Functional Programming Language – Haskell

Mohammed Aslam
CIS 24
Prof. Kopec
Presentation: 03
Date: 05/05/2003

Haskell is a general purpose, purely functional programming language named after the logician Haskell Brooks Curry. Haskell is based on *lambda calculus*, hence the lambda is used as a logo. It was designed in 1988 by a 15-member committee to satisfy, among others, the following constraints.

- It should be suitable for teaching, research, and applications, including building large systems.
- It should be freely available.
- It should be based on ideas that enjoy a wide consensus.
- It should reduce unnecessary diversity in functional programming languages.

It's features include higher-order functions, non-strict (lazy) semantics, static polymorphic typing, user-defined algebraic data types, type-safe modules, stream and continuation I/O, lexical, recursive scoping, curried functions, pattern-matching, list comprehensions, extensible operators and a rich set of primitive data types.

The Structure of Haskell Programs

A module defines a collection of values, data types, type synonyms, classes, etc. and *exports* some of these resources, making them available to other modules.

A Haskell *program* is a collection of modules, one of which, by convention, must be called `Main` and must export the value `main`. The *value* of the program is the value of the identifier `main` in module `Main`, and `main` must have type `Dialogue`.

Modules may reference other modules via explicit `import` declarations, each giving the name of a module to be imported, specifying its entities to be imported, and optionally renaming some or all of them. Modules may be mutually recursive.

The name space for modules is flat, with each module being associated with a unique module name.

There are no mandatory type declarations, although Haskell programs often contain type declarations. The language is strongly typed. No delimiters (such as semicolons) are required at the end of definitions - the parsing algorithm makes intelligent use of layout.

The notation for function application is simply juxtaposition, as in `sq n`.

Single line comments are preceded by ``--'' and continue to the end of the line. For example:

```
succ n = n + 1  -- this is a successor function
```

Multiline and nested comments begin with **{-** and end with **-}**. Thus

```
{- this is a
     multiline
        comment -}
```

Haskell is a typeful programming language: *types* are pervasive, and the newcomer is best of becoming well aware of the full power and complexity of Haskell's type system from the outset. For those whose only experience is with relatively *untypeful* languages such as Perl, Tcl, or Scheme, this may be a difficult adjustment; for those familiar with Java, C, Modula, or even ML, the adjustment should be easier but still not insignificant, since Haskell's type system is different and somewhat richer than most. In any case, *typeful programming* is part of the Haskell programming experience, and cannot be avoided.

Programs written in Haskell tend to be much more concise than their imperative counterparts. Quicksort is a rather extreme case, but in general functional programs are much shorter (two to ten times). Programs are often easier to understand. You should be able to understand the program without any previous knowledge of either Haskell or quicksort. The same certainly cannot be said of the C program. It takes quite a while to understand, and even when you do understand it, it is extremely easy to make a small slip and end up with an incorrect program. Here is a detailed explanation of the Haskell quicksort:

Quicksort in Haskell

```
qsort []     = []
qsort (x:xs) = qsort elts_lt_x ++ [x] ++ qsort elts_greq_x
                 where
                   elts_lt_x   = [y | y <- xs, y < x]
                   elts_greq_x = [y | y <- xs, y >= x]
```

The first line reads: "The result of sorting an empty list (written []) is an empty list". The second line reads: "To sort a list whose first element is x and the rest of which is called xs, just sort all the elements of xs which are less than x (call them elts_lt_x), sort all the elements of xs which are greater than or equal to x (call them elts_greq_x), and concatenate (++) the results, with x sandwiched in the middle."

The definition of elts_lt_x, which is given immediately below, is read like this: "elts_lt_x is the list of all y's such that y is drawn from the list xs, and y is less than x". The definition of elts_greq_x is similar. The syntax is deliberately reminiscent of standard mathematical set notation, pronouncing "|" as "such that" and " <-" as "drawn from".

When asked to sort a non-empty list, qsort calls itself to sort elts_lt_x and elts_greq_x. That's OK because both these lists are smaller than the one originally given to qsort, so the splitting-and-sorting process will eventually reduce to sorting an empty list, which is done rather trivially by the first line of qsort.

Quicksort in C

```
qsort( a, lo, hi ) int a[], hi, lo;
{
  int h, l, p, t;

  if (lo < hi) {
    l = lo;
    h = hi;
    p = a[hi];

    do {
      while ((l < h) && (a[l] <= p))
          l = l+1;
      while ((h > l) && (a[h] >= p))
          h = h-1;
      if (l < h) {
          t = a[l];
          a[l] = a[h];
          a[h] = t;
      }
    } while (l < h);

    t = a[l];
    a[l] = a[hi];
    a[hi] = t;

    qsort( a, lo, l-1 );
    qsort( a, l+1, hi );
  }
}
```

No core dumps

Haskell is strongly typed, eliminating a huge class of easy-to-make errors at compile time. In particular, strong typing means no core dumps! There is simply no possibility of treating an integer as a pointer, or following a null pointer.

Code re-use

Of course, strong typing is available in many imperative languages, such as Ada or Pascal. However, Haskell's type system is much less restrictive than, say, Pascal's, because it uses polymorphism. For example, the *qsort* program will not only sort lists of integers, but also lists of floating point numbers, lists of characters, lists of lists; indeed, it will sort lists of anything which can be compared by the less-than and greater-than operations. In contrast, the C version is restricted to sorting an array of integers. Polymorphism also enhances re-usability.

Strong glue

Haskell has another powerful feature: it only evaluate as much of the program as is required to get the answer – this is called *lazy evaluation*. This feature is rather like Unix pipes. For example, the Unix command

```
grep printf Foo.c | wc
```

counts the number of lines in the file *Foo.c* which include the string *printf* The command *"grep printf Foo.c"* produces all lines which contain the string *"printf",* while the *"wc"* command counts them. The pipe, written *"|"*, takes the output from the first command and delivers it to the second. The two commands execute together, so that the output of the first is consumed more-or-less immediately by the second. In this way, no large intermediate files need be produced. You can think of *wc* "demanding" lines from the *grep*

If the second command only needs some of the output of the first, then execution of the first command might never need to be completed. For example

```
grep printf Foo.c | head 5
```

just prints the first 5 lines which contain "*printf*". There is no need to modify the *grep* command to take account of the fact that its execution might be abandoned.

Haskell provides exactly this kind of demand-driven evaluation. Data structures are evaluated just enough to deliver the answer, and parts of them may not be evaluated at all. As in the case of Unix commands, this provides powerful "glue" with which to compose existing programs together. What this means is that it is possible to re-use programs, or pieces of programs, much more often than can be done in an imperative setting. *Lazy evaluation* allows us to write more modular programs.

Powerful abstractions

In general, Haskell offer powerful new ways to encapsulate abstractions. An abstraction allows defining an object whose internal workings are hidden; a C procedure, for example, is an abstraction. Abstractions are the key to building modular, maintainable programs, so much so that a good question to ask of any new language is "what mechanisms for abstraction does it provide?”

One powerful abstraction mechanism available in Haskell is the higher-order function. In Haskell a function is a first-class citizen: it can freely be passed to other functions, returned as the result of a function, stored in a data structure, and so on. It turns out that the judicious use of higher-order functions can substantially improve the structure and modularity of many programs.

Built-in memory management

Very many sophisticated programs need to allocate dynamic memory from a heap. In C this is done with a call to *malloc*, followed by code to initialize the store just allocated. The programmer is responsible for returning the store to the free pool when it isn't needed any more, a notorious source of "dangling-pointer" errors. Furthermore, *malloc* is fairly expensive, so programmers often *malloc* a single large chunk of store, and then allocate "by hand" out of this.

Haskell relieves the programmer of this storage management burden. Store is allocated and initialized implicitly, and recovered automatically by the garbage collector. The technology of storage allocation and garbage collection is now well developed, and the costs are rather slight.

The Haskell language has evolved significantly since its birth. By the middle of 1997, there had been four iterations of the language design (the latest at that point being Haskell 1.4). At the 1997 Haskell Workshop in Amsterdam, it was decided that a stable variant of Haskell was needed and is called Haskell 98. Therefore, older versions are now obsolete. Haskell users are encouraged to use Haskell 98. There are also many extensions to Haskell 98 that have been widely implemented. These are not yet a formal part of the Haskell language.

Table:

Attributes	**Description**	**Rating**
Author/Year	Haskell B. Curry/1988	N/A
Syntax	Easy to Understand	8
Clarity	Clear to read	7
Libraries	Yes	8
Compiled/Interpreted	Compiled	9
Fully Functional	Yes	10
Object Oriented	Yes	5
I/O	Complicated	5
Exception Handling	Yes	8
Memory Management	Yes	10
Garbage Collection	Yes	10
Application		N/A

Bibliography

Simon Peyton Jones (editor). *Report on the Programming Language Haskell 98, A Non-strict Purely Functional Language*. Yale University, Department of Computer Science Tech Report YALEU/DCS/RR-1106, Feb 1999.

Hudak, Paul et al. *Report on the Programming Language Haskell* Version 1.1 August 1991.

<*http://www.haskell.org*>