

Curried functions

By ordinary substitution,

$$(\lambda x \rightarrow (\lambda y \rightarrow 3*x + 2*y)) 10 \\ \rightsquigarrow \lambda y \rightarrow 3*10 + 2*y$$

Because

$$\lambda x \rightarrow (\lambda y \rightarrow 3*x + 2*y) = \lambda x y \rightarrow 3*x + 2*y$$

it follows that

$$(\lambda x y \rightarrow 3*x + 2*y) 10 \\ \rightsquigarrow \lambda y \rightarrow 3*10 + 2*y$$

That is,

Applying a function of two parameters to a single argument yields a function of one parameter.

Any n -parameter function can be applied to $k < n$ arguments:

$$(\lambda a b c \rightarrow a + b*c) \text{ arg} \\ \rightsquigarrow (\lambda b c \rightarrow \text{arg} + b*c)$$

The result is a function having $(n-k)$ parameters.

Functions (such as λ -expressions) that take their arguments one at a time are said to be *curried* functions (in honor of the mathematician H.B. Curry).

In Haskell, *all* functions are curried. Even infix operators are curried, allowing *operator sections*. For example,

$(2+)$	$= \lambda a \rightarrow 2+a$	Generalizing:	$(\otimes) x y = x \otimes y$
$(*y)$	$= \lambda x \rightarrow x*y$		
$(+)$	$= \lambda m n \rightarrow m + n$		

Evaluation policies for program-defined functions

The basic strategy:

Substitute each argument for occurrences of its parameter in the function body.

leaves open an important question:

Should arguments be evaluated before they are substituted?

If so, we have...

an **Innermost** evaluation policy:

0. Evaluate each argument (which may be another application, etc.) to obtain its value.
1. Substitute each argument's value for each occurrence of the corresponding parameter in (a copy of) the body.
2. Evaluate the resulting expression.

The resulting value is the value of the application.

Example:

$$2 + (\lambda x \rightarrow (x+3)/(5-x)) (7-3) \\ \rightsquigarrow 2 + (\lambda x \rightarrow (x+3)/(5-x)) 4 \\ \rightsquigarrow 2 + ((4+3)/(5-4)) \\ \rightsquigarrow 2 + (7/(5-4)) \\ \rightsquigarrow 2 + (7/1) \\ \rightsquigarrow 2 + 7/1 \\ \rightsquigarrow 9$$

Otherwise, we have...

an **Outermost** evaluation policy:

1. Substitute each argument —**unevaluated**— for each occurrence of the corresponding parameter in (a copy of) the body.
2. Evaluate the resulting expression, evaluating each subexpression *only* when (and if) its value is actually *needed*.

Example :

$$2 + (\lambda x \rightarrow (x+3)/(5-x)) (7-3) \\ \rightsquigarrow 2 + (((7-3)+3)/(5-(7-3))) \\ \rightsquigarrow 2 + ((4+3)/(5-(7-3))) \\ \rightsquigarrow 2 + (7/(5-(7-3))) \\ \rightsquigarrow 2 + (7/(5-4)) \\ \rightsquigarrow 2 + (7/1) \\ \rightsquigarrow 2 + 7 \\ \rightsquigarrow 9$$

It follows from the Church-Rosser Theorem) that

For any expression for which *innermost* and *outermost* both succeed, they get the *same result*.

Outermost is *safer* than innermost:

- it succeeds whenever innermost succeeds
- it succeeds on some expressions for which innermost fails

Example:

$$(\lambda x y \rightarrow \text{if } x \text{ then } 2 \text{ else } 3*y) (9>5) (7/0)$$

Innermost:

$$(\lambda x y \rightarrow \text{if } x \text{ then } 2 \text{ else } 3*y) (9>5) (7/0) \\ \rightsquigarrow (\lambda x y \rightarrow \text{if } x \text{ then } 2 \text{ else } 3*y) \text{ True } (7/0) \\ \rightsquigarrow (\lambda x y \rightarrow \text{if } x \text{ then } 2 \text{ else } 3*y) \text{ True } \perp \\ \rightsquigarrow \perp \quad (\text{i.e., "undefined"})$$

☞ Even an innermost evaluation *never* evaluates all three of **if**'s arguments.

Outermost:

$$(\lambda x y \rightarrow \text{if } x \text{ then } 2 \text{ else } 3*y) (9>5) (7/0) \\ \rightsquigarrow \text{if } 9>5 \text{ then } 2 \text{ else } 3 * (7/0) \\ \rightsquigarrow \text{if true then } 2 \text{ else } 3 * (7/0) \\ \rightsquigarrow 2$$

Outermost's safety is not a free lunch—

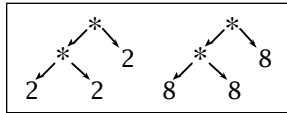
For some expressions, outermost takes *more work*.

Example: define

```
> cube = \x -> x * x * x
```

Innermost:

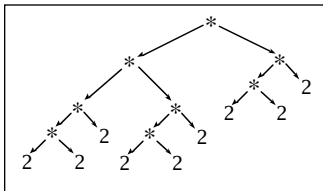
```
cube (cube 2)
=> cube (2 * 2 * 2)
=> cube (4 * 2)
=> cube 8
=> 8 * 8 * 8
=> 64 * 8
=> 512
```



number of multiplications = 4

Outermost:

```
cube (cube 2)
=> cube 2 * cube 2 * cube 2
=> 2 * 2 * 2 * cube 2 * cube 2
=> 4 * 2 * cube 2 * cube 2
=> 8 * cube 2 * cube 2
=> 8 * (2 * 2 * 2) * cube 2
=> 8 * (4 * 2) * cube 2
=> 8 * 8 * cube 2
=> 64 * cube 2
=> 64 * (2 * 2 * 2)
=> 64 * (4 * 2)
=> 64 * 8
=> 512
```



number of multiplications = 8

The reason for the difference:

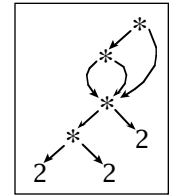
redundant evaluation of the argument (cube 2).

A “clever” variant of outermost evaluation avoids the repeated argument evaluation. It is called

Lazy evaluation (a.k.a. “call-by-need”).

Example of lazy evaluation:

```
cube (cube 2)
=> let x = cube 2 in x * x * x
=> let x = 2 * 2 * 2 in x * x * x
=> let x = 4 * 2 in x * x * x
=> let x = 8 in x * x * x
=> 8 * 8 * 8
=> 64 * 8
=> 512
```



4 multiplications

Using **let**-expressions to bind parameters to argument expressions enables all occurrences of a parameter to **share** the value of the corresponding argument.

Each argument is therefore evaluated *at most once*. If an argument’s value is not needed, it is never evaluated.

strategy	number of evaluations
innermost	exactly 1
outermost	0 .. ∞
lazy	0 or 1

So the *lazy* policy combines

- the *innermost* policy’s efficiency
- the *outermost* policy’s safety

Lazy Evaluation in Haskell

In Haskell the default evaluation policy is *lazy*.

This shows up in the way Haskell handles *sharing* and *procrastination*.

Sharing

Names are especially useful for expressions that are shared, i.e., used as subexpressions in several places.

```
> module HE2 where
> w = 16/0
> x = 6*8
> z = (x+y) * (x-y)
> y = 9-5
```

Naming repeated subexpressions makes them

- easier to read and write
- faster to evaluate

Laziness ⇒ When a name’s definition is evaluated for the first time, its value **replaces** the expression that defines it. Subsequent references to the name’s value get it at *no cost*.

Note the connection between Haskell’s *laziness* and its *referential transparency*:

- replacing any subexpression of *E* by any other expression of equal value has *no effect* on the value of *E*.

Without referential transparency, replacing expressions by their values could change the program’s meaning.

Procrastination

Haskell expressions are evaluated only when (and if) their values are needed.

Demonstration:

```
Hugs session for:
/usr/local/lib/hugs/lib/PreLude.hs
HE2.lhs
HE2> z
2288
HE2> w
Program error: {primDivDouble 16.0 0.0}
```

Because *z* does not depend on *w*, computing the value of *z* does not require the value of *w*.

The division by zero does not occur until the value of *w* is required.

In other words, evaluation is *demand-driven*: No computation happens until its result is needed.

Demand for values originates at the worksheet expression, and propagates to its subexpressions, their subexpressions, etc.

A script therefore must be read

- **not** as a sequence of statements
- but as a set of definitions.