Fun with Semirings

A functional pearl on the abuse of linear algebra

Stephen Dolan

Computer Laboratory, University of Cambridge
stephen.dolan@cl.cam.ac.uk

Abstract

Describing a problem using classical linear algebra is a very well-known problem-solving technique. If your question can be formulated as a question about real or complex matrices, then the answer can often be found by standard techniques.

It's less well-known that very similar techniques still apply where instead of real or complex numbers we have a *closed semiring*, which is a structure with some analogue of addition and multiplication that need not support subtraction or division.

We define a typeclass in Haskell for describing closed semirings, and implement a few functions for manipulating matrices and polynomials over them. We then show how these functions can be used to calculate transitive closures, find shortest or longest or widest paths in a graph, analyse the data flow of imperative programs, optimally pack knapsacks, and perform discrete event simulations, all by just providing an appropriate underlying closed semiring.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; G.2.2 [*Discrete Mathematics*]: Graph Theory—graph algorithms

Keywords closed semirings; transitive closure; linear systems; shortest paths

1. Introduction

Linear algebra provides an incredibly powerful problem-solving toolbox. A great many problems in computer graphics and vision, machine learning, signal processing and many other areas can be solved by simply expressing the problem as a system of linear equations and solving using standard techniques.

Linear algebra is defined abstractly in terms of fields, of which the real and complex numbers are the most familiar examples. Fields are sets equipped with some notion of addition and multiplication as well as negation and reciprocals.

Many discrete mathematical structures commonly encountered in computer science do not have sensible notions of negation. Booleans, sets, graphs, regular expressions, imperative programs, datatypes and various other structures can all be given natural notions of product (interpreted variously as intersection, sequencing or conjunction) and sum (union, choice or disjunction), but generally lack negation or reciprocals.

Such structures, having addition and multiplication (which distribute in the usual way) but not in general negation or reciprocals, are called semirings. Many structures specifying sequential actions can be thought of as semirings, with multiplication as sequencing and addition as choice. The distributive law then states, intuitively, a followed by a choice between b and c is the same as a choice between a followed by b and a followed by c.

Plain semirings are a very weak structure. We can find many examples of them in the wild, but unlike fields which provide the toolbox of linear algebra, there isn't much we can do with something knowing only that it is a semiring.

However, we can build some useful tools by introducing the *closed semiring*, which is a semiring equipped with an extra operation called *closure*. With the intuition of multiplication as sequencing and addition as choice, closure can be interpreted as iteration. As we see in the following sections, it is possible to use something akin to Gaussian elimination on an arbitrary closed semiring, giving us a means of solving certain "linear" equations over any structure with suitable notions of sequencing, choice and iteration. First, though, we need to define the notion of semiring more precisely.

2. Semirings

We define a semiring formally as consisting of a set R, two distinguished elements of R named 0 and 1, and two binary operations $+$ and $\cdot$, satisfying the following relations for any $a, b, c \in R$:

$$\begin{aligned} a + b &= b + a \\ a + (b + c) &= (a + b) + c \\ a + 0 &= a \\ a \cdot (b \cdot c) &= (a \cdot b) \cdot c \\ a \cdot 0 &= 0 \cdot a = 0 \\ a \cdot 1 &= 1 \cdot a = a \\ a \cdot (b + c) &= a \cdot b + a \cdot c \\ (a + b) \cdot c &= a \cdot c + b \cdot c \end{aligned}$$

We often write $a \cdot b$ as ab, and $a \cdot a \cdot a$ as a^3.

Our focus will be on *closed semirings* [12], which are semirings with an additional operation called *closure* (denoted *) which satisfies the axiom:

$$a^* = 1 + a \cdot a^* = 1 + a^* \cdot a$$

If we have an *affine map* $x \mapsto ax + b$ in some closed semiring, then $x = a^*b$ is a fixpoint, since $a^*b = (aa^* + 1)b = a(a^*b) + b$. So, a closed semiring can also be thought of as a semiring where affine maps have fixpoints.

The definition of a semiring translates neatly to Haskell:

ICFP '13, September 25–27, 2013, Boston, MA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2326-0/13/09…$15.00.
http://dx.doi.org/10.1145/2500365.2500613

```
infixl 9 @.
infixl 8 @+
class Semiring r where
  zero, one :: r
  closure :: r -> r
  (@+), (@.) :: r -> r -> r
```

There are many useful examples of closed semirings, the simplest of which is the Boolean datatype:

```
instance Semiring Bool where
  zero = False
  one = True
  closure x = True
  (@+) = (||)
  (@.) = (&&)
```

It is straightforward to show that the semiring axioms are satisfied by this definition.

In semirings where summing an infinite series makes sense, we can define a^* as:

$$1 + a + a^2 + a^3 + \dots$$

since this series satisfies the axiom $a^* = 1 + a \cdot a^*$. In other semirings where subtraction and reciprocals make sense we can define a^* as $(1 - a)^{-1}$. Both of these viewpoints will be useful to describe certain semirings.

The real numbers form a semiring with the usual addition and multiplication, where $a^* = (1 - a)^{-1}$. Under this definition, 1^* is undefined, an annoyance which can be remedied by adding an extra element ∞ to the semiring, and setting $1^* = \infty$.

The regular languages form a closed semiring where $\cdot$ is concatenation, $+$ is union, and * is the Kleene star. Here the infinite geometric series interpretation of * is the most natural: a^* is the union of a^n for all n.

3. Matrices and reachability

Given a directed graph G of n nodes, we can construct its adjacency matrix M, which is an $n \times n$ matrix of Booleans where M_{ij} is true if there is an edge from i to j.

We can add such matrices. Using the Boolean semiring's definition of addition (i.e. disjunction), the effect of this is to take the union of two sets of edges.

Similarly, we define matrix multiplication in the usual way, where $(AB)_{ij} = \sum_k A_{ik} \cdot B_{kj}$. The product of two Boolean matrices A, B is thus true at indices ij if there exists any index k such that A_{ik} and B_{kj} are both true. In particular, $(M^2)_{ij}$ is true if there is a path with two edges in G from node i to node j.

In general, M^k represents the paths of k edges in the graph G. A node j is reachable from a node i if there is a path with any number of edges (including 0) from i to j. This reachability relation can therefore be described by the following, where I is the identity matrix:

$$I + M + M^2 + M^3 + \dots$$

This looks like the infinite series definition of closure from above. Indeed, suppose we could calculate the closure of M, that is, a matrix M^* such that:

$$M^* = I + M \cdot M^*$$

M^* would include the paths of length 0 (the I term), and would be transitively closed (the $M \cdot M^*$ term). So, if we can show that $n \times n$ matrices of Booleans form a closed semiring, then we can use the closure operation to calculate reachability in a graph, or equivalently the reflexive transitive closure of a graph.

Remarkably, for any closed semiring R, the $n \times n$ matrices of elements of R form a closed semiring. This is a surprisingly powerful result: as we see in the following sections, the closure operation can be used to solve several different problems with a suitable choice of the semiring R.

We define addition and multiplication of $n \times n$ matrices in the usual way, where:

$$(A + B)_{ij} = A_{ij} + B_{ij}$$

$$(A \cdot B)_{ij} = \sum_{k=1}^{n} A_{ik} \cdot B_{kj}$$

The matrix $\mathbf{0}$ is the $n \times n$ matrix where every element is the underlying semiring's 0, and the matrix $\mathbf{1}$ has the underlying semiring's 1 along the main diagonal (so $\mathbf{1}_{ii} = 1$) and 0 elsewhere.

In Haskell, we use the type `Matrix`, which represents a matrix as a list of rows, each a list of elements, with a special case for the representation of scalar matrices (matrices which are zero everywhere but the main diagonal, and equal at all points along the diagonal). This special case allows us to define matrices `zero` and `one` without knowing the size of the matrix.

```
data Matrix a = Scalar a
              | Matrix [[a]]
```

To add a scalar to a matrix, we need to be able to move along the main diagonal of the matrix. To make this easier, we introduce some helper functions for dealing with block matrices.

A block matrix is a matrix that has been partitioned into several smaller matrices. We define a type for matrices that have been partitioned into four blocks:

```
type BlockMatrix a = (Matrix a, Matrix a,
                      Matrix a, Matrix a)
```

If `a`, `b`, `c` and `d` represent the $n \times n$ matrices A, B, C, D, then `BlockMatrix (a,b,c,d)` represents the $2n \times 2n$ block matrix:

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

Joining the components of a block matrix into a single matrix is straightforward:

```
mjoin :: BlockMatrix a -> Matrix a
mjoin (Matrix a, Matrix b,
       Matrix c, Matrix d) =
      Matrix ((a ‘hcat‘ b) ++ (c ‘hcat‘ d))
  where hcat = zipWith (++)
```

For any $n \times m$ matrix where $n, m \geq 2$, we can split the matrix into a block matrix by peeling off the first row and column:

```
msplit :: Matrix a -> BlockMatrix a
msplit (Matrix (row:rows)) =
    (Matrix [[first]], Matrix [top],
     Matrix left,      Matrix rest)
  where
    (first:top) = row
    (left, rest) = unzip (map (\(x:xs) -> ([x],xs))
                              rows)
```

Armed with these, we can start implementing a `Semiring` instance for `Matrix`.

```
instance Semiring a => Semiring (Matrix a) where
  zero = Scalar zero
  one = Scalar one

  Scalar a @+ Scalar b = Scalar (a @+ b)

  Matrix a @+ Matrix b =
    Matrix (zipWith (zipWith (@+)) a b)

  Scalar s @+ m = m @+ Scalar s
  Matrix [[a]] @+ Scalar b = Matrix [[a @+ b]]
```

```
m @+ s = mjoin (first @+ s, top,
                left,       rest @+ s)
  where (first, top,
         left,  rest) = msplit m

Scalar a @. Scalar b = Scalar (a @. b)
Scalar a @. Matrix b = Matrix (map (map (a @.)) b)
Matrix a @. Scalar b = Matrix (map (map (@. b)) a)
Matrix a @. Matrix b =
  Matrix [[foldl1 (@+) (zipWith (@.) row col)
          | col <- cols] | row <- a]
  where cols = transpose b
```

Defining closure for matrices is trickier. Lehmann [12] gave a definition of M^* for an arbitrary matrix M which satisfies the axioms of a closed semiring, and two algorithms for calculating it. The first of these generalises the Floyd-Warshall algorithm for all-pairs shortest paths [6], while the second is a semiring-flavoured form of Gaussian elimination.

Both are specified imperatively, via indexing and mutation of matrices represented as arrays. However, an elegant functional implementation can be derived almost directly from a lemma used to prove the correctness of the imperative algorithms. Given a block matrix

$$M = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

Lehmann shows that its closure M^* will satisfy

$$M^* = \begin{pmatrix} A^* + B'\Delta^* C' & B'\Delta^* \\ \Delta^* C' & \Delta^* \end{pmatrix}$$

where $B' = A^*B$, $C' = CA^*$ and $\Delta = D + CA^*B$. The closure of a 1×1 matrix is easily calculated since $\begin{pmatrix} a \end{pmatrix}^* = \begin{pmatrix} a^* \end{pmatrix}$, so this leads directly to an implementation of closure for matrices:

```
closure (Matrix [[x]]) = Matrix [[closure x]]
closure m = mjoin
  (first’ @+ top’ @. rest’ @. left’, top’ @. rest’,
   rest’ @. left’,                   rest’)
  where
    (first, top, left, rest) = msplit m
    first’ = closure first
    top’ = first’ @. top
    left’ = left @. first’
    rest’ = closure (rest @+ left’ @. top)
```

Multiplying a $p \times q$ matrix by a $q \times r$ matrix takes $O(pqr)$ operations from the underlying semiring. The closure function, when given a $n \times n$ matrix, does $O(n^2)$ semiring operations via matrix multiplication (by multiplying $1 \times n$ and $n \times n$ matrices, or $n \times 1$ and $1 \times n$), plus $O(n^2)$ semiring operations via matrix addition and msplit, plus one recursive call.

The recursive call to closure is passed a $(n-1) \times (n-1)$ matrix, and so the total number of semiring operations done by closure for an $n \times n$ matrix is $O(n^3)$. Thus, closure has the same complexity as calculating transitive closure using the Floyd-Warshall algorithm.

However, since it processes the entire graph and always produces an answer for all pairs of nodes, it is slower than standard algorithms for checking reachability between a single pair of nodes.

4. Graphs and paths

We've already seen that the reflexive transitive closure of a graph can be found using the above closure function, but it seems like a lot of work just to define reachability! However, choosing a richer underlying semiring allows us to calculate more interesting properties of the graph, all with the same closure algorithm.

The tropical semiring (more sensibly known as the min-plus semiring) has as its underlying set the nonnegative integers augmented with an extra element ∞, and defines its $+$ and $\cdot$ operators as $\min$ and addition respectively. This semiring describes the length of the shortest path in a graph: ab is interpreted as a path through a and then b (so we sum the distances), and $a + b$ is a choice between a path through a and a path through b (so we pick the shorter one). We express this in Haskell as follows, using the value Unreachable to represent ∞:

```
data ShortestDistance = Distance Int | Unreachable
instance Semiring ShortestDistance where
  zero = Unreachable
  one = Distance 0
  closure x = one

  x @+ Unreachable = x
  Unreachable @+ x = x
  Distance a @+ Distance b = Distance (min a b)

  x @. Unreachable = Unreachable
  Unreachable @. x = Unreachable
  Distance a @. Distance b = Distance (a + b)
```

For a directed graph with edge lengths, we make a matrix M such that M_{ij} is the length of the edge from i to j, or Unreachable if there is none. M is represented in Haskell with the type Matrix ShortestDistance, and calling closure calculates the length of the shortest path between any two nodes.

To see this, we can appeal again to the infinite series view of closure: $(M^k)_{ij}$ is the length of the shortest path with k edges from node i to node j, and M^* is the sum (which in this semiring means "minimum") of M^k for any k. Thus, $(M^*)_{ij}$ is the length of the shortest path with any number of edges from node i to node j.

Often we're interested in finding the actual shortest path, not just its length. We can define another semiring that keeps track of this data, where paths are represented as lists of edges, each represented as a pair of nodes.

There may not be a unique shortest path. If we are faced with a choice between two equally short paths, we must either have some means of disambiguating them or be prepared to return multiple results. In the following implementation, we choose the former: we assume nodes are ordered and choose the lexicographically least of multiple equally short paths.

```
data ShortestPath n = Path Int [(n,n)] | NoPath
instance Ord n => Semiring (ShortestPath n) where
  zero = NoPath
  one = Path 0 []
  closure x = one

  x @+ NoPath = x
  NoPath @+ x = x
  Path a p @+ Path a’ p’
    | a < a’            = Path a p
    | a == a’ && p < p’ = Path a p
    | otherwise         = Path a’ p’

  x @. NoPath = NoPath
  NoPath @. x = NoPath
  Path a p @. Path a’ p’ = Path (a + a’) (p ++ p’)
```

The @. operator given here isn't especially fast since ++ takes time linear in the length of its left argument, but this can be avoided by using an alternative data structure with constant-time appends such as difference lists.

We construct the matrix M, where M_{ij} is Path d [(i,j)] if there's an edge of length d between nodes i and j or NoPath if there's none. Calculating M^* in this semiring will calculate not only the length of the shortest path between all pairs of nodes, but give the actual route.

To calculate longest paths we can use a similar construction. We have to be slightly more careful here, because a graph with cycles contains arbitrarily long paths.

As well as nonnegative integer distances, we have two other possible values: `LUnreachable`, indicating that there is no path between two nodes, and `LInfinite`, indicating that there's an infinitely long path due to a cycle of positive length. This forms a semiring as shown:

```
data LongestDistance = LDistance Int
                     | LUnreachable
                     | LInfinite
instance Semiring LongestDistance where
  zero = LUnreachable
  one = LDistance 0

  closure LUnreachable = LDistance 0
  closure (LDistance 0) = LDistance 0
  closure _ = LInfinite

  x @+ LUnreachable = x
  LUnreachable @+ x = x
  LInfinite @+ _ = LInfinite
  _ @+ LInfinite = LInfinite
  LDistance x @+ LDistance y = LDistance (max x y)

  x @. LUnreachable = LUnreachable
  LUnreachable @. x = LUnreachable
  LInfinite @. _ = LInfinite
  _ @. LInfinite = LInfinite
  LDistance x @. LDistance y = LDistance (x + y)
```

We can find the widest path (also known as the highest-capacity path) by using min instead of addition as semiring multiplication (to pick the narrowest of successive edges in a path). By working with real numbers as edge weights, interpreted as the probability of failure of a given edge, we can calculate the most reliable path.

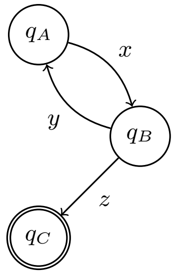
There is an intuition for closure for an arbitrary graph and an arbitrary semiring. Each edge of the graph is assigned an element of the semiring, which make up the elements of the matrix M. Any path (sequence of edges) is assigned the product of the elements on each edge, and M^*_{ij} is the sum of the products assigned to every path from i to j. The fact that product distributes over sum means we can calculate this, using the above `closure` algorithm, in polynomial time.

We can use this intuition to construct powerful graph analyses, simply by making an appropriate semiring and calculating the closure of a graph's adjacency matrix. For instance, we can make a semiring of subsets of nodes of a graph, where $+$ is intersection and $\cdot$ is union. We set $M_{ij} = \{i, j\}$, and calculate M^*. Each path is assigned the set of nodes visited along that path, and taking the "sum over all paths" calculates the intersection of those sets, or the nodes visited along all paths. Thus, M^*_{ij} gives the set of nodes that are visited along all paths from i to j, or in other words, the graph dominators of j with start node i.

5. "Linear" equations and regular languages

One of the sharpest tools in the toolbox of linear algebra is the use of matrix techniques to solve systems of linear equations.

Since we get to specify the semiring, we define what "linear" means. Many problems can then be described as systems of "linear" equations, even though they're far from linear in the classical sense.



$$\begin{pmatrix} A \\ B \\ C \end{pmatrix} = \begin{pmatrix} 0 & x & 0 \\ y & 0 & z \\ 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$$

Figure 1. A finite state machine and its matrix representation

Suppose we have a system of equations in some semiring on a set of variables $x_1, \ldots, x_n$, where each x_i is defined by a linear combination of the variables and a constant term. That is, each equation is of the following form, where a_{ij} and b_i are givens:

$$x_i = a_{i1}x_1 + a_{i2}x_2 + \cdots + b_i$$

We arrange the unknowns x_i into a column vector X, the coefficients A into a square matrix A and the constants b into a column vector B. The system of equations now becomes:

$$X = AX + B$$

This equation defines X as the fixpoint of an affine map. As we saw in section 2, it therefore has a solution $X = A^*B$, which can be calculated with our definition of `closure` for matrices.

The above was a little cavalier with matrix dimensions. Technically, our machinery for solving such equations is only defined for $n \times n$ matrices, not column vectors. However, we can extend the column vectors X and B to $n \times n$ matrices by making a matrix all of whose columns are equal. Solving the equation with such matrices comes to the same answer as using column vectors directly, so we keep working with column vectors. Happily, our Haskell code for manipulating matrices accepts column and row vectors without problems, as long as we don't try to calculate the closure of anything but a square matrix.

If we have some system that maps input to output, where the mapping can be described as a linear map $X \mapsto AX + B$, then the fixpoint $X = A^*B$ gives a "stable state" of the system.

As we see below, Kleene's proof that all finite state machines accept a regular language and the McNaughton-Yamada algorithm [15] for constructing a regular expression to describe a state machine can also be described as such "linear" systems.

Given a description of a finite state machine, we can write down a regular grammar describing the language it accepts. For every transition $q_A \xrightarrow{x} q_B$, we have a grammar production $A \to xB$, and for every accepting state q_A we have a production $A \to \varepsilon$.

We can group these productions by their left-hand sides to give a system of equations. For instance, in the machine of Figure 1 there is a state q_B with transitions $q_B \xrightarrow{y} q_A$ and $q_B \xrightarrow{z} q_C$, so we get the equation:

$$B = yA + zC$$

In the semiring of regular languages (where addition is union, multiplication is concatenation, and closure is the Kleene star), these are all linear equations. For an n-state machine, we define the $n \times n$ matrix $\mathbf{M}$ where $\mathbf{M}_{ij}$ is the symbol on the transition from the ith state to the jth, or 0 (the empty language) if there is no such transition. The vector $\mathbf{A}$ is constructed by setting $\mathbf{A}_i$ to 1 (the language containing only ε) when the ith state is accepting, and 0 otherwise. The languages are represented by the vector of unknowns L, where L_i is the language accepted starting in the ith state. For our example machine, $\mathbf{M}$ and $\mathbf{A}$ are shown at the right of Figure 1.

Then, the regular grammar is described by:

$$\mathbf{L} = \mathbf{M} \cdot \mathbf{L} + \mathbf{A}$$

This equation has a solution given by $\mathbf{L} = \mathbf{M}^* \cdot \mathbf{A}$. We can use our existing `closure` function to solve these equations and build a regular expression that describes the language accepted by the finite state machine.

In Haskell, we define a "free" semiring which simply records the syntax tree of semiring expressions. To qualify as a semiring we must have `a @+ b == b @+ a`, `a @. one == a`, and so on. We sidestep this by cheating: we don't define an `Eq` instance for `FreeSemiring`, and consider two `FreeSemiring` values equal if they are equal according to the semiring laws. However, to make our `FreeSemiring` values more compact, we do implement certain simplifications like $0x = 0$.

```
data FreeSemiring gen =
    Zero
  | One
  | Gen gen
  | Closure (FreeSemiring gen)
  | (FreeSemiring gen) :@+ (FreeSemiring gen)
  | (FreeSemiring gen) :@. (FreeSemiring gen)

instance Semiring (FreeSemiring gen) where
  zero = Zero
  one = One

  Zero @+ x = x
  x @+ Zero = x
  x @+ y = x :@+ y

  Zero @. x = Zero
  x @. Zero = Zero
  One @. x = x
  x @. One = x
  x @. y = x :@. y

  closure Zero = One
  closure x = Closure x
```

If we construct $\mathbf{M}$ as a `Matrix (FreeSemiring Char)`, then calculating $\mathbf{M}^* \cdot \mathbf{A}$ will give us a vector of `FreeSemiring Char`, each element of which can be interpreted as a regular expression describing the language accepted from a particular state.

For the example of Figure 1, `closure` then tells us that the language accepted with state A as the starting state is $x(yx)^*z$. This algorithm produces a regular expression that accurately describes the language accepted by a given state machine, but it is not in general the shortest such expression.

6. Dataflow analysis

Many program analyses and optimisations can be computed by *dataflow analysis*. As an example, we consider the classical liveness analysis, which computes which assignments in an imperative program assign values which will never be read ("dead"), and which ones may be used again ("live").

We construct the program's control flow graph by dividing it into control-free basic blocks and with edges indicating where there are jumps between blocks. For a given basic block b, the set of variables live at the start of the block (IN_b) are those used by the basic block itself (USE_b) before their first definition, and those which are live at the end of the block (OUT_b) but not assigned a new value during the block (DEF_b). The variables live at the end of a basic block are those live at the start of any successor.

This gives a system of equations:

$$\mathrm{IN}_b = (\mathrm{OUT}_b \cap \overline{\mathrm{DEF}_b}) \cup \mathrm{USE}_b$$
$$\mathrm{OUT}_b = \bigcup_{b' \in \mathrm{succ}(b)} \mathrm{IN}_{b'}$$

```
x := 1
z := 0

while x < y:

    x := x * 2
    z := z + 1

return x
```

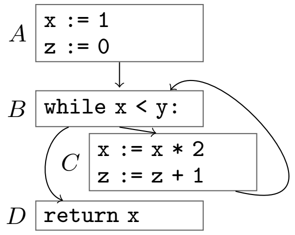


Figure 2. A simple imperative program to calculate the smallest power of two greater than the input `y`, and its control flow graph. The variable `z` does not affect the output.

An example program is given in Figure 2, where DEF and USE are as follows:

$$\begin{array}{ll}
\mathrm{DEF}_A = \{\mathtt{x}, \mathtt{z}\} & \mathrm{USE}_A = \emptyset \\
\mathrm{DEF}_B = \emptyset & \mathrm{USE}_B = \{\mathtt{x}, \mathtt{y}\} \\
\mathrm{DEF}_C = \{\mathtt{x}, \mathtt{z}\} & \mathrm{USE}_C = \{\mathtt{x}, \mathtt{z}\} \\
\mathrm{DEF}_D = \emptyset & \mathrm{USE}_D = \{\mathtt{x}\}
\end{array}$$

If we solve for IN_b and OUT_b, we find that `z` is not live upon entry to D (that is, $\mathtt{z} \notin \mathrm{IN}_D$). However, `z` is considered live on entry to C, even though it is never affects the output of the program. We see how to remedy this using *faint variables analysis* below, but first we show how the classical live variables analysis can be calculated using our semiring machinery.

We define a semiring of sets of variables, where 0 is the empty set, 1 is the set of all variables in the program, $+$ is union, $\cdot$ is intersection, and $x^* = 1$ for all sets x. Our system of equations can be represented as follows in this semiring:

$$\begin{aligned}
\mathrm{OUT}_b &= \sum_{b' \in \mathrm{succ}(b)} \mathrm{IN}_{b'} \\
&= \sum_{b' \in \mathrm{succ}(b)} \overline{\mathrm{DEF}_{b'}} \cdot \mathrm{OUT}_{b'} + \mathrm{USE}_{b'} \\
&= \left(\sum_{b' \in \mathrm{succ}(b)} \overline{\mathrm{DEF}_{b'}} \cdot \mathrm{OUT}_{b'} \right) + \left(\sum_{b' \in \mathrm{succ}(b)} \mathrm{USE}_{b'} \right)
\end{aligned}$$

This is a system of affine equations over the variables OUT_b, with coefficients $\overline{\mathrm{DEF}_{b'}}$ and constant terms $\sum_{b' \in \mathrm{succ}(b)} \mathrm{USE}_{b'}$. As before, we can solve it by building a matrix M containing the coefficients and a column vector A containing the constant terms. The solution vector OUT_b is given by $M^* \cdot A$, using the same `closure` algorithm.

Dataflow analyses can be treated more generally by studying the *transfer functions* of each basic block. We consider backwards analyses (like liveness analysis), where the transfer functions specify IN_b given OUT_b (the discussion applies equally well to forwards analyses, with suitable relabelling).

The equations have the following form:

$$\mathrm{IN}_b = f_b(\mathrm{OUT}_b)$$
$$\mathrm{OUT}_b = \operatorname*{join}_{b' \in \mathrm{succ}(b)} \mathrm{IN}_{b'}$$

or, more compactly,

$$\mathrm{OUT}_b = \operatorname*{join}_{b' \in \mathrm{succ}(b)} f(\mathrm{OUT}_{b'})$$

For many standard analyses, we can define a semiring where f_b is linear, and join is summation. This semiring is often the semiring of sets of variables (as above) or expressions, or its dual where $+$

is intersection, $\cdot$ is union, 0 is the entire set of variables and 1 is the empty set.

Such analyses are often referred to as bit-vector problems [11], as the sets can be represented efficiently using bitwise operations.

The available expressions analysis can be expressed as a linear system using this latter semiring, where the set of available expressions at the start of a block is the intersection of the sets of available expressions at the end of each predecessor.

Other analyses don't have such a simple structure. The *faint variables analysis* is an extension of live variables analysis which can detect that certain assignments are useless even though they are considered "live" by the standard analysis. For instance, in Figure 2, the variable z was considered live at the start of block C, even though all statements involving z can be deleted without affecting the meaning of the program. Live variable analysis will consider x "live" since it may be used on the next iteration. Faint variable analysis finds the *strongly live* variables: those used to compute a value which is not dead.

We can write transfer functions for faint variables analysis, which when given OUT_b compute IN_b. For instance, in our example $\mathtt{x} \in \mathrm{IN}_C$ if $\mathtt{x} \in \mathrm{OUT}_C$ (since x is used to compute the new value of x), and similarly $\mathtt{z} \in \mathrm{IN}_C$ if $\mathtt{z} \in \mathrm{OUT}_C$.

These transfer functions don't fall into the class of bitvector problems, and so our previous tactic won't work directly. However, they are in the more general class of *distributive dataflow* problems [10]: they have the property that $f_b(A \cup B) = f_b(A) \cup f_b(B)$. Happily, such transfer functions form a semiring.

We define a datatype to describe such functions which distribute over set union. For more generality, we consider an arbitrary commutative monoid instead of limiting ourselves to set union. Haskell has a standard definition of monoids, but they are not generally required to be commutative. We define the typeclass CommutativeMonoid for those monoids which are commutative. It has no methods and instances are trivial; it serves only as a marker by which the programmer can certify his monoid does commute.

```
class Monoid m => CommutativeMonoid m
instance Ord a => CommutativeMonoid (Set a)
```

With that done, we can define the semiring of transfer functions:

```
newtype Transfer m = Transfer (m -> m)
instance (Eq m, CommutativeMonoid m) =>
          Semiring (Transfer m) where
  zero = Transfer (const mempty)
  one = Transfer id
  Transfer f @+ Transfer g =
     Transfer (\x -> f x ‘mappend‘ g x)
  Transfer f @. Transfer g = Transfer (f . g)
```

Multiplication in this semiring is composition and addition is pointwise mappend, which is union in the case of sets. The distributive law is satisfied assuming all of the transfer functions themselves distribute over mappend.

The closure of a transfer function is a function f^* such that $f^* = 1 + f \cdot f^*$. When applied to an argument, we expect that $f^*(x) = x + f(f^*(x))$. The closure can be defined as a fixpoint iteration, which will give a suitable answer if it converges:

```
  closure (Transfer f) =
    Transfer (\x -> fixpoint (\y -> x ‘mappend‘ f y) x)
    where fixpoint f init =
              if init == next
                then init
                else fixpoint f next
            where next = f init
```

Convergence of this fixpoint iteration is not automatically guaranteed. However, it always converges when the transfer functions and the monoid operation are monotonic increasing in an order of finite height (such as the set of variables in a program), so this gives us a valid definition of closure for our transfer functions.

We can then calculate M^*, where M is the matrix of basic block transfer functions. M^* gives us their "transitive closure", which are the transfer functions of the program as a whole. Calling these functions with a trivial input (say, the empty set of variables in the case of faint variable analysis) allows us to generate the solution to the dataflow equations.

7. Polynomials, power series and knapsacks

Given any semiring R, we can define the semiring of polynomials in one variable x whose coefficients are drawn from R, which is written $R[x]$. We represent polynomials as a list of coefficients, where the ith element of the list represents the coefficient of x^i. Thus, $3 + 4x^2$ is represented as the list $[3, 0, 4]$.

We can start defining an instance of Semiring for such lists. The zero and unit polynomials are given by (where the one on the right-hand-side refers to r's one):

```
instance Semiring r => Semiring [r] where
  zero = []
  one = [one]
```

Addition is fairly straightforward: we add corresponding coefficients. If one list is shorter than the other, it's considered to be padding with zeros to the correct length (since $1 + 2x = 1 + 2x + 0x^2 + 0x^3 + \ldots$).

```
  [] @+ y = y
  x @+ [] = x
  (x:xs) @+ (y:ys) = (x @+ y):(xs @+ ys)
```

The head of the list representation of a polynomial is the constant term, and the tail is the sum of all terms divisible by x. So, the Haskell value a:p corresponds to the polynomial $a + px$, where p is itself a polynomial. Multiplying two of these gives us:

$$(a + px)(b + qx) = ab + (aq + pb + pqx)x$$

This directly translates into an implementation of polynomial multiplication:

```
  [] @. _ = []
  _ @. [] = []
  (a:p) @. (b:q) = (a @. b):(map (a @.) q @+
                             map (@. b) p @+
                             (zero:(p @. q)))
```

If we multiply a polynomial with coefficients a_i (that is, the polynomial $\sum_i a_i x^i$) by one with coefficients b_i resulting in the polynomial with coefficients c_i, then the coefficients are related by:

$$c_n = \sum_{i=0}^{n} a_i b_{n-i}$$

This is the discrete convolution of two sequences. Our definition of @. is in fact a pleasantly index-free definition of convolution.

In order to give a valid definition of Semiring, we must define the operation s^* such that $s^* = 1 + s \cdot s^*$. This seems impossible: for instance, there is no polynomial p such that $p = 1 + xp$, since the degrees of both sides don't match.

To form a closed semiring, we need to generalise somewhat and consider not just polynomials, but arbitrary *formal power series*, which are polynomials which may be infinite, giving us the semiring $R[[x]]$.

Our power series are purely formal, representing sequences of elements from a closed semiring. We have no notion of "evaluating" such a series by specifying x. We think of the formal power series $1 + x + x^2 + x^3 \ldots$ as the sequence $1, 1, 1, \ldots$, and require no infinite sums, limits or convergence. As such, "multiplication by x" simply means "shifting the series by one place", and we can write $pxq = pqx$ (but not $pqx = qpx$) even when the underlying semiring does not have commutative multiplication.

Since Haskell's lists are lazily defined and may be infinite, our existing definitions for addition and multiplication work for such power series, as demonstrated by McIlroy in his functional pearl [14].

Given a power series $s = a + px$, we seek its closure $s^* = b + qx$, such that $s^* = 1 + s \cdot s^*$:

$$\begin{aligned} b + qx &= 1 + (a + px)(b + qx) \\ &= 1 + ab + aqx + p(b + qx)x \end{aligned}$$

The constant terms must satisfy $b = 1 + ab$, so a solution is given by $b = a^*$. The other terms must satisfy $q = aq + ps^*$. This states that q is the fixpoint of an affine map, so a solution is given by $q = a^*ps^*$ and thus $s^* = a^*(1 + ps^*)$. This translates neatly into lazy Haskell as follows:

```
closure [] = one
closure (a:p) = r
  where r = [closure a] @. (one:(p @. r))
```

This allows us to solve affine equations of the form $x = bx + c$, where the unknown x and the parameters b and c are all power series over an arbitrary semiring. This form of problem arises naturally in some dynamic programming problems. As an example, we consider the unbounded integer knapsack problem.

We are given n types of item, with nonnegative integer weights $w_1, \dots, w_n$ and nonnegative integer values $v_1, \dots, v_n$. Our knapsack can only hold a weight W, and we want to find out the maximal value that we can fit in the knapsack by choosing some number of each item, while remaining below or equal to the weight limit W. This problem is NP-hard, but admits an algorithm with complexity polynomial in W (this is referred to as a *pseudo-polynomial time algorithm* since in general W can be exponentially larger than the description of the problem).

The algorithm calculates a table t, where $t(w)$ is the maximum value possible within a weight limit w. We set $t(0)$ to 0, and for all other w:

$$t(w) = \max_{0 \le w_i \le w} (v_i + t(w - w_i))$$

This expresses that the optimal packing of the knapsack to weight w can be found by looking at all of the elements which could be inserted and choosing the one that gives the highest value.

The algebraic structure of this algorithm becomes more clear when we rewrite it using the max-plus semiring that we earlier used to calculate longest paths, which we implemented in Haskell as `LongestDistance`. Confusingly, in this semiring the unit element is the number 0, since that is the identity of the semiring's multiplication, which is addition of path lengths. The zero element of this semiring is ∞, which is the identity of $\max$.

We take v_i and $t(w)$ to be elements of this semiring. Then, in this semiring's notation,

$$\begin{aligned} t(0) &= 1 \\ t(w) &= \sum_{i=0}^{w} v_i \cdot t(w - w_i) \end{aligned}$$

We can combine the two parameters v_i and w_i into a single polynomial $V = \sum_i v_i x^{w_i}$. For example, suppose we fill our knapsack with four types of coin, of values 1, 5, 7 and 10 and weights 3, 6, 8 and 6 respectively. The polynomial V is given by:

$$V = x^3 + 5x^6 + 7x^8 + 10x^6$$

Since we are using the max-plus semiring, this is equivalent to:

$$V = x^3 + 10x^6 + 7x^8$$

Represented as a list, the wth element of V is the value of the most valuable item with weight w (which is zero if there are no items of weight w). Similarly, we represent $t(w)$ as the power series $T = \sum_i t(i)x^i$. The list representation of T has as its wth element the maximal value possible within a weight limit w.

We can now see that the definition of $t(w)$ above is in fact the convolution of T and V. Together with the base case, that $t(0)$ is the semiring's unit, this gives us a simpler definition of $t(w)$:

$$T = 1 + V \cdot T$$

The above can equally be written as $T = V^*$, and so we get the following elegant solution to the unbounded integer knapsack problem (where `!!` is Haskell's list indexing operator):

```
knapsack values maxweight = closure values !! maxweight
```

Note that our previous intuition of x^* being the infinite sum $1 + x + x^2 + \dots$ applies nicely here: the solution to the integer knapsack problem is the maximum value attainable by choosing no items, or one item, or two items, and so on for any number of items.

Instead of using the `LongestDistance` semiring, we can define `LongestPath` in the same way that we defined `ShortestPath` above, with $\max$ in place of $\min$. Using this semiring, our above definition of `knapsack` still works and gives the set of elements chosen for the knapsack, rather than just their total value.

8. Linear recurrences and Petri nets

The power series semiring has another general application: it can express linear recurrences between variables. Since the definition of "linear" can be drawn from an arbitrary semiring, this is quite an expressive notion.

As we are discussing functional programming, we are obliged by tradition to calculate the Fibonacci sequence.

The nth term of the Fibonacci sequence is given by the sum of the previous two terms. We construct the formal power series F whose nth coefficient is the nth Fibonacci number. Multiplying this sequence by x^k shifts along by k places, so we can rewrite the recurrence relation as:

$$1 + xF + x^2F = F$$

This defines $F = 1 + (x + x^2)F$, and so $F = (x + x^2)^*$. So, we can calculate the Fibonacci sequence as `closure [0,1,1]`.

There are of course much more interesting things that can be described as linear recurrences and thus as formal power series. Cohen et al. [4] showed that a class of Petri nets known as *timed event graphs* can be described by linear recurrences in the max-plus semiring (the one we previously used for longest paths and knapsacks).

A timed event graph consists of a set of *places* and a set of *transitions*, and a collection of directed edges between places and transitions. Atomic, indistinguishable *tokens* are consumed and produced by transitions and held in places.

In a timed event graph, unlike a general Petri net, each place has exactly one input transition and exactly one output transition, as well as a nonnegative integer delay, which represents the processing time at that place. When a token enters a place, it is not eligible to leave until the delay has elapsed.

When all of the input places of a transition have at least one token ready to leave, the transition "fires". One token is removed from each input place, and one token is added to each output place of the transition. For simplicity, we assume that transitions are instant, and that a token arrives at all of the output places of a transition as soon as one is ready to leave each of the input places. If desired, transition processing times can be simulated by adding extra places.

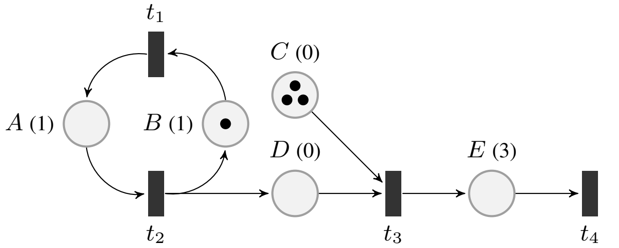


Figure 3. A timed event graph with four transitions t_1, t_2, t_3, t_4 and five places A, B, C, D, E with delays in parentheses, where all but two of the places are initially empty.

In Figure 3, the only transition which is ready to fire at time 0 is t_1. When it fires, it removes a token from B and adds one to A. This makes transition t_2 fire at time 1, which adds a token to B causing t_1 to fire at time 2, then t_2 to fire at time 3 and so on.

When t_2 fires at times $1, 3, 5, \dots$, a token is added to place D. The first three times this happens, t_3 fires, but after that the supply of tokens from C is depleted. t_4 fires after the tokens have waited in E for a delay of three steps, so t_4 fires at times 4, 6 and 8.

Simulating such a timed event graph requires calculating the times at which tokens arrive and leave each place. For each place p, we define the sequences $\mathrm{IN}(p)$ and $\mathrm{OUT}(p)$. The ith element of $\mathrm{IN}(p)$ is the time at which a token arrives into p for the ith time. The ith element of $\mathrm{OUT}(p)$ is the time at which a token becomes available from p for the ith time, which may be some time before it actually leaves p.

In the example of Figure 3, we have:

$$\begin{aligned}
\mathrm{IN}(A) &= 0, 2, 4, 6 \dots & \mathrm{OUT}(A) &= 1, 3, 5, 7 \dots \\
\mathrm{IN}(B) &= 1, 3, 5, 7 \dots & \mathrm{OUT}(B) &= 0, 2, 4, 6 \dots \\
\mathrm{IN}(C) &= \text{—} & \mathrm{OUT}(C) &= 0, 0, 0 \\
\mathrm{IN}(D) &= 1, 3, 5, 7 \dots & \mathrm{OUT}(D) &= 1, 3, 5, 7 \dots \\
\mathrm{IN}(E) &= 1, 3, 5 & \mathrm{OUT}(E) &= 4, 6, 8
\end{aligned}$$

We say that a place p' is a predecessor of p (and write $p' \in \mathrm{pred}(p)$) if the output transition of p' is the input transition of p. Since transitions fire instantly, a place receives a token as soon as all of its predecessors are ready to produce one.

$$\mathrm{IN}(p)_i = \max_{p' \in \mathrm{pred}(p)} \mathrm{OUT}(p')_i$$

Exactly when the ith token becomes available from a place p depends on the amount of time tokens spend processing at p, which we write as $\mathrm{delay}(p)$, and on the number of tokens initially in p, which we write as $\mathrm{nstart}(p)$. The times at which the first $\mathrm{nstart}(p)$ tokens become ready to leave p are given by the sequence $\mathrm{START}(p)$, which is nondecreasing and each element of which is less than $\mathrm{delay}(p)$. In the example we assume the initial tokens of B and C are immediately available, so we have $\mathrm{START}(B) = 0$ and $\mathrm{START}(C) = 0, 0, 0$.

Thus, the time that the ith token becomes available from p is given by:

$$\mathrm{OUT}(p)_i = \begin{cases} \mathrm{START(p)}_i & i < \mathrm{nstart}(p) \\ \mathrm{IN}(p)_{i - \mathrm{nstart}(p)} + \mathrm{delay}(p) & i \geq \mathrm{nstart}(p) \end{cases}$$

By adopting the convention that $\mathrm{IN}(p)_i$ is $-\infty$ when $i < 0$ and that $\mathrm{START}(p)_i$ is $-\infty$ when $i < 0$ or $i \geq \mathrm{nstart}(p)$, we can write the above more succinctly as:

$$\mathrm{OUT}(p)_i = \max(\mathrm{START(p)}_i, \mathrm{IN}(p)_{i - \mathrm{nstart}(p)} + \mathrm{delay}(p))$$

This gives a set of recurrences between the sequences: the value of $\mathrm{OUT}(p)$ depends on the previous values of $\mathrm{IN}(p)$.

We now shift notation to make the semiring structure of this problem apparent. We return to the max-plus algebra, previously used for longest distances and knapsacks, where we write $\max$ as $+$, and addition as $\cdot$. Instead of sequences, let's talk about formal power series, where the ith element of the sequence is now the coefficient of x^i. With our semiring goggles on, the above equations now say:

$$\begin{aligned}
\mathrm{IN}(p) &= \sum_{p' \in \mathrm{pred}(p)} \mathrm{OUT}(p') \\
\mathrm{OUT}(p) &= \mathrm{delay}(p) \cdot x^{\mathrm{nstart(p)}} \cdot \mathrm{IN}(p) + \mathrm{START}(p)
\end{aligned}$$

We can eliminate $\mathrm{IN}(p)$ by substituting its definition into the second equation:

$$\mathrm{OUT}(p) = \sum_{p' \in \mathrm{pred}(p)} \mathrm{delay}(p) \cdot x^{\mathrm{nstart(p)}} \cdot \mathrm{OUT}(p') + \mathrm{START}(p)$$

What we're left with is a system of affine equations, where the unknowns, the coefficients and the constants are all formal power series over the max-plus semiring.

We can solve these exactly as before. We build the matrix $\mathbf{M}$ containing all of the $\mathrm{delay}(p) \cdot x^{\mathrm{nstart}(p)}$ coefficients, and the column vector $\mathbf{S}$ containing all of the $\mathrm{START}(p)$ sequences, and then calculate $\mathbf{M}^* \cdot \mathbf{S}$ (which, as before, can be done with a single call to `closure` and a multiplication by $\mathbf{S}$). The components of the resulting vector are power series; their coefficients give $\mathbf{OUT}(p)$ for each place p.

Thus, we can simulate a timed event graph by representing it as a linear system and using our previously-defined semiring machinery.

9. Discussion

It turns out that very many problems can be solved with linear algebra, for a definition of "linear" suited to the problem at hand. There are surprisingly many questions that can be answered with a call to `closure` with the right `Semiring` instance. Even still, this paper barely scratches the surface of this rich theory. Much more can be found in books by Gondran and Minoux [9], Golan [7, 8] and others.

The connections between regular languages, path problems in graphs, and matrix inversion have been known for some time. The relationship between regular languages and semirings is described in Conway's book [5]. Backhouse and Carré [3] used regular algebra to solve path problems (noting connections to classical linear algebra), and Tarjan [17] gave an algorithm for solving path problems by a form of Gaussian elimination.

A version of closed semiring was given by Aho, Hopcroft and Ullman [2], along with transitive closure and shortest path algorithms. The form of closed semiring that this paper discusses was given by Lehmann [12], with two algorithms for calculating the closure of a matrix: an algorithm generalising the Floyd-Warshall all-pairs shortest-paths algorithm [6], and another generalising Gaussian elimination, demonstrating the equivalence of these two in their general form. More recently, Abdali and Saunders [1] reformulate the notion of closure of a matrix in terms of "eliminants", which formalise the intermediate steps of Gaussian elimination.

The use of semirings to describe path problems in graphs is widespread [9, 16]. Often, the structures studied include the extra axiom that $a + a = a$, giving rise to *idempotent semirings* or *dioids*. Such structures can be partially ordered, and it becomes possible to talk about least fixed points of affine maps. These have proven strong enough structures to build a variant of classical real analysis [13].

Cohen et al. [4], as well as providing the linear description of Petri nets we saw in section 8, go on to develop an analogue of classical linear systems theory in a semiring. In this theory, they explore semiring versions of many classical concepts, such as stability of a linear system and describing a system's steady-state as an eigenvalue of a transfer matrix.

Acknowledgments

The author is grateful to Alan Mycroft for suffering through early drafts of this work and offering helpful advice, and to Raphael Proust, Stephen Roantree and the anonymous reviewers for their useful comments, and finally to Trinity College, Cambridge for financial support.

References

[1] S. Abdali and B. Saunders. Transitive closure and related semiring properties via eliminants. *Theoretical Computer Science*, 40:257–274, 1985.

[2] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[3] R. Backhouse and B. Carré. Regular algebra applied to path-finding problems. *IMA Journal of Applied Mathematics*, 2(15):161–186, 1975.

[4] G. Cohen and P. Moller. Linear system theory for discrete event systems. In *23rd IEEE Conference on Decision and Control*, pages 539–544, 1984.

[5] J. Conway. *Regular algebra and finite machines*. Chapman and Hall (London), 1971.

[6] R. Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[7] J. S. Golan. *Semirings and their applications*. Springer, 1999.

[8] J. S. Golan. *Semirings and affine equations over them*. Kluwer Academic Publishers, 2003.

[9] M. Gondran and M. Minoux. *Graphs, dioids and semirings*. Springer, 2008.

[10] J. Kam and J. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.

[11] U. Khedker and D. Dhamdhere. A generalized theory of bit vector data flow analysis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(5):1472–1511, 1994.

[12] D. Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76, 1977.

[13] V. P. Maslov. *Idempotent analysis*. American Mathematical Society, 1992.

[14] M. D. Mcilroy. Power series, power serious. *Journal of Functional Programming*, 9(3):325–337, 1999.

[15] R. McNaughton and H. Yamada. Regular expressions and state graphs for automata. *IRE Transactions on Electronic Computers*, 9(1):39–47, 1960.

[16] M. Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7(3): 321–350, 2002.

[17] R. Tarjan. Solving path problems on directed graphs. Technical report, Stanford University, 1975.