatment of differential aspects is incomplete, and it is not clear how to proceed. We instead propose a simpler rule and show that it correctly achieves the goal of arbitrary prioritized computation.

We conclude in Section 6, and discuss some possible future work.

2 Mathematical foundations

The mathematical foundations of differential dataflow concern: data organised into abelian groups; version-indexed streams of data and their differentials, which are obtained by Möbius transformation; and stream operations and their differentials, which, in their turn, operate on stream differentials. These three topics are covered in Sections 2.1, 2.2, and 2.3.

2.1 Abelian groups

Abelian groups play a major role in our theory, arising from negative multiplicities. The set of collections, or multisets, $\mathcal{C}(X)$ over a set X can be defined as the functions $c : X \rightarrow \mathbb{N}$ that are 0 almost everywhere. It forms a commutative monoid under multiset union, defined pointwise by: $(c \cup d)(x) = c(x) + d(x)$. The set of multisets $\mathcal{A}(X)$ with possibly negative multiplicities is obtained by replacing \mathbb{N} by \mathbb{Z} ; it forms an abelian group under pointwise sum. A similar example is provided by the commutative monoid \mathcal{R}_+ of the positive reals and the abelian group \mathcal{R} of all reals.

A function between commutative monoids is *linear* if it preserves finite sums, e.g., selection and aggregation can provide linear functions from $\mathcal{C}(X)$ to commutative monoids such as $\mathcal{C}(Y)$ and \mathcal{R}_+ . These functions lift to the corresponding groups: every linear $f : \mathcal{C}(X) \rightarrow G$, with G an abelian group, has a unique linear extension $\bar{f} : \mathcal{A}(X) \rightarrow G$ given by $\bar{f}(c) = \sum_{x \in X} c(x)f(x)$. These observations exemplify a well-known general construction universally embedding cancellative commutative monoids in abelian groups.

2.2 Versions, streams, and Möbius inversion.

We work with a general notion of version, viz. *locally finite partial orders*, that is partial orders T such that, for $t \in T$, $\downarrow t$ is finite ($\downarrow t$ is $\{t' \mid t' \leq t\}$). Examples include finite products of \mathbb{N} , as indicated above, and the finite powerset partial order $\mathcal{P}_{fin}(I)$. We think of functions from T to G as T -indexed streams of elements of G .

The Möbius coefficients $\mu_T(t', t) \in \mathbb{Z}$, with $t' \leq t$, are given by the following recursion:

$$\mu_T(t', t) = \begin{cases} 0 & (t' \not\leq t) \\ 1 & (t' = t) \\ -\sum_{t' \leq r < t} \mu_T(t', r) & (t' < t) \end{cases}$$

For example for $T = \mathbb{N}$ (the natural numbers with their usual ordering), $\mu_{\mathbb{N}}(n', n)$ is 1, if $n' = n$, is -1 , if $n' = n - 1$, and is 0, otherwise; for $X \subseteq Y \subseteq_{fin} I$, $\mu(X, Y) = -1^{\#(Y \setminus X)}$. For product partial orders one has: $\mu_{S \times T}((s', t'), (s, t)) = \mu_S(s', s)\mu_T(t', t)$.

The Möbius transformation of a function $f : T \rightarrow G$, where G is an abelian group, is given by:

$$\delta_T(f)(t) = \sum_{t' \leq t} \mu_T(t', t) f(t')$$

For example $\delta_{\mathbb{N}}(f)(n) = f(n) - f(n - 1)$, if $n > 0$, and $= f(0)$ if $n = 0$.

Defining

$$S_T(f)(t) = \sum_{t' \leq t} f(t')$$

we obtain the famous Möbius inversion formulas:

$$S_T(\delta_T(f)) = f = \delta_T(S_T(f))$$

See, for example, [5,6]. Expanded out, these formulas read:

$$f(t) = \sum_{t' \leq t} \sum_{t'' \leq t'} \mu_T(t'', t') f(t'') \quad f(t) = \sum_{t' \leq t} \mu_T(t', t) \sum_{t'' \leq t'} f(t'')$$

The collection G^T of all T -indexed streams of elements of G forms an abelian group under pointwise addition. We would further like to iterate this function space construction to obtain the doubly indexed functions mentioned in the introduction; we would also like to consider products of such groups. It is therefore natural to generalise to abelian groups G

equipped with linear inverses $G \xrightarrow{\delta_G} G \xrightarrow{S_G} G$. A simple example is any abelian group G , such as $\mathcal{A}(X)$, with $\delta_G = S_G = \text{id}_G$, the identity on G .

For such a G and a locally finite partial order T we define linear inverses $G^T \xrightarrow{\delta_{G^T}} G^T \xrightarrow{S_{G^T}} G^T$ on G^T by setting:

$$\delta_{G^T}(f) = \sum_{t' \leq t} \mu_T(t', t) \delta_G(f(t')) \quad \text{and} \quad S_{G^T}(f) = \sum_{t' \leq t} S_G(f(t'))$$

It is clear that δ_{G^P} and S_{G^P} are linear; we check they are mutually inverse:

$$\begin{aligned} \delta_{G^P}(S_{G^P}(x))(t) &= \sum_{t' \leq t} \mu(t', t) \delta_G(S_{G^P}(x)(t')) \\ &= \sum_{t' \leq t} \mu(t', t) \delta_G(\sum_{t'' \leq t'} S_G(x(t''))) \\ &= \sum_{t' \leq t} \sum_{t'' \leq t'} \mu(t', t) \delta_G(S_G(x(t''))) \quad (\text{as } \delta_G \text{ is linear}) \\ &= \sum_{t' \leq t} \mu(t', t) \sum_{t'' \leq t'} x(t'') \\ &= x(t) \quad (\text{by the M\"obius inversion formula}) \end{aligned}$$

$$\begin{aligned} S_{G^P}(\delta_{G^P}(x))(t) &= \sum_{t' \leq t} S_G(\delta_{G^P}(x)(t')) \\ &= \sum_{t' \leq t} S_G(\sum_{t'' \leq t'} \mu(t'', t') \delta_G(x(t''))) \\ &= \sum_{t' \leq t} \sum_{t'' \leq t'} \mu(t'', t') S_G(\delta_G(x(t''))) \quad (\text{as } S_G \text{ is linear}) \\ &= \sum_{t' \leq t} \sum_{t'' \leq t'} \mu(t'', t') x(t'') \\ &= x(t) \quad (\text{by the M\"obius inversion formula}) \end{aligned}$$

Iterating the stream construction enables us to avoid the explicit use of product partial orders, as the group isomorphism $(G^T)^{T'} \cong G^{T \times T'}$ extends to an isomorphism of their linear inverses.

As for products, given two abelian groups G and H with linear inverses δ_G, S_G and δ_H, S_H , we construct linear inverses $\delta_{G \times H}$ and $S_{G \times H}$ for $G \times H$ by setting: $\delta_{G \times H}(c, d) = (\delta_G(c), \delta_H(d))$ and $S_{G \times H}(c, d) = (S_G(c), S_H(d))$. We write π_0 and π_1 for the first and second projections.

2.3 Function differentials.

The *differential* of a function $f : G \rightarrow H$ is the function $\delta(f) : G \rightarrow H$ where:

$$\delta(f) =_{\text{def}} \delta_H \circ f \circ S_G .$$

The definition applies to n-ary functions, e.g., for $f : G \times H \rightarrow K$ we have $\delta(f)(c, d) = \delta_K(f(S_G(c), S_H(d)))$. With this definition we have $\delta(f)(\delta_G(c)) = \delta_H(f(c))$, suggesting that straight-line programs written using such functions can be recast differentially by replacing both streams and functions with their corresponding differentials. Efficient differential

implementations were developed in [2] for several important classes of primitive functions (e.g., selection, projection, relational joins).

A function $f : G \rightarrow H$ can be lifted by a partial order T to a function $f^T : G^T \rightarrow H^T$ defined element-wise as:

$$f^T(c)_t = f(c_t)$$

The most common case is when $T = \mathbb{N}$, corresponding to the lifting of an arbitrary function f to one whose inputs may vary sequentially, either because the function is to be placed within a loop or because the inputs may change due to external stimulus.

The following proposition relates the differential of a lifted function to its own differential using Möbius coefficients. It proves the correctness of some implementations from [2], specifically that some lifted linear functions, such as selection and projection, are their own differentials.

Proposition 1. *For any $c \in G^T$ and $t \in T$ we have:*

1.

$$\delta(f^T)(c)_t = \sum_{t' \leq t} \mu(t', t) \delta(f) \left(\sum_{t'' \leq t'} c_{t''} \right)$$

2. *If, further, f is linear then we have: $\delta(f^T)(c)_t = \delta(f)(c_t)$.*

3. *If, yet further, $\delta(f) = f$ then $\delta(f^T) = f^T$, that is, $\delta(f^T)(c)_t = f(c_t)$.*

Proof. 1. We calculate:

$$\begin{aligned} \delta(f^T)(c)_t &= \sum_{t' \leq t} \mu(t', t) \delta_H(f^T(S_{G^T}(c))_{t'}) \\ &= \sum_{t' \leq t} \mu(t', t) \delta_H(f(S_{G^T}(c)_{t'})) \\ &= \sum_{t' \leq t} \mu(t', t) \delta_H(f(\sum_{t'' \leq t'} S_G(c)_{t''})) \\ &= \sum_{t' \leq t} \mu(t', t) \delta_H(f(S_G(\sum_{t'' \leq t'} c_{t''}))) \\ &= \sum_{t' \leq t} \mu(t', t) \delta(f)(\sum_{t'' \leq t'} c_{t''}) \end{aligned}$$

2. If f is linear so is $\delta(f)$ and then, continuing the previous calculation:

$$\begin{aligned} \delta(f^T)(c)_t &= \sum_{t' \leq t} \mu(t', t) \delta(f)(\sum_{t'' \leq t'} c_{t''}) \\ &= \sum_{t' \leq t} \mu(t', t) \sum_{t'' \leq t'} \delta(f)(c_{t''}) \\ &= \delta(f)(c_t) \end{aligned}$$

3. This is an immediate consequence of the previous part. \square

For binary functions $f : G \times H \rightarrow K$, we define $f^T : G^T \times H^T \rightarrow K^T$ by $f^T(c, d)_t = f(c_t, d_t)$. In the case $T = \mathbb{N}$ a straightforward calculation shows that if f is bilinear (i.e., linear in each of its arguments) then:

$$\delta(f^{\mathbb{N}})(c, d)_n = \delta(f)(c_n, \delta(d)_n) + \delta(f)(\delta(c)_n, d_n) - \delta(f)(\delta(c)_n, \delta(d)_n)$$

justifying the implementations in [2] of differentials of lifted bilinear functions such as relational join. The equation generalises to forests, i.e., the locally finite partial orders whose restriction to any $\downarrow t$ is linear.

The following proposition (proof omitted) applies more generally. Part 2 justifies the implementation of binary function differentials in [2].

Proposition 2. *For any $c \in G^T$, $d \in H^T$, and $t \in T$ we have:*

1.

$$\delta(f^T)(c, d)_t = \sum_{t' \leq t} \mu(t', t) \delta(f) \left(\sum_{t'' \leq t'} c_{t''}, \sum_{t'' \leq t'} d_{t''} \right)$$

2. *If, further, f is bilinear (i.e., linear in each argument separately), and T has binary sups then we have:*

$$\delta(f^T)(c, d)_t = \sum_{\substack{r, s \\ r \vee s = t}} \delta(f)(c_r, d_s)$$

3. *If, yet further, $\delta(f) = f$ we have:*

$$\delta(f^T)(c, d)_t = \sum_{\substack{r, s \\ r \vee s = t}} f(c_r, d_s)$$

3 Loops

To develop the differential of an iterative computation we use the same construction as in [2], but require additional formalism to establish the correctness of the construction, as well as to be able to generalize it sufficiently to correctly support prioritization. Loops follow the dataflow computation pictured in Figure 1.

The Ingress node introduces input to a loop, and is modelled by the function $\mathbf{in} : G \rightarrow G^{\mathbb{N}}$ where:

$$\mathbf{in}(c)_i =_{\text{def}} \begin{cases} c & (i = 0) \\ 0 & (i > 0) \end{cases}$$

The Feedback node advances values from one iteration to the next, and is modelled by the function $\mathbf{fb} : G^{\mathbb{N}} \rightarrow G^{\mathbb{N}}$ where:

$$\mathbf{fb}(c)_i =_{\text{def}} \begin{cases} 0 & (i = 0) \\ c_{i-1} & (i > 0) \end{cases}$$

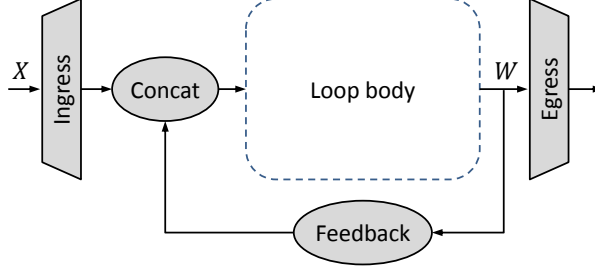


Fig. 1. A loop (reproduced with permission from [2])

The Concat node merges the input and feedback streams, and is modelled by the function $+_{G^T} : G^T \times G^T \rightarrow G^T$. The Egress node effects the fixed-iteration-number loop egress policy, returning the value at some k th iteration, and is modelled by the function $\text{out}_k : G^{\mathbb{N}} \rightarrow G$ where:

$$\text{out}_k(c) = c_k$$

In addition, the loop body is modelled by a function $f^{\mathbb{N}} : G^{\mathbb{N}} \rightarrow G^{\mathbb{N}}$ for a given function f on G .

The loop is intended to output an \mathbb{N} -indexed stream $s \in G^{\mathbb{N}}$ at W , starting at $f(c)$, where $c \in G$ is input at X , and then successively output $f^2(c)$, $f^3(c)$, \dots . It is more convenient, and a little more general, to take the output just after Concat, obtaining instead the sequence $c, f(c), f^2(c), \dots$. This s is a solution of the fixed-point equation:

$$d = \text{in}(c) + \text{fb}(f^{\mathbb{N}}(d)) \quad (1)$$

Indeed it is the unique solution, as one easily checks that the equation is equivalent to the following iteration equations:

$$d_0 = c \quad d_{n+1} = f(d_n)$$

which recursively determine d . The output of the loop is obtained by applying out_k to s , and so the whole loop construct computes $f^k(c)$.

The differential version of the loop employs the differential versions of in , fb , and out , so we first check these agree with [2].

Proposition 3. *The differentials of in , fb , and out satisfy:*

$$\delta(\text{in})(c)_i = \begin{cases} c & (i = 0) \\ -c & (i = 1) \\ 0 & (i \geq 2) \end{cases} \quad \delta(\text{fb}) = \text{fb} \quad \delta(\text{out}_k)(c) = \sum_{m \leq k} c_m$$

Proof. 1. We have:

$$\delta(\mathbf{in})(c)(j) = \delta_{G^{\mathbb{N}}}(\mathbf{in}(S_G(c)))(j) = \sum_{i \leq j} \mu(i, j) \delta_G(\mathbf{in}(S_G(c)))(i)$$

Then we see that if $j = 0$, this is $\delta_G(\mathbf{in}(S_G(c)))(0) = \delta_G(S_G(c)) = c$; if $j = 1$, this is $\delta_G(\mathbf{in}(S_G(c)))(1) - \delta_G(\mathbf{in}(S_G(c)))(0) = 0 - c$; and if $j \geq 2$, this is $\delta_G(\mathbf{in}(S_G(c)))(j) - \delta_G(\mathbf{in}(S_G(c)))(j-1) = 0 - 0$.

2. It suffices to show \mathbf{fb} preserves S , i.e., $\mathbf{fb}(S_{G^T}(c))_j = S_{G^T}(\mathbf{fb}(c))_j$, for all $j \in \mathbb{N}$. In case $j = 0$, both sides are 0. Otherwise we have:

$$\begin{aligned} \mathbf{fb}(S_{G^T}(c))_j &= S_{G^T}(c)_{j-1} \\ &= \sum_{i \leq j-1} S_G(c_i) \\ &= \sum_{1 \leq i \leq j} S_G(c_{i-1}) \\ &= \sum_{i \leq j} S_G(\mathbf{fb}(c)_i) \\ &= S_{G^T}(\mathbf{fb}(c))_j \end{aligned}$$

3. We calculate:

$$\begin{aligned} \delta(\mathbf{out}_k)(c) &= \delta_G(\mathbf{out}_k(S_{G^T}(c))) \\ &= \delta_G(\mathbf{out}_k(m \mapsto \sum_{m' \leq m} S_G(c_{m'}))) \\ &= \delta_G(\sum_{m \leq k} S_G(c_m)) \\ &= \sum_{m \leq k} c_m \end{aligned}$$

□

As the differential version of the loop employs the differential versions of \mathbf{in} , etc, one expects $\delta(s)$ to satisfy the following equation:

$$d = \delta(\mathbf{in})(\delta(c)) + \mathbf{fb}(\delta(f^{\mathbb{N}})(d)) \quad (2)$$

since $+$ and \mathbf{fb} are their own differentials. In fact exactly this equation arises if we differentiate Equation 1. More precisely, Equation 1 specifies that d is a fixed point of F , where $F(d) =_{\text{def}} \mathbf{in}(c) + \mathbf{fb}(f^{\mathbb{N}}(d))$. One then calculates $\delta(F)$:

$$\begin{aligned} \delta(F)(d) &= \delta(F(S(d))) \\ &= \delta(\mathbf{in}(c) + \mathbf{fb}(f^{\mathbb{N}}(Sd))) \\ &= \delta(\mathbf{in}(c)) + \delta(\mathbf{fb}(f^{\mathbb{N}}(Sd))) \\ &= \delta(\mathbf{in})(\delta(c)) + \mathbf{fb}(\delta(f^{\mathbb{N}}(Sd))) \\ &= \delta(\mathbf{in})(\delta(c)) + \mathbf{fb}(\delta(f^{\mathbb{N}})(\delta(Sd))) \\ &= \delta(\mathbf{in})(\delta(c)) + \mathbf{fb}(\delta(f^{\mathbb{N}})(d)) \end{aligned}$$

So Equation 2 specifies that $\delta(s)$ is a fixed-point of $\delta(F)$. It is immediate, for any G and $F : G \rightarrow G$, that d is a fixed-point of F iff $\delta(d)$ is a fixed-point of $\delta(F)$; so $\delta(s)$ is the unique solution of the second equation. As

$s_n = f^n(c)$, differentiating we obtain an explicit formula for $\delta(s)$:

$$\delta(s)_n = \sum_{m \leq n} \mu(m, n) \delta(f)^m(\delta(c))$$

equivalently:

$$\delta(s)_n = \begin{cases} \delta(c) & (n = 0) \\ \delta(f)^n(\delta(c)) - \delta(f)^{n-1}(\delta(c)) & (n > 0) \end{cases}$$

Finally, combining the differential versions of the loop and the egress policy, we find:

$$\begin{aligned} \delta(\mathbf{out}_k)(\delta(s)) &= \sum_{m \leq k} \delta(s)_m \\ &= \sum_{m \leq k} \sum_{l \leq m} \mu(l, m) \delta(f)^l(\delta(c)) \\ &= \delta(f)^k(\delta(c)) \end{aligned}$$

so the differential of the loop followed by the differential of egress is, as expected, the differential of the k th iteration of the loop body.

4 The programming language

The language has expressions of various types, given as follows.

Types

$$\sigma ::= b \mid \sigma \times \tau \mid \mathbf{unit} \mid \sigma^+$$

where b is taken from a set of *base types*. Types will be interpreted as abelian groups with linear inverses; in particular σ^+ will be interpreted as a group of \mathbb{N} -streams.

Expressions

$$\begin{aligned} e ::= & x \mid f(e_1, \dots, e_n) \mid \mathbf{let} \ x : \sigma \ \mathbf{be} \ e \ \mathbf{in} \ e' \mid \\ & 0_\sigma \mid e + e' \mid -e \mid \\ & \langle e, e' \rangle \mid \mathbf{fst}(e) \mid \mathbf{snd}(e) \mid * \mid \\ & \mathbf{iter} \ x : \sigma \ \mathbf{to} \ e \ \mathbf{in} \ e' \mid \mathbf{out}_k(e) \quad (k \in \mathbb{N}) \end{aligned}$$

where we have a given signature $f : \sigma_1, \dots, \sigma_n \rightarrow \sigma$ of *basic function symbols*. (The basic types and function symbols are the built-ins.)

Typing Environments $\Gamma = x_1 : \sigma_1, \dots, x_n : \sigma_n$ are sequences of variable bindings, with no variable repetition. We are given rules to establish *typing judgments*. These judgments have the form:

$$\Gamma \vdash e : \sigma$$

Typing Rules

$$\begin{array}{c}
\Gamma \vdash x : \sigma \quad (x : \sigma \in \Gamma) \\
\\
\frac{\Gamma \vdash e_i : \sigma_i \quad (i = 1, \dots, n)}{\Gamma \vdash f(e_1, \dots, e_n) : \sigma} \quad (f : \sigma_1, \dots, \sigma_n \rightarrow \sigma) \\
\\
\frac{\Gamma \vdash e : \sigma \quad \Gamma, x : \sigma \vdash e' : \tau}{\Gamma \vdash \text{let } x : \sigma \text{ be } e \text{ in } e' : \tau} \\
\\
\Gamma \vdash 0_\sigma : \sigma \quad \frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash e + e' : \sigma} \quad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash -e : \sigma} \\
\\
\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash e' : \tau}{\Gamma \vdash \langle e, e' \rangle : \sigma \times \tau} \quad \frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash \text{fst}(e) : \sigma} \quad \frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash \text{snd}(e) : \tau} \\
\\
\frac{\Gamma, x : \sigma \vdash e : \sigma \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash \text{iter } x : \sigma \text{ to } e \text{ in } e' : \sigma^+} \quad \frac{\Gamma \vdash e : \sigma^+}{\Gamma \vdash \text{out}_k(e) : \sigma}
\end{array}$$

Proposition 4. (*Unique typing*) For any Γ, e , there is at most one σ such that $\Gamma \vdash e : \sigma$.

In fact, there will also be a unique derivation of $\Gamma \vdash e : \sigma$.

4.1 Language semantics

Types Types are modelled by abelian groups with inverses, as described in Section 2. For for each basic type b we assume given an abelian group with inverses $(\mathcal{B}[[b]], \delta_b, S_b)$. The denotational semantics of types is then:

$$\begin{array}{l}
\mathcal{D}[[b]] = \mathcal{B}[[b]] \\
\mathcal{D}[[\sigma \times \tau]] = \mathcal{D}[[\sigma]] \times \mathcal{D}[[\tau]] \\
\mathcal{D}[[\text{unit}]] = \mathbb{1} \\
\mathcal{D}[[\sigma^+]] = \mathcal{D}[[\sigma]]^{\mathbb{N}}
\end{array}$$

Expressions For each basic function symbol $f : \sigma_1, \dots, \sigma_n \rightarrow \sigma$ we assume given a map:

$$\mathcal{B}[[f]] : \mathcal{D}[[\sigma_1]] \times \dots \times \mathcal{D}[[\sigma_n]] \longrightarrow \mathcal{D}[[\sigma]] .$$

We do *not* assume these are linear, multilinear, or preserve the δ 's or S 's.

Let $\mathcal{D}[[\Gamma]] = \mathcal{D}[[\sigma_1]] \times \dots \times \mathcal{D}[[\sigma_n]]$ for $\Gamma = x : \sigma_1, \dots, x_n : \sigma_n$. Then for each $\Gamma \vdash e : \sigma$ we define its semantics with type:

$$\mathcal{D}[[\Gamma \vdash e : \sigma]] : \mathcal{D}[[\Gamma]] \longrightarrow \mathcal{D}[[\sigma]]$$

In case Γ, σ are evident, we may just write $\mathcal{D}[[e]]$.

Definition of \mathcal{D} We define $\mathcal{D}[\Gamma \vdash e : \sigma](\alpha) \in \mathcal{D}[\sigma]$, for each $\alpha \in \mathcal{D}[\Gamma]$ by structural induction on e as follows:

$$\begin{aligned}
\mathcal{D}[\Gamma \vdash x_i : \sigma_i](\alpha) &= \alpha_i \\
\mathcal{D}[\Gamma \vdash f(e_1, \dots, e_n) : \sigma](\alpha) &= \mathcal{B}[f](\mathcal{D}[e_1](\alpha), \dots, \mathcal{D}[e_n](\alpha)) \\
\mathcal{D}[\Gamma \vdash \text{let } x : \sigma \text{ be } e \text{ in } e' : \tau](\alpha) &= \mathcal{D}[\Gamma, x : \sigma \vdash e'](\alpha, \mathcal{D}[e](\alpha)) \\
\mathcal{D}[\Gamma \vdash 0_\sigma : \sigma](\alpha) &= 0_{\mathcal{D}[\sigma]} \\
\mathcal{D}[\Gamma \vdash e + e' : \sigma](\alpha) &= \mathcal{D}[e](\alpha) +_{\mathcal{D}[\sigma]} \mathcal{D}[e'](\alpha) \\
\mathcal{D}[\Gamma \vdash -e : \sigma](\alpha) &= -_{\mathcal{D}[\sigma]}(\mathcal{D}[e](\alpha)) \\
\mathcal{D}[\Gamma \vdash \langle e, e' \rangle : \sigma \times \tau](\alpha) &= (\mathcal{D}[e](\alpha), \mathcal{D}[e'](\alpha)) \\
\mathcal{D}[\Gamma \vdash \text{fst}(e) : \sigma](\alpha) &= \pi_0(\mathcal{D}[e](\alpha)) \\
\mathcal{D}[\Gamma \vdash \text{snd}(e) : \tau](\alpha) &= \pi_1(\mathcal{D}[e](\alpha)) \\
\mathcal{D}[\Gamma \vdash * : \text{unit}](\alpha) &= * \\
\mathcal{D}[\Gamma \vdash \text{iter } x : \sigma \text{ to } e \text{ in } e' : \sigma^+](\alpha)_n &= (\lambda a : \mathcal{D}[\sigma]. \mathcal{D}[e](\alpha, a))^n(\mathcal{D}[e'](\alpha)) \\
\mathcal{D}[\Gamma \vdash \text{out}_k(e) : \sigma](\alpha) &= \text{out}_k(\mathcal{D}[e](\alpha))
\end{aligned}$$

The semantics of iteration is in accord with the discussion of the solution of Equation 1 for loops.

4.2 Differential semantics

We next define the differential semantics of our expressions. It has the same form as the ordinary semantics:

$$\mathcal{D}^\delta[\Gamma \vdash e : \sigma] : \mathcal{D}[\Gamma] \longrightarrow \mathcal{D}[\sigma]$$

The semantics of types is not changed from the non-differential case.

First for $f : \sigma_1, \dots, \sigma_n \rightarrow \sigma$ we set

$$\mathcal{B}^\delta[f](\alpha_1, \dots, \alpha_n) = \delta_{\mathcal{D}[\sigma]}(\mathcal{B}[f](S_{\mathcal{D}[\sigma_1]}(\alpha_1), \dots, S_{\mathcal{D}[\sigma_n]}(\alpha_n)))$$

Then \mathcal{D}^δ is defined exactly as for the non-differential case except for iteration and egress where, following the discussion of loops, we set:

$$\begin{aligned}
\mathcal{D}^\delta[\Gamma \vdash \text{iter } x : \sigma \text{ to } e \text{ in } e' : \sigma^+](\alpha)(n) &= \\
&\sum_{n' \leq n} \mu(n', n)(\lambda a : \mathcal{D}[\sigma]. \mathcal{D}^\delta[e](\alpha, a))^{n'}(\mathcal{D}^\delta[e'](\alpha))
\end{aligned}$$

and

$$\mathcal{D}^\delta[\Gamma \vdash \text{out}_k(e) : \sigma](\alpha) = \sum_{n \leq k} \mathcal{D}^\delta[e](\alpha)(n)$$

Theorem 1. (*Correctness of differential semantics*) Suppose $\Gamma \vdash e : \sigma$.
Then:

$$\mathcal{D}^\delta[\Gamma \vdash e : \sigma](\alpha) = \delta_{\mathcal{D}[\sigma]}(\mathcal{D}[\Gamma \vdash e : \sigma](S_{\mathcal{D}[\sigma]}(\alpha)))$$

equivalently:

$$\mathcal{D}^\delta[\Gamma \vdash e : \sigma](\delta_{\mathcal{D}[\sigma]}(\alpha)) = \delta_{\mathcal{D}[\sigma]}(\mathcal{D}[\Gamma \vdash e : \sigma](\alpha))$$

Proof. The first of these equivalent statements is proved by structural induction on expressions. We assume Γ has the form $x_1 : \sigma_1, \dots, x_n : \sigma_n$. We only give the last two cases of the proof.

Iteration:

$$\begin{aligned} & \mathcal{D}^\delta[\Gamma \vdash \text{iter } x : \sigma \text{ to } e \text{ in } e' : \sigma^+](\alpha)(n) \\ &= \sum_{n' \leq n} \mu(n', n) (\lambda a : \mathcal{D}[\sigma]. \mathcal{D}^\delta[e](\alpha, a))^{n'} (\mathcal{D}^\delta[e'](\alpha)) \\ &= \sum_{n' \leq n} \mu(n', n) (\lambda a : \mathcal{D}[\sigma]. \delta(\mathcal{D}[e](S\alpha, Sa)))^{n'} (\delta(\mathcal{D}[e'](S\alpha))) \text{ (by IH)} \\ &= \sum_{n' \leq n} \mu(n', n) (\delta \circ (\lambda a : \mathcal{D}[\sigma]. \mathcal{D}[e](S\alpha, a)) \circ S)^{n'} (\delta(\mathcal{D}[e'](S\alpha))) \\ &= \sum_{n' \leq n} \mu(n', n) \delta((\lambda a : \mathcal{D}[\sigma]. \mathcal{D}[e](S\alpha, a))^{n'} (\mathcal{D}[e'](S\alpha))) \\ &= \sum_{n' \leq n} \mu(n', n) \delta(\mathcal{D}[\text{iter } x : \sigma \text{ to } e \text{ in } e'](S\alpha)(n')) \\ &= \delta(\mathcal{D}[\text{iter } x : \sigma \text{ to } e \text{ in } e'](S\alpha))(n) \end{aligned}$$

Egress:

$$\begin{aligned} \mathcal{D}^\delta[\Gamma \vdash \text{out}_k(e) : \sigma](\alpha) &= \sum_{n \leq k} \mathcal{D}^\delta[e](\alpha)(n) \\ &= \sum_{n \leq k} \delta(\mathcal{D}[e](S\alpha))(n) \text{ (by IH)} \\ &= \sum_{n \leq k} \sum_{n' \leq n} \mu(n', n) \delta(\mathcal{D}[e](S\alpha)(n')) \\ &= \delta(\mathcal{D}[e](S\alpha))(k) \\ &= \delta(\mathcal{D}[\text{out}_k(e)](S\alpha)) \end{aligned}$$

□

5 Priorities

In “prioritized iteration” [2], a sequence of fixed-point computations consumes the input values in batches; each batch consists of the set of values assigned a given priority, and each fixed-point computation starts from the result of the previous one, plus all input values in the next batch.

Such computations can be much more efficient than ordinary iterations, but it was left open in [2] how to implement them correctly for anything more complicated than loop bodies with no nested iteration.

The proposed notion of time was the lexicographic product of \mathbb{N} with any nested T , i.e., the partial order on $\mathbb{N} \times T$ with

$$(e, s) \leq (e', s') \equiv (e < e') \vee (e = e' \wedge s \leq s')$$

where a pair (e, s) is thought of as “stage s in epoch e ”. Unfortunately, the construction in [2] appears incorrect for $T \neq \mathbb{N}$. Moreover, the lexicographic product is not locally finite, so our theory cannot be applied.

We propose instead to avoid these difficulties using a simple generalization of iteration where new input can be introduced at each iteration. One use of this generality is for prioritized iteration, in which elements with priority i are introduced at iteration $i \times k$; this scheme provides exactly k iterations for each priority, before moving to the next priority starting from where the previous priority left off. This is exactly the prioritized iteration strategy from [2] with the fixed-iteration-number loop-egress policy, but cast in a framework where we can verify its correctness.

The generalisation of Equation 1 is:

$$d = c + \mathbf{fb}(f^{\mathbb{N}}(d)) \tag{3}$$

where now c is in $G^{\mathbb{N}}$ (rather than in G , and placed at iteration 0 by **in**). This equation is equivalent to the equations:

$$d_0 = c_0 \quad d_{n+1} = c_{n+1} + f(d_n)$$

so has a unique solution s . Differentiating Equation 3, we obtain:

$$d = \delta(c) + \mathbf{fb}(\delta(f^{\mathbb{N}}))(d)$$

By the remark in Section 3 on fixed-points of function differentials, this also has a unique solution, viz. $\delta(s)$.

To adapt the language, one can simply change the typing rule for the iteration construct to:

$$\frac{\Gamma, x : \sigma \vdash e : \sigma^+ \quad \Gamma \vdash e' : \sigma}{\Gamma \vdash \mathbf{iter} \ x : \sigma \ \mathbf{to} \ e \ \mathbf{in} \ e' : \sigma^+}$$

We assume the ingress function is available as a built-in function; other built-in functions can enable the use of priority functions. The semantics of this version of iteration is given by:

$$\mathcal{D}[\Gamma \vdash \mathbf{iter} \ x : \sigma \ \mathbf{to} \ e \ \mathbf{in} \ e' : \sigma^+](\alpha) = \mu d : \mathcal{D}[\sigma^+]. \mathcal{D}[e](\alpha) + \mathbf{fb}(\mathcal{D}[e](\alpha, d))$$

where we are making use of the usual notation for fixed points; that is justified here by the discussion of Equation 3. The differential semantics has exactly the same form, and Theorem 1 extends.

6 Discussion

We have given mathematical foundations for differential dataflow, introduced in [2]. By accounting for differentials using Möbius inversion, we systematically justified various operator and loop differentials discussed there. Using the theory we could also distinguish the difficult case of lexicographic products, and justify an alternative.

Via a schematic language we showed that a differential semantics was the differential of the ordinary semantics, verifying the intuition that to compute the differential of a computation, one only changes how individual operators are computed, but not its overall shape. (We could have given a more concrete language with selection and other such operators, but we felt our more schematic approach would bring out the underlying ideas more clearly.)

There are some natural possibilities for further work. As mentioned above, it would be interesting to formulate a small-step operational semantics that propagates differences in a dataflow graph; one could then prove a soundness theorem linking it to the above denotational semantics.

It would also be interesting to consider the egress policy of exiting on reaching a fixed point, that is at the first k such that $d_k = d_{k+1}$, where d is the output stream. As no such time may exist, one is naturally led to consider partial streams, as mentioned above. This would need a theory of Möbius inversion for partial functions. It would also give the possibility, via standard domain theory, of a general recursion construct, so of more general loops. Finally, as regards priorities, one wonders if the approach with lexicographic products can be rescued.

References

1. Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A. Acar, and Rafael Pasquin, Incoop: MapReduce for incremental computations, *Proc. 2nd ACM Symposium on Cloud Computing*, 7pp., 2011.
2. Frank McSherry, Derek Gordon Murray, Rebecca Isaacs, and Michael Isard, Differential dataflow, *Proc. Sixth Biennial Conference on Innovative Data Systems Research*, www.cidrdb.org, 2013.
3. Svilen R. Mihaylov, Zachary G. Ives, and Sudipto Guha, REX: recursive, delta-based data-centric computation, *Proc. VLDB Endowment*, **5**(11), 1280–1291, 2012.
4. Derek Gordon Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi, Naiad: a timely dataflow system, *Proc. ACM SIGOPS 24th. Symposium on Operating Systems Principles* (eds. Michael Kaminsky and Mike Dahlin), 439–455, 2013.
5. Gian-Carlo Rota, On the foundations of combinatorial theory I, Theory of Möbius functions, *Probability theory and related fields*, **2**(4), 340–368, 1964.
6. Richard P. Stanley, *Enumerative Combinatorics*, Vol. 1, CUP, 2011.

Appendix: Proofs

We give the proofs omitted from the paper.

Proof of Proposition 2

Proof. 1. We calculate:

$$\begin{aligned}
\delta(f^T)(c, d)_t &= \sum_{t' \leq t} \mu(t', t) \delta_K(f^T(S_{G^T}(c), S_{H^T}(d))_{t'}) \\
&= \sum_{t' \leq t} \mu(t', t) \delta_K(f(S_{G^T}(c)_{t'}, S_{H^T}(d)_{t'})) \\
&= \sum_{t' \leq t} \mu(t', t) \delta_K(f(\sum_{t'' \leq t'} S_G(c)_{t''}, \sum_{t'' \leq t'} S_H(d)_{t''})) \\
&= \sum_{t' \leq t} \mu(t', t) \delta_K(f(S_{G \times K}(\sum_{t'' \leq t'} c_{t''}, \sum_{t'' \leq t'} d_{t''}))) \\
&= \sum_{t' \leq t} \mu(t', t) \delta(f)(\sum_{t'' \leq t'} c_{t''}, \sum_{t'' \leq t'} d_{t''})
\end{aligned}$$

2. Continuing the previous calculation, now using the bilinearity of f , we have:

$$\begin{aligned}
\delta(f^T)(c)_t &= \sum_{t' \leq t} \mu(t', t) \delta(f)(\sum_{t'' \leq t'} c_{t''}, \sum_{t'' \leq t'} d_{t''}) \\
&= \sum_{t' \leq t} \mu(t', t) \sum_{r \leq t', s \leq t'} \delta(f)(c_r, d_s) \\
&= \sum_{t' \leq t} \mu(t', t) \sum_{t'' \leq t'} \sum \{\delta(f)(c_r, d_s) \mid r \vee s = t''\} \\
&= \sum \{\delta(f)(c_r, d_s) \mid r \vee s = t\}
\end{aligned}$$

3. This is an immediate consequence of the previous part. □

Proof of Theorem 1

Proof. We give the remaining cases of the proof.

Case 1. e is x_i

$$\begin{aligned}
\mathcal{D}^\delta[\Gamma \vdash x_i : \sigma](\alpha) &= \alpha_i \\
&= \delta(S(\alpha_i))
\end{aligned}$$

Case 2. e is $f(e_1, \dots, e_n)$

$$\begin{aligned}
\mathcal{D}^\delta[\Gamma \vdash f(e_1, \dots, e_n) : \sigma](\alpha) &= \mathcal{B}^\delta[f](\mathcal{D}^\delta[e_1](\alpha), \dots, \mathcal{D}^\delta[e_n](\alpha)) \\
&= \delta(\mathcal{B}[f](S\delta(\mathcal{D}[e_1](S\alpha)), \dots, S\delta(\mathcal{D}[e_n](S\alpha)))) \quad (\text{by IH}) \\
&= \delta(\mathcal{B}[f](\mathcal{D}[e_1](S\alpha), \dots, \mathcal{D}[e_n](S\alpha))) \\
&= \delta(\mathcal{D}[f(e_1, \dots, e_n)](S\alpha))
\end{aligned}$$

Case 3. e is $\text{let } x : \sigma \text{ be } e \text{ in } e'$

$$\begin{aligned}
\mathcal{D}^\delta[\Gamma \vdash \text{let } x : \sigma \text{ be } e \text{ in } e' : \sigma](\alpha) &= \mathcal{D}^\delta[\Gamma, x : \sigma \vdash e'](\alpha, \mathcal{D}^\delta[e](\alpha)) \\
&= \delta(\mathcal{D}[\Gamma, x : \sigma \vdash e'](S\alpha, S\delta\mathcal{D}[e](S\alpha))) \quad (\text{by IH}) \\
&= \delta(\mathcal{D}[\Gamma, x : \sigma \vdash e'](S\alpha, \mathcal{D}[e](S\alpha))) \\
&= \delta(\mathcal{D}[\Gamma \vdash \text{let } x : \sigma \text{ be } e \text{ in } e'](S\alpha))
\end{aligned}$$

Cases 4, 5, and 6. These are all much the same. We only give case 5.

$$\begin{aligned}
\mathcal{D}^\delta[\Gamma \vdash e + e' : \sigma](\alpha) &= \mathcal{D}^\delta[e](\alpha) + \mathcal{D}^\delta[e'](\alpha) \\
&= \delta(\mathcal{D}^\delta([e](S\alpha)) + \delta(\mathcal{D}^\delta[e'](S\alpha))) \quad (\text{by IH}) \\
&= \delta(\mathcal{D}[e + e'](S\alpha))
\end{aligned}$$

Case 7.

$$\begin{aligned}
\mathcal{D}^\delta[\Gamma \vdash \langle e, e' \rangle : \sigma \times \tau](\alpha) &= (\mathcal{D}^\delta[e](\alpha), \mathcal{D}^\delta[e'](\alpha)) \\
&= (\delta(\mathcal{D}^\delta[e](S\alpha)), \delta(\mathcal{D}^\delta[e'](S\alpha))) \quad (\text{by IH}) \\
&= \delta((\mathcal{D}^\delta[e](S\alpha), \mathcal{D}^\delta[e'](S\alpha))) \\
&= \delta(\mathcal{D}^\delta[\langle e, e' \rangle](S\alpha))
\end{aligned}$$

Cases 8 and 9. Only case 8 is shown.

$$\begin{aligned}
\mathcal{D}^\delta[\Gamma \vdash \text{fst}(e) : \sigma](\alpha) &= \pi_0(\mathcal{D}^\delta[e](\alpha)) \\
&= \pi_0(\delta(\mathcal{D}[e](S\alpha))) \quad (\text{by IH}) \\
&= \delta(\pi_0(\mathcal{D}[e](S\alpha))) \\
&= \delta(\mathcal{D}[\text{fst}(e)](S\alpha))
\end{aligned}$$

Case 10. This is trivial, so omitted.

□