Reprinted from *Proceedings of the 2012 ACM SIGPLAN International Conference on Functional Programming* (ICFP'12)

Experience Report: Haskell in Computational Biology

Noah M. Daniels   Andrew Gallant   Norman Ramsey

Department of Computer Science, Tufts University

{ndaniels, agallant, nr}@cs.tufts.edu

Abstract

Haskell gives computational biologists the flexibility and rapid prototyping of a scripting language, plus the performance of native code. In our experience, higher-order functions, lazy evaluation, and monads really worked, but profiling and debugging presented obstacles. Also, Haskell libraries vary greatly: memoization combinators and parallel-evaluation strategies helped us a lot, but other, nameless libraries mostly got in our way. Despite the obstacles and the uncertain quality of some libraries, Haskell's ecosystem made it easy for us to develop new algorithms in computational biology.

Categories and Subject Descriptors D.1.1 [*Applicative (Functional) Programming*]; J.3 [*Biology and genetics*]

Keywords memoization, stochastic search, parallel strategies, QuickCheck, remote homology detection

1. Introduction

Computational biologists write software that answers questions about sequences of nucleic acids (genomic data) or sequences of amino acids (proteomic data). When performance is paramount, software is usually written in C or C++. When convenience, readability, and productivity are more important, software is usually written in a dynamically typed or domain-specific language like Perl, Python, Ruby, SPSS, or R. In this paper, we report on experience using a third kind of language, Haskell:

- We had to reimplement an algorithm already implemented in C++, and the Haskell code is slower. But the Haskell code was easy to write, clearly implements the underlying mathematics (Section 3.1), was easy to parallelize, and performs well enough (Section 3.3). And our new tool solves a problem that could not be solved by the C++ tool which preceded it.
- Higher-order functions made it unusually easy to create and experiment with new stochastic-search algorithms (Section 3.2).
- Haskell slowed us down in only one area: understanding and improving performance (Section 3.4).
- Although the first two authors are computational biologists with little functional-programming experience, Haskell made it easy for us to explore new research ideas. By contrast, our group's C++ code has made it hard to explore new ideas (Section 4).
- The Haskell community offers libraries and tools that promise powerful abstractions. Some kept the promise, saved us lots of effort, and were a pleasure to use. Others, not so much. We couldn't tell in advance which would be which (Section 5.2).

2. The biology

Proteins, by interacting with one another and with other molecules, carry out the functions of living cells: metabolism, regulation, signaling, and so on. A protein's function is determined by its structure, and its structure is determined by the sequence of amino acids that form the protein. The amino-acid sequence is ultimately determined by a sequence of nucleic acids in DNA, which we call a gene. Given a gene, biologists wish to know the cellular function of the protein the gene codes for. One of the best known methods of discovering such function is to find other proteins of similar structure, which likely share similar function. Proteins that share structure and function are expected to be descended from a common ancestor—in biological terms, *homologous*—and thus the problem of identifying proteins similar to a *query sequence* is called *homology detection*.

Computational biologists detect homologies by building algorithms which, given a query sequence, compare it with known proteins. When the known proteins have amino-acid sequences that are not too different from the query sequence, homology can be detected by a family of algorithms called *hidden Markov models* (Eddy 1998). But in real biological systems, proteins with similar structure and function may be formed from significantly different amino-acid sequences, which are not close in edit distance. Our research software, MRFy (pronounced "Murphy"), can detect homologies in amino-acid sequences that are only distantly related. MRFy is available at `mrfy.cs.tufts.edu`.

3. The software

Homology-detection software is most often used in one of two ways: to test a hypothesis about the function of a single, newly discovered protein, or to compare every protein in a genome against a library of known protein structures. Either way, the software is *trained* on a group of proteins that share function and structure. These proteins are identified by a biologist, who puts their amino-acid sequences into an *alignment*. This alignment relates individual amino acids in a set of homologous proteins. An alignment may be represented as a matrix in which each row corresponds to the amino-acid sequence of a protein, and each column groups amino acids that play similar roles in different proteins (Figure 1).

An alignment may contain *gaps*, which in Figure 1 are shown as dashes. A gap in row 2, column j indicates that as proteins evolved, either protein 2 lost its amino acid in position j, or other proteins gained an amino acid in position j. If column j contains few gaps, it is considered a *consensus column*, and the few proteins with gaps probably lost amino acids via *deletions*. If column j contains *mostly* gaps, it is considered a *non-consensus column*, and the few proteins without gaps probably gained amino acids via *insertions*.

Once a protein alignment is constructed, it is used to train a *hidden Markov model*. A hidden Markov model is a probabilistic finite-state machine which can assign a probability to any query sequence. A protein whose query sequence has a higher probability

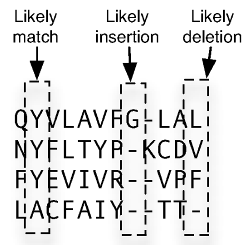


Figure 1. A structural alignment of four proteins ($C = 12$)

is more likely to be homologous to the proteins in the alignment. We write a query sequence as $x_1, \ldots, x_N$, where each x_i is an amino acid. The number of amino acids, N, can differ from the number of columns in the alignment, C.

A hidden Markov model carries probabilities on some states and on all state transitions. Both the probabilities and the states are determined by the alignment:

- For each column j of the alignment, the hidden Markov model has a *match state* M_j. The match state contains a table $e_{M_j}(x)$ which gives the probability that a homologous protein has amino acid x in column j.
- For each column j of the alignment, the hidden Markov model has an *insertion state* I_j. The insertion state contains a table $e_{I_j}(x)$ which gives the probability that a homologous protein has gained amino acid x by insertion at column j.
- For each column j of the alignment, the hidden Markov model has a *deletion state* D_j. The deletion state determines the probability that a homologous protein has lost an amino acid by deletion from column j.

The probabilities $e_{M_j}(x)$ and $e_{I_j}(x)$ are *emission probabilities.*

A hidden Markov model also has distinguished "begin" and "end" states. In our representation, each state contains a probability or a table of probabilities, and it is also labeled with one of these labels:

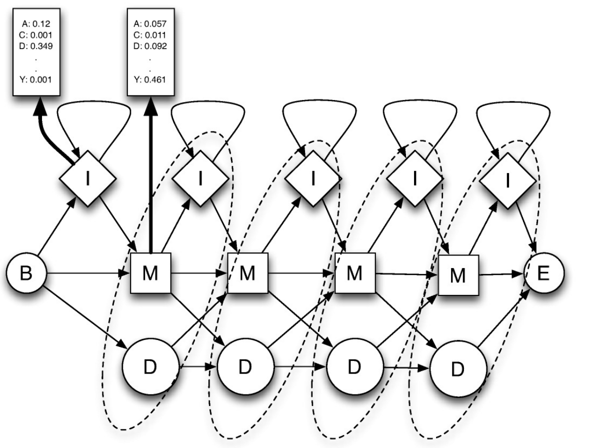
```
data StateLabel = Mat | Ins | Del | Beg | End
```

We use the "Plan7" hidden Markov model, which forbids direct transitions between insertion states and deletion states (Eddy 1998). "Plan7" implies that there are exactly 7 possible transitions into the states of any column j. Each transition has its own probability:

- A transition into a match state is more likely when column j is a consensus column. Depending on the predecessor state, the probability of such a transition is $a_{M_{j-1}M_j}$, $a_{I_{j-1}M_j}$, or $a_{D_{j-1}M_j}$.
- A transition into a deletion state is more likely when column j is a non-consensus column. The probability of such a transition is $a_{M_{j-1}D_j}$ or $a_{D_{j-1}D_j}$.
- A transition into an insertion state is more likely when column j is a non-consensus column. The probability of such a transition is $a_{M_{j-1}I_j}$ or $a_{I_{j-1}I_j}$.

3.1 Computing probabilities using perspicuous Haskell

Given a hidden Markov model, an established software package called HMMER (pronounced "hammer") can compute the probability that a new protein shares structure with the proteins used to train the model. The computation finds the most likely path through the hidden Markov model. To make best use of floating-point arithmetic, the software computes the *logarithm* of the probability of



This model has begin and end states B and E, as well as four nodes, each containing an insertion state I, a match state M, and a deletion state D.

Figure 2. A hidden Markov model ($C = 4$)

each path, by summing the logs of the probabilities on the states and edges of the path (Viterbi 1967). The path that maximizes the log of the probability is the most likely path.

The computation is specified on the left-hand side of Figure 3. A probability $V_j^M(i)$ represents the probability of the most likely path of the first i amino acids in the query sequence, terminating with placement of amino acid x_i in state M_j. Probabilities $V_j^I(i)$ and $V_j^D(i)$ are similar. The equations are explained very clearly by Durbin et al. (1998, Chapter 5)

To be able to use Haskell, we had to reimplement the standard algorithm for solving Viterbi's equations. Haskell made it possible for us to write code that looks like the math, which made the code easy to write and gives us confidence that it is correct.

Our code represents a query sequence as an immutable array of amino acids. In idiomatic Haskell, we might represent an individual amino acid x_i using a value of algebraic data type:

```
data Amino = Ala | Cys | Asp | Glu | ...   -- not used
```

But our models use a legacy file format in which each amino acid is a small integer used only to index arrays. We therefore chose

```
newtype AA = AA Int
```

Our legacy file format also *negates* the log of each probability, making it a positive number. The negated logarithm of a probability is called a *score*.

```
newtype Score = Score Double
```

Type `Score` has a limited `Num` instance which permits scores to be added and subtracted but not multiplied.

In a real hidden Markov model, each probability is represented as a score. Our code implements a transformed version of Viterbi's equations which operates on scores. The transformed equations are shown on the right-hand side of Figure 3. They minimize the score (the negated log probability) for each combination of column j, amino acid x_i, and state M_j, I_j, or D_j.

A model is represented as a sequence of *nodes*; node j includes states M_j, I_j, and D_j, as well as the probabilities of transitions out of that node. Each node contains tables of emission scores

$$V_j^M(i) = \log \frac{e_{M_j}(x_i)}{q_{x_i}} + \max \begin{cases} \log a_{M_{j-1}M_j} + V_{j-1}^M(i-1) \\ \log a_{I_{j-1}M_j} + V_{j-1}^I(i-1) \\ \log a_{D_{j-1}M_j} + V_{j-1}^D(i-1) \end{cases}$$

$$V_j^I(i) = \log \frac{e_{I_j}(x_i)}{q_{x_i}} + \max \begin{cases} \log a_{M_j I_j} + V_j^M(i-1) \\ \log a_{I_j I_j} + V_j^I(i-1) \end{cases}$$

$$V_j^D(i) = \max \begin{cases} \log a_{M_{j-1}D_j} + V_{j-1}^M(i) \\ \log a_{D_{j-1}D_j} + V_{j-1}^D(i) \end{cases}$$

$$V_j'^M(i) = e'_{M_j}(x_i) + \min \begin{cases} a'_{M_{j-1}M_j} + V_{j-1}'^M(i-1) \\ a'_{I_{j-1}M_j} + V_{j-1}'^I(i-1) \\ a'_{D_{j-1}M_j} + V_{j-1}'^D(i-1) \end{cases}$$

$$V_j'^I(i) = e'_{I_j}(x_i) + \min \begin{cases} a'_{M_j I_j} + V_j'^M(i-1) \\ a'_{I_j I_j} + V_j'^I(i-1) \end{cases}$$

$$V_j'^D(i) = \min \begin{cases} a'_{M_{j-1}D_j} + V_{j-1}'^M(i) \\ a'_{D_{j-1}D_j} + V_{j-1}'^D(i) \end{cases}$$

$$a'_{s\hat{s}} = -\log a_{s\hat{s}} \qquad e'_s(x) = -\log \frac{e_s(x)}{q_x} \qquad V_j'^M(i) = -V_j^M(i)$$

Figure 3. Viterbi's equations, in original and negated forms

e'_{M_j} and e'_{I_j}. These tables are read by function `eScore`, whose specification is `eScore` $s\ j\ i = e'_{s_j}(x_i)$. We place the transition probabilities into a record in which each field is labeled s_$\hat{s}$, where s and $\hat{s}$ form one of the 7 permissible pairs of state labels:

```
newtype TProb = TProb { logProbability :: Score }
data TProbs = TProbs
  { m_m :: TProb, m_i :: TProb, m_d :: TProb
  , i_m :: TProb, i_i :: TProb
  , d_m :: TProb, d_d :: TProb }
```

These scores are read by function `aScore`, whose specification is `aScore` $s\ \hat{s}\ (j-1) = a_{s_{j-1}\hat{s}_j}$.

Scores can be usefully attached to many types of values, so we have defined a small abstraction:

```
data Scored a = Scored { unScored :: !a, scoreOf :: !Score}
(/+/) :: Score -> Scored a -> Scored a
```

Think of a value of type `Scored a` as a container holding an "a" with a score written on the side. The `/+/` function adds to the score without touching the container. Function `fmap` is also defined; it applies a function to a container's contents. Finally, we made `Scored` an instance of `Ord`. Containers are ordered by score alone, so applying `minimum` to a list of scored things chooses the thing with the smallest (and therefore best) score.

Armed with our models and with the `Scored` abstraction, we attacked Viterbi's equations. The probability in each state is a function of the probabilities in its predecessor states, and all probabilities can be computed by a classic dynamic-programming algorithm. This algorithm starts at the begin state, computes probabilites in nodes 1 through C in succession, and terminates at the end state. One of us implemented this algorithm, storing the probabilities in an array. The cost was $O(|N| \times |C|)$; in MRFy, C and N range from several hundred to a few thousand.

Another of us was curious to try coding Viterbi's equations directly as recursive functions. Like a recursive Fibonacci function, Viterbi's functions, when implemented naïvely, take exponential time. But like the Fibonacci function, Viterbi's functions can be *memoized*. For example, to compute $V_j'^M(i)$ using the equation at the top right of Figure 3, we define `vee' Mat` $j\ i$. The equation adds $e'_{M_j}(x_i)$, computed with `eScore`, to a minimum of sums. The sum of an $a'_{s\hat{s}}$ term and a $V_{j-1}'^s(i-1)$ term is computed by function `avSum`, in which the terms are computed by `aScore` and `vee''`, respectively:

```
vee' Mat j i = fmap (Mat `cons`) $
   eScore Mat j i /+/ minimum (map avSum [Mat, Ins, Del])
 where avSum prev =
         aScore prev Mat (j-1) /+/ vee'' prev (j-1) (i-1)
```

What about the call to `fmap (Mat `cons`)`? This call performs a computation *not* shown in Figure 3: MRFy computes not only the probability of the most likely path but also the path itself. Function `(Mat `cons`)` adds M to a path; we avoid `(Mat :)` for reasons explained in Section 3.3 below.

Function `vee''` is the memoized version of `vee'`. Calling `vee''` produces the same result as calling `vee'`, but faster:

```
vee'' = Memo.memo3 (Memo.arrayRange (Mat, End))
                   (Memo.arrayRange (0, numNodes))
                   (Memo.arrayRange (-1, seqlen))
                   vee'
```

Functions `Memo.memo3` and `Memo.arrayRange` come from Luke Palmer's `Data.MemoCombinators` package. The value `numNodes` represents C, and `seqlen` represents N.

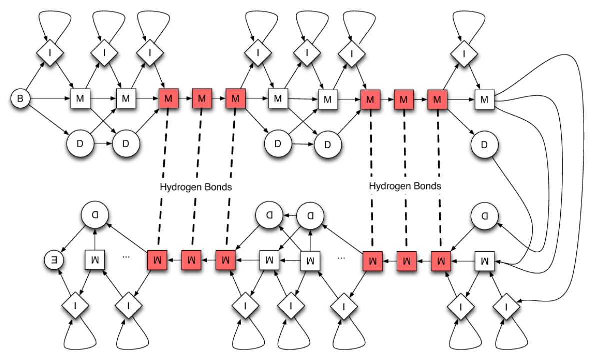
Memoization makes `vee'` perform as well as our classic dynamic-programming code. And the call to `Memo.memo3` is the *only* part of the code devoted to dynamic programming. By contrast, standard implementations of Viterbi's algorithm, such as in HMMER, spend much of their code managing dynamic-programming tables. Haskell enabled us write simple, performant code with little effort. Because the memoized version so faithfully resembles the equations in Figure 3, we retired the classic version.

3.2 Exploring new algorithms using higher-order functions

We use Viterbi's algorithm to help detect homologies in proteins with specific kinds of structure. When a real protein folds in three dimensions, amino acids that are far away in the one-dimensional sequence can be adjacent in three-dimensional space. Some groups of such acids are called *beta strands*. Beta strands can be hydrogen-bonded to each other, making them "stuck together." These beta strands help identify groups of homologous proteins. MRFy detects homologous proteins that include hydrogen-bonded beta strands; using prior methods, many instances of this problem are intractable.

Beta strands require new equations and richer models of protein structure. When column j of an alignment is part of a beta strand and is paired with another column $\pi(j)$, the probability of finding amino acid x_i in column j depends on the amino acid x' in column $\pi(i)$. If x' is in position i' in the query sequence, Viterbi's equations are altered; for example, $V_j'^M(i)$ depends not only on $V_{j-1}'^M(i-1)$ but also on $V_{\pi(j)}'^M(i')$. The distance between j and $\pi(j)$ can be as small as a few columns or as large as a few hundreds of columns. Because $V_j'^M(i)$ depends not only on nearby values but also on $V_{\pi(j)}'^M(i')$, dynamic programming cannot compute the maximum likelihood quickly (Menke et al. 2010; Daniels et al. 2012).

The new equations are accompanied by a new model. Within a beta strand, amino acids are not inserted or deleted, so a bonded



Each shaded node represents a beta-strand position. Nodes connected by dashed edges are hydrogen-bonded.

Figure 4. A Markov random field with two beta-strand pairs

pair of beta strands is modeled by a pair of sequences of match states. Between beta strands, the model is structured as before. The combined model, an example of which is shown in Figure 4, is called a *Markov random field*.

MRFy treats the beta strands in the model as "beads" which can slide along the query sequence. A positioning of the beta strands is called a *placement*. A placement's likelihood is computed based on frequencies of amino-acid pairs observed in hydrogen-bonded beta strands (Cowen et al. 2002). Given a placement, the maximum likelihood of the rest of the query sequence, between and around beta strands, is computed quickly and exactly using Viterbi's algorithm. This likelihood is *conditioned* on the placement.

MRFy searches for likely placements stochastically. MRFy implements random hill climbing, simulated annealing, multistart simulated annealing, and a genetic algorithm. These algorithms share much code, and MRFy implements them using higher-order functions, existentially quantified types, and lazy evaluation.

We describe MRFy's search abstractly: MRFy computes a sequence of *points* in a search space. The type of point is existentially quantified, but it is typically a single placement or perhaps a population of placements. Each point also has a `Score`; MRFy looks for points with good scores.

Ideally, MRFy would use the now-classic, lazy, modular technique advocated by Hughes (1989), in which one function computes an infinite sequence of points, and another function uses a finite prefix to decide on an approximation. But because MRFy's search is stochastic, making MRFy's search modular is not so easy.

To illustrate the difficulties, we discuss our simplest search: random hill climbing. From any given point in the search space, this search moves a random distance in a random direction. If the move leads to a better point, we call it *useful*; otherwise it is *useless*.

```
data Utility a = Useful a | Useless
```

(We also use `Useful` and `Useless` to tag *points*.) With luck, an infinite sequence of useful moves converges at a local optimum.

MRFy's search path follows only useful moves; if a move is useless, MRFy abandons it and moves again (in a new random direction) from the previous point. Ideally, MRFy would search by composing a *generator* that produces an infinite sequence of moves, a *filter* that selects the useful moves, and a *test* function that enumerates finitely many useful moves and returns the final destination. But a generator may produce an infinite sequence of useless moves. (For example, if MRFy should stumble upon a global optimum, every move from that point would be useless.) Given an infinite sequence of useless inputs, a filter would not produce any values, and the search would diverge.

We address this problem by combining "generate and filter" into a single abstraction, which has type `SearchGen pt r`. Type variable `pt` is a point in the search space, and `r` is a random-number generator. `Rand r` is a lazy monad of stochastic computations:

```
data SearchGen pt r =
 SG { pt0     :: Rand r (Scored pt)
    , nextPt  :: Scored pt -> Rand r (Scored pt)
    , utility :: Move pt -> Rand r (Utility (Scored pt))
    }
```

The monadic computation `pt0` randomly selects a starting point for search; `nextPt` produces a new point from an existing point. Because scoring can be expensive, both `pt0` and `nextPt` use *scored* points, and they can reuse scores from previous points.

To tell if a point returned by `nextPt` is useful, we call the `utility` function, which scrutinizes a move represented as follows:

```
data Move pt = Move { older        :: Scored pt
                    , younger      :: Scored pt
                    , youngerCCost :: CCost }
```

The decision about utility uses not only a source of randomness but also the *cumulative cost* of the point, which we define to be the number of points explored previously. The cumulative cost of the current point is also the age of the search, and in simulated annealing, for example, as the search ages, the `utility` function becomes less likely to accept a move that worsens the score.

Using these pieces, function `everyPt` produces an infinite sequence containing a mix of useful and useless moves:

```
everyPt :: RandomGen r
        => SearchGen pt r -> CCost -> Scored pt
        -> Rand r [CCosted (Utility (Scored pt))]
everyPt sg cost startPt = do
  successors <- mapM (nextPt sg) (repeat startPt)
  tagged <- zipWithM costedUtility successors [succ cost..]
  let (useless, CCosted (Useful newPt) newCost : _) =
                    span (isUseless . unCCosted) tagged
  (++) (CCosted (Useful startPt) cost : useless) <$>
                                everyPt sg newCost newPt
 where costedUtility pt cost =
         utility sg move >>= \u -> return $ CCosted u cost
        where move = Move { older = startPt, younger = pt
                          , youngerCCost = cost }
```

Both `nextPt` and `utility` are monadic, but we can still exploit laziness: from its starting point, `everyPt` produces an infinite list of randomly chosen successor points, then calls `costedUtility` to tag each one with a cumulative cost and a utility. We hope that if you look carefully at how `successors` is computed, you will understand why we separate `pt0` from `nextPt` instead of using a single function that produces an infinite list: We don't *want* the infinite list that would result from applying `nextPt` to many points in succession; we want the infinite list that results from applying `nextPt` to `startPt` many times in succession, each time with a different source of randomness.

Once the successors have been computed and tagged, `span` finds the first useful successor. In case there *is* no successor, `everyPt` also returns all the useless successors. If we do find a useful successor, we start searching anew from that point, with a recursive call to `everyPt`. (Because `everyPt` is monadic, the points accumulated so far are appended to its result using the `<$>` operator.) The most informative part of `everyPt` is last expression of the `do` block, which shows that the result begins with a useful point, is followed by a (possibly infinite, possibly empty) list of useless points, and then continues recursively with another call to `everyPt`.

The rest of the search uses Hughes's classic composition of generator and test function. Because our code is monadic, we use the monadic composition operator `=<<`, which is the bind operator with its arguments swapped:

```
search :: RandomGen r => SearchGen pt r -> SearchStop pt
       -> Rand r (History pt)
search strat test =
  return . test =<< everyPt strat 0 =<< pt0 strat
```

The `test` function has type `SearchStop pt`:

```
type SearchStop pt =
       [CCosted (Utility (Scored pt))] -> History pt
```

Type `History pt` retains only the useful points. (Internally, MRFy needs only the *final* useful point, but because we want to study how different search algorithms behave, we keep all the useful points.)

The definition of `SearchStop` reveals two forms of non-modularity which are inherent in MRFy's search algorithm. First, we need `Utility`, because if we omit the useless states, search might not terminate. Second, we need `CCosted`, because some of our test functions decide to terminate based either on the cumulative cost of the most recent point or on the difference between costs of successive useful points.

Despite these non-modular aspects, the search interface provides ample scope for experiments. Random hill climbing took 50 lines of code and one day to implement. Simulated annealing required only a new `utility` function, which took 15 lines of code and half an hour to implement. (Hill climbing accepts a point if and only if it scores better than its predecessor; simulated annealing may accept a point that scores worse.) Our genetic algorithm uses very similar functions, except for `nextPt`: recombination of parent placements took forty lines of code and a full day to implement.

We're not entirely happy with the way we're writing all the individual functions. In particular, `SearchStop` functions aren't composable; we can't, for example, combine two functions to say that we'd like to stop if scores aren't improving or if we've tried a thousand points, whichever comes first. Eventually, we'd like to have combinator libraries for `SearchStop` and `nextPt`, at least.

3.3 Performance

At each point in its search, MRFy calls `vee'` several times. Our `vee'` function computes a `Scored [StateLabel]`, that is, an optimal path and its score. But at intermediate points in MRFy's search, MRFy uses only the score. Even though Haskell evaluation is lazy, `vee'` still allocates thunks that could compute paths. To measure the relevant overhead, we cloned `vee'` and modified it to compute only a score, with no path. This change improved run time by nearly 50%.

Could we keep the improvement without maintaining two versions of `vee'`? In Lisp or Ruby we would have used macros or metaprogramming, but we were not confident of our ability to use Template Haskell. Instead, we used higher-order functions. As shown in Section 3.1, `vee'` does not use primitive `(:)` but instead uses an unknown function `cons`, which is passed in. To get a path, we pass in primitive `(:)`; to get just a score, we pass in `\_ _ -> []`. This trick is simple and easy to implement, and it provides the same speedup as the cloned and modified code. But we worry that it may work only because of undocumented properties of GHC's inliner, which may change.

Even with this trick, MRFy's implementation of Viterbi's algorithm is much slower than the C++ version in MRFy's predecessor, SMURF. For example, on a microbenchmark that searches for a structural motif of 343 nodes in a protein of 2000 amino acids using only Viterbi's algorithm and no beta-strand information, MRFy takes 2.32 seconds and SMURF takes 0.29 seconds.

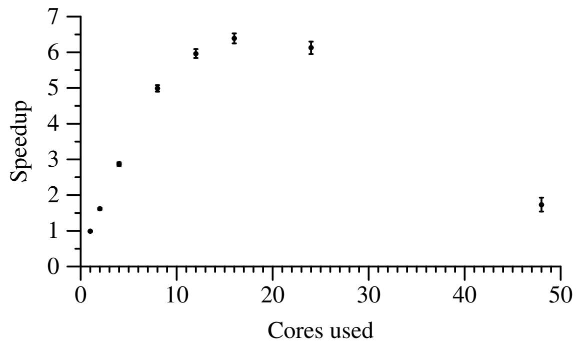


Figure 5. MRFy's parallel speedup on an 8-bladed beta propeller

But MRFy's job is not to run Viterbi's algorithm on large models; MRFy's job is to detect homologies to structures for which both Viterbi's algorithm and SMURF's more complex algorithm are unsuited. MRFy can solve problems that SMURF cannot. For example, we tried both programs on a complex, 12-stranded "beta sandwich" model. The model contains 252 nodes, 97 of which appear in the 12 beta strands. MRFy computes an alignment in under a minute, but SMURF allocates over 16GB of memory and does not terminate even after eight hours.

We also benchmarked MRFy using a model of an "8-bladed beta propeller." The model has 343 nodes, of which 178 appear in 40 beta strands. The segments between beta strands typically have at most 10 nodes. We used a query sequence of 592 amino acids, but each placement breaks the sequence into 41 pieces, each of which typically has at most 20 amino acids. Because MRFy can solve the models between the beta strands independently, this benchmark has a lot of parallelism, which Haskell made it easy to exploit. Using `Control.Parallel`, parallelizing the computation was as easy as substituting `parmap rseq` for `map`. Figure 5 shows speedups when using from 1 to 48 of the cores on a 48-core, 2.3GHz AMD Opteron 6176 system. Errors are estimated from 5 runs. After about 12 cores, where MRFy runs 6 times as fast as sequential code, speedup rolls off. By running 4 instances of MRFy in parallel on different searches, we hope to be able to use all 48 cores with about 50% efficiency.

3.4 Awkward debugging and testing

Our experience writing code and trying new ideas was excellent, as was the ease of parallelizing MRFy. Higher-order functions, memoization, laziness, and parallel strategies really worked. But we also encountered obstacles that prevented functional programming from working as well as we would have liked. The most significant obstacles were in debugging and testing.

We had a hard time diagnosing run-time errors. We expected some run-time errors; our group's legacy file format is poorly documented and hard to deal with. (When beta strands overlap and are doubly paired, even the invariants of the format are unclear.) But using Haskell, we found the errors hard to diagnose. Calls to `trace` littered our code, even when relegated to wrapper functions. We didn't know about the backtrace feature of GHC's profiler, and even after we learned about it, it didn't help: the profiler can be used only on very small test cases, which didn't always trigger the errors. This same limitation affected GHCi's debugger; in GHCi, our `vee'` function is too slow to be runnable on nontrivial inputs. Moreover, GHCi's debugger can set breakpoints only at top-level functions or at specific column/line positions, which made debugging the memoized `vee''` function impractical. In August 2011, Lennart Augustsson said that the biggest advantage of Strict Haskell is getting a stack trace on error, and Simon Marlow said that he may have figured out how to track call stacks properly in a lazy functional language. We can't wait.

Our difficulties with debugging led to internal disagreements. The junior members of our team wanted to apply the debugging skills they had honed through years of imperative programming. But these skills did not transfer well to Haskell. The senior member of the team kept repeating that a proper approach to debugging Haskell code should involve QuickCheck. But mere exhortation was unhelpful.

Only the senior member of our team was able to use QuickCheck easily. In retrospect, we have identified some obstacles that prevented the junior people from using QuickCheck.

- The examples and tutorials we found focused predominantly on writing and testing properties using data types that already implemented class `Arbitrary`. We didn't understand the `Arbitrary` class very well, perhaps because the overloaded `arbitrary` value is not a function. (For programmers accustomed to object-oriented dispatch on *arguments*, it is hard to grasp how Haskell's type-class system can find the proper version of `arbitrary` using only the type of a *value*.)
- Our difficulties were compounded by a weak understanding of monads. We were too baffled by QuickCheck's `Gen` monad to grasp its importance.
- We continually overlooked the critical role of *shrinking*. As a result, on the one or two occasions we did use QuickCheck, the counterexamples were too large to be informative.

Because of these obstacles, we wrote thousands of lines of code without ever defining an instance of `Arbitrary` and therefore without looking hard at QuickCheck. After the fact, we were overwhelmed by the work involved in writing and testing instances of `Arbitrary`. The work got done only when the whole team pitched in to meet the deadlines for this paper.

At the last minute, QuickCheck did find a bug in our implementation of Viterbi's algorithm: we had omitted the score for the transition from the final node of the hidden Markov model to the special "end" state. Without QuickCheck, we probably wouldn't have known anything was wrong.

4. Our previous experience compared

Our Haskell code for hidden Markov models and Viterbi's algorithm solves the same problems as existing C++ code. Other researchers using Haskell may also have to reimplement code, but in computational biology, reimplementing existing algorithms is unremarkable. For example, both SMURF and HMMER also contain new implementations of hidden Markov models and Viterbi's algorithm.

When performance has mattered, members of our group, like other computational biologists, have used C++. To compare our Haskell experience with our C++ experience, we discuss three tools:

- Matt (Menke et al. 2008) is used to create alignments like that shown in Figure 1. It comprises about 12,000 lines of C++. The only external tools or libraries it uses are `zlib` and `OpenMP`, and its initial development took two years.
- SMURF (Menke et al. 2010) is used to detect homologous proteins in the presence of paired beta strands. It comprises about 9,000 lines of C++, of which about 1,000 lines are shared with Matt. It uses no external tools or libraries, and its initial development took a year and a half. It uses multidimensional dynamic programming to exactly compute the alignments for which MRFy relies on stochastic search. As a result, in the presence of complex beta-strand topologies, SMURF is computationally intractable.
- MRFy is used to detect homologous proteins in the presence of paired beta strands; it effectively supplants SMURF. It comprises about 2,500 lines of Haskell, about 500 of which are devoted to tests, QuickCheck properties, and generators. Neither Matt nor SMURF includes test code. MRFy uses several external tools and libraries, of which the most notable are Parsec, the BioHaskell bioinformatics library, and the libraries `Data.MemoCombinators`, `Control.Parallel`, and `Data.Vector`. MRFy's initial development took about three months.

Like much research software, all three tools were written in haste. We have experience modifying the older tools.

We modified Matt to use information about sequences as well as structure. The modification added 2,000 lines of code, and it calls external sequence aligners that we did not write. We thought the modification would take three months, but it took most of a year. Matt uses such data structures as mutable oct-trees, vectors, and arrays. It uses clever pointer arithmetic. The mutable data structures were difficult to repurpose, and the pointer arithmetic was *too* clever: nearly every change resulted in new segfaults.

We had hoped to extend Matt further, with support for partial alignments, which we expected to require only a cosmetic manipulation of the output. But this feature wound up requiring deep information about Matt's data structures, and we had to give up. We believe we could write an equivalent tool in Haskell, with most of Matt's performance, in at most nine months.

Our most painful experience was adding "simulated evolution" to SMURF (Daniels et al. 2012). Although simulated evolution represents a relatively minor enhancement, just understanding the existing code took several months.

We built MRFy quickly, and we expect that higher-order functions will make MRFy easy to extend. Each new addition to MRFy's stochastic search has taken at most a day to implement.

Haskell encourages hasty programmers to slow down. We have to get the types right, which makes it hard to write very large functions. To get the code to typecheck, we have to write type signatures, which also serve as documentation. And once the types are accepted by the compiler, it is not much more work to write contracts for important functions. MRFy is still hasty work. Many types could be simplified; we're sure we've missed opportunities for abstraction; and we know that MRFy's decomposition into modules could be improved. But despite being hasty programmers, we produced code that is easy to understand and easy to extend. Our hastily written Haskell beats our hastily written Ruby and C++.

Looking beyond our research group to computational biology more broadly, our experience with other software is better. Little of it is written in functional languages, but much of the software shared by the community is excellent. MRFy's training component was derived from that of HMMER, and working with the HMMER codebase was pleasant; data structures and their mutators are well documented. There is a BioHaskell library, part of which we use, but it is not nearly as complete as BioPython or BioRuby, which are heavily used in the community. We hope that tools for computational biology in Haskell continue to mature.

5. What can you learn from our experience?

If you are a computational biologist and you are interested in functional programming, you don't need extensive preparation to be productive in Haskell. Two of us (Daniels and Gallant) are graduate students. Daniels has taken a seminar in functional programming, which included some Haskell; Gallant has taken a programming-languages course which included significant functional programming but no Haskell. Ramsey is a professor who has used Haskell for medium-sized projects, but his contributions to MRFy have been limited, mostly to *post hoc* refactoring and testing.

5.1 Obstacles to be overcome

We had quite some difficulty profiling, but we hope that this difficulty may be mitigated by new profiling tools released early in 2012 with GHC 7.4. GHC assigns costs to "cost centers" (Sansom and Peyton Jones 1997), and in GHC 7.0, which we used for most of MRFy's development, cost-center annotations had to be added manually to nested functions. Although these annotations made our code so ugly that we felt compelled to remove them, they did enable us to improve the performance of `vee'` as discussed in Section 3.3. GHC 7.4 provides more sophisticated profiling tools, which we look forward to using. Difficulties using Cabal to enable profiling of installed libraries may remain.

Like other functional programmers, we have found that once we have our types right, our code is often right. But MRFy computes with arrays and array indices, and in that domain, types don't help much. Bounds violations lead to run-time errors, which we have not been able to identify any systematic way to debug. GHC's profiler can provide stack traces, but we found this information difficult to discover, and as noted above, there are obstacles to profiling. We're aware that debugging lazy functional programs has been a topic of some research, but one of the biggest obstacles we encountered to using Haskell is that we have had to abandon our old approaches to debugging.

Ideally we would use QuickCheck to find bugs, but as we mention in Section 3.4, we found obstacles. We have now overcome these obstacles, but we sorely regret not doing so earlier. In light of our experience, we will institute a new programming practice: whenever we introduce a new data type, we will write the instance of `Arbitrary` right away, while relevant invariants are still fresh in memory. When invariants are not enforced by Haskell's static type system, we will write them as Haskell predicates. We can then immediately run QuickCheck on each predicate, to verify that our Arbitrary instance agrees with the predicate. For each predicate `p` we can also check `fmap (all p . shrink) arbitrary`.

5.2 Information that will help you succeed

If you want to use Haskell in your research, we believe that you must have enough experience with functional programming that you can build *all* the code you need, not only the code that is easy to write in a functional language. Implementing Viterbi's equations in Haskell was pure joy. Writing an iterative search in purely functional style was easy. Transforming data in the HMMER file format, *without* using mutable state the way the C++ code does, was difficult.

While the Haskell community offers many enticing tools, libraries, and packages, not all of them are worth using. Some are not ready for prime time, and some were once great but are no longer maintained. The great packages, like `Data.MemoCombinators` and Parallel Strategies, are truly great. But for amateurs, it's not always easy to tell the great packages from the wannabes and the has-beens. And even some of the great packages could be better documented, with more examples.

As in any endeavor, access to experts helps. We would have been better off if our in-house expert had been an enthusiastic student and not a busy professor. But we have been surprised and pleased by the help available from faraway experts on Stack Overflow and on Haskell mailing lists. Although a local expert makes things easier, one is not absolutely necessary.

6. Conclusion

A little knowledge of and a lot of love for functional programming enabled us to carry out a successful research project in a language that computational biologists seldom use. If you *want* to use Haskell—or one of your graduate students wants to use Haskell—you can succeed.

Acknowledgments

Anonymous referees spurred us to think more deeply about laziness and to write more carefully about performance. Josef Svenningsson suggested a correspondence between MRFy's search and stream fusion, which led us eventually to the `Utility` type. Koen Claessen instantly diagnosed an inappropriately strict implementation of the `Rand` monad. We also thank Lenore Cowen, Kathleen Fisher, Ben Hescott, Brad Larsen, and Nathan Ricci.

This work was funded in part by NIH grant 1R01GM080330.

References

Lenore Cowen, Philip Bradley, Matt Menke, Jonathan King, and Bonnie Berger. Predicting the beta-helix fold from protein sequence data. *Journal of Computational Biology*, 2002.

Noah Daniels, Raghavendra Hosur, Bonnie Berger, and Lenore Cowen. SMURFLite: combining simplified Markov random fields with simulated evolution improves remote homology detection for beta-structural proteins into the twilight zone. *Bioinformatics*, March 2012.

Rirchard Durbin, Sean Eddy, Anders Krogh, and Graeme Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, May 1998.

Sean Eddy. Profile hidden Markov models. *Bioinformatics*, 14: 755–763, 1998.

John Hughes. Why functional programming matters. *The Computer Journal*, 32(2):98–107, April 1989.

Matthew Menke, Bonnie Berger, and Lenore Cowen. Matt: local flexibility aids protein multiple structure alignment. *PLoS Computational Biology*, 2008.

Matthew Menke, Bonnie Berger, and Lenore Cowen. Markov random fields reveal an N-terminal double beta-propeller motif as part of a bacterial hybrid two-component sensor system. *Proceedings of the National Academy of Science*, 2010.

Patrick Sansom and Simon L Peyton Jones. Formally based profiling for higher-order functional languages. *ACM TOPLAS*, 19 (2):334–385, 1997.

Andrew Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, April 1967.

You can live to surf the Haskell wave,
but if you slide off the crest, you drown.