Lazy evaluation

Gofer 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:

```haskell
ignoreArgument x = "Didn’t evaluate x"
```

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

```haskell
? ignoreArgument (1/0)
I didn’t need to evaluate x
(1 reduction, 31 cells)
```

We can force strict evaluation when that is necessary:

```haskell
? strict ignoreArgument (1/0)
{primDivInt 1 0}
(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 * 3 * 3 * 3`:

```haskell
? square * square where square = 3 * 3
 81
(3 reductions, 9 cells)

? (3 * 3) * (3 * 3)
81
(4 reductions, 11 cells)
```

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

The sequences of reductions:

\[ \text{square * square where square} = 3 * 3 \]
\[ \quad \text{-- calculate the value of square by} \]
\[ \quad \text{-- reducing 3*3==}>9 \text{ and replace each} \]
\[ \quad \text{-- occurrence of square with this result} \]
\[ \quad \Rightarrow 9 * 9 \]
\[ \Rightarrow 81 \]

\[ (3 * 3) * (3 * 3) \quad \text{-- evaluate first (3*3)} \]
\[ \Rightarrow 9 * (3 * 3) \quad \text{-- evaluate second (3*3)} \]
\[ \Rightarrow 9 * 9 \]
\[ \Rightarrow 81 \]

Lazy evaluation means that only the minimum amount of calculation is used to determine the result of an expression.

Lazy evaluation — Example...

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:

? sort [100,99..1]
[1, 2, 3, 4, 5, 6, 7, 8, ..., 99, 100]
(10712 reductions, 21519 cells)
?

Infinite data structures

Lazy evaluation makes it possible for functions in Gofer 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:

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:

```haskell
? 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, ...]`:

```haskell
powersOfTwo = 1 : map double powersOfTwo
  where double n = 2 * 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`:

```haskell
twoToThe n = powersOfTwo !! n
```

User-defined Datatypes

Gofer allows the definition of new datatypes:

```haskell
data Datatype a1 ... an = constr1 | ... | constrm
```

where

1. `Datatype` is the name of a new type constructor of arity `n` ≥ 0,
2. `a1, ..., an` are distinct type variables representing the arguments of `DatatypeName` and
3. `constr1, ..., constrm` (m ≥ 1) describe the way in which elements of the new datatype are constructed.

Each `constr` can take one of two forms:

1. `Name type1 ... type_r`, where `Name` is a previously unused constructor function name (i.e. an identifier beginning with a capital letter). This declaration introduces `Name` as a new constructor function of type:

```latex
\text{type}_1 \rightarrow \ldots \rightarrow \text{type}_r \rightarrow \text{Datatype} \, a_1 \ldots a_n
```

2. `type1 \oplus type2`, where `\oplus` is a previously unused constructor function operator (i.e. an operator symbol beginning with a colon). This declaration introduces `\oplus` as a new constructor function of type:

```latex
\text{type}_1 \rightarrow \text{type}_2 \rightarrow \text{Datatype} \, a_1 \ldots a_n
```
User-defined Datatypes...

The following definition introduces a new type \( \text{Day} \) with elements \( \text{Sun}, \text{Mon}, \text{Tue}, \ldots \):

\[
\text{data Day} = \text{Sun} \mid \text{Mon} \mid \text{Tue} \mid \text{Wed} \mid \text{Thu} \mid \text{Fri} \mid \text{Sat}
\]

Simple functions manipulating elements of type \( \text{Day} \) can be defined using pattern matching:

- `what_shall_I_do Sun = "relax"`
- `what_shall_I_do Sat = "go shopping"`
- `what_shall_I_do _ = "go to work"`

User-defined Datatypes...

Another example uses a pair of constructors to provide a representation for temperatures which may be given using either of the centigrade or fahrenheit scales:

\[
\text{data Temp} = \text{Centigrade} \ 	ext{Float} \mid \text{Fahrenheit} \ 	ext{Float}
\]

- `freezing :: Temp -> Bool`
- `freezing (\text{Centigrade} \ \text{temp}) = \text{temp} <= 0.0`
- `freezing (\text{Fahrenheit} \ \text{temp}) = \text{temp} <= 32.0`

User-defined Datatypes...

Datatype definitions may also be recursive.

The following example defines a type representing binary trees with values of a particular type at their leaves:

\[
\text{data Tree a} = \text{Lf} \ a \mid \text{Tree} \ a \ :^\ : \ \text{Tree} \ a
\]

For example,

\[
(\text{Lf} \ 12 :^\ : (\text{Lf} \ 23 :^\ : \text{Lf} \ 13)) :^\ : \text{Lf} \ 10
\]

has type \( \text{Tree \ Int} \) and represents the binary tree:

```
          10
         /   \  \
       12    23
       /     /  \
      13    23
```

User-defined Datatypes...

Calculate the list of elements at the leaves of a tree traversing the branches of the tree from left to right.

\[
\text{leaves} :: \text{Tree \ a} \rightarrow [\text{a}]
\]

- `leaves (Lf 1) = [1]`
- `leaves (l:^\ :r) = leaves l ++ leaves r`

Using the binary tree above as an example:

? leaves ((Lf 12:^\ : (Lf 23:^\ : Lf 13)):^\ : Lf 10)
[12, 23, 13, 10]
(24 reductions, 73 cells)
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.