Multilinear Programming with Big Data

Mihai Budiu Gordon D. Plotkin
Microsoft Research

Abstract

Systems such as MapReduce have become enormously popular for processing massive data sets since they substantially simplify the task of writing many naturally parallelizable parallel programs. In this paper we identify the computations carried out by such programs as *linear transformations* on distributed collections. To this end we model collections as multisets with a union operation, giving rise to a commutative monoid structure. The results of the computations (e.g., filtering, reduction) also lie in such monoids, (e.g., multisets with union, or the natural numbers with addition). The computations are then modelled as linear (i.e., homomorphic) transformations between the commutative monoids. Binary computations such as join are modelled in this framework by *multilinear* transformations, i.e., functions of several variables, linear in each argument.

We present a typed higher-order language for writing multilinear transformations; the intention is that all computations written in such a programming language are naturally parallelizable. The language provides a rich assortment of collection types, including collections whose elements are negatively or fractionally present (in general it permits modules over any given semiring). The type system segregates computations into linear and nonlinear phases, thereby enabling them to "switch" between different commutative monoids over the same underlying set (for example between addition and multiplication on real numbers). We use our language to derive linear versions of standard computations on collections; we also give several examples, including a linear version of MapReduce.

1 Introduction

As has been famously demonstrated by MapReduce, Dean and Ghemawat (2004), and followed up by related systems, such as DryadLINQ, Yu et al. (2008), big data computations can be accelerated by using massive parallelism. Parallelization is justified for simple mathematical reasons: big data has a natural commutative

monoid structure with respect to which the transformations carried out by computations are linear (i.e., homomorphic). We present a programming language which seeks to expose this linearity; we intend thereby to lay the foundations for (multi)linear programming with big data.

Our language manipulates two kinds of types: ordinary and linear. In our setting linear types are commutative monoids (i.e., sets with a commutative associative operation with a zero). A typical example of such a monoid is provided by the positive reals $\mathbf{R}^+$ with addition.

Big data is usually manipulated as collections; these are unordered bags, or multisets, of values (sometimes represented as lists); we write $X^\star$ for the type of collections of elements of a set X. While the elements of a collection may be from an ordinary type, the collection type itself is a commutative monoid if endowed with multiset union (indeed $X^\star$ is the free commutative monoid over X).

Turning to transformations, given a function $f : X \to Y$ between ordinary types the Map operator yields a transformation $\mathrm{Map}(f) : X^\star \to Y^\star$ mapping X-collections to Y-collections. Again, given $g : Y \to \mathbf{R}^+$, the Reduce operator yields a transformation $\mathrm{Reduce}(g) : Y^\star \to \mathbf{R}^+$ mapping Y-collections to $\mathbf{R}^+$. Both of these are linear, preserving the monoid structures, i.e., we have:

$$\begin{array}{lcllcl} \mathrm{Map}(f)(\emptyset) & = & \emptyset & \mathrm{Map}(f)(c \cup c') & = & \mathrm{Map}(f)(c) \cup \mathrm{Map}(f)(c') \\ \mathrm{Reduce}(g)(\emptyset) & = & 0 & \mathrm{Reduce}(g)(c \cup c') & = & \mathrm{Reduce}(g)(c) + \mathrm{Reduce}(g)(c') \end{array}$$

(In fact, since $X^\star$ is the free commutative monoid over X, these are the unique such maps extending f and g, respectively.)

These equations justify the use of parallelism. For example, the linearity of Map implies that one can split a collection into two parts, map them in parallel, and combine the results to obtain the correct result.

As another example, suppose we have a binary tree of processors, and a collection c partitioned across the leaves of the tree. We map and then reduce at the leaves, and then reduce the results at the internal nodes. The final result is $\mathrm{Reduce}(g)(\mathrm{Map}(f)(c))$ irrespective of the data distribution at leaves, and of the shape of the tree. This fact depends on both the associativity and commutativity of the monoid operations and the linearity of the transformations; in practice this translates into the ability to do arbitrary load balancing of computations.

Our language is typed and higher-order. The language accommodates binary functions, such as joins, which have multilinear types (they are linear in each of their arguments). The language provides rich collection type constructors: in particular, for any linear types A and (certain) ordinary types X, we can construct the linear type $A[X]$ whose elements can be thought of variously as *A-ary X-collections*, or as *key-value dictionaries*, with X as the type of keys (or indices) and A as the type of values.

For example, by taking A to be the integers, we obtain multisets with elements having positive and negative counts; these are useful in modelling differential dataflow computations, see McSherry et al. (2013). Taking A to be the nonnegative reals, we obtain weighted collections, which are useful for modelling differential privacy, see Prospero et al. (2014). Dictionaries enable one to express GroupBy computations. The language further provides a mechanism for programming computations with both linear and nonlinear phases, possibly switching between different commutative monoids over the same carrier.

In the rest of this paper, after some remarks on commutative monoids, we present the syntax and denotational semantics of our language. We then argue practicality by modelling MapReduce and LINQ distributed computations through a series of examples (see Meijer et al. (2006) for an account of LINQ).

2 Remarks on commutative monoids

We work with commutative monoids $M = (|M|, +, 0)$ and linear (i.e., homomorphic functions) between them. We write $U(M)$ for $|M|$, the *carrier* of M (i.e., its underlying set). For any $n \in \mathbb{N}$ and $m \in M$ we write nm for the sum of m with itself n times.

The product of two commutative monoids is another, with addition and zero defined coordinatewise. Various sets of functions with range a commutative monoid M also form commutative monoids, with addition and zero defined pointwise. Examples include: $M[X]$ the monoid of all functions from a given set X to M which are zero except, possibly, at finitely many arguments; $X \to M$, the monoid of all functions from a given set X to M; and $M_1, \ldots, M_n \multimap M$ the monoid of all multilinear functions from given commutative monoids $M_1, \ldots, M_n$ to M. We write a typical element of $A[X]$ with value 0 except possibly at n arguments $x_1, \ldots, x_n$ as $\{x_1 \mapsto a_1, \ldots, x_n \mapsto a_n\}$.

Categorically, U is (the object part of) the forgetful functor to the category of sets. The product of two commutative monoids is also their sum, and so we employ the biproduct notation $M_1 \oplus M_2$. The commutative monoid $M[X]$ is the categorical sum $\sum_{x \in X} M$ and can also be viewed as the tensor $X \otimes M$ (the corresponding cotensor is $X \to M$).

$$\frac{X = Y}{Y = X} \qquad b = b \qquad |\mathrm{c}| = \mathrm{U}(\mathrm{c})$$

$$\frac{X = Y \quad X' = Y'}{X \times X' = Y \times Y'} \qquad \frac{A = B}{\mathrm{U}(A) = \mathrm{U}(B)} \qquad \frac{X = X' \quad Y = Y'}{X \to Y = X' \to Y'}$$

$$\frac{X = \mathrm{U}(A) \quad Y = \mathrm{U}(B)}{X \times Y = \mathrm{U}(A \oplus B)} \qquad \frac{X = X' \quad Y = \mathrm{U}(A)}{X \to Y = \mathrm{U}(X' \to A)}$$

Figure 1: Definitional equality rules for ordinary types.

$$\mathrm{c} = \mathrm{c} \qquad \frac{A = A' \quad B = B'}{A \oplus B = A' \oplus B'} \qquad \frac{A = A' \quad X = X'}{A[X] = A'[X']}$$

$$\frac{X = X' \quad A = A'}{X \to A = X' \to A'} \qquad \frac{\vec{A} = \vec{A'} \quad B = B'}{\vec{A} \multimap B = \vec{A'} \multimap B'}$$

Figure 2: Definitional equality rules for linear types.

3 The language

Types

The language has two kinds of type expressions: *ordinary* and *linear*, ranged over by $X, Y, \ldots$ and $A, B, \ldots$, respectively. They are given by:

$$X \quad ::= \quad \mathrm{b} \mid X \times Y \mid \mathrm{U}(A) \mid X \to Y$$

$$A \quad ::= \quad \mathrm{c} \mid A \oplus B \mid A[X] \mid X \to B \mid A_1, \ldots, A_m \multimap B$$

where b and c range over given *basic* ordinary and linear types, respectively. The basic ordinary types always contain bool and nat. The basic linear types always contain nat_+; other possibilities are $\mathrm{nat}_{\max}$ and real_+. In $A[X]$ we restrict X to be an *equality type*, meaning one not containing any function types.

We also assume given a syntactic *carrier* function $|\cdot|$, mapping basic linear types c to basic ordinary types $|\mathrm{c}|$. For example $|\mathrm{nat}_+| = |\mathrm{nat}_{\max}| = \mathrm{nat}$. This is used to obtain a notion of definitional equality of types which will enable computations to move between different linear structures on the same carrier; the rules for definitional equality are given in Figures 1 and 2; note the use there of vector notation for sequences of linear types.

Semantics of Types

Ordinary types X denote sets $[\![X]\!]$ and linear types A denote commutative monoids $[\![A]\!]$. The denotations of basic type expressions are assumed to be given. For example, bool and nat would denote, respectively, the booleans and the natural numbers; nat_+, would denote the natural numbers with addition; and $\mathrm{nat}_{\max}$ and real_+ would denote the natural numbers with maximum, and the reals with addition. We assume, for any basic linear type c that $[\![|c|]\!]$ is $U([\![c]\!])$ the carrier of $[\![c]\!]$.

The other type expressions have evident denotations. For example $[\![X \to Y]\!]$ is the set of all functions from $[\![X]\!]$ to $[\![Y]\!]$; $[\![\mathrm{U}(A)]\!]$ is $U([\![A]\!])$; $[\![A[X]]\!]$ is $[\![A]\!][[\![X]\!]]$; $[\![A_1, \ldots, A_n \multimap B]\!]$ is $[\![A_1]\!], \ldots, , [\![A_n]\!] \multimap [\![B]\!]$; and so on. One can check that definitionally equal types have equal denotations.

Terms

The language has *ordinary* terms ranged over by $t, u, \ldots$ and *multilinear* terms ranged over by $M, N, \ldots$. They are given by:

$$
\begin{array}{rcl}
t & ::= & x \mid \mathrm{d}_X(M) \mid f(t_1, \ldots, t_n) \mid \\
& & \langle t, u \rangle \mid \ \mathrm{fst}(t) \mid \ \mathrm{snd}(t) \mid \\
& & \lambda x : X.\, t \mid t(u)
\end{array}
$$

$$
\begin{array}{rcl}
M & ::= & a \mid \mathrm{u}_A(t) \mid g(M_1, \ldots, M_n) \mid \\
& & 0_A \mid M + N \mid MN \mid \\
& & \text{if } t \text{ then } M \text{ else } N \mid \text{match } x : X, y : Y \text{ as } t \text{ in } M \mid \\
& & \langle M, N \rangle \mid \ \mathrm{fst}(M) \mid \ \mathrm{snd}(M) \mid \\
& & M \cdot t \mid \text{sum } a : A, x : X \text{ in } M.\, N \mid \\
& & \lambda x : X.\, M \mid M(t) \mid \\
& & \lambda a_1 : A_1, \ldots, a_n : A_m.\, M \mid M(N_1, \ldots, N_n)
\end{array}
$$

In the above we use the letters $x, y, \ldots, a, b \ldots$ to range over variables. In the "match" construction x, y have scope extending over M; and in the "sum" construction a, x have scope extending over N. We assume given two signatures: one of ordinary basic functions $f : \mathrm{b}_1, \ldots, \mathrm{b}_n \to \mathrm{b}$ and the other of linear basic functions $g : \mathrm{c}_1, \ldots, \mathrm{c}_n \to \mathrm{c}$.

We introduce three "let" constructions as standard syntactic sugar:

$$
\begin{array}{lcl}
\text{let } x : X \text{ be } t \text{ in } u & =_{\mathrm{def}} & (\lambda x : X.\, u)(t) \\
\text{let } x : X \text{ be } t \text{ in } M & =_{\mathrm{def}} & (\lambda x : X.\, M)(t) \\
\text{let } \vec{a} : \vec{A} \text{ be } \vec{M} \text{ in } N & =_{\mathrm{def}} & (\lambda \vec{a} : \vec{A}.\, N)(\vec{M})
\end{array}
$$

Instead of sum $a : A, x : X$ in $M.\, N : B$ we may write in a more "mathematical" way:

$$\sum_{a \cdot x \in M} N$$

Finally we may write unary function applications $M(N)$ in an "argument-first" manner, as $N.M$, associating such applications to the left.

Environments

The language has *ordinary* environments ranged over by Γ and *multilinear* environments ranged over by Δ. These environments are sequences of variable bindings of respective forms:

$$\Gamma ::= x_1 : X_1, \ldots, x_m : X_n \qquad \Delta ::= a_1 : A_1, \ldots, a_n : A_n$$

where the x_i are all different, as are the a_j. Below we write $\Delta || \Delta'$ for the set of all merges (interleavings) of the two sequences of variable bindings Δ and Δ'.

Typing Rules

We have two kinds of judgements, *ordinary* and *multilinear*

$$\Gamma \vdash t : X \qquad \text{and} \qquad \Gamma \mid \Delta \vdash M : A$$

where, in the latter, Γ and Δ have no variables in common. The rules are either structural, casting, ordinary, or multilinear, and are as follows:

Structural

$$\Gamma, x : X, \Gamma' \vdash x : X \qquad \Gamma \mid a : A \vdash a : A$$

Casting

$$\frac{\Gamma \mid \vdash M : A \quad \mathrm{U}(A) = X}{\Gamma \vdash \mathrm{d}_X(M) : X} \qquad \frac{\Gamma \vdash t : X \quad X = \mathrm{U}(A)}{\Gamma \mid \vdash \mathrm{u}_A(t) : A}$$

Ordinary

$$\frac{\Gamma \vdash \vec{\mathrm{t}} : \vec{\mathrm{b}}}{\Gamma \vdash f(\vec{\mathrm{t}}) : \mathrm{b}} \ (f : \vec{\mathrm{b}} \to \mathrm{b})$$

$$\frac{\Gamma \vdash t : X \quad \Gamma \vdash u : Y}{\Gamma \vdash \langle t, u \rangle : X \times Y} \qquad \frac{\Gamma \vdash t : X \times Y}{\Gamma \vdash \ \mathrm{fst}(t) : X} \qquad \frac{\Gamma \vdash t : X \times Y}{\Gamma \vdash \ \mathrm{snd}(t) : Y}$$

$$\frac{\Gamma, x : X \vdash t : Y}{\Gamma \vdash \lambda x : X.\, t : X \to Y} \qquad \frac{\Gamma \vdash t : X \to Y \quad \Gamma \vdash u : X}{\Gamma \vdash t(u) : Y}$$

Linear

$$\frac{\Gamma \mid \Delta_i \vdash M_i : \mathrm{c}_i \quad (i = 1, n)}{\Gamma \mid \Delta \vdash g(M_1, \ldots, M_n) : \mathrm{c}} \quad (g : \mathrm{c}_1, \ldots, \mathrm{c}_n \to \mathrm{c}, \Delta \in \Delta_1 || \ldots || \Delta_n)$$

$$\Gamma \mid \Delta \vdash 0_A : A \qquad \frac{\Gamma \mid \Delta \vdash M : A \quad \Gamma \mid \Delta \vdash N : A}{\Gamma \mid \Delta \vdash M + N : A}$$

$$\frac{\Gamma \mid \Delta' \vdash M : \mathrm{nat}_+ \quad \Gamma \mid \Delta'' \vdash N : A}{\Gamma \mid \Delta \vdash MN : A} \quad (\Delta \in \Delta' || \Delta'')$$

$$\frac{\Gamma \vdash t : \mathrm{bool} \quad \Gamma \mid \Delta \vdash M : A \quad \Gamma \mid \Delta \vdash N : A}{\Gamma \mid \Delta \vdash \text{if } t \text{ then } M \text{ else } N : A} \qquad \frac{\Gamma \vdash t : X \times Y \quad \Gamma, x : X, y : Y \mid \Delta \vdash M : A}{\Gamma \mid \Delta \vdash \text{match } x : X, y : Y \text{ as } t \text{ in } M : A}$$

$$\frac{\Gamma \mid \Delta \vdash M : A \quad \Gamma \mid \Delta \vdash N : B}{\Gamma \mid \Delta \vdash \langle M, N \rangle : A \oplus B} \qquad \frac{\Gamma \mid \Delta \vdash M : A \oplus B}{\Gamma \mid \Delta \vdash \ \mathrm{fst}(M) : A} \qquad \frac{\Gamma \mid \Delta \vdash M : A \oplus B}{\Gamma \mid \Delta \vdash \ \mathrm{snd}(M) : B}$$

$$\frac{\Gamma \mid \Delta \vdash M : A \quad \Gamma \vdash t : X}{\Gamma \mid \Delta \vdash M \cdot t : A[X]} \qquad \frac{\Gamma \mid \Delta' \vdash M : A[X] \quad \Gamma, x : X \mid \Delta'', a : A \vdash N : B}{\Gamma \mid \Delta \vdash \text{sum } a : A, x : X \text{ in } M.\, N : B} \quad (\Delta \in \Delta' || \Delta'')$$

$$\frac{\Gamma, x : X \mid \Delta \vdash M : B}{\Gamma \mid \Delta \vdash \lambda x : X.\, M : X \to B} \qquad \frac{\Gamma \mid \Delta \vdash M : X \to B \quad \Gamma \vdash t : X}{\Gamma \mid \Delta \vdash M(t) : B}$$

$$\frac{\Gamma \mid \Delta, \vec{a} : \vec{A} \vdash M : B}{\Gamma \mid \Delta \vdash \lambda \vec{a} : \vec{A}.\, M : \vec{A} \multimap B}$$

$$\frac{\Gamma \mid \Delta' \vdash M : A_1, \ldots, A_n \multimap B \qquad \Gamma \mid \Delta_i \vdash N_i : A_i \quad (i = 1, n)}{\Gamma \mid \Delta \vdash M(N_1, \ldots, N_n) : B} \quad (\Delta \in \Delta' || \Delta_1 || \ldots || \Delta_n)$$

The use of the merge operator $||$ on linear environments ensures that derivable typing judgments are closed under permutation of linear environments; as may be expected, they are not closed under weakening or duplication. Typing is unique in that for any Γ, Δ and M there is at most one A such that $\Gamma \mid \Delta \vdash M : A$ and similarly for judgments $\Gamma \vdash t : X$. There is also a natural top-down type-checking algorithm.

We sketch the denotational semantics of terms below, but their intended meaning should be clear from the previous section. For example, the term MN with

$M : \mathrm{nat}_+$ indicates the addition of N with itself M times. The terms $\mathrm{d}_X(M)$ and $\mathrm{u}_A(t)$ should be read as "down" and "up" casts, which convert back and forth between a linear type A and an ordinary type X definitionally equal to $\mathrm{U}(A)$. Using terms of the forms $\mathrm{d}_X(M)$, $\mathrm{u}_A(t)$ one can construct conversions between any two definitionally equal types.

Some constructions that may seem missing from the biproduct are in fact definable. The first injection $\mathrm{inl}(M)$ can be defined by $\langle M, 0\rangle$, similarly for the second, and we can define a cases construction by:

$$\begin{array}{lll} \text{cases } K \text{ fst } a : A.\, M,\ \text{snd } b : B.\, N & =_{\mathrm{def}} & \text{let } c : A \oplus B \text{ be } K \text{ in} \\ & & (\text{let } a : A \text{ be } \mathrm{fst}(c) \text{ in } M) \\ & & + (\text{let } b : B \text{ be } \mathrm{snd}(c) \text{ in } N) \end{array}$$

So given "product" and "plus" we get "sum"; in fact, given any two of "product", "sum", and "plus" one can define the third.

Semantics of terms

For the basic functions, f, g, one assumes available given functions $[\![f]\!]$, $[\![g]\!]$ of suitable types.

For environments $\Gamma = x_1 : X_1, \ldots, x_m : X_m$ and $\Delta = a_1 : A_1, \ldots, a_n : A_n$, we write $[\![\Gamma]\!]$ for the set $[\![X_1]\!] \times \ldots \times [\![X_m]\!]$, and $[\![\Delta]\!]$ for the carrier of $[\![A_1]\!] \times \ldots \times [\![A_n]\!]$, respectively. Then, much as usual, the denotational semantics assigns to each typing judgement $\Gamma \vdash t : Y$ a function

$$[\![\Gamma \vdash t : Y]\!] : [\![\Gamma]\!] \longrightarrow [\![Y]\!]$$

and to each typing judgement $\Gamma \mid \Delta \vdash M : B$ a function

$$[\![\Gamma \mid \Delta \vdash M : B]\!] : [\![\Gamma]\!] \times [\![\Delta]\!] \longrightarrow [\![B]\!]$$

linear in each of the Δ coordinates (this is why Δ is called a "multilinear environment"). The definition is by structural definition on the terms; we just illustrate a few cases.

The type conversions are modelled by the identity function, for example:

$$[\![\Gamma \mid \vdash \mathrm{d}_X(M) : X]\!](\vec{v}, \vec{\alpha}) = [\![\Gamma \vdash M : A]\!](\vec{v})$$

As one might expect the syntactic monoid operations are modelled by the semantic ones, for example:

$$[\![\Gamma \mid \Delta \vdash M{+}N : B]\!](\vec{v}, \vec{\alpha}) = [\![\Gamma \mid \Delta \vdash M : B]\!](\vec{v}, \vec{\alpha}) +_{[\![B]\!]} [\![\Gamma \mid \Delta \vdash N : B]\!](\vec{v}, \vec{\alpha})$$

For the collection syntax we have first that:

$$[\![\Gamma \mid \Delta \vdash M \cdot t : A[X]]\!](\vec{v}, \vec{\alpha}) = \{[\![\Gamma \vdash t : X]\!](\vec{v}) \mapsto [\![\Gamma \mid \Delta \vdash M : A]\!](\vec{v}, \vec{\alpha})\}$$

Next if $\Gamma \mid \ \Delta \vdash \text{sum } a : A, x : X \text{ in } M. N : X$ holds then there are, necessarily unique, Δ', Δ'' such that $\Gamma \mid \Delta' \vdash M : A[X]$ and $\Gamma, x : X \mid \Delta'', a : A \vdash N : B$ and $\Delta \in \Delta' || \Delta''$ all hold. We use the fact that $\Delta \in \Delta' || \Delta''$ to obtain canonical projections $\pi' : [\![\Delta]\!] \to [\![\Delta']\!]$ and $\pi'' : [\![\Delta]\!] \to [\![\Delta'']\!]$.

Suppose that

$$[\![\Gamma \mid \Delta' \vdash M : A[X]]\!](\vec{v}, \pi'(\vec{\alpha})) = \{v_1 \mapsto a_1, \ldots, v_n \mapsto a_n\}$$

Then

$$[\![\Gamma \mid \vdash \text{sum } a : A, x : X \text{ in } M. Nt : X]\!](\vec{v}, \vec{\alpha}) \quad = \\ \textstyle\sum_{i=1,n} [\![\Gamma, x : X \mid \Delta'', a : A \vdash N : B]\!]((\vec{v}, v_i), (\pi''(\vec{\alpha}), a_i))$$

The semantics of the other terms pose no surprises; in cases where linear environments Δ are split up, one again makes use of canonically available projections.

Implementation considerations

It very much remains to be seen how useful our ideas prove. In the meantime, it seems worthwhile saying a little about possible implementation datatypes. One could use lists, possibly spread among different processors, to represent collections. Representations would be recursively defined: if R represented X, and S represented the carrier of A then $(R \times S)^*$ could represent $A[X]$, with $(r_1, s_1) \ldots (r_n, s_n)$ representing $\sum_{i=1,n}\{x_i \mapsto a_i\}$ if r_i, s_i represented x_i, a_i, for all $i \in \{1, \ldots, n\}$.

Such representations have a natural normal form: assuming the r_i and s_i are already in normal form, one adds the s_i together (using a representation of addition on A) to produce a list $(r'_1, s'_1), \ldots, (r'_n, s'_n)$ with the r'_j all different, and then orders the list using a total ordering of S, itself recursively defined.

When evaluating $\mathrm{u}_A(t)$ one needs to have the value of t in normal form, as otherwise the addition implied by the representation relation is that of A, which may not generally be correct (for example, t may itself be $\mathrm{d}_X(M)$ where the (linear) type of M has a different addition from that of A). So when evaluating $\mathrm{d}_X(M)$, one should put the value of M into normal form as the correct addition is then known from the linear type of M.

4 Operators

As stated above, the type $A[X]$ can be regarded variously as that of A-valued X-collections or of key-value dictionaries over A and indexed by X. In particular, taking A to be nat_+ we get the usual unordered collections, i.e., finite multisets of elements of X; we write this type as $X^\star$. We now look at linear versions of standard operators such as Map, Fold, Reduce, GroupBy, and Join. As our notion of collection is more general than that used in traditional programming languages we obtain corresponding generalisations of these operators.

Map

We can define a family of Map operators which operate on both the elements of a collection and their coefficients. Associating function type arrows to the right, they have type

$$(A \multimap B) \multimap (X \to Y) \to (A[X] \multimap B[Y])$$

and are given by:

$$\mathrm{Map}_{X,Y,A,B} =_{\mathrm{def}} \lambda f : (A \multimap B).\, \lambda g : (X \to Y).\, \lambda c : A[X].\, \sum_{a \cdot x \in c} f(a) \cdot g(x)$$

where we are making use of the summation notation introduced above. Note that here, and below, operators are often linear in their function arguments.

Specialising to the case where $B = A$ and $f : A \multimap B$ is the identity id_A (i.e., $\lambda a : A.\, a$), we obtain a family of operators

$$\mathrm{Map}_{X,Y,A} : (X \to Y) \to (A[X] \multimap A[Y])$$

where we are overloading notation. When $A = \mathrm{nat}_+$ these are the usual Map operators, but with their linearity made explicit in their type:

$$(X \to Y) \to (X^\star \multimap Y^\star)$$

Actions and their extensions

We define an *action (term)* of a linear type A on another B to be a term of type $A, B \multimap B$. Such an action always exists when $A = \mathrm{nat}_+$, viz., the term $\lambda n : \mathrm{nat}_+, b : B.\, nb$. In general, we may only be given "multiplication" terms $m_A : A, A \multimap A$ providing an action of A on itself; we may then, as we will see below, use the given multiplication to obtain actions on other linear types.

For example in the case of real_+, the multiplication term would denote the usual multiplication on the positive reals. When we have a multiplication term on a linear type A we may also have a "unit" term $1_A : A$ (e.g., a term denoting the usual unit in the case of the positive reals). The unit provides a generalisation of the multiset singleton map $\{-\} : X \to X^\star$, namely $\lambda x : X.\, 1_A x : X \to A[X]$.[1]

Given an action of A on B we can obtain an action of A on $B[X]$ using a family of Extend operators. They have type

$$(A, B \multimap B) \multimap (A, B[X] \multimap B[X])$$

and are given by:

$$\mathrm{Extend}_{X,A,B} =_{\mathrm{def}} \lambda f : (A, B \multimap B).\, \lambda a : A, c : B[X].\, \sum_{b \cdot x \in c} f(a, b) \cdot x$$

Actions can be extended to other types. In the case of biproducts, given an action of A on both B and C, then there is an action of A on $B \oplus C$; in the case of function types, given an action of A on C, there are actions of A on $X \to C$ and $\vec{B} \multimap C$. We leave their definition as an exercise for the reader. Combining such extensions, one can build up actions on complex datatypes.

Folding

We define a family of Fold operators with type

$$(A, B \multimap B), (X \to B) \multimap (A[X] \multimap B)$$

They are given by:

$$\mathrm{Fold}_{X,A,B} =_{\mathrm{def}} \lambda m : (A, B \multimap B), f : X \to B.\, \lambda c : A[X].\, \sum_{a \cdot x \in c} m(a, f(x))$$

Note that the fold operator needs an action of of A on B.

SelectMany Using Fold we can define a family of SelectMany operators that generalise those of LINQ analogously to the above Map operators. They have type

$$(A \multimap B), (X \to B[X]) \multimap (A[X] \multimap B[X])$$

[1] We would expect such a multiplication and unit to make A a semiring (i.e., to provide a bilinear associative multiplication operation with a unit) and we would expect the actions of A on other linear types to make them A-modules. If such algebraic assumptions are fulfilled, some natural program equivalences hold.

and are given by:

$$\begin{array}{rcl}\mathrm{SelectMany}_{X,A,B} & = & \lambda f : A \multimap B, g : X \to B[X].\, \lambda c : A[X]. \\ & & \quad \mathrm{let}\ e : A, B[X] \multimap B[X]\ \mathrm{be} \\ & & \quad\ \mathrm{Extend}(\lambda a : A, b : B.\, m_B(f(a), b)) \\ & & \quad \mathrm{in}\ c.\mathrm{Fold}(e, g)\end{array}$$

where we have made use of the reverse application notation introduced above, and have also assumed available a multiplication term $m_B : B, B \multimap B$.

Taking $B = A$ and specialising $f : A \multimap B$ to the identity, we obtain a family of operators of types

$$\mathrm{SelectMany}_{X,A} : (X \to A[X]) \multimap (A[X] \multimap A[X])$$

again overloading notation. When $A = \mathrm{nat}_+$ these have type

$$(X \to X^\star) \multimap (X^\star \multimap X^\star)$$

and are the usual LINQ SelectMany operators (these are the same as MapReduce's improperly-named Map operators).

Reduction For general A-valued collections, we may already regard Fold as a reduction (or aggregation) operator. We can obtain analogues of the more usual reductions by taking both A and B to be basic linear types where there is an action of A on B; an example would be to take them both to be real_+ and the action to be m_{real_+}.

We can specialise the first argument of Fold to the action of nat_+ on linear type's B and obtain a family of operators

$$\mathrm{Reduce}_{X,B} : (X \to B) \multimap (X^\star \multimap B)$$

In this generality, these include SelectMany, if we take B to be $Y^\star$. Taking B to be a basic linear type such as real_+ we obtain more usual reductions.

Note that in all the cases considered above, the reduction operations are fixed to be the sum operations of the target linear types.

GroupBy

As already indicated, one can regard elements of the linear type $A[X]$ as key-value dictionaries of elements of A, indexed by elements of X. In particular, given a

type of keys K, we can regard $A[X][K]$ as the type of K-indexed A-valued X-collections. With this understanding, we have a family of GroupBy operators using of key function $X \to K$. These have type

$$(X \to K) \to (A[X] \multimap A[X][K])$$

and are given by:

$$\mathrm{GroupBy}_{K,X,A} = \lambda k : (X \to K).\, \lambda c : A[X].\, \sum_{a \cdot x \in c} (ax) \cdot k(x)$$

Lookup

Lookup functions extract the element with a given key from a K-indexed dictionary. They have type

$$K \to (A[K] \multimap A)$$

and are given by:

$$\mathrm{Lookup}_{K,X,A} =_{\mathrm{def}} \lambda x : K.\, \lambda c : A[K].\, \sum_{a \cdot x' \in c} \text{if } x' = x \text{ then } a \text{ else } 0$$

where we have assumed available an equality function on K.

Join

We first define cartesian product operations on collections; they require actions of linear types on themselves in order to combine values with the same index. We have a family of operations of type

$$(A, A \multimap A), A[X], A[Y] \multimap A[X \times Y]$$

given by:

$$\mathrm{CartProd}_{X,A} =_{\mathrm{def}} \lambda m : A, A \multimap A, c : A[X], c' : A[X].\, \sum_{a \cdot x \in c} \sum_{a' \cdot y \in c'} m(a, a') \cdot \langle x, y \rangle$$

We further have a family of Join operations which operate on pre-grouped-by collections, with a type of keys K for which an equality function is assumed available. They have type

$$(A, A \multimap A), A[X][K], A[Y][K] \multimap A[X \times Y][K]$$

and are given by:

$$\begin{aligned}\mathrm{Join}_{X,Y,K,A} =_{\mathrm{def}} \lambda m : (A, A \multimap A), d : A[X][K], d' : A[Y][K].& \\ \textstyle\sum_{c \cdot k \in d} \text{let } c' : A[Y] \text{ be } \mathrm{Lookup}(k)(d') \text{ in } \mathrm{CartProd}(m, c, c') \cdot k&\end{aligned}$$

Zip

Our final example is another family of binary functions on key-value dictionaries, which model the LINQ Zip operation:

$$\mathrm{Zip}_{X,A,B} : A[X] \oplus B[X] \multimap (A \oplus B)[X]$$

They take two X-indexed dictionaries and pair entries with the same index. They are given by:

$$\begin{aligned}\mathrm{Zip}_{X,A,B} \quad =_{\mathrm{def}} \quad & \lambda d : A[X] \oplus B[X]. \\ & \mathrm{Map}(\mathrm{inl})(\mathrm{id}_X)(\mathrm{fst}(d)) +_{A \oplus B} \mathrm{Map}(\mathrm{inr})(\mathrm{id}_X)(\mathrm{snd}(d))\end{aligned}$$

equivalently:

$$\begin{aligned}\mathrm{Zip}_{X,A,B} \quad =_{\mathrm{def}} \quad & \lambda d : A[X] \oplus B[X]. \, \mathrm{cases}\ d\ \mathrm{fst}\ a : A[X]. \, \mathrm{Map}(\mathrm{inl})(\mathrm{id}_X)(\mathrm{a}), \\ & \qquad\qquad\qquad\qquad \mathrm{snd}\ b : B[X]. \, \mathrm{Map}(\mathrm{inr})(\mathrm{id}_X)(\mathrm{b})\end{aligned}$$

5 Some Example Programs

In this section we give some example programs. The first two compute non-linear functions. However they are composed from linear subcomputations, and these are exposed as linear subterms. The last example is a linear version of MapReduce.

5.1 Counting

Our first example illustrates the utility of being able to move non-linearly between different monoids with the same carrier. Given a collection, we can count its elements, taking account of their multiplicity, using $\mathrm{Count}_X : X^\star \multimap \mathrm{nat}_+$, given by:

$$\lambda c : X^\star. \, \mathrm{Reduce}_{X,\mathrm{nat}_+}(\lambda x : X. \, \mathrm{u}_{\mathrm{nat}_+}(1))(c)$$

However, if we instead want to count ignoring multiplicity (i.e., to find the number of distinct elements), the computation proceeds in two linear phases separated by a non-linear one, as follows:

- the input collection, read as a nat_+-collection by a type conversion from an ordinary to a linear type, is mapped to a $\mathrm{nat}_{\max}$-collection c' to record only the presence or absence of an item (by a 1 or a 0),
- c' is then transformed nonlinearly to a nat_+-collection c'', using the type conversions, and, only then,

- the Count function is applied to c''.

To do this we use a term $\mathrm{SetCount}_X : \mathrm{U}(X^\star) \to \mathrm{nat}_+$, namely

$$\begin{array}{l}\lambda x : \mathrm{U}(X^\star).\\ \quad \mathrm{let}\ c' : \mathrm{nat}_{\max}[X]\ \mathrm{be}\ \mathrm{u}_{X^\star}(x).\mathrm{Map}(\mathrm{Conv})(\mathrm{id}_X)\ \mathrm{in}\\ \quad \mathrm{let}\ c'' : X^\star\ \mathrm{be}\ \mathrm{u}_{X^\star}(\mathrm{d}_{\mathrm{U}(X^\star)}(c'))\ \mathrm{in}\ c''.\mathrm{Count}\end{array}$$

where $\mathrm{Conv} : \mathrm{nat}_+ \multimap \mathrm{nat}_{\max}$ is a linear term converting between nat_+ and $\mathrm{nat}_{\max}$, namely $\lambda n : \mathrm{nat}_+.\, n(\mathrm{u}_{\mathrm{nat}_{\max}}(1))$ (it sends 0 to 0 and all other natural numbers to 1), and where we have dropped operator indices.

5.2 Histograms

Our second example is a simple histogram computation. Suppose we have a collection c of natural numbers and wish to plot a histogram of them spanning the range 0 to m, the maximum element in the collection (which we assume to be > 0). The histogram is to have $k > 0$ buckets, starting at 0, so each bucket will have width m/k. We model histograms by multisets of natural numbers, where each element corresponds to a bucket, and has multiplicity corresponding to the number of values in that bucket.

The following function $\mathrm{Hist}^k_{\max} : \mathrm{nat} \to (\mathrm{nat}^\star \multimap \mathrm{nat}^\star)$ is provided the maximum element value and computes the histogram linearly over c:

$$\lambda m : \mathrm{nat}.\, \lambda c : \mathrm{nat}^\star.\, c.\mathrm{SelectMany}(\lambda n : \mathrm{nat}.\, \{\lfloor kn/m \rfloor\})$$

The maximum element can be found linearly from c using a reduction:

$$c.\mathrm{Reduce}(\lambda n : \mathrm{nat}.\, \mathrm{u}_{\mathrm{nat}_{\max}}(n))$$

Putting these two together, we obtain a function $\mathrm{Hist}^{\mathrm{k}} : \mathrm{nat}^\star \to \mathrm{nat}^\star$ computing the required histogram:

$$\begin{array}{l}\lambda c : \mathrm{U}(\mathrm{nat}^\star).\\ \quad \mathrm{let}\ m : \mathrm{nat}\ \mathrm{be}\ \mathrm{d}_{\mathrm{nat}}(\mathrm{u}_{\mathrm{nat}^\star}(c).\mathrm{Reduce}(\lambda n : \mathrm{nat}.\, \mathrm{u}_{\mathrm{nat}_{\max}}(n)))\ \mathrm{in}\\ \quad \mathrm{u}_{\mathrm{nat}^\star}(c).\mathrm{Hist}^k_{\max}(m)\end{array}$$

Note the double occurrence of c, signalling nonlinearity.

5.3 A linear MapReduce

We present a linear version of MapReduce. It models the distributed nature of the data by using a dictionary indexed by machine names to model partitioned collections. The MapReduce computation begins with the initial collection distributed

over the machines, carries out a computation in parallel on each machine, redistributes the data between the machines by a shuffle operation, and then performs a final reduction.

We begin with the computation carried out on each machine. This applies to a collection c, and consists of a SelectMany, then a GroupBy using a basic type K of keys, and then a Reduce at each key. It is given by the following term MR:

$$\lambda c : X^\star.\ c.\mathrm{SelectMany}(m).\mathrm{GroupBy}(k).\mathrm{Map}(\mathrm{Reduce}(r))(\mathrm{id}_\mathrm{K})$$

which has typing:

$$k : Y \to \mathrm{K} \mid m : X \to Y^\star, r : Y \to A \vdash \mathrm{MR} : X^\star \multimap A[\mathrm{K}]$$

We next need to model data of any given type B spread across machines. To do this we assume available a basic type M of machine names and model such data by an M-indexed dictionary of type $B[\mathrm{M}]$. With this in mind the parallel computation is given by the following term PMR:

$$\mathrm{Map}(\mathrm{MR})(\mathrm{id}_\mathrm{M})$$

which maps MR across the machines and which has typing

$$k : Y \to \mathrm{K} \mid m : X \to Y^\star, r : Y \to A \vdash \mathrm{PMR} : X^\star[\mathrm{M}] \multimap A[\mathrm{K}][\mathrm{M}]$$

The shuffle operation employs a key-to-machine function, $h : \mathrm{K} \to \mathrm{M}$, and is given by the following term SH:

$$\begin{array}{l}\lambda e : A[\mathrm{K}][\mathrm{M}].\\ \quad \mathrm{sum}\ d : A[\mathrm{K}], m : \mathrm{M}\ \mathrm{in}\ e.\\ \quad \mathrm{sum}\ a : A, k : \mathrm{K}\ \mathrm{in}\ d.\ (\{a\} \cdot k) \cdot h(k)\end{array}$$

which has typing:

$$h : \mathrm{K} \to \mathrm{M} \mid \ \vdash \mathrm{SH} : A[\mathrm{K}][\mathrm{M}] \multimap A^\star[\mathrm{K}][\mathrm{M}]$$

The final reduction is carried out in parallel on each machine and is given by the following term FR:

$$\mathrm{Map}(\mathrm{Map}(\mathrm{Reduce}(\mathrm{id}_A))(\mathrm{id}_\mathrm{K}))(\mathrm{id}_\mathrm{M})$$

which maps the reduction at each key across the machines and which has typing

$$\vdash \mathrm{FR} : A^\star[\mathrm{K}][\mathrm{M}] \multimap A[\mathrm{K}][\mathrm{M}]$$

Putting everything together we obtain the entire MapReduce computation. It is given by the following term MapReduce:

$$\lambda b : X^\star[\mathrm{M}].\, b.\mathrm{PMR.SH.FR}$$

which has typing:

$$k : Y \to \mathrm{K}, h : \mathrm{K} \to \mathrm{M} \mid f : X \to Y^\star, r : Y \to A \vdash \mathrm{MapReduce} : X^\star[\mathrm{M}] \multimap A[\mathrm{K}][\mathrm{M}]$$

One could evidently abstract on the various functions, or choose specific ones.

6 Discussion

One can imagine a number of extensions and developments. Most immediately, as well as rules for type-checking, one would like an equational system, as is usual in type theories. This would open up the possibility of proving programs such as MapReduce correct. Regarding the language design, the reduction facilities depend on the built-in monoid structures. However in, e.g., LINQ, programmers can choose their own. In order to continue exposing linearity, it would be natural to introduce linear types of the form (X, z, m) where $z : X$ and $m : X^2 \to X$ are intended to provide X with a commutative monoid structure.

The mathematics suggests further possibilities. For example when working with A-valued collections (but not dictionaries) it is natural to suppose one has a semiring structure on A. Perhaps it would be worthwhile to add a kind of semirings (possibly even programmable) and to have separate linear types of collections and dictionaries.

Again, commutative monoids have a tensor product $A \otimes B$ classifying bilinear functions. One wonders if this would provide a useful datatype for big data programming. The tensor product enjoys various natural isomorphisms, for example: $A[X] \otimes B[Y] \cong (A \otimes B)[X \times Y]$, in particular $X^\star \otimes Y^\star \cong (X \times Y)^\star$.

The mathematics suffers if one were to drop commutativity, and just work with monoids, as in Nesl — see Blelloch (2011). One no longer has linear function spaces or tensor products. However it is not clear that one would not thereby enjoy benefits for programming with big data.

There are yet other possibilities for further development. It would be useful to add probabilistic choices to the language, however the interaction between probability and linearity is hardly clear. It would be interesting to consider differential aspects, as in McSherry et al. (2013). This would involve passing from monoids and semirings to abelian groups and rings. Compilers might well benefit from language facilities to indicate intended parallelism; an example is the use of

machine-indexed collections used above to model MapReduce. One could imagine a programmer-specified machine architecture, with machine-located datatypes $A@m$, see Jia and Walker (2004) and Murphy VII (2008).

References

G. E. Blelloch. Nesl. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1278–1283. Springer, 2011. ISBN 978-0-387-09765-7.

J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th OSDI*, pages 137–150. ACM, 2004. URL `http://labs.google.com/papers/mapreduce.html`.

L. Jia and D. Walker. Modal proofs as distributed programs (extended abstract). In *ESOP*, pages 219–233, 2004.

F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *CIDR*. www.cidrdb.org, 2013.

E. Meijer, B. Beckman, and G. M. Bierman. LINQ: reconciling object, relations and XML in the .NET framework. In *Proc. SIGMOD Int. Conf. on Manage. Data*, page 706. ACM, 2006. ISBN 1-59593-434-0. doi: http://doi.acm.org/10.1145/1142473.1142552. URL `http://doi.acm.org/10.1145/1142473.1142552`.

T. Murphy VII. *Modal types for mobile code*. PhD thesis, CMU, 2008.

D. Prospero, S. Goldberg, and F. McSherry. Calibrating data to sensitivity in private data analysis, a platform for differentially-private analysis of weighted datasets. To appear in VLDB14, 2014.

Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. Gunda, K. Pradeep, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proc. 8th OSDI*, pages 1–14. ACM, 2008.