3: Functional paradigm (II)

Programming Languages, Technologies and Paradigms

Escuela Técnica Superior de Ingeniería **Informática** etsinf

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 $\{x_1 \rightarrow t_1, \ldots, x_n \rightarrow t_n\}$ with $x_i \neq t_i$.
  Example: $\sigma = \{x \rightarrow 1, y \rightarrow 0\}$ is a substitutión
- 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:

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

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

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

Example 2

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

$e = f(x,y)$

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

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 σ (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:

$\underbrace{\underline{\text{sixtimes } 1}}_{\boxed{\textbf{Redex}}} \rightarrow$ double (triple 1)

Equation:

sixtimes x = double (triple x)

Substitution:

$\{x \rightarrow 1\}$

Reduction

Example:

$$\underline{\text{sixtimes 1}} \rightarrow \text{double } (\overbrace{\underline{\text{triple 1}}}^{\text{Redex}})$$
$$\rightarrow \text{double } (3*1)$$

Equation:
triple y = 3 * y
Substitution:
{y→1}

Reduction

Example:

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

Redex

Equation:
predefined: product

Reduction

Example:

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

Redex

Equation:
double x = x+x
Substitution:
{x→3}

Reduction

Example:

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

Redex

Equation:
predefined: addition

Reduction

Example:

$$
\begin{aligned}
\underline{\text{sixtimes 1}} \quad &\rightarrow \quad \text{double } (\underline{\text{triple 1}}) \\
&\rightarrow \quad \text{double } (\underline{3*1}) \\
&\rightarrow \quad \underline{\text{double 3}} \\
&\rightarrow \quad \underline{3+3} \\
&\rightarrow \quad \boxed{\mathbf{6}}
\end{aligned}
$$

Normal form

Summary:

Functional Program

List of equations:

add 0 x = x

add (S x) y = S (add x y)

Initial Expression:

add (add 0 0) 0

It cannot contain free variables

Summary:

Functional Program

List of equations:

add 0 x = x

add (S x) y = S (add x y)

Initial Expression:

add (add 0 0) 0

It cannot contain free variables

redex chosen by the evaluation strategy

Summary:

Functional Program

List of equations:

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

add (S x) y = S (add x y)

Initial Expression:

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

add 0 x

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:

add 0 x = x

add (S x) y = S (add x y)

Initial Expression:

add (add 0 0) 0

It cannot contain free variables

redex chosen by the evaluation strategy

add 0 x

add 0 0

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

substitution

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

Summary:

Functional Program

List of equations:

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

add (S x) y = S (add x y)

Initial Expression:

add (add 0 0) 0

It cannot contain free variables

redex chosen by the evaluation strategy

add 0 x

add 0 0

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

substitution

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

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

reduction step

$\sigma(x)=0$

add (add 0 0) 0 → 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 succesive **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*

<u>sixtimes 1</u>	$\rightarrow$	<u>double (triple 1)</u>
	$\rightarrow$	(<u>triple 1</u>)+(triple 1)
	$\rightarrow$	(<u>3*1</u>)+(triple 1)
	$\rightarrow$	3+(<u>triple 1</u>)
	$\rightarrow$	3+(<u>3*1</u>)
	$\rightarrow$	<u>3+3</u>
	$\rightarrow$	6

Evaluation Modes

- Which strategy is more efficient?

It depends on the program!

<u>Sometimes **eager evaluation is more efficient than lazy evaluation**</u>

double x = x + x

Lazy evaluation:

double (3+5) → (3+5)+(3+5)
→ 8+(3+5)
→ 8+8
→ 16

Eager evaluation:

double (3+5) → double (8)
→ 8+8
→ 16

Evaluation Modes

- Which strategy is more efficient?

It depends on the program!

Sometimes **lazy evaluation is more efficient than eager evaluation**

mult 0 x = 0

mult 0 (3+5) → 0

Lazy evaluation

mult 0 (3+5) → mult 0 8
→ 0

Eager evaluation

Evaluation Modes

- Which strategy is more efficient?

It depends on the program!

Sometimes **lazy evaluation is as efficient as eager evaluation**

mult 0 x = 0

Lazy evaluation	Eager evaluation
mult (2-2) 5 → mult 0 5 → 0	mult (2-2) 5 → mult 0 5 → 0

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

```
tail (x:xs) = xs
```

The expression

```
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 <u>loop</u> $\rightarrow$ mult 0 <u>loop</u> $\rightarrow$ $\cdots$

Lazy evaluation

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

```
loop      = loop
mult 0 x = 0
```

<u>mult 0 loop</u> → 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

```
insert x [ ] = [x]
insert x (y:ys)
        |x<=y = (x:y:ys)
        |otherwise = y : (insert x ys)


inorder [ ] = [ ]
inorder (x:xs) = insert x (inorder xs)
```