Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science
6.827 Multithreaded Parallelism: Languages and Compilers

Problem Set 3

Please Remember: 1) You can work in groups of up to three people; include the names of all group members on all problems. 2) Turn in the answer to each problem as a separate packet. 3) Comment your code carefully and include output from sample runs.

We believe this problem set is rather challenging, but you should have no serious difficulties if you start early! We'll continue to use pH to write some larger functional programs than in Problem Set 2. The first problem asks you to implement the Huffman coding algorithm to compress files. This problem combines algebraic types, arrays, and a bit of I/O. In the second problem, we explore the concept of *generalized arrays* and make extensive use of the Haskell/pH array data type. In the final two problems, we implement an applicative interpreter and a basic type-checker for a "core" version of pH. We have provided basic templates to get you started on these problems in the files `/mit/6.827/ps-data/ps3-n.hs`.

Problem 1 — Huffman Codes

Huffman codes are *variable-length codes* used to compress data. *Fixed-length codes* like ASCII use a fixed number of bits to represent characters. Huffman codes use a variable number of bits to represent characters, and the encoding is chosen according to the frequency of occurrences of the characters in the data.

Huffman codes are often depicted as trees. Suppose we have an alphabet containing only the three symbols `a, b, c` and that our algorithm has produced the encoding shown in the figure. The encoding of a character according to this particular Huffman code is the path followed to reach the character from the root of the tree. For example, the code for `a` is *L* since we go down the left subtree from the root to reach `a`. Similarly, the codes for `b` and `c` are *RL* and *RR* respectively.

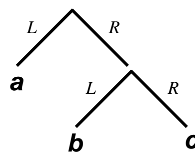

Figure 1: Tree.

Throughout this problem, you should assume that alphabets contain at least two symbols and that data to be encoded uses at least two different symbols.

Part a:
In this first part, we'll look at the process of encoding and decoding data using a Huffman code. We define the following data types for this purpose:

```
data Bit   = L | R    deriving Show
type HCode = [Bit]
type Table = Array Char HCode
data Tree  = Left Char Int | Node Int Tree Tree   deriving (Read,Show)
```

The `Bit` data type represents the components of the route down the encoding tree, and the `HCode` data type represents the encoding of a particular character or characters.[1] Next, the `Table` data type defines a mapping from characters to Huffman encodings via an array. Finally, the `Tree` data structure is used to map codes to characters; ignore the `Int` field in each disjunct for now. In our example, the code corresponds to the following `Tree` and `Table` objects.

```
encoding = array ('a','c') [('a',[L]),('b',[R,L]),('c',[R,R])]
decoding = Node 0 (Leaf 'a' 0) (Node 0 (Leaf 'b' 0) (Leaf 'c' 0))
```

Write the functions

```
encodeData :: Table -> String -> HCode
decodeData :: Tree -> HCode -> String
```

that encode and decode data. Try these out using the `encoding` and `decoding` objects give above.

Part b:

Now we consider the problem of building Huffman coding trees and encoding tables. We give the algorithm in several steps:

1. The first step in this process is to build a histogram of the number of occurrences of each symbol in the data to be encoded. Write a function

   ```
   countChars :: String -> Array Char Int
   ```

   that creates the histogram. Assume that the alphabet is composed of all characters in the Haskell/pH `Char` data type. Characters can be represented using single quotes as in `'a'` or they can be created using the integers 0-255 and the `toEnum` method of the `Enum` typeclass, a class that has `Char` as one of its members. For the message `aabcccaba`, the counts for characters `a`, `b`, and `c` are 4, 2, and 3, respectively. Your job will be much easier if you use Haskell's accumulator comprehension accumArray (which is implemented using M-structures as shown in lecture).

2. Given the character counts, we build a separate coding tree, containing only a leaf node, for each character that occurs at least once in the input. As you may have guessed by now, the count for a character is entered into the `Int` field we ignored above. For the data `aabcccaba`, the set of encoding trees can be represented as the list (sorted by increasing count):

[1] Huffman codes are *prefix codes* so parsing is unambiguous.

```
[  Leaf 'b' 2 , Leaf 'c' 3 , Leaf 'a' 4 ]
```

3. We then start combining the two trees with the lowest counts into a new tree by making the two trees subtrees of a new `Node`. The character count field for the new node is the sum of the counts of its subtrees. The process continues until a single tree results. In the example above, the set of trees resulting from the first combination, again sorted by count, would be:

   ```
   [ Leaf 'a' 4, Node 5 (Leaf 'b' 2) (Leaf 'c' 3) ]
   ```

   And the final tree would be:

   ```
   Node 9 (Leaf 'a' 4)  (Node 5 (Leaf 'b' 2) (Leaf 'c' 3))
   ```

   Write a function

   ```
   makeCodingTree :: Array Char Int -> Tree
   ```

   that takes as input the frequency histogram of the characters in the file and produces a single coding tree.

   As you can see, the key to the Huffman coding algorithm is that characters that occur most often in the input data are pushed to the top of the encoding tree. In this way, their encoding will require fewer bits. Less frequent characters are pushed to deeper levels in the tree and will require more bits to encode.

4. Finally, write a function

   ```
   makeCodeTable :: Tree -> Table
   ```

   that converts an encoding tree into an encoding table suitable for the `encodeData` routine. If a particular character does not appear in an input file, make sure that its entry in the `Table` is `[]`.

Part c:

Now that we've got the tools to create encodings from inputs, we need a way to represent encoded messages (lists of `Bits`) as real bit streams. Such a representation will allows us to output encoded data compactly to files. The idea is to divide a list of `Bits` into chunks of eight `Bits`; to convert each chunk into an integer; and then to use the integer with `toEnum` to create a `Char`. Assume the `L` constructor represents a 0 bit while `R` represents a 1 bit. Write a function

```
hCodeToString :: HCode -> String
```

to perform the conversion.

What happens if the length of the Huffman encoding of your input data (the list of `Bits`) is not a multiple of eight? Make sure you handle this case properly so that you can write the function

```
stringToHCode :: String -> HCode
```

to perform the inverse operation of `hCodeToString`. The function `stringToHCode` takes a string (a list of `Char`) and decodes it into an `HCode` (a list of `Bits`). Assume that all the inputs to `stringToHCode` have been generated using your version of `hCodeToString`. In other words, `hCodeToString` and `stringToHCode` must agree on a way to handle encodings with lengths that are not multiples of eight bits.

Part d:
Write two functions

```
encode :: String -> (Tree,String)
decode :: (Tree,String) -> String
```

to serve as the top-level interface to your Huffman coding routines. The `encode` function takes a string as input, encodes it, and returns the encoding tree and the encoded string. The `decode` function takes an encoding tree and an encoded string and produces a decoded string as output. These functions are called by

```
encodeFile :: FilePath -> FilePath -> IO ()
decodeFile :: FilePath -> FilePath -> IO ()
```

which we have provided for you. The parameters to `encodeFile` and `decodeFile` are the name of an input file and the name of an output file (both are strings).

Try encoding and decoding several files, including the sample files (`ps3-sample{1-3}`) in the `/mit/6.827/ps-data` directory. However, don't try to encode or decode very large files as pH will run out memory (Why?).

How much compression do you get? All the sample files contain the same information, but `ps3-sample2` and `ps3-sample3` have been preprocessed in special ways. Can you guess how?

Problem 2 Generalized Arrays

Dr. Bunsen Honeydew[2] of Jee-whiz Projects Laboratories (JPL) grabs you in the hallway one morning... the conversation goes something like this:

> Hey, *your name here*, you're a pH wizard, but I think pH is terrible... and I'll tell you why. I need to compute some Fourier transforms—sometimes of a vector, sometimes of a matrix, sometimes even of a cube or an object with still more dimensions! Furthermore, in some cases I need to use an entire vector—row or column—and sometimes I need to use only *part* of a vector. In pH, I need to write a bunch of Fast Fourier Transforms (FFT's), one for each of these cases.
>
> Even `FORTRAN` handles all my FFT needs with one routine because it allows me to treat rows and columns of matrices as vectors, and it even allows me to select pieces of vectors.
>
> Hummmph....

[2]All names have been altered to protect the innocent.

At this point, Dr. Honeydew walks off mumbling something about the evils of functional programming...

In a flash of inspiration, you remember *generalized arrays* (see the pH book), and you think "hey, these would do the trick!" The algorithms to do these transformations all use the basic 1D FFT, sometimes on rows, sometimes on columns. You realize that if you implement the 1D FFT algorithm using generalized arrays, the doctor will be able to use it to compute 2D and 3D FFTs.

The 1D FFT of an N-vector $\vec{x}$ of complex numbers is the vector $\vec{y} = F_N\vec{x}$, where F_N is

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega & \omega^2 & \dots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \dots & \omega^{2(N-1)} \\ 1 & \omega^3 & \omega^6 & \dots & \omega^{3(N-1)} \\ . & . & . & \dots & . \\ . & . & . & \dots & . \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \dots & \omega^{(N-1)(N-1)} \end{bmatrix}$$

In general, the $(i, j)th$ entry of the matrix is $\omega^{ij}, 0 \leq i, j \leq N - 1$, where $\omega = \omega_N$ is a principal Nth root of unity, $w^N = 1$ and for $0 < j < N, w^j \neq 1$. For this problem, we will assume that N is a power of 2.

Though the FFT transformation is expressed as a matrix multiply, we can design a much more efficient algorithm using a divide-and-conquer strategy. This algorithm works by dividing the input array into two equal parts, applying itself recursively and finally combining the results of the two halves to yield the FFT of the original array.

A pH implementation of this algorithm is given in the file `/mit/6.827/ps-data/ps3-2.hs`. This implementation uses ordinary arrays. Your task is to write the same algorithm using two different generalized array representations: the first is the one given in the pH book, and second is developed below.

Part a:

Write the function `fft` which takes two arguments:

- The input array (in generalized form).
- The roots of unity array `roU`.

and returns the FFT in a *normal array* (you needn't generalize the result).

Your function should do the following steps:

- If the size of the input array is 4, it computes the FFT directly.
- If not, it solves the problem recursively:

- It *shuffles* the input generalized array into two generalized arrays of half the size, one consisting of odd elements `v_left` and the other consisting of even elements `v_right`.
- Arrays `fft_left, fft_right` are computed using recursive calls to `fft`.
- The final result is returned using the function `combine` which combines these arrays.

Part b:

You show your solution to the doctor, and he spots an efficiency problem with generalized arrays. He points out that a single access of an array element at the bottom of the recursion takes many (how many?) function applications to yield the value of that array element. You want to improve the efficiency of the program by making sure that the number of applications needed to access an array element is bounded by a constant (irrespective of the input array size). You think hard and come up with a more general vector representation which achieves this goal. Re-implement your FFT functions using this new representation and measure the reduction count. Are we doing better than before? How do we compare with the array copying version of FFT? Which implementation is the best?

Hint: The new generalized array is of the form `(low,n,stride,f)`. The array has `n` elements and is accessed by array indices 1 through n. The first element is `(f l)`. What is `stride`?

Part c:

You might also notice that the `fft` function requires a lot of storage, as the FFT array is computed and discarded at each stage of the recursion. We will need to make a tradeoff between parallelism and storage use in order to use machine resources effectively. We can circumvent this problem in several ways, but the simplest is to use an IVector. Rewrite the FFT code so that only a single IVector is allocated to hold the result of the transform.

Problem 3 Core pH: Type Checker

In this problem, you will implement the type-checking algorithm described in class. This process is somewhat long, but it will be broken down into small steps. Debugging your program will be much easier if you have a thorough understanding of the algorithm. You should work out the algorithm by hand on several examples before trying to program it. However, you need to submit only your program with its test cases.

Types in our language are represented as follows. Types are either type variables, type constants or function types.

```
data Type = TVar TIdent | TConst BaseType | Arrow Type Type
            deriving (Eq,Show)
data TIdent = TName Int   deriving (Eq,Show)
data BaseType = IntType | BoolType   deriving (Eq,Show)
```

Type schemes are generalized types. The type is generalized with respect to the variables in the list.

```
data TScheme = TScheme [TIdent] Type   deriving (Eq,Show)
```

A type environment is a list of (variable,type) scheme pairs. In our algorithm, we use the environment to store types of variables seen "before" in the program.

```
type TEnv = [(Ident,TScheme)]
```

A substitution is a list of (type-variable, type) pairs. The substitution is actually applied on a type in the reverse order, i.e. the substitution last in the list is the one which is applied first.

```
type Subst = [(TIdent,Type)]
```

Part a:

The first step in writing the type-checking algorithm is to write functions to manipulate environments. We need two functions:

```
extendEnv :: TEnv -> Ident -> TScheme -> TEnv
-- This function extends the environment with the
-- (variable, type scheme) pair.

getEnv :: TEnv -> Ident -> TScheme
-- This function looks up the environment for the variable, generates
-- an error if it is not present.
```

Part b:

That was easy! Next, we need functions to manipulate type variables. In particular, we will be dealing with *sets* of type variables. For this, you will want to use the `List` library which provides functions to treat lists as sets. Write the following functions:

```
freeVars :: Type -> [TIdent]
-- Returns free variables of a type.

freeVarsScheme :: TScheme -> [TIdent]
-- Returns free variables of a type scheme.

freeVarsEnv :: TEnv -> [TIdent]
-- Returns free variables of a type environment.

newTypeVar :: NameSupply -> TIdent
-- Generates a new type variable, not used anywhere before.
```

Note that you will need to provide a "name supply" for your code—a means of generating new type variables. It is up to you how you choose to implement this.

Part c:

Now, we will write functions to apply substitutions on the various entities in the type system. You need to write the following functions. Note that these type signatures omit any mention of a `NameSupply`—you should add arguments and results to deal with this "plumbing" where it is necessary. Later on we will discuss nice ways of encapsulating such plumbing.

```
applySubToT :: Type -> Subst -> Type
-- Apply a substitution to a type.

applySubToScheme :: TScheme -> Subst -> TScheme
-- Apply a substitution to a type scheme. Be careful to avoid
-- variable capture!

applySubToEnv :: TEnv -> Subst -> TEnv
-- Apply a substitution to a type environment.

combine :: Subst -> Subst -> Subst
-- Combine two substitutions. Note that (combine s2 s1)
-- is the substitution which applies s1 first and then s2.
```

Part d:

Next, we write functions which convert type schemes to types and vice versa.

```
instantiate :: TScheme -> Type
-- This function instantiates the type scheme, i.e. it replaces
-- the generalized type variables by new type variables.

generalize :: TEnv -> Type -> TScheme
-- This function generalizes (or closes) a type to generate a type
-- scheme.
```

Part e:

The next step in the algorithm is to write the unification procedure.

```
unify :: Type -> Type -> Subst
-- This function returns a substitution which unifies the argument
-- types. If the types are not unifiable, it generates an error.
```

Part f:

Finally, we are ready to write the type inference algorithm `w`.

```
w :: TEnv -> Exp -> (Subst,Type)
-- This implements the inference algorithm.
```

In addition to these functions given above, you can write “glue” functions to make your program more readable. You’ll want to use the features of the `List` library and the prelude extensively in your code; functions like `foldr, foldl, map, lookup` etc. are very helpful in writing concise programs.

Test out your type inference function on the following inputs:

```
let
   k = \x -> \y -> x
in
  k 1 (k True 2)
```

```
\k -> k 1 (k True 2)

\k ->
  let
     k1 = k
  in
     k1 1 (k1 True 2)

\x -> \y -> plus x y
```

You do not need to parse these functions, just convert them by hand to the expression representation (`Exp`) used by your program. Feel free to include any other test cases.