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

Paul Hudak
Steve Anderson

Yale University*
Department of Computer Science
New Haven, CT

May 1989

1 Introduction

Haskell is a new functional language, named after the logician Haskell B. Curry, that was designed by a 14-member international committee representative of the functional programming research community [HWe88].[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 common language. The committee's goals were that Haskell should:

1. Be suitable for teaching, research, and applications, including building large systems.
2. Be completely described via the publication of a formal syntax and semantics.
3. Be freely available, such that anyone is permitted to implement the language and distribute it to whomever they please.
4. Be based on ideas that enjoy a wide consensus.
5. Be useable as a basis for further programming language research.

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

[1]The committee members are Arvind (MIT), Brian Boutel (Victoria University of Wellington), Jon Fairbairn (Cambridge University), Joseph Fasel (Los Alamos National Laboratory), Paul Hudak (Yale University), John Hughes (Glasgow University), Thomas Johnsson (Chalmers Institute of Technology), Dick Kieburtz (Oregon Graduate Center), Rishuyar Nikhil (MIT), Simon Peyton-Jones (University College London), Mike Reeve (Imperial College), Philip Wadler (Glasgow University), David Wise (Indiana University), and Jonathan Young (Yale and MIT).

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, and a rich set of primitive data types, including arbitrary and fixed precision integers, and complex, rational, and floating-point numbers. In addition it has several novel features that give it additional expressivness, including an elegant form of overloading using a notion of *type classes*, a flexible I/O system that unifies the two most popular functional I/O models, and an array datatype that allows purely functional, monolithic arrays to be constructed using "array comprehensions."

The reader will note that we did not describe Haskell as a *parallel* programming language; indeed it is not. However, much research in recent years has centered on the implementation of functional languages on parallel machines, including the building of special-purpose hardware such as dataflow and reduction machines. We will say little about these issues here, other than noting how the solutions to the problems presented have considerable inherent parallelism.

Given our space constraints, it is impossible for us to describe Haskell in its entirety; our goal is only to give the reader some familiarity with the language by giving solutions to the four language session problems presented at the 1988 Salishan Conference on High-Speed Computing. The reader is referred to the Haskell Report [HWe88] for a complete definition of the language. Of course, studying the solutions presented here will give the reader an idea of what programming in any of a number of modern functional languages is like; indeed, all of the solutions given have been run on our implementation of Alfl, a functional language designed and implemented at Yale.

2 Brief Overview of Haskell

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

Haskell is an "equational" language similar to Miranda[2], 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
'       (y:ys) = if x==y then True
                         else member x ys
```

[2]Miranda is a trademark of Research Software Ltd.

The "tick mark" on the second line is a convenient abbreviation for the initial subsequence on the preceding line (so that the arity of the two equations is the same).

A function `f x = x+1` may also be defined "anonymously" with the expression `\x -> x+1`, and thus `(\x -> x+1) 2` returns `3`.

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

```
[ (x,y) | x<-xs, y<-ys ]
```

which constructs the list of all pairs whose first element is from `xs`, and second is from `ys`. "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. Aside from `member` defined earlier, the ones we need in this paper are the following:

```
takewhile pred []   = []       -- takes elements of list while pred is true
'            (a:as) = if (pred a) then (a : takewhile pred as)
                                  else []

foldl f a []   = a             -- folds list from left
'       (x:xs) = foldl f (f a x) xs

foldr f a []   = a             -- folds list from right
'       (x:xs) = f x (foldr f a xs)

zip   []     bs   = []         -- forms list of pairs from pair of lists
'     as     []   = []
'   (a:as) (b:bs) = (a,b) : zip as bs

nodups []   = []               -- removes duplicates from list
'    (x:xs) = x : nodups [ y | y <- xs, y /= x ]
```

In Haskell, function application always has higher precedence than any infix operator, and thus "`a : takewhile pred as`" is parsed as "`a : (takewhile pred as)`." Note in `zip` the use of *tuples*, which in Haskell are constructed in arbitrary but finite length by writing "`(a,b, ..., c)`" (the parentheses are mandatory); tuples may be pattern-matched like lists. Finally, 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) ... ))
```

Haskell also has *arrays* and a special syntax for manipulating them. A two-dimensional array `a` is indexed at position `(i,j)` via the expression `a!(i,j)`. New arrays are constructed using the primitive function `array`, which takes a set of bounds and a list comprehension as arguments; the list comprehension specifies the set of index/value pairs for the new array. For example:

```
array ((1,1),(n,n))
  [ ((i,j) , k*a!(i,j)) | i<-[1..n], j<-[1..n] ]
```

returns a nXn matrix representing the matrix `a` multiplied by the scalar `k`.

This description of 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 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 problem is as an inductive definition:

- 1 is in the output sequence.

Figure 1: Naive Hamming Solution

Figure 2: Hamming Solution Without Duplicates

- 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 1, 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 2 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`:

```
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) = if a<b then a : merge as (b:bs)
                             else b : merge (a:as) bs
'      [] bs = bs
'      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 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 rewrite Turner's KRC solution in Haskell (we also made a few simplifications), and refer to the paper referenced above for a detailed description of it:

```
main = foldr (++) [] (map paraffin [1..])

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

para = ["H"] : 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]]
'          ("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 fs xs = xs ++ closure' fs xs xs
closure' fs xs ys = closure'' fs xs (nodups [a | f <- fs, a <- map f ys,
                                                 not (member a xs) ])
closure'' fs xs [] = []
'               ys = ys ++ closure' fs (xs ++ ys) ys

quotient f []   = []
```

```
'          (a:x) = a : [ b | b <- quotient f x, not (f a b) ]
```

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 somewhat with the statement made in the *last* sentence of the problem description. Thus we also provide a solution that uses explicit non-determinism. Unfortunately, non-determinsm is not part of the Haskell standard, and thus we assume a primitive operator called `choose` which non-determistically chooses an element from a list.

The two solutions are radically different, and reflect very different characteristics of 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
      '   (p',t') rest@((p,t):xs) | t'<t = (p',t') : rest
      '                                  = (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) <- zip 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) <- zip3 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 world = choose_loop world processes

processes = [sickening_process, curing_process, receptionist]

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

curing_process w@(h,s,[],c,d,r) = w                 -- nobody being cured
'              w@(h,s,b, c,d,r) = (h,s,dps,dp:c,d,r)
                                  where (dp,dps) = cure_one b

receptionist w = choose [help_the_sick,move_the_cured] w
  where help_the_sick w@(h,[],  b,c,d,   r) = w     -- nobody's sick
        '             w@(h,s,   b,c,[],  r) = w     -- no free doctors
        '             w@(h,p:ss,b,c,d:ds,r) = (h,ss,(d,p):b,c,ds,r)

        move_the_cured w@(h,s,b,[],       ds,r) = w -- no recent curing
        '              w@(h,s,b,(d,p):dps,ds,r) =
```

```
                                              (p:h,s,b,dps,ds++[d],(d,p):r)

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

choose_and_remove lst = (el, [y | y<-lst, y\=el])
                        where el = choose lst

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 also listed in the Appendix of [HA88].

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 Haskell using incremental updates to purely functional arrays (see [Hud86] for a discussion of incremental 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.

The incremental functional array solution is presented in [HA88], and illustrates 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 incremental program, when run on our Alpha-Tau implementation of Alfl [Hud84], achieves the same space complexity as 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. 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. This style of solution is presented below.

6.1 Monolithic Array Solution

The skyline problem illustrates well some of the special strengths of 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 dependences 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. 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.

The incremental solution was essentially a transcribed version of Sherman's code, and thus we included no description of the data representations or the algorithm. For the monolithic solution we will instead start from the very basics, and develop the final program via step-wise refinement of the specification.

6.1.1 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 5-tuple `(n, pl, d, pu, irl, iru)` where:

- `n` = the order of `A`.
- `pl, d, pu` = 1-D floating point vectors representing 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`.

The base address vectors require some explanation. For access into lower triangle `pl`, suppose we defined the vectors:

- `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 desirable to make this summation the innermost loop to preserve locality of reference and therefore achieve good cache and virtual memory hit rates. 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 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.

6.1.2 The "first nonzero" functions.

We will show how the first nonzero function `fl` for the row-oriented envelopes in the lower triangle can be recovered from the vector `irl`. 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`. Let `(pl_env_len i)` = the row `i` envelope size. Then the column index of row `i`'s first nonzero is

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

The index values `(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.
```

Since there is no `irl!0` entry, `fl` is only defined for `i <- [2..n]`, 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_l 1) - (fl 1) = 1 - 1 = 0`, therefore `irl!0` must satisfy the equation:

```
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.

6.1.3 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 $k \epsilon [1..n]$, each step forming an intermediate matrix $A(k)$; this particular sequential approach to Gaussian elimination is very obscure, hiding the essential data dependences 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 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 dependences are clear, and it is easy to justify and debug optimizations.

Each $a(i,j)$ is the inner product of l's row i and u's column j: $a(i,j) = \sum_{k=1}^{n} l(i,k) * u(k,j),\quad i \epsilon [1..n],\ j \epsilon [1..n]$. 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 $k \epsilon [1..n]$, we only need to sum over $k \epsilon [1..(min\ i\ j)]$.

Equivalently, we can separate the definitions for $a(i,j)$ in the lower triangle and diagonal ($i \leq j$, and therefore use j as the summation limit):

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

or in the upper triangle ($i < j$, and therefore use i as the summation limit).

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

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

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

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

The L equations show us that whatever the operational sequencing, $l(i,j)$ depends on $a(i,j)$ and recursively depends on 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 make them 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.

6.1.4 $L*U$ factorization in dense array format.

For clarity, we introduce some new syntax into list comprehensions. In an `array` function's list comprehension argument, the `(i,x)` pair can be written in the form `i = x`, similar to the declarations in a Haskell `where`. For example,

```
array ((1,1), (N,N))
  [ (i,j) = k*a!(i,j) | i <- [1..N], j <- [1..N] ]
```

From the equations in the previous section let us write a functional program to compute L and U. Let us define a higher-order function for the mathematical summation sign:

```
sum i j accum f  =  if j<i then accum else sum (i+1) j (accum + (f i)) f
```

Let us define a function `l` that given index `(i,j)`, computes the element value in L:

```
l (i,j)  =  a!(i,j) - (sum 1 (j-1) 0. l_exp)
    where l_exp k  =  L!(i,k) * U!(k,j)
```

This definition of `l` is very similar to a DO loop in Fortran, and can be compiled as efficiently.

Instead we will use an alternative definition of `sum` that operates over a list.

```
sum xs  =  sum1 0. xs
where
    sum1 accum []      =  accum
    sum1 accum (x:xs)  =  sum1 (accum+x) xs
```

Used with a list comprehension argument, this version gives a somewhat more legible way of writing `l i j`, and more closely resembles the mathematical summation sign.

```
l (i,j)  =  a!(i,j) - sum [ L!(i,k) * U!(k,j) | k <- [1..j-1] ]
```

We can think of the list as a multiset, and of `sum` as summing the elements of the set (although strictly speaking, floating point addition is not associative). Techniques such as Wadler's listlessness and deforestation transformations can ensure that such an expression gets converted to a semantically equivalent expression in which the lists are eliminated; the expression can be compiled as efficiently as a DO loop ([Wad84, Wad85]).

This definition of `l` 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 `d`, which computes L's diagonal elements, is a simplified version of the definition for `l` above (since `i = j`), 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 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` $= (L-D) + D^{-1} + (U-I)$ in dense matrix format (where $D = L$'s diagonal, $D^{-1} = D$'s inverse):

```
plu a  =  lu
where
    ((1,1),(n,n)) = bounds a
    lu = array ((1,1),(n,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  =  a!(i,j) - sum [ lu!(i,k) * lu!(k,j) | k <- [1..j-1] ]
    d i    =  1./s
        where s = a!(i,i) - sum [ lu!(i,k) * lu!(k,i) | k <- [1..i-1] ]
    u i j  =  lu!(i,i) * ( a!(i,j)
                - sum [ lu!(i,k) * lu!(k,j) | k <- [1..i-1] ] )
```

6.1.5 Lazy arrays and strict 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 `lu` should be evaluated.

In fact, we could let `lu` be a *lazy array*. Each element in a lazy array is represented by a thunk which is evaluated only when demanded. 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 thunks 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 dependence.

But if an element has already been forced once, the thunk modifies itself so that its value is returned immediately, without recomputation. We can think of an array `a` as a function of `d` integer arguments (where `d` is the the array's dimension), 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 Haskell and arrays in a language like Fortran. 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, Haskell's monolithic definition requires that the output array be computed in a separate space from the input array.

For 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 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 research topic is the subject of [AH89] and [And89].

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 dependence will have already been computed. Both the 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. But 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. See [AH89, And89].

6.1.6 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. If a sparse matrix is organized such that most nonzeros are close to the main diagonal, considerable work can be saved by ignoring terms in which either `l!(i,k)` falls to the left of row `i`'s first nonzero or if `u!(k,j)` falls above column `j`'s first nonzero. In other words, skip terms for which `k < (fl i)` or `k < (fu j)`).

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

```
l (i,j)  =  a!(i,j) - sum [ lu!(i,k) * lu!(k,j) } | k <- [kmin_l..j-1] ]
where kmin_l   =  max (fl i) (fu j)
```

We have defined `l (i,j)` for `i <- [2..n], j <- [1..(i-1)]`. But recall the domains of the first nonzero functions: `(fl i)` is defined over `i <- [2..n]`, which causes no problem, but `(fu j)` is defined only over `j <- [2..n]`, which makes `(l (i,j))` undefined in column 1.

We can put a run-time test in `(fu j)` for the case column `j = 1`, but then this test gets executed for every one of the $O(n^2)$ lower triangle elements. But we notice that whenever `j` is at or to the left of the row's first nonzero (i.e., when `j <= (fl i)`), the summation must terminate immediately. Then we are left with `(l (i,j)) = a!(i,j)`, which is zero for `j < (fl i)`, nonzero for `j = (fl i)`.

Therefore, when we know `j <= (fl i)`, we can return `a!(i,j)` immediately, avoiding calls to the summation altogether. This case includes column `j = 1`, so we do not need special treatment for this column.

We partition the lower triangle into different regions which use separately tailored element definitions.

```
lu = array ((1,1),(n,n))
        [ (i,j) = l i j  | i <- [2..n], j <- [(fl i)+1..i-1] ]  ++
        [ (i,j) = a[i,j] | i <- [2..n],
                           j <- [ (fl i) ], j < i ]  ++
        [ (i,j) = 0.0    | i <- [2..n], j <- [1..(fl i)-1] ]
        . . .
```

Similar partitionings hold for the diagonal and upper triangle. These clauses partition the lower triangle into three regions:

1. `j <- [(fl i)+1..i-1]` : inside the envelope except for the envelope's first column (the empty set if first nonzero is on or immediately beside the diagonal),

2. `j <- [ (fl i) ], j < i` : the envelope's first column (the empty set if first nonzero is on the diagonal).

3. `j <- [1..(fl i)-1]` : outside the envelope (the empty set if first nonzero is in column 1). These zero elements of `a` are guaranteed not to fill in for `lu`.

6.1.7 $L * U$ factorization using envelope representation.

Finally let us convert our definitions from dense format to envelope format. Let input matrix `a` be represented by the tuple `(n, old_pl, old_d, old_pu, irl, iru)` and result matrix `lu` be represented by `(n, new_pl, new_d, new_pu, irl, iru)`.

The auxiliary vectors `iru` and `irl` are the same for both input matrix `a` and output matrix `lu`. Zero elements in `a` that become non-zero in `lu` are called *fill-in*. Fill-in can occur only inside the envelope; all zeroes outside `a`'s envelope are guaranteed to remain zero in `lu` and are therefore also outside `lu`'s envelope; therefore `a` and `lu` have the same envelope structure, as represented by vectors `iru` and `irl`.

The vector bounds for `new_d` are simply `(1,n)`. We can get the new vector bounds for `new_pl` (and similarly for `new_pu` by fetching the old vector bounds: `(bounds pl)`. Alternatively, we can observe that the index of `pl`'s last element is `irl!n + n - 1`.

Example substitutions for references to the input matrix `a` are:

When `i <- [2..n], j <- [(fl i)..(i-1)]`,
`a!(i,j)` becomes `old_pl!(irl!i + j)`.

When `i <- [1..n]`,
`a!(i,i)` becomes `old_d!(i)`.

When `j <- [2..n], i <- [(fu j)..(j-1)]`,
`a!(i,j)` becomes `old_pu!(i + iru!j)`.

Similar substitutions can be made for references to `l`, `d`, and `u` regions of `lu!(i,j)`. The definition of the lower triangle then becomes:

```
l (i,j)   = s
    where
        kmin_l      =  max (fl i) (fu j)
        l_exp k     =  new_pl!(irl!i + k) * new_pu!(k + iru!j)
        accum_init  =  old_pl!(irl!i + j)
        s           = a_init - sum [ l_exp k | k <- [kmin_l..j-1] ]
```

```
new_pl  = array (bounds old_pl)
            [ irl!i+j  =  l (i,j)
                    | i <- [2..n], j <- [(fl i)+1..i-1] ]  ++
            [ irl!i + j  =  old_pl!(irl!i + j)
                    | i <- [2..n], j <- [(fl i)], j < i ]
```

There are many opportunities for common subexpression elimination. For a given element `(l (i,j))`, every term `k` in the summation uses the same base address values `irl!i` and `iru!j`, so we could save 2 vector lookups per term.

Also notice that for a given row `i`, the expressions `irl!i` and `(fl i)` appear both in the definition of `(l (i,j))` and in the array comprehension for `new_pl`. By abstracting these two expressions out of the definition of `(l (i,j))` and computing them at the level of the array comprehension, we not only share between these two parts of the program, we also ensure that these expressions are computed only once for a given row, instead of getting recomputed for each element in the row.

Here is our final complete version of the $L * U$ factorization, giving the code for the lower triangle. The code for the main diagonal and the upper triangle are similar.

```
plu (n, old_pl, old_d, old_pu, irl, iru)  =
    (n, new_pl, new_d, new_pu, irl, iru)
where
    l (i,j) irli fli = s
    where
        kmin_l  = max fli (fu j)
        iruj    = iru!j
        l_exp k = new_pl!(irli + k) * new_pu!(k + iruj)
        a_init  = old_pl!(irli + j)
        s       = a_init - sum [ l_exp k | k <- [kmin_l..j-1] ]
    new_pl  = array (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 ] ]
    . . .
```

An extension of the list comprehension that treats the nesting of generators and loop-invariant subexpressions more clearly and elegantly is discussed in [AH89, And89].

If the compiler treats `i` as an outer loop index and `j` as an inner loop index, we have achieved the equivalent of lifting loop-invariant computations in Fortran. There remain a few more opportunities for lifting common subexpressions in this program fragment, but we have taken all opportunities that lift loop-invariant subexpressions.

6.1.8 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_vec  =  array (1,n) [ i = y i  | i <- [1..n] ]
y i  =  s * d!i
where s  =  b!i - sum [ l!(i,j) * y_vec!j | j <- [1..i-1] ]
```

Finally we solve the upper triangular system $U * x = y$, recalling again that U is unit diagonal. The entire function, assuming the matrix `lu` $= (L-D)+D^{-1}+(U-I)$ 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    =  s * lu!(i,i)
        where s  =  b!i - sum [ lu!(i,j) * y_vec!j | j <- [1..i-1] ]
    x i    =  y_vec!i - sum [ lu!(i,j) * x_vec!j | j <- [i+1..n] ]
    y_vec  =  array (1,n) [ i = y i  | i <- [1..n] ]
    x_vec  =  array (1,n) [ i = x i  | i <- [1..n] ]
```

Now let us stay with dense format, but incorporate the "first nonzero" functions `fl` and `fu` to avoid multiplies and subtracts 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  =  s * lu!(i,i)
where s  =  b!i - sum [ lu!(i,j) * y_vec!j | j <- [(fl i)..i-1]
```

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.

```
x i  =  y_vec!i - sum [ lu!(i,j) * x_vec!j
                      | j <- [i+1..n], (fu j) <= i ]
x_vec  =  array (1,n) [ i = x i  | i <- [1..n] ]
```

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 approach at this stage is to switch to column-oriented view of U, which was convenient for the factorization phase, to a row-oriented view more appropriate to the backsolve phase. [HA88] gives a program that performs this column-oriented to row-oriented reorganization of U's envelope representation. 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 $O(n)$.

Another solution is to imitate the Fortran solution, which walks through U column by column, performing successive updates on the `x` vector as each `x!j` becomes available. Keeping for now the dense representation of `lu`, we can transform the definition of `xvec` given above to this form:

```
x_vec  =  j_loop n y_vec
j_loop j x_vec  =
    if j < 1
        then x_vec
        else j_loop (j-1)
                (array (1,n)
                    [ i = x_vec!i | i <- [1..fuj-1] ] ++
                    [ i = x_vec!i - lu(i,j) * x_vec!j
                                  | i <- [fuj..j-1] ] ++
                    [ i = x_vec!i | i <- [j..n] ])
                where fuj = fu j
```

For each iteration of `j` we define a new monolithic array, but we can easily transform this version to loop over `i` using an element-at-a-time incremental update function: `(upd x_vec i new_value)`. Or we can define a function `bigupdate` that has the same semantics as the `array` expression above, but it is only necessary to specify the elements that are *different* from `x_vec`.

```
. . .
else j_loop (j-1)
            (bigupdate x_vec
               [ i = x_vec!i - lu(i,j) * x_vec!j | i <- [(fu j)..j-1] ] )
```

A `bigupdate` function could perform an in-place update if the compiler determines this is safe (see [AH89, And89]). The semantics of `bigupdate` takes a middle ground between the `upd` function's incremental view of functional arrays and the `array` constructor's monolithic view.

We finally convert our program to use a envelope format version by making the appropriate substitutions to convert references to `lu` into references to `pl, pu`, and `d`. Here is the final complete version of the $L * U * x = b$ solver.

```
plub (n, pl, d, pu, irl, iru) b  =  x_vec
where
    fl i   =  i - 1 + irl!(i-1) - irl!i
    y i    =  s * d!i  where
            s  =  b!i - sum [ pl!(irl!i + j) * y_vec!j
                                  | j <- [(fl i)..i-1] ]
    y_vec  =  array (1,n) [ 1 = b!1 * d!1 ] ++
                          [ i  =  y i  |  i <- [2..n] ]

    fu j   =  j - 1 + iru!(j-1) - iru!j
    x_vec  =  j_loop n y_vec
    j_loop j x_vec  =
        if j < 1
            then x_vec
            else j_loop (j-1)
                    (bigupdate x_vec
                        [ i = x_vec!i - pu!(i + iru!j) * x_vec!j
                                      | i <- [(fu j)..j-1] ] )
```

Andy Sherman's version of `plub` also deals with reordering the unknowns to achieve a narrower envelope. The linear system $A * x = b$ may have been poorly organized for the envelope representation, and the equivalent system $P * A * P^{-1} * P * x = P * b$ may require

a smaller envelope to store A and its factorization. The permutation matrix P reorganizes the rows (P^{-1} the columns) using reverse Cuthill-McKee (RCM) heuristic to minimize the envelope size.

Sherman's version of `plub` assumes that the LU-factorized envelope-format matrix is in RCM order, whereas input vector `b` and result vector `x` are in the original 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.

References

[AH89] S. Anderson and P. Hudak. *Efficient Compilation of Haskell Array Comprehensions*. Technical Report YALEU/DCS/RR693, Yale University, Department of Computer Science, March 1989.

[And89] S. Anderson. *Compiling Monolithic Functional Arrays into DO Loops*. Technical Report YALEU/DCS/TR-, Yale University, 1989.

[HA88] P. Hudak and S. Anderson. *Haskell Solutions to the Language Session Problems at the 1988 Salishan High-Speed Computing Conference*. Technical Report YALEU/DCS/RR-627, Yale University, Department of Computer Science, January 1988.

[Hud84] P. Hudak. *ALFL Reference Manual and Programmer's Guide*. Research Report YALEU/DCS/RR-322, Second Edition, Yale University, October 1984.

[Hud86] P. Hudak. Arrays, non-determinism, side-effects, and parallelism: a functional perspective. In *Proceedings of the Santa Fe Graph Reduction Workshop*, pages 312–327, Los Alamos/MCC, Springer-Verlag LNCS 279, October 1986.

[HWe88] P. Hudak and P. Wadler (editors). *Report on the Functional Programming Language Haskell*. Technical Report YALEU/DCS/RR666, Yale University, Department of Computer Science, November 1988.

[Wad84] P. Wadler. Listlessness is better than laziness: lazy evaluation and garbage collection at compile time. In *Proceedings 1984 ACM Conference on LISP and Functional Programming*, pages 45–52, ACM, August 1984.

[Wad85] P. Wadler. Listlessness is better than laziness ii: composing listless functions. In *LNCS 217: Programs as Data Objects*, pages 282–305, Springer-Verlag, 1985.