

Numerical methods

A widely used algorithm for computing \sqrt{N} is Newton's method, which computes a sequence of ever-improving approximations:

$$a_{i+1} = \frac{a_i + \frac{N}{a_i}}{2}; \text{ as } i \rightarrow \infty, a_i^2 \rightarrow N$$

In Haskell, the sequence of approximations can be represented by an infinite list.

A Haskell function which computes a better approximation of \sqrt{N} from a given approximation:

```
> next :: Float -> Float -> Float
> next n a = (a + n/a) / 2.0
```

Call the initial approximation `a0`; then the sequence of approximations is the list

```
> iterate (next n) a0
```

Now we need a function that traverses the list:

```
> within :: Float -> [Float] -> Float
> within eps (a:b:bs)
>   | abs(a-b) <= eps = b
>   | otherwise       = within eps (b:bs)
```

The parameter `eps` determines the closeness of the approximation.

Then

```
> sqrt eps n = within eps (iterate (next n) 1.0)
```

The traversal of the list causes elements to be computed, but only as far as needed.

Problem: A fixed value of `eps` will allow a relatively large error when `N` is very small, and may result in non-termination when `N` is very large.

If we expect `N` to cover a wide range, we prefer an alternative to `within`:

```
> relative :: Float -> [Float] -> Float
> relative eps (a:b:bs)
>   | abs(a/b-1.0) <= eps = b
>   | otherwise           = relative eps (b:bs)
```

and then

```
> rsqrt eps n
>   = relative eps (iterate (next n) 1.0)
```

Because of the floating-point division (`/`), `relative` is slower than `within`.

Note that only the list-traversal function is changed; the list of approximations

```
(iterate (next n) 1.0)
```

is reused without alteration.

Moreover, `within` and `relative` could be applied, without alteration, to other converging approximation sequences (cube roots, differentials, integrals, etc.).

Conclusion: Infinite lists play a role like that of loops, and functions applied to them serve as termination conditions. By enabling loops and their termination conditions to be defined *separately*, infinite lists provide a new way to modularize functional programs.

Memoization

Infinite lists can be used to speed up functions by "caching" their results for reuse, avoiding the cost of recomputing them.

Given: a function `f :: Int -> Int` defined for all arguments `x ∈ ℕ` (the natural numbers).

Define

```
> fMem :: Int -> Int
> fMem x = fList !! x
> fList = [ f n | n <- [0..] ]
```

The list `fList` is an infinite list of applications:

```
f 0 : f 1 : f 2 : f 3 : ...
```

Thanks to laziness,

- each list element `f n` will be evaluated only when —and if— its value is needed
- when an element is evaluated, its value replaces the function application

A famous example of redundant computation is the "naïve" definition of the Fibonacci function:

```
> fib :: Int -> Integer
> fib 0     = 0
> fib 1     = 1
> fib n | n > 1 = fib (n-1) + fib (n-2)
```

The double recursion makes this algorithm exponentially expensive.

To avoid the recomputations, we use a list to "memoize" the function:

```
> ffib :: Int -> Integer
> ffib 0     = 0
> ffib 1     = 1
> ffib n | n > 1 = fibList!!(n-1) + fibList!!(n-2)
> fibList = [ fib n | n <- [0..] ]
```

But computing each list element is still exponentially expensive. That can be remedied by a small change:

```
> fibList = [ ffib n | n <- [0..] ]
```

Each list element is computed from its two predecessors.

Laziness \Rightarrow The list grows only as long as needed.

Because the cost of `xs!!k` is in $\Theta(k)$, we've reduced the cost from exponential ($\Theta(\backslash n \rightarrow k^n)$) to quadratic ($\Theta(\backslash n \rightarrow n^2)$).

A further improvement, to linear cost ($\Theta(\backslash n \rightarrow n)$), results from eliminating most of the list indexing:

```
> vFib :: Int -> Integer
> vFib n = fibs!!n
> fibs = 0:1:[ a+b | (a,b) <- zip fibs (tail fibs) ]
```

... a nice example of recursive stream processing.

How can I/O be Referentially Transparent?

How can one define a Haskell program that reads from the standard input stream?

The Standard-ML approach:

```
inputChar :: Char
```

Each evaluation of `inputChar` reads the next keystroke from the input stream; the value received determines `inputChar`'s value.

Now consider the expression

```
inputChar == inputChar
```

and suppose that the next two values in the input stream are different.

- ☛ This would be a violation of referential transparency, which requires that every occurrence of an expression in a given scope have the same value.

Another problem arises when we consider

```
y == y where y = inputChar
```

How many characters are read from the input stream?

If two characters are read, then

- ☛ we have the same violation of referential transparency as noted above.

If only one character is read, then

- ☛ we have another violation of referential transparency: `y` and `inputChar` are defined to be equal, but replacing `y` by `inputChar` *changes* the value of `(y == y)`.

Haskell I/O

Rather than abandon referential transparency, Haskell rejects the idea of obtaining input values by expression evaluation.

Instead, Haskell provides a polymorphic *abstract data type* (ADT)

```
IO a
```

whose values represent *I/O actions*.

The operations applicable to this ADT's values are carefully designed so that no side-effects of performing I/O actions can be detected within a Haskell program.

The simplest kind of I/O action merely displays a single character on the worksheet, or obtains a single character from the keyboard. I/O actions can be combined in sequence to make more complex I/O actions.

IO itself is, like `[]`, a *type constructor*.

```
[a] lists containing elements of type a
```

```
IO a I/O actions delivering values of type a
```

A value of the type `IO a` is an action which—if it ever takes place—

- performs some I/O
- *delivers* a value of type `a`.

The prelude provides

- some elementary I/O actions
- several ways of combining actions into sequences of actions

Two elementary IO actions which read from the keyboard:

```
getChar :: IO Char
```

```
getLine :: IO String
```

These *deliver* a character and a string, respectively.

Output actions (i.e., I/O actions that write in the worksheet) don't deliver anything, but the type constructor `IO` requires a type as an argument.

For this purpose, Haskell provides the *unit type* `()`, whose only element is the *unit value* `()`. Thus

```
() :: ()
```

Output actions then have the type

```
IO ()
```

Two I/O actions which write in the worksheet are returned by these functions:

```
putChar :: Char -> IO ()
```

```
putStr :: String -> IO ()
```

A function which returns an action which writes a given line in the worksheet:

```
> putStrLn :: String -> IO ()
```

```
> putStrLn s = putStr (s ++ "\n")
```

A function which returns an action which writes a given value's `String` representation in the worksheet:

```
> print :: Show a => a -> IO ()
```

```
> print x = putStrLn (show x)
```

A function which returns an elementary I/O action which performs *no I/O* and *delivers* a given value:

```
> return :: a -> IO a
```

(`return` is useful in defining composite input actions).

Invoking I/O actions

Up to this point, we've tacitly assumed that a Haskell program's value is a `String` which is displayed as the program's output. That's not actually true.

Every Haskell program's value is actually an *I/O action* which displays the program's value.

An I/O action takes place when you type it at the Hugs prompt:

```
Prelude> print "Hello, World!"
"Hello, World!"
```

```
Prelude> :t print "Hello, World!"
print "Hello, World!" :: IO ()
```

If you type an expression of some non-IO type at the prompt, Hugs tacitly applies the function

```
\x -> putStr (show x)
```

to it.

This explains how `putStr` gets rid of unwanted punctuation in output `Strings`:

Queries/results	actual values
<pre>Prelude> "abc\ndef" "abc\ndef"</pre>	<code>abc</code> ↵ <code>def</code>
<pre>Prelude> putStr "abc\ndef" abc def</pre>	<code>abc</code> ↵ <code>def</code>
<pre>Prelude> show "abc\ndef" "\"abc\\ndef\""</pre>	<code>"abc\ndef"</code>

(Here '↵' denotes a new-line character.)