CSc 372

Comparative Programming Languages

29: Haskell — Lazy Evaluation

Christian Collberg
collberg@cs.arizona.edu

Department of Computer Science
University of Arizona

Copyright © 2004 Christian Collberg
Lazy evaluation

Haskell evaluates expressions using a technique called lazy evaluation:

- No expression is evaluated until its value is needed.
- No shared expression is evaluated more than once; if the expression is ever evaluated then the result is shared between all those places in which it is used.

The first of these ideas is illustrated by the following function:

\[
\text{ignoreArgument } x = \text{"Didn’t evaluate x"}
\]
Lazy evaluation...

Since the result of the function ignoreArgument doesn’t depend on the value of its argument \( x \), that argument will not be evaluated:

\[
\text{? ignoreArgument (1/0)}\\
\text{I didn’t need to evaluate } x\\
\text{(1 reduction, 31 cells)}
\]

We can force strict evaluation when that is necessary:

\[
\text{? strict ignoreArgument (1/0)}\\
\text{\{primDivInt 1 0\}}\\
\text{(4 reductions, 29 cells)}
\]
The second basic idea behind lazy evaluation is that no shared expression should be evaluated more than once. For example, the following two expressions can be used to calculate $3 \times 3 \times 3 \times 3$:

```
> /usr/local/hugs98/bin/hugs +s
> square*square where square = 3*3
81
28 reductions, 39 cells)
> (3*3)*(3*3)
81
34 reductions, 45 cells)
```

Notice that the first expression requires fewer reduction than the second.
Lazy evaluation...

The sequences of reductions:

\[\text{square} \times \text{square \ where \ square} = 3 \times 3\]

-- calculate the value of square by
-- reducing \(3 \times 3\) ==> 9 and replace each
-- occurrence of square with this result

==> 9 \times 9
==> 81

\((3 \times 3) \times (3 \times 3)\) -- evaluate first \((3 \times 3)\)

==> 9 \times (3 \times 3) -- evaluate second \((3 \times 3)\)

==> 9 \times 9
==> 81

Lazy evaluation means that only the minimum amount of calculation is used to determine the result of an expression.
Consider the task of finding the smallest element of a list of integers.

? minimum [100,99..1]
1
(809 reductions, 1322 cells)
?

[100,99..1] denotes the list of integers from 1 to 100 arranged in decreasing order. Instead, we could first sort and then take the head of the result:

? :load List
? sort [100,99..1]
[1, 2, 3, 4, 5, 6, 7, 8, ..., 99, 100]
(10712 reductions, 21519 cells)
?
Lazy evaluation — Example...

However, thanks to lazy-evaluation, calculating just the first element of the sorted list actually requires less work in this particular case than the first solution using \texttt{minimum}:

\begin{verbatim}
? head (sort [100,99..1])
1
(713 reductions, 1227 cells)
?
\end{verbatim}
Infinite data structures

Lazy evaluation makes it possible for functions in Haskell to manipulate ‘infinite’ data structures.

The advantage of lazy evaluation is that it allows us to construct infinite objects piece by piece as necessary.

Consider the following function which can be used to produce infinite lists of integer values:

```haskell
countFrom n = n : countFrom (n+1)

? countFrom 1
[1, 2, 3, 4, 5, 6, 7, 8, ^C{Interrupted!}]
(53 reductions, 160 cells)
? 
```
Infinite data structures...

For practical applications, we are usually only interested in using a finite portion of an infinite data structure.

We can find the sum of the integers 1 to 10:

```
? sum (take 10 (countFrom 1))
55
(62 reductions, 119 cells)
?
```

- `take n xs` evaluates to a list containing the first \( n \) elements of the list \( xs \).
Infinite data structures enables us to describe an object without being tied to one particular application of that object.

The following definition for infinite list of powers of two \([1, 2, 4, 8, \ldots]\):

\[
powersOfTwo = 1 : \text{map double powersOfTwo}
\]

\[
\text{where double } n = 2 \ast n
\]

\(xs!!n\) evaluates to the \(n\)th element of the list \(xs\).

We can define a function to find the \(n\)th power of 2 for any given integer \(n\):

\[
twoToThe n = powersOfTwo !! n
\]
Acknowledgements

These slides were derived directly from the Gofer manual.

Functional programming environment, Version 2.20
© Copyright Mark P. Jones 1991.

A copy of the Gofer manual can be found in

/home/cs520/2003/gofer/docs/goferdoc.ps.