Constructive Logic
Fall 2000

Carnegie Mellon University
Frank Pfenning

Assignment 8

Out: Thursday Nov 9

Due: Thursday Nov 16

Note: You may reuse any function introduced in lecture or the assignments without giving the definition again!

1. Vectors and Matrices (50 Points)

Many numerical applications, e.g., in statics, involve vectors and matrices. Vector algebra defines arithmetical operations on vectors, but most of them impose restrictions on their arguments. E.g., the inner product of two vectors is only defined if they have equal length. It turns out that these kinds of restrictions can be captured elegantly by dependent types.
First an short introduction into vectors and matrices. Given a semi-ring R (basically a structure with addition and multiplication, e.g., $\mathbb{N}$, $\mathbb{Q}$ or $\mathbb{R}$) we define vectors $v \in R^n$ and matrices $a \in R^{k \times m}$ as follows:

$$v = (v_i)_{0 \le i < n} = \begin{pmatrix} v_0 \\ v_1 \\ \vdots \\ v_{n-1} \end{pmatrix}$$

$$a = (a_{i,j})_{0 \le j < m}^{0 \le i < k} = \begin{pmatrix} a_{0,0} & a_{0,1} & \cdots & a_{0,m-1} \\ a_{1,0} & a_{1,1} & \cdots & a_{1,m-1} \\ \vdots & \vdots & \ddots & \vdots \\ a_{k-1,0} & a_{k-1,1} & \cdots & a_{k-1,m-1} \end{pmatrix}$$

Several operations are defined on vectors and matrices, we present four of them:

1. *Inner product.* Given two vectors $v, w \in R^n$ of the same length we define

$$\langle v, w \rangle := \sum_{i=0}^{m-1} v_i w_i \in R$$

2. *Application of a matrix to a vector.* Given a matrix $a \in R^{n \times k}$ and a vector $v \in R^k$ we define

$$av := \left(\sum_{j=0}^{k-1} a_{i,j} v_j \right)_{0 \le i < n} \in R^n$$

3. *Matrix transposition.* Given a matrix $a = (a_{i,j})_{0 \le j < m}^{0 \le i < k} \in R^{k \times m}$, we define

$$a^T := (a_{j,i})_{0 \le i < k}^{0 \le j < m} \in R^{m \times k}$$

4. *Matrix multiplication.* Given two matrics $a \in R^{n \times k}$ and $b \in R^{k \times m}$ we define

$$ab := \left(\sum_{l=0}^{k-1} a_{i,l} b_{l,j} \right)_{0 \le j < m}^{0 \le i < n} \in R^{n \times m}$$

In the following we define the datatypes **vec** and **matr** (vectors and matrices over natural numbers) by introduction rules. We overload **nil** and :: because our data structures are closely related to lists.

$$\frac{}{\Gamma \vdash \mathbf{nil} \in \mathbf{vec}(\mathbf{0})}\,\mathbf{vec}\,\mathrm{I_n} \qquad \frac{\Gamma \vdash x \in \mathbf{nat} \quad \Gamma \vdash v \in \mathbf{vec}(n)}{\Gamma \vdash x :: v \in \mathbf{vec}(\mathbf{s}\ n)}\,\mathbf{vec}\,\mathrm{I_c}$$

$$\frac{\Gamma \vdash n \in \mathbf{nat}}{\Gamma \vdash \mathbf{nil} \in \mathbf{matr}(\mathbf{0}, n)}\,\mathbf{matr}\,\mathrm{I_n} \qquad \frac{\Gamma \vdash v \in \mathbf{vec}(n) \quad \Gamma \vdash a \in \mathbf{matr}(m, n)}{\Gamma \vdash v :: a \in \mathbf{matr}(\mathbf{s}\ m, n)}\,\mathbf{matr}\,\mathrm{I_c}$$

A vector is basically a list of natural numbers and a matrix a list of rows where each row is represented by a vector.

1. (20 points) Give the formation and elimination rules for **vec** and **matr**.
2. (10 points) Give the equational specification[1] of the inner product of two vectors.

$$\mathit{inner} \in \Pi n \in \mathbf{nat}.\ \mathbf{vec}(n) \to \mathbf{vec}(n) \to \mathbf{nat}$$

3. (10 points) Give the equational specification of the application of a matrix to a vector.

$$\mathit{app} \in \Pi k \in \mathbf{nat}.\, \Pi n \in \mathbf{nat}.\ \mathbf{matr}(n, k) \to \mathbf{vec}(k) \to \mathbf{vec}(n)$$

4. (10 Points) Matrix multiplication cannot be defined in a straightforward way with our data structures. This is because each element of the resulting matrix is computed by folding a row of the first matrix with a column of the second matrix. However, our data structure only gives us easy access to the rows of a matrix. We now assume we have already implemented a function

$$\mathit{transpose} \in \Pi k \in \mathbf{nat}.\, \Pi n \in \mathbf{nat}.\ \mathbf{matr}(n, k) \to \mathbf{matr}(k, n)$$

[1] That is defining a function by a set of equations, as we did in lecture and assignments before. You do not have to give an implementation as a primitive recursive function.

which takes a matrix a and returns a matrix b whose rows are the columns of a. Now multiplication is implemented with the help of an auxiliary function *mult′* as follows:

$$\begin{aligned} \mathit{mult} &\in \Pi k \in \mathbf{nat}.\ \Pi n \in \mathbf{nat}.\ \mathbf{matr}(n,k) \to \Pi m \in \mathbf{nat}.\ \mathbf{matr}(k,m) \\ &\quad \to \mathbf{matr}(n,m) \\ \mathit{mult} &= \lambda k.\ \lambda n.\ \lambda a.\ \lambda m.\ \lambda b.\ \mathit{mult}'\ k\ n\ a\ m\ (\mathit{transpose}\ m\ k\ b) \end{aligned}$$

Give the equational specification for *mult′*.

$$\begin{aligned} \mathit{mult}' &\in \Pi k \in \mathbf{nat}.\ \Pi n \in \mathbf{nat}.\ \mathbf{matr}(n,k) \to \Pi m \in \mathbf{nat}.\ \mathbf{matr}(m,k) \\ &\quad \to \mathbf{matr}(n,m) \end{aligned}$$

2. Queues (50 Points)

Consider the following interface for queues:

$$\begin{array}{lll} \tau\ \mathit{queue} & \mathit{type} & \\ \mathit{empty} & \in & \tau\ \mathit{queue} \\ \mathit{snoc} & \in & \tau\ \mathit{queue} \to \tau \to \tau\ \mathit{queue} \\ \mathit{head} & \in & \tau\ \mathit{queue} \to 1 + \tau \\ \mathit{tail} & \in & \tau\ \mathit{queue} \to \tau\ \mathit{queue} \end{array}$$

A *queue* is a data structure that stores elements of type τ. Queues can be constructed by *empty*, which returns an empty queue, and *snoc* which adds one element to the end of the queue.[2] Destructors for queues are *head*, which returns the first element of the list resp. an indication that the queue is empty, and *tail* which returns the queue minus its first element. Alternative names for *queue* are **F**irst **I**n **F**irst **O**ut data structures or *pipes*. There are plenty applications for queues, e.g., dispatching processes.
Straightforward implementation by $\mathit{queue} = \mathbf{list}$ runs us into efficiency problems: Either construction or destruction of a queue will be expensive. E.g., we could choose to implement *snoc* by

$$\mathit{snoc}\ q\ a = \mathit{append}\ q\ (a :: \mathbf{nil})$$

Then taking the head or the tail could be done by one operation in $O(1)$ time, but *snoc* would have complexity of $O(|q|)$, since we have to traverse the whole list to add a single element. ($|\cdot|$ denotes the length of a list here.)
But we can do better: We can implement a queue by a pair of lists observing an invariant

$$\begin{array}{l} \tau\ \mathit{queue} = \tau\ \mathbf{list} \times \tau\ \mathbf{list} \\ \text{Invariant: } |f| \geq |r| \text{ for } (f,r) \in \tau\ \mathit{queue} \end{array}$$

[2] *snoc* is *cons* read backwards and is a silly but common name for this operation. To come up with clear and expressive names is not a gift of every programmer!

The idea is as follows: *head* takes the first element of the first list f, *tail* returns a queue where the head element is removed from f and *snoc* adds an element to the front of r. Now all these operations can be done in $O(1)$ time, with one exception: In some cases we have to rearrange the data in our queue. If an operation results in a queue which violates the invariant, then we have to reverse r and append it to f. This happens, e.g., if we take the last element from f and r is not empty. Then we rearrange the data to make the elements of r available for *head* and *tail*. This operations has complexity $O(n)$ where n is the number of elements in the queue. Still this is a very effective implementation of queues.

1. (10 points) Refine the datatype *queue* s.th. it additionally carries the lengths $n, m \in \mathbf{nat}$ of the lists f and r. Use dependent types to ensure that $n =_N |f|$ and $m =_N |r|$ always hold. The invariant $|f| \geq |r|$ does *not* have to be captured by dependent types.

2. (40 points) Implement the interface given above by equationally specifying the four objects *empty*, *snoc*, *head* and *tail*. Make sure that the invariant $|f| \geq |r|$ always holds. You may use the following two functions (and, of course, all other functions given in lecture).

$$\begin{array}{lcl} \mathit{lt} & \in & \mathbf{nat} \to \mathbf{nat} \to \mathbf{bool} \\ \mathit{apprev} & \in & \Pi n \in \mathbf{nat}.\, \tau\, \mathbf{list}(n) \to \Pi m \in \mathbf{nat}.\, \tau\, \mathbf{list}(m) \\ & & \to \tau\, \mathbf{list}(\mathit{plus}\ n\ m) \end{array}$$

   The result is of evaluating $\mathit{lt}\ n\ m$ is **true** if $n < m$ and **false** otherwise. The function *apprev* appends the reverse of its second argument to the first.

Have fun, even without tutch!