

3: Functional paradigm (II)

Programming Languages, Technologies and Paradigms

Summary



Introduction to Functional Programming

PART I: Types in Functional Programming

1. Functional types. Algebraic types.
2. Predefined types.
3. Polymorphism: genericity, overloading and coercion. Inheritance in Haskell.

PART II: Models of computation in functional programming.

4. Operational model.

PART III: Advanced features

5. Anonymous functions and composition.
6. Iterators and compressors (foldl, foldr).

Operational model



- A **functional program** consists of:
 - A list of **equations defining functions** (possibly with additional equations defining types)
 - An **initial expression** (without free variables)
- The **execution** of a functional program consists of the evaluation of the **initial expression**
- The evaluation itself consists of a sequence of **reduction steps**

Operational model

- We use the notion of *substitution* to formalize the *parameter passing* as a matching from the expression to be evaluated against the (left-hand side of) equation $l=r$ which is used in the reduction step.
- A substitution σ is a mapping from variables into expressions such that $\sigma(x) \neq x$ holds for a finite set of variables.
- Substitutions are then represented by just giving the non-trivial bindings $\{x_1 \rightarrow t_1, \dots, x_n \rightarrow t_n\}$ with $x_i \neq t_i$.
Example: $\sigma = \{x \rightarrow 1, y \rightarrow 0\}$ is a substitution
- The *identity* or ‘empty’ substitution is denoted by ε

Operational model

- The application $\sigma(e)$ of a substitution σ to an expression e is called *instantiation*

Example 1:

$$\sigma = \{x \rightarrow 1, y \rightarrow 0\}$$

$$e = f(x, g(y))$$

$$\sigma(e) = f(1, g(0))$$

Example 2

$$\sigma = \{x \rightarrow s(y), y \rightarrow 0\}$$

$$e = f(x, y)$$

$$\sigma(e) = f(s(y), 0)$$

Reduction

- A **redex** is an instance $\sigma(l)$ of a left-hand side l of an equation $l = r$
(or $l \mid c = r$ for conditional equations)
- The expression e **reduces** to e' if:
 - It contains a redex $\sigma(l)$ of an equation $l \mid c = r$
 - The condition c holds (i.e., it reduces to True) after applying σ to it
 - e' is obtained as the replacement of $\sigma(l)$ by $\sigma(r)$ in e
- Expressions that cannot be further reduced are called **normal forms**

Reduction

Example:

sixtimes 1 \rightarrow double (triple 1)

Redex

Equation:

sixtimes x = double (triple x)

Substitution:

$\{x \rightarrow 1\}$

Reduction

Example:

Redex

sixtimes 1 \rightarrow double (triple 1)
 \rightarrow double (3*1)

Equation:

triple $y = 3 * y$

Substitution:

$\{y \rightarrow 1\}$

Reduction

Example:

sixtimes 1 → double (triple 1)
→ double (3*1)
→ double 3

Redex

Equation:

predefined: product

Reduction

Example:

sixtimes 1 \rightarrow double (triple 1)
 \rightarrow double (3*1)
 \rightarrow double 3
 \rightarrow 3+3

Redex

Equation:

double x = x+x

Substitution:

{x \rightarrow 3}

Reduction

Example:

sixtimes 1 → double (triple 1)
→ double (3*1)
→ double 3
→ 3+3
→ 6

Redex

Equation:

predefined: addition

Reduction

Example:

sixtimes 1 → double (triple 1)
 → double (3*1)
 → double 3
 → 3+3
 → **6**

Normal form

Summary:

Functional Program

List of equations:

$\text{add } 0 \ x = x$

$\text{add } (S \ x) \ y = S \ (\text{add } x \ y)$

Initial Expression:

$\text{add } (\text{add } 0 \ 0) \ 0$

It cannot contain
free variables

Summary:

Functional Program

List of equations:

$\text{add } 0 \ x = x$

$\text{add } (S \ x) \ y = S \ (\text{add } x \ y)$

Initial Expression:

$\text{add } (\text{add } 0 \ 0) \ 0$

It cannot contain
free variables

redex chosen by the
evaluation strategy

Summary:

Functional Program

List of equations:

$\text{add } 0 \ x = x$

$\text{add } (S \ x) \ y = S \ (\text{add } x \ y)$

Initial Expression:

$\text{add } (\text{add } 0 \ 0) \ 0$

$\text{add } 0 \ x$
 $\text{add } 0 \ 0$

Does the chosen redex match the left-hand side of an equation by means of some substitution?

It cannot contain free variables

redex chosen by the evaluation strategy

Summary:

Functional Program

List of equations:

$\text{add } 0 \ x = x$

$\text{add } (S \ x) \ y = S \ (\text{add } x \ y)$

Initial Expression:

$\text{add } (\text{add } 0 \ 0) \ 0$

It cannot contain
free variables

redex chosen by the
evaluation strategy

$\text{add } 0 \ x$
 $\text{add } 0 \ 0$

Does the chosen redex
match the left-hand side
of an equation by means
of some substitution?

substitution

$\sigma = \{x \rightarrow 0\}$

$\sigma(\text{add } 0 \ x) = \text{add } 0 \ 0$

Summary:

Functional Program

List of equations:

$$\text{add } 0 \ x = x$$

$$\text{add } (S \ x) \ y = S \ (\text{add } x \ y)$$

Initial Expression:

$$\text{add } (\text{add } 0 \ 0) \ 0$$

$$\left. \begin{array}{l} \text{add } 0 \ x \\ \text{add } 0 \ 0 \end{array} \right\}$$

Does the chosen redex match the left-hand side of an equation by means of some substitution?

substitution

$$\sigma = \{x \rightarrow 0\}$$

$$\sigma(\text{add } 0 \ x) = \text{add } 0 \ 0$$

$$\sigma(x) = 0$$

It cannot contain free variables

redex chosen by the evaluation strategy

reduction step

$$\text{add } (\text{add } 0 \ 0) \ 0 \rightarrow \text{add } 0 \ 0$$

Evaluation



- The **evaluation** of an expression proceeds by applying successive **reduction steps** until a **normal form** is reached

Evaluation



- The **evaluation** of an expression proceeds by applying successive **reduction steps** until a **normal form** is reached
- The final result may depend on the selected **reduction strategy**

Evaluation modes



- Given a function call:

$$f\ e_1 \cdots e_k$$

We can distinguish two essential evaluation modes:

- ▣ Eager evaluation
- ▣ Lazy evaluation

Evaluation modes

- **Eager** evaluation (*call-by-value*): first evaluate the arguments; then use an equation defining the function f

```
sixtimes 1 → double (triple 1)  
              → double (3*1)  
              → double 3  
              → 3+3  
              → 6
```

Evaluation modes

- Lazy evaluation (call-by-name): the arguments are evaluated only if this is necessary to apply some of the equations defining f

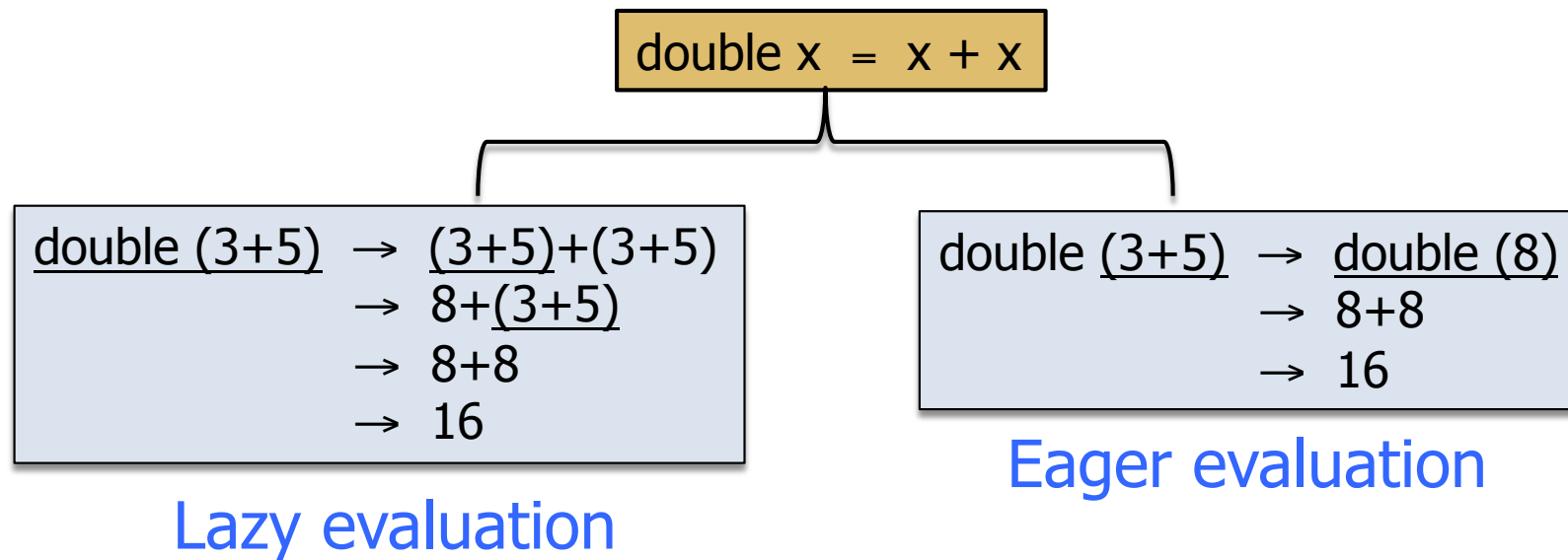
```
sixtimes 1    →  double (triple 1)  
                →  (triple 1)+(triple 1)  
                →  (3*1)+(triple 1)  
                →  3+(triple 1)  
                →  3+(3*1)  
                →  3+3  
                →  6
```

Evaluation Modes

- Which strategy is more efficient?

It depends on the program!

Sometimes eager evaluation is more efficient than lazy evaluation

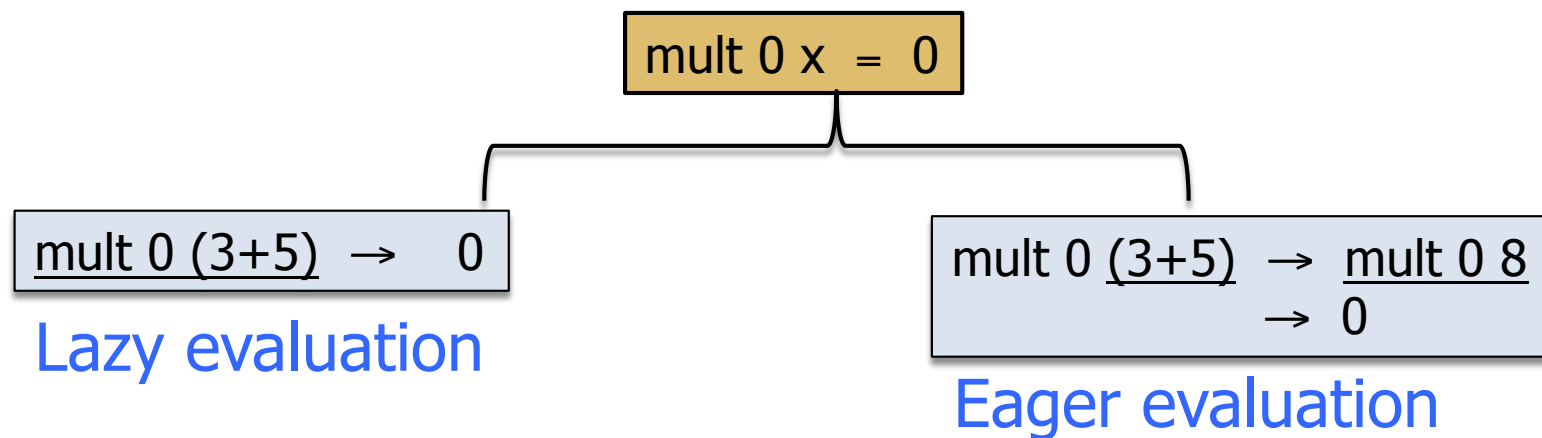


Evaluation Modes

- Which strategy is more efficient?

It depends on the program!

Sometimes lazy evaluation is more efficient than eager evaluation

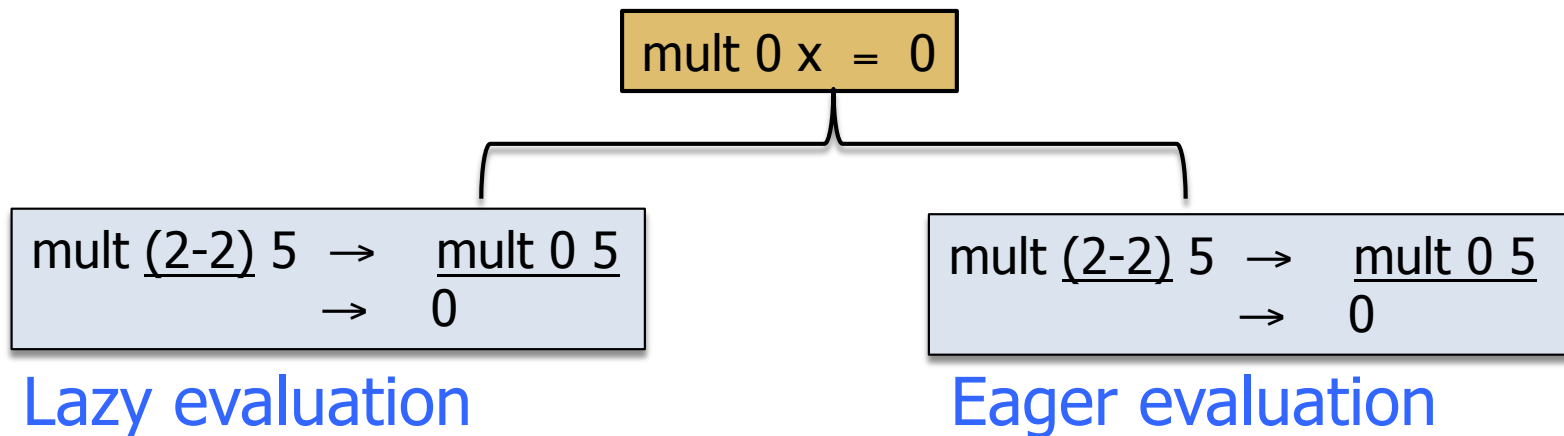


Evaluation Modes

- Which strategy is more efficient?

It depends on the program!

Sometimes lazy evaluation is as efficient as eager evaluation



The evaluation process



- ☐ Possible outcomes of the evaluation process:

The evaluation process

- The evaluation process can be:
 - ▣ **Successful:** it terminates and yields a value

```
sixtimes 1 →* 6
```

The evaluation process

- The evaluation process can be:
 - ▣ **Successful:** it terminates and yields a value
 - ▣ **Failed:** it terminates but no value is obtained

$\text{tail } (x:xs) = xs$

The expression

$\text{tail } []$

is a **normal form** but it is not a value.

The evaluation process

- The evaluation process can be:
 - ▣ **Successful:** it terminates and yields a value
 - ▣ **Failed:** it terminates but no value is obtained
 - ▣ **Incomplete:** it does not terminate

loop = loop

mult 0 x = 0

An incomplete evaluation sequence:

mult 0 loop → mult 0 loop → . . .

Lazy evaluation



- With lazy evaluation we can **avoid *nontermination***

loop = loop

mult 0 x = 0

mult 0 loop → 0

Lazy evaluation

- With lazy evaluation we can deal with **infinite data structures**

```
from n      = n:from (n+1)
```

```
sel 0 (x:xs) = x
```

```
sel n (x:xs) = sel (n-1) xs
```

The expression **from 0** denotes an infinite list containing all natural numbers

Lazy evaluation

sel 1 (from 0)

→ sel 1 (0:from (0+1))

→ sel (1-1) (from (0+1))

→ sel 0 (from (0+1))

→ sel 0 ((0+1):from (0+1+1))

→ 0+1

→ 1

With lazy evaluation we can evaluate expressions involving infinite values

Exercise

Indicate the reduction sequence of the expression:

inorder [2,6,1]

with both lazy and eager evaluation

$\text{insert } x \ [] = [x]$

$\text{insert } x \ (y:ys)$

$\quad | x \leq y = (x:y:ys)$

$\quad | \text{otherwise} = y : (\text{insert } x \ ys)$

$\text{inorder } [] = []$

$\text{inorder } (x:xs) = \text{insert } x \ (\text{inorder } xs)$