Haskell Solutions to the Language Session Problems at the 1988 Salishan High-Speed Computing Conference

Paul Hudak and Steve Anderson

Research Report YALEU/DCS/RR-627
May 1988

This research was supported primarily by DOE grant FG02-86ER25012

Haskell Solutions to the Language Session Problems at the 1988 Salishan High-Speed Computing Conference

Paul Hudak
Steve Anderson

Research Report YALEU/DCS/RR-627*
Yale University
Department of Computer Science

May 11, 1988

1 Introduction

Haskell is a new functional language, named after the logician Haskell B. Curry, being designed by a 14-member international committee representative of the functional programming research community.[1] The committee was formed because it was felt that research and application of modern functional languages was being hampered by the lack of a standard language. Having a standard would allow faster communication of new ideas, provide a stable foundation for real applications development, and encourage other people to learn and use functional languages. The committee's goals in designing Haskell include:

1. It should be suitable for teaching, research, and applications, including building large systems.
2. It should be completely described via the publication of a formal syntax and semantics.
3. It should be freely available. Anyone should be able to implement the language and distribute it to whomever they please.
4. It should be based primarily on well-tried ideas that enjoy a wide consensus.
5. It should be useable as a basis for further research. It is expected and desirable that extensions or variants of the language may appear, incorporating experimental features (for example, para-functional programming constructs).

*This research was supported primarily by DOE grant FG02-86ER25012.

[1]The committee members are Arvind (MIT), Richard Bird (Oxford University), Guy Cousineau (INRIA), Jon Fairbairn (Cambridge University), Joe Fasel (Los Alamos National Laboratory), Paul Hudak (Yale University), John Hughes (Glasgow University), Thomas Johnsson (Chalmers Institute of Technology), Dick Kieburtz (Oregon Graduate Center), Simon Peyton-Jones (University College London), Rinus Plasmeijer (University of Nijmegen), Mike Reeve (Imperial College), Philip Wadler (Glasgow University), and David Wise (Indiana University).

Haskell is a general purpose, purely functional programming language exhibiting many of the recent innovations in programming language research, including higher-order functions, non-strict functions and data structures, static polymorphic typing, user-definable algebraic data types, pattern-matching, list comprehensions, a module system, state-transition–based I/O, and a rich set of primitive data types, including lists, arrays, arbitrary and fixed precision integers, and complex, rational, and floating-point numbers.

This paper is intended to give the reader some familiarity with Haskell by giving solutions to the four language session problems presented at the 1988 Salishan Conference on High-Speed Computing. Unfortunately, the Haskell standard is not yet available! The committee has been designing the language since September of '87, and a preliminary draft of the standard will be completed by June 1, 1988; the final draft is expected to appear around August, 1988.

Since the committee has concentrated foremost on the core semantical issues, the concrete syntax is currently the least defined component of the language. Therefore we will present the solutions in a syntax that is typical of modern functional languages, although the reader should be aware that the actual Haskell syntax may be quite a bit different. In any case, the solutions will give the reader an idea of what programming in a modern functional language is like, and that is our main goal. To emphasize the preliminary nature of the syntax, from now on we will refer to the language as "pre-Haskell." We will precede each section with a description of the problem as presented at the Workshop. All of the solutions given have been run on the Alpha-Tau implementation of Alfl, our functional language implementation at Yale.

2 Brief Overview of Pre-Haskell Syntax

In this section we will describe enough pre-Haskell syntax to allow understanding the programs given later. As a result, there are significant parts of pre-Haskell that won't be described at all, most notably user-defined data types, modules, and I/O.

Pre-Haskell is an "equational" language similar to Miranda, Hope, and several other modern functional languages. A function is defined by a set of equations which can *pattern-match* against their arguments. Lists are written `[a,b,c]` with `[]` being the empty list. An element `a` may be added to the front of the list `as` by writing `a:as`. Two lists may be appended together by `l1::l2`. Here is an example of pattern-matching:

```
member x []    = false
member x (y:l) = x=y -> true,
                      member x l
```

Note in this example the use of a *conditional*, which has the general form "`predicate -> consequence, alternate.`"

List comprehensions are a concise way to define lists, and are best explained by example:

```
[ (x,y) | x<-l1; y<-l2 ]
```

which constructs the list of all pairs whose first element is from l1, and second is from l2. "Infinite lists" may also be defined, and thanks to lazy evaluation, only that portion of the list that is needed by some other part of the program is actually computed. Thus the infinite list of ones can be defined by:

```
ones = 1:ones
```

The notation a..b denotes the list of integers from a to b, inclusive, and a.. is the infinite ascending list of integers beginning with a.

There are many standard utility functions defined on lists. The ones we need in this paper are the following:

```
takeWhile pred    []   = []
takeWhile pred (a:as) = pred a -> a : takeWhile pred as, []

foldl f a    []   = a
foldl f a (x:xs) = foldl f (f a x) xs

foldr f a    []   = a
foldr f a (x:xs) = f x (foldr f a xs)

zip   as      []    = []
zip   []      bs    = []
zip (a:as) (b:bs) = (a,b):(zip as bs)
```

Note in zip the use of *tuples*, which in pre-Haskell are constructed in arbitrary but finite length by writing "a,b, ..., c" and may be pattern-matched like lists.

Note that for foldl and foldr the following relationships hold:

```
foldl f a [x1, x2, ..., xn]  ==>  (f ... (f (f a x1) x2) ... xn)
foldr f a [x1, x2, ..., xn]  ==>  (f x1 (f x2 ... (f xn a) ... ))
```

Pre-Haskell also has *arrays* and a special syntax for manipulating them, best explained by an example:

```
{ 2D_array (1,n),(1,n)
     | [i,j] = k*a[i,j] || i<-1..n, j<-1..n }
```

which returns a two dimensional array representing the matrix a multiplied by the scalar k.

A function f x = x+1 may be defined "anonymously" with expression \ x. x+1 and thus (\ x. x+1) 2 returns 3.

This description of pre-Haskell is quite brief, but should be enough to make the programs given later self-explanatory. Nevetheless, experience with at least one other functional language would be beneficial.

3 Hamming's Problem (Extended)

"Given as input a finite increasing sequence of primes $\langle a, b, c, ...\rangle$ and an integer n, output in order of increasing magnitude and without duplication all integers less than or equal to n of the form:

$$a^i b^j c^k ..., \quad i, j, k, ... \geq 0$$

Notice that if m is in the output sequence then so are:

$$am, \ bm, \ cm, \ ... \ \leq n$$

Our intention in posing the problem is to see how each language expresses such mutually recursive stream computations."

A natural way to solve this problem in pre-Haskell is to generate an infinite increasing sequence of hamming numbers, and then filter out those less than n. But how do we create that infinite sequence? To start, let's define a function `scale` that multiplies every element in a stream by a certain number:

```
scale p xs = [ p*x | x<-xs ]
```

Now note that a "constructive" way to express the hint is as an inductive definition:

- 1 is in the output sequence.
- For each prime p, if k is in the output sequence, then so is $k * p$.

We can construct a dataflow diagram for this as shown in Figure ??a, where the repeating pattern has been highlighted in a box. Capturing the box's functionality in a function `f`, and using `foldl` to "unfold" `f` over the list of primes, we arrive at this straightforward program to realize the dataflow diagram:

```
hamming primes =
    h where h = 1 : (foldl f [] primes)
            f xs p = merge xs (scale p h)
```

where `merge` merges a list of streams in increasing numeric order. Unfortunately, `merge` must also *remove duplicates*, since this simple definition will construct every permutation of the factors for a particular number. For example, it will generate three 12's: 2*2*3, 2*3*2, and 3*2*2. This is of course inefficient, and we'd prefer a solution that avoided the extra multiplications.

The problem stems from the fact that the sub-streams are generated recursively from the *entire* list `h`. What we really want is something that "chases its tail" so as to avoid generating all of the combinations. The dataflow diagram in Figure ??b in fact does just that – note how the result of each merge is fed back only to itself, thus avoiding the duplicates. As before we can express this result by abstracting the repeating functionality and using `foldl`:

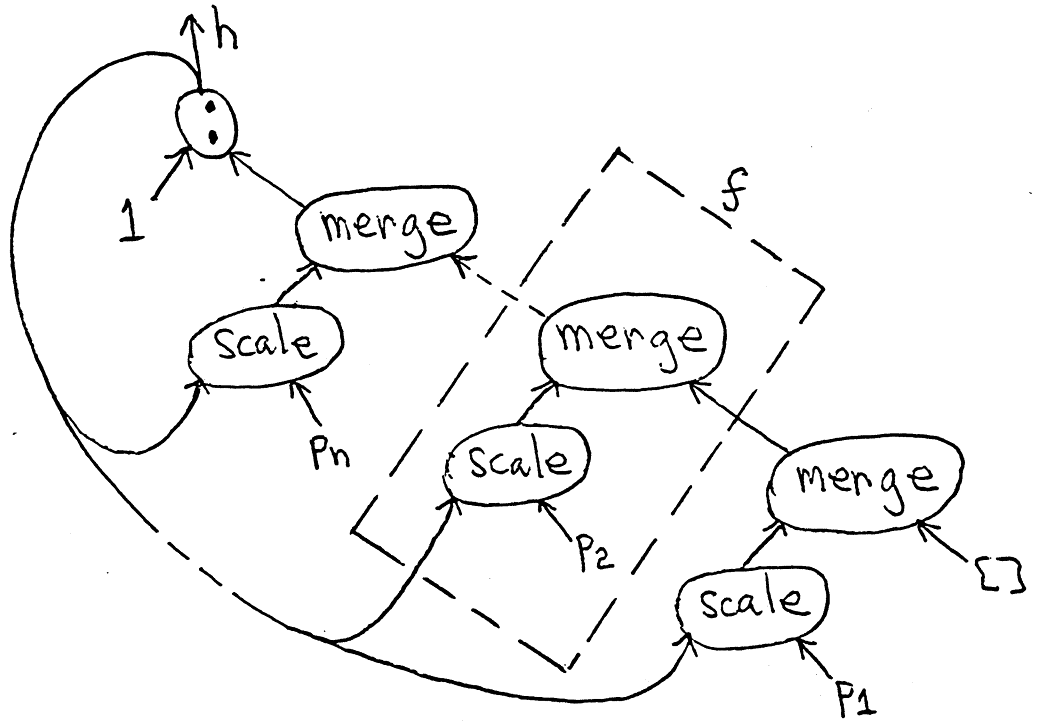

Figure 1: Naive Hamming Solution

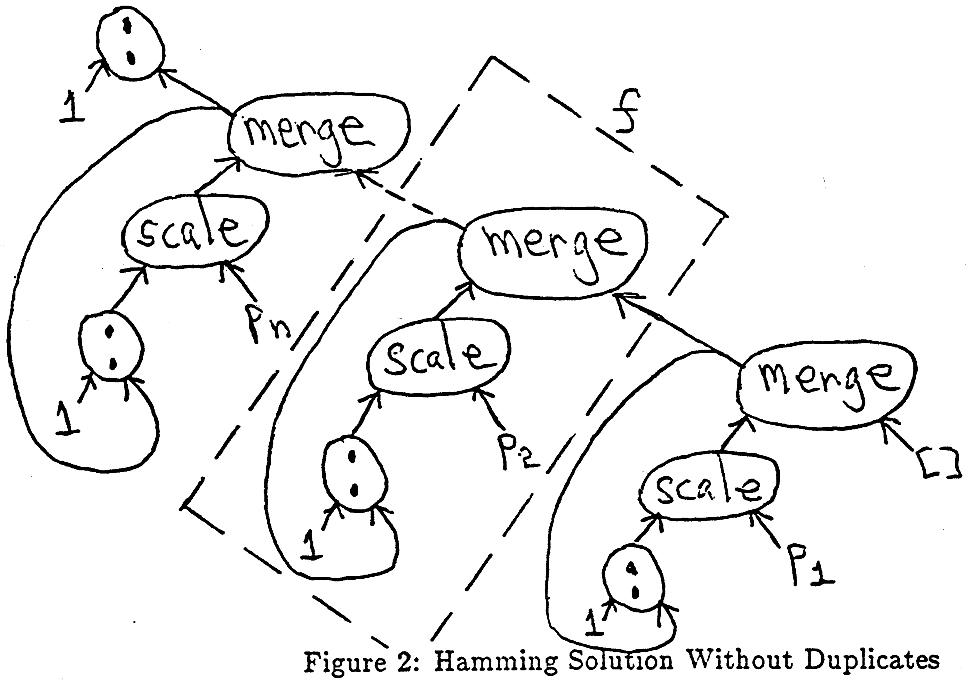

Figure 2: Hamming Solution Without Duplicates

```
    hamming primes = 1 : (foldl f [] primes)
                     where f xs p = h where
                                    h = merge (scale p (1:h)) xs
```

in which case merge is defined simply by:

```
    merge (a:as) (b:bs) = a<b -> a : (merge as (b:bs)),
                                 b : (merge (a:as) bs)
    merge [] bs = bs
    merge as [] = as
```

and the result is just:

```
    takeWhile (\ x. x<n) (hamming primes)
```

using the utility takeWhile defined in the introduction.

Here is a sample output transcript, run on our Alfl implementation:

```
  takeWhile (\ x. x<46) (hamming [2,3,5]);

  Result: (1 : 2 : 3 : 4 : 5 : 6 : 8 : 9 : 10 : 12 : 15 : 16 : 18 :
          20 : 24 : 25 : 27 : 30 : 32 : 36 : 40 : 45 : [])
```

4 The Paraffin Problem

"The chemical formula for paraffin molecules is C_iH_{2i+2}. Given an integer n, output without repetition and in order of increasing size, structural representations of all paraffin molecules for $i \leq n$. Include all isomers, but no duplicates. You may choose any representation for the molecules you wish, so long as it clearly distinguishes among isomers." The problem is discussed in:

Turner, D. A., The semantic elegance of applicative languages. Proc. Conf. on Functional Programming Languages and Computer Architecture, Portsmouth, NH, 1981 Oct., pp. 85-92.

This problem was solved in the above reference using the functional language Miranda, which happens to be similar to pre-Haskell, and thus our job is already done for us! Actually there are more efficient algorithms for solving this problem, but no more insight into understanding Haskell will be gained by giving them. Thus we will simply duplicate Turner's solution below, and refer to the paper referenced above for a detailed description of it:

```
    output = layn (append (map paraffin 1..))

    paraffin n = quotient equiv [ [x,"H","H","H"] | x<-(para (n-1)) ]
```

```
para 0 = ["H"]
para n = paralist n

paralist = map genpara (1..)

genpara n = [ [a,b,c] | i<-0..(n-1)/3; j<-i..(n-1-i)/2;
                        a<-(para i); b<-(para j); c<-(para (n-1-i-j)) ]

equiv a b = member (equivclass a) b
equivclass x = closure_under_laws [invert, rotate, swap] [x]

invert [[a,b,c],d,e,f] = [a,b,c,[d,e,f]]
invert     ("H":x)     = "H":x
rotate [a,b,c,d] = [b,c,d,a]
swap   [a,b,c,d] = [b,a,c,d]

closure_under_laws f s = s :: closure' f s s
closure'  f s t = closure'' f s (mkset [ a | f'<-f; a<-(map f' t);
                                                         ~member s a ])
closure'' f s t = (t=[]) -> [],
                  t :: closure' f (s :: t) t

quotient f (a:x) = a : [ b | b<-(quotient f x); ~f a b ]
quotient f [] = []
```

5 A Doctor's Office

"Given a set of patients, a set of doctors, and a receptionist, model the following interactions: Initially, all patients are well, and all doctors are in a queue awaiting sick patients. At random times, patients become sick and enter a queue for treatment by one of the doctors. The receptionist handles the two queues, assigning patients to doctors in a first-in-first-out manner. Once a doctor and patient are paired, the doctor diagnoses the illness and, in a randomly chosen period of time, cures the patient. Then, the doctor and patient return to the receptionist's desk, where the receptionist records pertinent information. The patient is then released until such time as he or she becomes sick again, and the doctor returns to the queue to await another patient.

You may use any distribution functions you wish to decide when a patient becomes sick and how long a patient sees a doctor, but the code that models doctors must have no knowledge of the distribution function for patients, and vice versa, and that for the receptionist should know nothing of either. The receptionist may record any information you wish: patient's name, doctor assigned, illness, cure, wait times, queue lengths, etc. The purpose of the problem is to evaluate how each language expresses asynchronous communications from multiple sources."

Of the four problems, this is probably the least well-defined. The main difficulty lies in just what is meant by the verb "model" in the first sentence. Perhaps the most common kind of modelling is a simulation of the actual time/event pairs, and that is what the first solution (written by Joe Fasel) presented below does. However, such a solution removes completely the non-determinism and asynchrony of the problem (since they are being simulated!), which conflicts with the statement made in the *last* sentence of the problem description. Thus we also provide a solution that uses explicit non-determinism. The two solutions are radically different, and reflect very different characteristics of pre-Haskell.

5.1 Time/event Simulation

This model of the doctors' office takes as input a number of patients, a number of doctors, an initial list of times at which patients get sick, and two infinite lists of durations, representing the distributions of times that patients remain well and of the times doctors take to cure patients. An infinite list of tuples is returned, containing the following information for each office visit:

```
(patient, sick-time, doctor, start-treatment-time, cure-time)
```

That is, a patient number, the time the patient got sick and entered the patient queue, the number of the doctor assigned, the time at which the patient was assigned a doctor, and the time the doctor finished treating the patient.

The style of this solution is to create mutually recursive streams of time/event pairs, merging them together at appropriate places while preserving the temporal order. The main streams of events are patients (`patientQ`), doctors (`doctorQ`), and cured people (`cured`), as shown below. `insert` and `makeQ` are utilities for handling queues of time-event pairs.

```
doctors n m initialWellDist WellDist CureDist = cured

where insert  y       []         = [y]  -- insert y into time-ordered queue
      insert (p',t') ((p,t):xs) = t'<t -> (p',t'):(p,t):xs,
                                   (p,t):(insert (p',t') xs)

      makeQ (x:xs) yys                  -- initial queue (in order),
                                        -- subsequent entries (not in order)
          = x : makeQ (insert y xs) ys where y:ys = yys

      patientQ                          -- [(patient, sick-time)]
          = makeQ (foldr insert [] (zip (1..n) initialWellDist))
                  [(p,c+x) | (p,s,d,t,c),x <- cured,wellDist]

      doctorQ                           -- [(doctor, time-available)]
          = makeQ [(d,0) | d <- 1..m]
                  [(d,c) | (p,s,d,t,c) <- cured]

      cured
```

```
= [(p,s,d,t,t+x) where t = max s a
   | (p,s),(d,a),x <- patientQ,doctorQ,cureDist]
```

5.2 Asynchronous Process Model

In the following solution the "world" is modelled as a 6-tuple:

```
[healthy_people, -- list of healthy people
 sick_people,    -- queue of sick people
 being_cured,    -- list of sick-people/doctor pairs
 cured_people,   -- queue of cured-people/doctor pairs
 doctor_q,       -- queue of available doctors
 record]         -- receptionist's record of pertinent data
```

This representation is actually more detailed, and thus more realistic, than the previous one. In particular, note the presence of a record book, as well as a queue to hold the doctor/patient pairs reporting back to the receptionist after a curing session (this queue is not called for in the specification, but seems more realistic). The initial state of the world should be obvious:

```
initial_world = (1..n,    -- everybody's healthy
                 [],      -- nobody's sick
                 [],      -- nobody's being cured
                 [],      -- nobody's just been cured
                 1..m,    -- every doctor is idle
                 [])      -- no record of curing
```

The dynamics of this model are captured by three "processes" that operate non-deterministically (i.e. asynchronously) and in parallel. Each process takes as input a world and outputs a "new" world. Simulation of the doctors office proceeds by starting with the initial world and iteratively choosing a process non-deterministically with which to generate a new world on each step of the simulation. The result is an infinite stream of worlds.

```
    doctors_office world = choose_loop world processes

    processes = [sickening_process, curing_process, receptionist]

    sickening_process w = h=[] -> w,  -- everybody's already sick!!
                          [hs,per:s,b,c,d,r]
                          where [h,s,b,c,d,r] = w
                                (per:hs) = sicken_one h

       curing_process w = b=[] -> w,  -- nobody's being cured
                          [h,s,rest,[doc,per]:c,d,r]
                          where [h,s,b,c,d,r] = w
                                ([doc,per]:rest) = cure_one b

         receptionist w = choose [process_the_sick,process_the_cured] w
         where process_the_sick w = s=[] -> w,       -- nobody's sick
                                    d=[] -> w,       -- no available doctors
                                    [h,ss,[doc,per]:b,c,ds,r]
                                    where [h,s,b,c,d,r] = w
                                          per:ss,doc:ds = s,d
              process_the_cured w = c=[] -> w,       -- nobody's just been cured
                                    [per:h,s,b,rest,d::[doc],[doc,per]:r]
                                    where [h,s,b,c,d,r] = w
                                          ([doc,per]:rest) = c

      cure_one = choose_and_remove     -- random curing function
      sicken_one = choose_and_remove   -- random sickening function

      choose_and_remove lst = el : remove el lst
                              where el = choose lst
                                    remove x [] = []
                                    remove x (y:xs) = x=y -> xs, remove x xs

      choose_loop obj fs = new_obj : choose_loop new_obj fs
                           where new_obj = (choose fs) obj
```

Note that the non-deterministic utility functions are built from a single non-deterministic primitive called choose that non-deterministically selects an element from a list.

This non-deterministic process model, by the way, could be made deterministic by providing lists of sickness and wellness distributions as in the time/event simulation. Similarly, the time/event simulation could be made non-deterministic by suitably merging the event streams non-deterministically.

6 Skyline Matrix Solver

"Solve the system of linear equations:

$$A\,x = b$$

where A is an n by n skyline matrix. A skyline matrix has nonzero elements in column j in rows i through j, $1 \leq i \leq j$, and has nonzero elements in row i in columns j through i, $1 \leq j \leq i$. The first constraint defines the skyline above the diagonal, which is towards the top, and the second constraint defines the skyline below the diagonal, which is towards the left. For example, if

$$A = \begin{bmatrix} X & 0 & 0 & x & 0 & 0 & 0 \\ 0 & X & 0 & x & 0 & x & 0 \\ 0 & x & X & x & x & x & 0 \\ 0 & 0 & 0 & X & x & x & 0 \\ 0 & x & x & x & X & x & 0 \\ 0 & 0 & 0 & x & x & X & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & X \end{bmatrix}$$

then the i vector of the first constraint is:

$$\langle 1,2,3,1,3,2,7 \rangle$$

and the j vector of the second constraint is:

$$\langle 1,2,2,4,2,4,7 \rangle$$

You may assume any input form for A and b you wish, and you may assume the i and j vectors as input parameters. A rather obscure reference for the problem (available on request) is:

Eisenstat, S.C., and Sherman, A.H. *Subroutines for envelope solution of sparse linear systems.* Research Report 35, Yale University, New Haven CT, October 1974.

The intention of this problem is to test each language's ability to manipulate arrays, to use the structure of arrays to avoid unnecessary computations, and to express array operations."

Our understanding of this problem was aided greatly not only by the above tech report, but also a copy of some Fortran code written by Andy Sherman which implements an envelope method for solving a linear system. That code, complete with documentation, is listed in the Appendix (actually only the subrountines PLU and PLUB are listed, but they are the only two relevant to the probelm as given).

Having Sherman's code provided us with an opportunity to study Fortran-style incremental array manipulations in a functional language, and to contrast that with the preferred monolithic array approach. We think the results are quite interesting. To conduct the study we first converted, as faithfully as possible, the Fortran code into pre-Haskell using incremental updates to purely functional arrays. We then rewrote the program in a monolithic style, adhering more closely to the matrix algebra, but using the same envelope representations used by Sherman.

6.1 Incremental Array Solution

Strict incrementally-updated arrays will probably not be part of the Haskell standard, but are the closest thing to conventional arrays in imperative programming languages, and thus are the easiest way to translate a Fortran program into a functional language. An incremental array `a` of dimensionality `d` is created by `new_array d`, and is updated with value `x` at position `i` by `upd a i x`. This updated array is conceptually an entirely new array, and in fact references to `a` will still reveal `a`'s previous state. Fortunately, most uses of such arrays (especially in scientific computing) are *single-threaded*, meaning that after the update no references to the old array will take place, and thus an optimizing compiler if often able to implement the update destructively, thus achieving the same efficiency as the equivalent imperative program.

To further aid in the translation of Sherman's code it is convenient to have an idiom for simulating DO loops, which we do by defining the following higher-order function:

```
do i j acc f = j>i -> acc, do (i+1) j (f i acc) f;
```

Thus, for example:

```
do 1 n (new_array n)
(\ i a. upd a i 0)
```

initializes a new array to zero, and is equivalent to the Fortran code:

```
dimension a(n)
do 10 i=1,n
a(i) = 0
```

If `(pl,d,pu)` is the original `n` by `n` skyline matrix A, whose structure is described by `irl` and `iru`; `b` is the original B vector before reordering; and `iord` is the reordering matrix; then the result can be computed using Sherman's code as follows:

```
dimension x(n)
call plu(n,pl,d,pu,irl,iru)
call plub(n,pl,d,pu,irl,iru,x,b,iord)
```

which leaves the result in `x`. In the pre-Haskell version of the Fortran code presented below, the solution is computed via:

```
plub n pl' d' pu' irl iru x b iord
where x = new_array n
      (d',pl',pu') = plu n pl d pu irl iru
```

Here is the pre-Haskell program that simulates Andy Sherman's Fortran program using strict incremental arrays. It is written completely without comments, and should be compared directly against Sherman's code (which is commented) in the Appendix to see the correspondence.

```
plu n pl d pu irl iru =          -- n, irl, and iru are "pure input"
do 2 n (upd d 1 (1./d[1]),pl,pu)
\ i (d,pl,pu). (d',pl',pu')
where irli = irl[i]
      irui = iru[i]
      ifli = i - 1 + irl[i-1] - irli
      jminl = ifli + 1
      ifui = i - 1 + iru[i-1] - irui
      jminu = ifui + 1
      jmax = i - 1
      pl' = do jminl jmax pl
            \ j pl. upd pl (irli+j) (-plij) where
                        iruj = iru[j]
                        ifuj = j - 1 + iru[j-1] - iruj
                        kmin = max0 ifli ifuj
                        kmax = j - 1
                        plij = do kmin kmax (-pl[irli+j])
                               \ k plij. plij + pl[irli+k]*pu[iruj+k]
      pu' = do jminu jmax (upd pu (irui+jminu-1) (pu[irui+jminu-1]*d[jminu-1]))
            \ j pu. upd pu (irui+j) (-puji*d[j]) where
                        irlj = irl[j]
                        iflj = j - 1 + irl[j-1] - irlj
                        kmin = max0 ifui iflj
                        kmax = j - 1
                        puji = do kmin kmax (-pu[irui+j])
                               \ k puji. puji + pl[irlj+k]*pu[irui+k]
      jmin = max0 ifli ifui
      di = do jmin jmax (-d[i])
           \ j di. di + pl'[irli+j]*pu'[irui+j]
      d' = upd d i (-1./di)

plub n pl d pu irl iru x b iord = (x3,iord2)
where x1 = do 2 n (upd x 1 (b[iord[1]]*d[1]))
           \ j x. upd x j (-xj*d[j])
                  where irlj = irl[j]
                        kmin = j - 1 + irl[j-1] - irlj
                        kmax = j - 1
                        xj = do kmin kmax (-b[iord[j]])
                             \ k xj. xj + pl[irlj+k]*x[k]
      x2 = do 1 (n-1) x1
           \ i x. do kmin kmax x
                  \ k x. upd x k (x[k] + xj*pu[iruj+k])
                  where j = n + 1 - i
                        iruj = iru[j]
                        kmin = j - 1 + iru[j-1] - iruj
```

```
                         kmax = j - 1
                         xj = - x[j]
       x3,iord1 = do 1 n (x2,iord)
                  \ k (x,iord). iord[k]<0 -> (x,iord),
                                loop k iord x
                                where loop i iord x =
                                      iordi=k -> (x',iord'),
                                                 loop iordi iord' x'
                                      where iordi = iord[i]
                                            x' = upd (upd x iordi x[k])
                                                     k x[iordi]
                                            iord' = upd iord i (-iord[i])
       iord2 = do 1 n iord1
             \ i iord. upd iord i (-iord[i])
```

6.2 Monolithic Array Solution

The above solution is given primarily to illustrate how one *could* do incremental array operations in a functional language that "have the feel" of side effects to arrays in an imperative language. In fact, the above program, when run on our Alpha-Tau implementation of Alfl, achieves the same space complexity of the Fortran program. That is, our optimizer is able to infer that every array is "single-threaded" and thus updates can be done destructively rather than by copying.

On the other hand, this is not the *preferred* way to program with arrays in a functional language. Pre-Haskell has a primitive data type for arrays together with special syntax that allows the specification of an array instance *monolithically* rather than incrementally. That is, the entire final array is specified in one monolithic declaration, yielding a declarative reading more in line with the philosophy of functional programming.

The skyline problem in fact illustrates well some of the special strengths of pre-Haskell arrays. In particular, the array specifications can be derived from the original mathematical definition of the problem in a clear and straightforward way. The essential data dependencies are clear, rather than obscured by extraneous operational sequencing. The recursive definition of arrays, including mutually recursive definitions of multiple arrays, permit elegant specifications as well as efficient implementations. Pre-Haskell arrays permit separate definitions for elements in different regions of an array, which permits optimizations similar to the lifting of computations from Fortran loops, and which clearly correspond to the mathematical function domain specifications.

In the last section we gave a solution which was essentially a transcribed version of Sherman's code, and thus we included no description of the data representations or the algorithm. In this section we will instead start from the very basics, and develop the final program via step-wise refinement of the specification. In particular, we derive all of the code optimizations that Sherman implements, in an attempt to show how functional languages can be "fine-tuned" for performance improvements in much the same way Fortran programs are.

6.2.1 Introduction to Sherman's envelope format for sparse matrices

Sherman's envelope format works best when the sparse linear system `A*x = b` has its equations and variables ordered such that most of A's nonzeros are close to the main diagonal. Each row `i` of the lower triangle is stored as an envelope from the leftmost nonzero in the row up to the last column `j = i-1` before the diagonal. Likewise, each column `j` of the upper triangle is stored as an envelope from the uppermost nonzero in the column down to the last row `i = j-1` before the diagonal. The main diagonal itself is stored as a 1-D vector of length `n`.

Sherman represents a sparse matrix as the tuple `[n, pl, d, pu, irl, iru]` where:

- `n` = the order of `A`.
- `pl, d, pu` = 1-D floating point vectors representing respectively the lower triangle's consecutively stored row envelopes, the main diagonal elements, and the upper triangle's consecutively stored column envelopes.
- `irl, iru` = 1-D length `n` integer vectors of base addresses into `pl` and `pu` respectively.

The base address vectors require some explanation. For access into lower triangle `pl`, let:

- `fl[i]` = the column index of the first nonzero in row `i`;
- `begin_l[i]` = the index into `pl` of row `i`'s first nonzero.

Then we would access `a[i,j]` in the lower triangle by: `pl [ begin_l[i] + j - fl[i] ] = a[i,j]`. But the value `begin_l[i] - fl[i]` is the same for every `j` in row `i`. In a later section we will see that computing an element `[i,j]` of either the lower or upper triangle factor requires an inner product summation that runs along the lower triangle's row `i` and the upper triangle's column `j`. For a sequential program it is simplest to make this summation the innermost loop, therefore we would like to raise this loop-invariant computation out of the innermost loop, replacing the $O(n^2)$ evaluations of the expression `begin_l[i] - fl[i]` by $O(n)$ evaluations. We can also save space in the representation by replacing the 2 length-n vectors with a single length-n vector.

```
irl [i]              =  begin_l[i] - fl[i]
pl [ irl[i] + j ]    =  a[i,j]
```

The value `irl[i]` can be thought of as the row `i` envelope's base address into `pl`. The "first nonzero" function `fl` is useful as a limit for the summation over all `j` in row `i`, but can be easily recovered from `irl`. The upper triangle's column-oriented envelopes are stored in a similar fashion.

6.2.2 The "first nonzero" functions

We will derive the first nonzero function `fl` for the row-oriented envelopes in the lower triangle. A similar function `fu` can be derived for the column-oriented upper triangle envelopes.

The last column stored for row `i` is `j = i-1`, so if `(pl_env_len i)` is the row `i` envelope size, the column index of row `i`'s first nonzero is

```
fl i  =  i - (pl_env_len i)
```

The indices `irl[i-1] + i-2` and `irl[i] + i-1` into `pl` point to the end of the row `i-1` and row `i` envelopes respectively. Since the envelopes are stored consecutively in `pl`, we have `pl_env_len i = (irl[i] + i-1) - (irl[i-1] + i-2)`, therefore,

```
fl i  =  i - 1 + irl[i-1] - irl[i]
```

Notice that `fl` is only defined for `i <- 2..n`, since there is no `irl[0]` entry. The row 1 envelope is always empty in the lower triangle. For an empty row the first nonzero is in column `(fl i) = i`; so the envelope contains columns `j <- (fl i)..(i-1) =` ϕ.

We could put a conditional in `fl` to make it defined for row 1, but this imposes a run-time test for every row. A better alternative is to define a bogus `irl[0]` that causes `(fl 1)` to return 1. Entry `irl[1]` always has the value `irl[1] = (begin 1) - (fl 1) = 1 - 1 = 0`, therefore `irl[0]` must satisfy:

```
1  =  fl 1  =  1 - 1 + irl[0] - irl[1]  =  irl[0]
```

However, we will discover later that we can always avoid any calls to `fl` for row 1, or to `fu` for column 1.

6.2.3 A functional derivation of L*U factorization

The problem is to solve the linear system `A*x = b`: given `A` and `b`, what is `x`? If `A` is invertible, there exists a unique factorization `A = L*U` where `L` is lower triangular and `U` is unit upper triangular, which reduces the original problem to the easier problem of solving the triangular linear systems `L*y = b, U*x = y`.

But this leaves the problem: given `A`, what are `L` and `U`? The usual derivation of `L` and `U` is presented as a sequence of steps `k <- 1..n`, each step forming an intermediate matrix `A(k)`; this particular sequential approach to Gaussian elimination is very obscure, hiding the essential data dependencies under non-essential operational details. Instead we will first write out the equation `A = L*U` as if we were finding `A` given `L` and `U`, then by algebraic manipulation, derive mutually recursive equations for L and U given A. We will see that the pre-Haskell program mimics closely the mathematical notation we use to derive the equations for L and U. Because of this close resemblance, the program is easy to reason about, the essential data dependencies are clear, and it is easy to justify and debug optimizations. Equally important, the inherent parallelism becomes immediately apparent, since the only sequentiality in the program is that inherent in the data dependencies.

Each `a[i,j]` is the inner product of `l`'s row `i` and `u`'s column `j`: $a[i,j] \;=\; \sum_{k=1}^{n} l[i,k] * u[k,j]$, $i \in 1..n$, $j \in 1..n$. But there is no contribution to `a[i,j]` for terms in which `l[i,k] = 0` (for columns `k` to the right of the diagonal: `i < k`) or in which `u[k,j] = 0` (for rows `k` below the diagonal: `j < k`). Therefore, instead of summing over $k \in 1..n$, we only need to sum over $k \in 1..(min\ i\ j)$.

Equivalently, we can separate the definitions for a[i,j] in the lower triangle and diagonal (i >= j, and therefore use j as the summation limit) or in the upper triangle (i < j, and therefore use i as the summation limit).

$$\begin{aligned} a[i,j] &= \textstyle\sum_{k=1}^{j} l[i,k] * u[k,j] \\ &= l[i,j] * u[j,j] + \textstyle\sum_{k=1}^{j-1} l[i,k] * u[k,j], \qquad i \in 1..n,\ j \in 1..i, \end{aligned}$$

$$\begin{aligned} a[i,j] &= \textstyle\sum_{k=1}^{i} l[i,k] * u[k,j] \\ &= l[i,i] * u[i,j] + \textstyle\sum_{k=1}^{i-1} l[i,k] * u[k,j], \qquad i \in 1..n,\ j \in i+1..n \end{aligned}$$

But we can immediately rearrage these equations to define the elements of L and U (recall that we require u[j,j] = 1.0 for all j):

$$l[i,j] = a[i,j] - \textstyle\sum_{k=1}^{j-1} l[i,k] * u[k,j], \qquad i \in 1..n,\ j \in 1..i,$$

$$u[i,j] = (\ a[i,j] - \textstyle\sum_{k=1}^{i-1} l[i,k] * u[k,j]\ )\ /\ l[i,i], \qquad i \in 1..n,\ j \in i+1..n$$

The L equations show us that whatever the operational sequencing, l[i,j] depends on a[i,j] and recursively depends on depends on other L elements in the same row i and to the left, and on U elements in the same column j and above. The recursion terminates upon the leftmost column of L and the topmost row of U. Similar reasoning holds for the U equation.

In the following we will replace l[i,i] with the name d[i]. There are optimizations we can perform on L's diagonal elements that cause them to deserve special treatment. The d[i]'s are *not* to be confused with the elements of diagonal matrix D in the L*D*U factorization, where both L and U are unit triangular.

6.2.4 L*U factorization in dense array format

From the equations above let us write a functional program to compute L and U. Let us define the functional form of the mathematical summation sign:

```
sum i j accum f  =  j<i  ->  accum,  sum (i+1) j (accum + (f i)) f
```

We save one addition by letting a[i,j] be the accumulator's initial value and subtracting each term:

```
subtract_sum i j accum f
    =  j<i  ->  accum,  subtract_sum (i+1) j (accum - (f i)) f
```

Then our definition of an element in L is:

```
l[i,j]  =  subtract_sum 1 (j-1) a[i,j] l_exp)
    where l_exp k  =  l[i,k]*u[k,j]
```

This definition is exactly what we would write using the mathematical summation sign; it holds for `i <- 2..n, j <- 1..(i-1)`. Row 1 is skipped since `l[1,1]` is on the diagonal and we wish to define the diagonal elements separately.

The definition for L's diagonal elements is a simplified version of the definition above (since `i = j`), defined for `i <- 1..n`. The definition for `U` is nearly the same as for `L` except that summation stops at `k = i-1`. Then the entire row is scaled by `1./d[i]` to normalize `U`'s diagonal to 1.0. `U`'s definition holds for `i <- 1..(n-1), j <- (i+1)..n`, or equivalently, for `j <- 2..n, i <- 1..(j-1)`. See the complete program at the end of this section.

Each element `d[i]` appears as a divisor in the definition for every `u[i,j]` in the same row `i` ($(n^2-n)/2$ divisions altogether), as well as in definition of `x[i]` in the same row for the `L*U*x = b` backsolve stage (n divisions). Since division is expensive compared with multiplication, we instead store the inverse of each `d[i]`, replacing $O(n^2)$ divisions with n divisions and $O(n^2)$ multiplications. This is a classic example of using an array to store expensive shared computations.

The definitions of `L` and `U` are mutually recursive, both in the mathematical definition and in the pre-Haskell array definition. We do not need to store L's upper or U's lower triangle, which are zero, or U's unit diagonal, so we can store all the essential results in a single n^2 array. We can recursively define the matrix `lu = (L-D) + (1/D) + (U-I)` in dense matrix format (where `D` = L's diagonal, `1/D = D`'s inverse):

```
plu a  =  lu
where
    ((1,n),(1,n)) = bounds a
    lu = { 2D_array (1,n),(1,n)
            | [i,j] = l i j    || i <- 2..n, j <- 1..(i-1)
            | [i,i] = d i      || i <- 1..n
            | [i,j] = u i j    || i <- 1..(n-1), j <- (i+1)..n }
    l i j = let l_exp k  =  lu[i,k] * lu[k,j]
            in  subtract_sum 1 (j-1) a[i,j] l_exp
    d i   = let d_exp k  =  lu[i,k] * lu[k,i]
                sum      =  subtract_sum 1 (i-1) a[i,i] d_exp
            in 1./sum
    u i j = let u_exp k  =  lu[i,k] * lu[k,j]
            in (subtract_sum 1 (i-1) a[i,j] u_exp ) * lu[i,i]
```

6.2.5 Strict vs. non-strict arrays

Some clarifying comments are in order concerning the semantics of pre-Haskell monolithic arrays.

Notice that the domain specifications in the the array comprehension correspond exactly to the domain specifications given in the mathematical function definitions. These domain specifications should *not* be thought of as looping constructs: they say nothing about the order in which elements of the array should be evaluated – that order is dictated solely by data dependencies. In fact, pre-Haskell monolithic arrays are *non-strict*, or *lazy* – each element in the array is in effect implemented as a "thunk" which is evaluated only when demanded.

Thus domain specifications such as `i <- 2..n, j <- 1..(i-1)` specify *where* thunks for a particular form of expression must be placed, but say nothing about the *order* in which the elements are evaluated. Evaluation of an element in a lazy array is forced only when it is explicitly requested. If a lazy array is recursively defined, evaluation of an element in turn forces the evaluation of other elements on which it has a data dependency.

Of course, once an element has been evaluated its value is stored directly in the array in place of the thunk. We can think of an array `a` as a function of `d` integer arguments (where `d` is the array's dimensionality), for which we know that any given function application `a i_1 ... i_d` (i.e., any given array element `a[i_1,...,i_d]`), will be requested many times. In this view an array is a **caching function**.

There are several essential differences between lazy arrays in pre-Haskell and arrays in a language like Fortran, which we will discuss briefly here. One difference is that pre-Haskell specifies the result array monolithically in terms of a definition for each element, whereas Fortran specifies the result array in terms of incremental updates to the input array. For the example program presented so far, pre-Haskell's monolithic definition requires that the output array be computed in a separate space from the input array.

For pre-Haskell to be able to reuse the input array `a` to store the output array `lu`, the compiler must know that reuse is safe. There must be no other outstanding references to `a` outside the definition of `lu`. Furthermore, an element `a[i,j]` must be dead at the time that it is replaced by element `lu[i,j]`, which means that either the compiler must determine or the programmer must specify a safe order of evaluation. This topic is an area of research, and will not be discussed further here.

Another difference is that the Fortran programmer must be careful to arrange the order of his computation so that whenever he evaluates an element `lu[i,j]`, the elements on which `lu[i,j]` has a direct data dependency will have already been computed. Both the pre-Haskell and the Fortran arrays can be viewed as cached functions, so although they may differ in the *order* in which array elements are evaluated, there is no difference in the *total amount* of computation time spent on array indexing and floating point arithmetic. On the other hand, pre-Haskell's thunks increase the time by a small constant factor. In addition to computing the element values, we must also create a thunk for each element when array storage is allocated; and whenever an element is demanded we must test whether or not its thunk as been forced yet.

Notice that we could eliminate the need for creating and testing element thunks if, like the Fortran programmer, we could guarantee a safe order of evaluation. Again, this is a research topic, and will not be treated further here.

6.2.6 Refinement of `L*U` factorization using "first nonzero" information

Notice that in the summations, the `k`-th term `l[i,k]*u[k,j]` makes no contribution if either factor is zero. More specifically, if a sparse matrix is organized such that most nonzeros are close to the main diagonal, then there is no contribution if either `l[i,k]` falls to the left of row `i`'s first nonzero (i.e., if `k < (fl i)`) or if `u[k,j]` falls above column `j`'s first nonzero (if `k < (fu j)`).

Assume we are given the "first nonzero" functions `fl` and `fu`. `L` is then defined by:

```
        l_exp k       = new_pl[ irli + k ] * new_pu[ k + iruj ]
        accum_init    = old_pl[ irli + j ]
        sum           = subtract_sum kmin_l (j-1) accum_init l_exp
    new_pl  = { vector (bounds old_pl)
                | [ irli + j ]    = l i j irli fli
                    || i <- 2..n,
                           irli <- [ irl[i] ],
                           fli  <- [ (fl i) ] ,
                           j <- fli+1..i-1
                | [ irli + j ] = old_pl [ irli + j ]
                    || i <- 2..n,
                           irli <- [ irl[i] ],
                           fli  <- [ (fl i) ],
                           j <- fli, fli < i  }
```

If we suppose that extra information has been supplied to the compiler directing it to treat i as an outer loop index and j as an inner loop index, we can see that the common subexpression lifting we have performed corresponds exactly to the Fortran idea of lifting loop-invariant computations. There exist a few more opportunities for lifting common subexpressions in this program fragment, but we have taken all opportunities that lift loop-invariant subexpressions.

6.2.8 Reorganizing the upper triangle from column- to row-oriented envelopes

For the L*U = A factorization phase, it is desirable to store the lower triangle as row-oriented envelopes and the upper triangle as column-oriented envelopes. But for the L*U*x = b backsolve phase, there are some advantages to reorganizing U into row-oriented envelopes. These advantanges will be discussed in the next section; here we describe how to achieve the reorganization.

If we store the upper triangle in row-oriented envelopes, then the row i envelope contains columns j <- (i+1)..(row_fu i) where row_fu returns the column index of row i's last nonzero. We want a real storage vector row_pu and a base address vector row_iru such that row_pu [row_iru[i] + j] = a[i,j]. Let us suppose that row_iru was already computed by the same program that prepared iru and irl. We can derive the function row_fu that given row index i returns the column index of i's *last* nonzero: row_fu i = i + 1 + row_iru[i+1] - row_iru[i]. This function is defined for rows i <- 1..(n-1). Notice also that size of row_pu is contained in row_iru[n]. So now we can reorganize pu into row_pu:

```
    row_pu   =  { vector (1,row_iru[n])
                  | [ row_iru[i] + j ] = fetch_col_pu i j
                        || i <- 1..(n-1), j <- (i+1)..(row_fu i) }
```

Why do we not simply replace (fetch_col_pu i j) with pu [i + iru[j]]? Because although both the column- and row-oriented representations store all the upper triangle's nonzeros (both before and after fill-in), the row form may store some zeros not stored in the column form, and vice versa. Our definition of row_pu demands every [i,j] pair stored in the row form, but fetch_col_pu must decide whether that [i,j] falls within the column form, and return a zero if not:

```
    fetch_col_pu i j  =  i<(fu j)  ->  0.0,  col_pu [i + col_iru[j]]
```

This function is defined only over the upper triangle `j <- 2..n, i <- 1..(j-1)`. The complete definition:

```
    reorg_pu n col_pu col_iru row_iru  =  row_pu  where
        col_fu j      = j - 1 + col_iru [j-1] - col_iru [j]
        row_fu i      = i + 1 + row_iru [i+1] - row_iru [i]
        fetch_col_pu i j    =  i <= (fu j)  ->  0.0,  col_pu [i + col_iru [j]]
        row_pu  =  { vector (1,n)
                       | [ row_iru [i] + j ]  =  fetch_col_pu i j
                              || i <- 1..(n-1), j <- (i+1)..(row_fu i) }
```

The reorganization takes time proportional to the size of the upper triangle's row-oriented envelope. For matrices in which the maximum size of any row envelope is independent of matrix size n, this time is $O(n)$.

6.2.9 The L*U*x = b solution phase

We will discuss two versions of the backsolve phase, one version using the column-oriented U, the other using the reorganized row-oriented U.

A*x = L*U*x = b is equivalent to solving the lower triangular system L*y = b , using the intermediate solution y as the righthand side for solving the upper triangular system U*x = y.

```
    l[1,1]*y[1]                                          = b[1]
    l[2,1]*y[1]  + l[2,2]*y[2]                           = b[2]
    . . .                                                . . .
    l[n,1]*y[1]  +        . . .        + l[n,n]*y[n]     = b[n]
```

Recall that we are storing the inverse of diagonal elements under the name d[i] = 1./l[i,i] for every i. For a typical row i, this system of equations can be recast as the function:

```
    y[i]   =  sum * d[i]
    where
        y_exp j  =  l[i,j]*y[j]
        sum      =  subtract_sum 1 (i-1) b[i] y_exp
```

We can interpret this as saying that the lefthand side unknown y[i] is contained in the righthand side known b[i], but to find y[i] we must subtract out the contribution of the other elements y[1], ..., y[i-1]. For j <- 1..(i-1) the coefficient u[i,j] gives the weight of y[j]'s contribution to b[i].

Finally we solve the upper triangular system U*x = y, recalling again that U is unit diagonal. The entire function, assuming the matrix lu = (L-D) + (1/D) + (U-I) is in dense matrix format, and doing the appropriate substitutions for l[i,j], d[i], and u[i,j]:

```
plub lu b  =  x_vec  where
    y i  =  sum * lu[i,i]
    where
        y_exp j  =  lu[i,j]*y_vec[j]
        sum      =  subtract_sum 1 (i-1) b[i] y_exp
    x i  =  subtract_sum (i+1) n y_vec[i] x_exp
        where x_exp j  =  lu[i,j]*x_vec[j]
    y_vec  =  { vector (1,n)   | [i] = y i    || i <- 1..n }
    x_vec  =  { vector (1,n)   | [i] = x i    || i <- 1..n }
```

Now let us stay with the dense format, but incorporate the "first nonzero" functions fl and fu to avoid multiplications and subtractions for terms in which the contribution weight l[i,j] or u[i,j] is outside the matrix envelope and therefore zero. The change is trivial for the lower triangular system, since the summation is along a row. For i <- 2..n, (fl i) tells us the first nonzero column in row i:

```
y i  =  sum * lu[i,i]
where
    y_exp j  =  lu[i,j]*y_vec[j]
    sum      =  subtract_sum (fl i) (i-1) b[i] y_exp
```

Otherwise y 1 = b[1].

But for the upper triangle we have the problem that the summation is also running along a row, but fl tells us the first nonzero in a given column. We cannot use it to give a bound on the summation for a row i the way we did for the lower triangular system. We could instead use fl as a predicate for each element of a row to see whether that element falls outside the upper triangle's column-oriented envelope. Unfortunately, although we avoid an expensive floating point multiply and subtract for each zero u[i,j], we still incur a predicate test for every element. The upper triangular envelope may be of size $O(n)$, but testing *every* element for inclusion in the envelope forces us to perform $O(n^2)$ work.

One solution is to imitate the Fortran solution, which walks through U column by column, performing successive updates on the x vector as each x[i] becomes available. It is possible to write functional programs in which a compiler can easily detect the opportunity for array updates, but we will not pursue that avenue any further here.

The solution we will pursue instead is to switch at this stage from a column-oriented view of U, which was convenient for the factorization phase, to a row-oriented view more appropriate to the backsolve phase. Let us stay for the moment with a dense format matrix, but let us recall the earlier section in which we discussed a row-oriented U: suppose we have available a function (row_fu i) that returns the column number j of the *last* nonzero in upper triangle row i, so we can limit how far the summation runs along row i:

```
x i  =  subtract_sum (i+1) (row_fu i) y_vec[i] x_exp
where  x_exp j  =  lu[i,j]*x_vec[j]
```

for i <- 1..(n-1), since row_fu is defined over that domain, otherwise x n = y_vec[n].

It is now a trivial matter to convert our program to use an envelope format version of lu. Assume the user has supplied both the column- and row-oriented base address vectors col_iru and row_iru for the upper triangle, and we have the column-oriented reals vector col_pu computed by plu; then call reorg_pu defined in the previous section:

```
row_pu  =  reorg_pu n col_pu col_iru row_iru
```

Recall this reorganization took time proportional to the size of U's envelope. Now lu is represented as the tuple [n, pl, d, row_pu, irl, row_iru]:

```
plub [n, pl, d, row_pu, irl, row_iru] b = x_vec  where
    fl i      =  i - 1 + irl [i-1] - irl [i]
    row_fu i  =  i + 1 + row_iru[i+1] - row_iru[i]
    y i  =  sum * d[i]  where
            irli     = irl[i]
            y_exp j  = pl [irli + j] * y_vec[j]
            sum      = subtract_sum (fl i) (i-1) b[i] y_exp
    x i  =  sum  where
            row_iru_i   = row_iru[i]
            x_exp j = row_pu [row_iru_i + j] * x_vec[j]
            sum      = subtract_sum (i+1) (row_fu i) y_vec[i] x_exp
    y_vec    = { vector (1,n) | [i] = y i      || i <- 2..n
                              | [i] = b[i]     || i <- [1] }
    x_vec    = { vector (1,n) | [i] = x i      || i <- 1..(n-1)
                              | [i] = y_vec[n]   || i <- [n] }
```

The linear system A*x = b may have been poorly organized for the envelope representation, and the equivalent system P*A*(1/P)*P*x = P*b may require a smaller envelope to store A and its factorization. The permutation matrix P reorganizes the rows (1/P the columns) using reverse Cuthill-McKee numbering to heuristically minimize the envelope size. Sherman's version of plub assumes that the L*U envelope format matrix is in RCM order, whereas b and the result x are in the orginal problem order. If we have a vector iord representing the permutation P mapping the original number i to RCM number iord[i], then the change to plub is trivial, and is left as an exercise.

7 Acknowledgements

We wish to thank Joe Fasel at Los Alamos for comments on various parts of this document, as well as for providing us with his solution to the doctors' office problem. Also thanks to Los Alamos and Lawrence Livermore National Laboratories for their sponsorship of the Salishan High-Speed Computing Conference.

Appendix: Andy Sherman's Fortran Code for the Skyline Problem

```
      subroutine plu(n,pl,d,pu,irl,iru)
      dimension pl(1),d(1),pu(1),irl(1),iru(1)
c
c  this subroutine performs an envelope lu decomposition on the
c  matrix c which is stored in pl, d, and pu in envelope form
c  (see subroutine genenv). the rows (columns) of the lower
c  (upper) triangle of c from the first nonzero up t
c  but not including the diagonal are stored sequentially
c  in pl (pu).  the diagonal entries of a are stored in d.
c  irl(i) (iru(i)) points to the nonexistent c(i,0) (c(0,i))
c  element of the i-th row (column).  on return, the strict lower
c  (upper) triangle of l (u) is stored in pl (pu), and the
c  inverses of the diagonal elements of l are stored in d.
c  (u is unit upper triangular.)
c
      d(1) = 1. / d(1)
      do 100 i=2,n
        irli = irl(i)
        irui = iru(i)
c
c  ifli (ifui) is the lowest off-diagonal index in the
c  i-th row (column). similar computations are used for other
c  rows and columns below. the first off-diagonal element in the
c
c  i-th row (column) never requires an inner product.
c
        ifli = i - 1 + irl(i-1) - irli
        jminl = ifli + 1
        ifui = i - 1 + iru(i-1) - irui
        jminu = ifui + 1
        jmax = i - 1
c
c  compute l(i,j) for j in i-th row
c
        if (jminl .ge. i) go to 30
        do 20 j=jminl,jmax
          iruj = iru(j)
          ifuj = j - 1 + iru(j-1) - iruj
          kmin = max0(ifli,ifuj)
          if (kmin .ge. j) go to 20
c
c  plij = -pl(ij) to force good code in loop
c
```

```
          plij = - pl(irli+j)
          kmax = j - 1
c
c  compute inner product for l(i,j)
c
          do 10 k=kmin,kmax
            plij = plij + pl(irli+k)*pu(iruj+k)
  10        continue
          pl(irli+j) = -plij
  20      continue
c
c  compute u(j,i) for j in i-th column
c
  30    if (jminu .gt. i) go to 70
c
c  compute first off-diagonal element of column
c
        pu(irui+jminu-1) = pu(irui+jminu-1) * d(jminu-1)
        if (jminu .eq. i) go to 70
        do 60 j=jminu,jmax
          irlj = irl(j)
          iflj = j - 1 + irl(j-1) - irlj
          kmin = max0(ifui,iflj)
c
c  puji = -pu(ji) to force good code in loop
c
          puji = -pu(irui+j)
          if (kmin .ge. j) go to 50
          kmax = j - 1
c
c  compute inner product for u(j,i)
c
          do 40 k=kmin,kmax
            puji = puji + pl(irlj+k)*pu(irui+k)
  40        continue
  50      pu(irui+j) = -puji * d(j)
  60      continue
c
c  compute l(i,i)
c
  70    jmin = max0(ifli,ifui)
        di = -d(i)
        if (jmin .gt. jmax) go to 90
        do 80 j=jmin,jmax
          di = di + pl(irli+j)*pu(irui+j)
```

```
   80     continue
c
c  store 1/l(i,i) in d(i)
c
   90   d(i) = -1./di
  100   continue
c
      return
      end

c ------------------------------------------------------------------

cdate of version:  04/05/79
      subroutine plub(n,pl,d,pu,irl,iru,x,b,iord)
      dimension pl(1),d(1),pu(1),irl(1),iru(1)
      dimension x(1),b(1),iord(1)
c
c  this subroutine performs the backsolves for the solution of
c  l u p x = p b. l and u are stored in pl, d, and pu as
c  described in subroutine plu.
c
c  solve l x = p b
c
      iordj = iord(1)
      x(1) = b(iordj) * d(1)
      do 30 j=2,n
        iordj = iord(j)
        xj = - b(iordj)
        irlj = irl(j)
c
c  kmin is the lowest off-diagonal column index in the j-th row
c  of pl. similar computations are used for other rows and
c  columns below.
c
        kmin = j - 1 + irl(j-1) - irlj
        if (kmin .ge. j) go to 20
        kmax = j - 1
        do 10 k=kmin,kmax
          xj = xj + pl(irlj+k) * x(k)
   10     continue
   20   x(j) = - xj * d(j)
   30   continue
c
c  solve ux = x
c
```

```
      imax = n - 1
      do 50 i=1,imax
        j = n + 1 - i
        iruj = iru(j)
        kmin = j - 1 + iru(j-1) - iruj
        if (kmin .ge. j) go to 50
        kmax = j - 1
        xj = - x(j)
        do 40 k=kmin,kmax
          x(k) = x(k) + xj * pu(iruj+k)
   40     continue
   50   continue
c
c  reorder x to solve p x = x
c
      do 70 k=1,n
c
c  iord(k) .lt. 0 means that x(k) is proper element already.
c  otherwise, interchange x(i) and x(iord(k)). the effect
c  of this is to rotate every cycle of the permutation one
c  position so that it is properly oriented.
c
        if (iord(k) .lt. 0) go to 70
        i = k
   60     iordi = iord(i)
          t = x(iordi)
          x(iordi) = x(k)
          x(k) = t
          iord(i) = - iord(i)
          i = iordi
          if (i .ne. k) go to 60
   70   continue
c
c  at this point all entries of iord are negative
c
      do 80 i=1,n
        iord(i) = - iord(i)
   80   continue
c
      return
      end
```