G6021 Comparative Programming

Part 2 - Functional Programming

Functional Programming

General Concepts

- Functional programs consist entirely of functions.
- A function can be defined in terms of other functions (previously defined by the programmer, in the libraries, or language primitives).
- The focus is in *what* is to be computed, not *how* it should be computed.
- Modern functional languages are *strongly typed* (no run-time type errors) and have built-in memory management.
- Advantages: shorter programs, easier to understand, easier to design and maintain than imperative programs.
- Disadvantage: often slower than imperative programs.

Reading list

- See web page for books
- `www.haskell.org`

Examples of applications

- Banking
- Music composition
- Theorem provers and proof assistants
- Graphical interfaces
- Expert Systems
- Telephony (Ericsson)

Haskell

We use Haskell as the main example functional programming language in this module.

- A modern functional programming language
- Several interpreters and compilers available.
- More information about Haskell, including the language report, learning resources, user manuals, etc. can be found on the web site `www.haskell.org`

Syntax of functional programs

The notation is inspired by the mathematical definition of functions, using equations.

Example

We can compute the square of a number using a function *square* defined by the equation:


```
square x = x * x
```

Execution of programs

The role of the computer is to evaluate and display the results of the expressions that the programmer writes, using the available functions (like a sophisticated calculator).

Example

- When we enter an expression, such as

```
square 6
```

the computer will display the result: `36`

- As in mathematics, expressions may contain numbers, variables or names of functions.
- For instance `36`, `square 6`, `x * x` are expressions.

Evaluation

- To evaluate `square 6` the computer will use the definition of `square` and replace this expression by `6*6`. Then using the predefined `*` operation, it will find the result `36`.
- The process of evaluating an expression is a *simplification process*, also called *reduction* or *evaluation*.
- The goal is to obtain the *value* or *normal form* associated to the expression, by a series of reduction steps.

Examples

1. `square 6` $\rightarrow$ `6 * 6` $\rightarrow$ `36`
   `36` is the value or normal form of `square 6`
2. `((3 + 1) + (2 + 1))` $\rightarrow$ `((3 + 1) + 3)` $\rightarrow$ `(4 + 3)` $\rightarrow$ `7`
   `7` is the value denoted by the expression
   `((3 + 1) + (2 + 1))`

The *meaning* of an expression is its value.

Remark

- There may be several reduction sequences for an expression.

Example

The following is also a correct reduction sequence:

$$((3 + 1) + (2 + 1)) \rightarrow (4 + (2 + 1)) \rightarrow (4 + 3) \rightarrow 7$$

In both cases the value is the same.

Exercise

Can you draw a *reduction graph* of all possible reductions starting from

$$((3 + 1) + (2 + 1))$$

Properties

1. **Unicity of normal forms:**
   In (pure) functional languages the value of an expression is uniquely determined by its components, and is independent of the order of reduction.
   Advantage: readability of programs.
2. **Non-termination:**
   Not all reduction sequences lead to a value, some reduction sequences *do not terminate*.

Example

Let us define the constant function

```
fortytwo x = 42
```

and the function *infinity*

```
infinity = infinity + 1
```


- The evaluation of `infinity` never reaches a normal form.
- For the expression

  ```
  fortytwo infinity
  ```

  some reduction sequences do not terminate but those which terminate give the value `42` (unicity of normal forms).

Exercise

Can you draw a *reduction graph* of all possible reductions starting from `fortytwo infinity` ?

Strategies

- Although the normal form is unique, the order of reductions is important.
- The *strategy of evaluation* defines the reduction sequence that the language implements.
- Most popular strategies:
  1. *Call-by-name (Normal order):* reduce first the application using the definition of the function, and then the argument
  2. *Call-by-value (Applicative order):* evaluate first the argument and then the application using the definition of the function

Strategies, continued

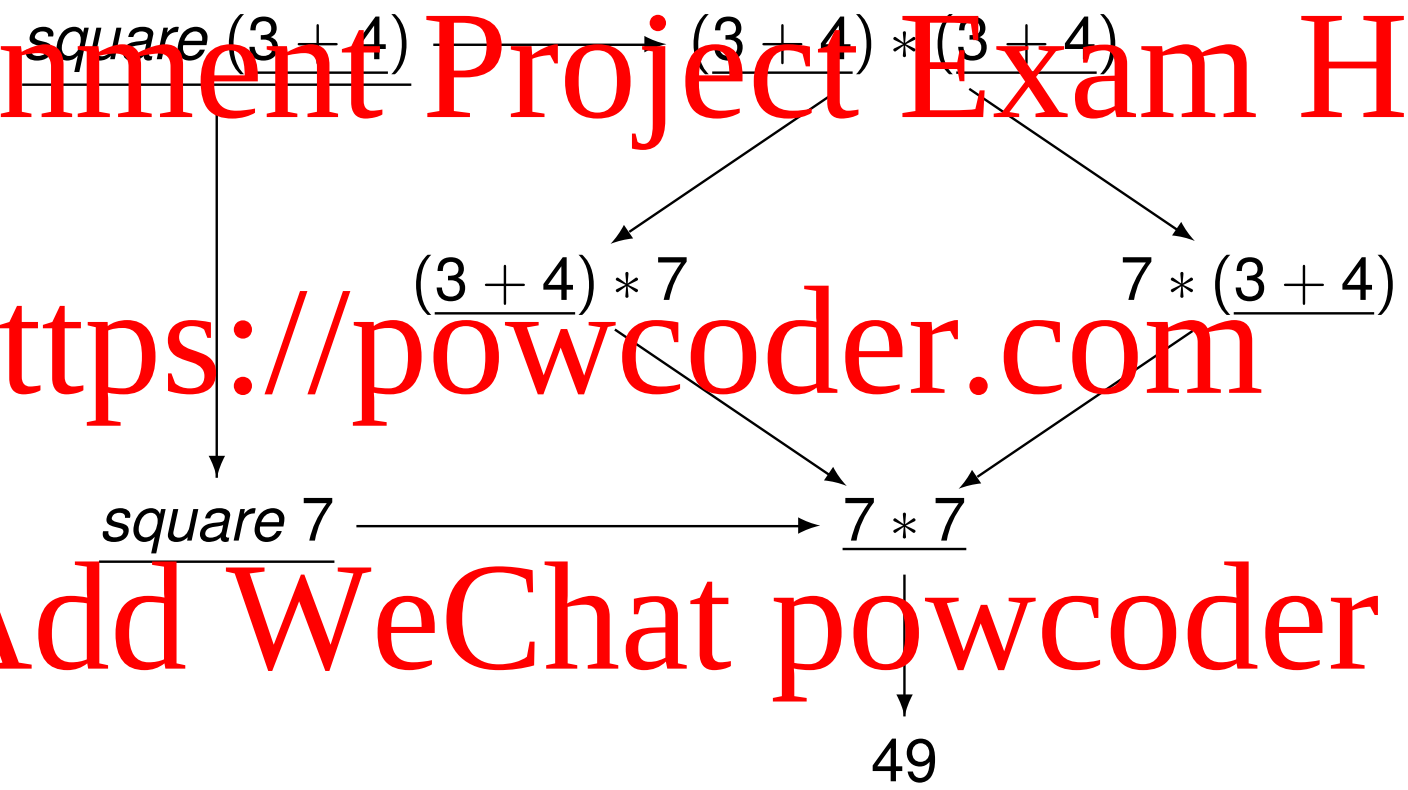
Remarks

- Different strategies require different number of reduction steps (*efficiency*).
- Call-by-name always finds the value, if there is one.
- Call-by-value is in general more efficient, but may fail to find a value.
- Haskell uses a strategy called *lazy evaluation*, which guarantees that if an expression has a normal form, the evaluator will find it.

lazy evaluation = call-by-name + sharing

Example: Strategies

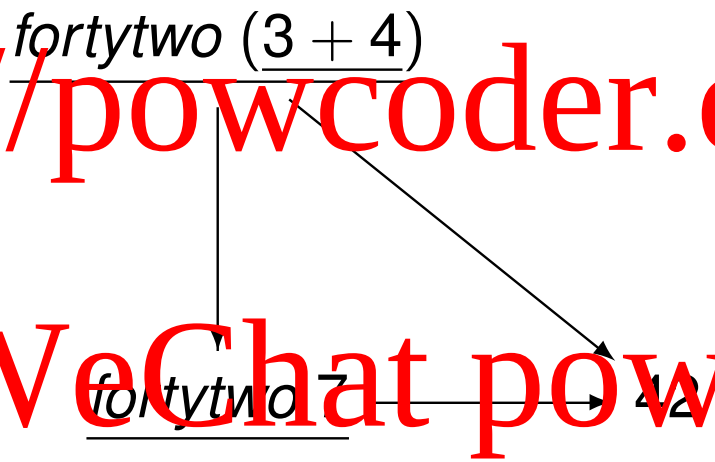
An example reduction graph for the expression: `square (3+4)`



Note: we underline each reducible expression (redex), all reductions lead to the same answer, and some reductions are longer than others.

Example: Strategies

- The previous example had the shortest path for call-by-value.
- But let's look at another example:



In this case, call-by-value gives the longest reduction path...

Functional Values

- Functions are also values, even though we cannot display them or print them.
- A function is a mapping that associates to each element of a given type A, called the *domain* of the function, an element of type B, called the *codomain*.

$$f : A \to B$$

- If a function f of type $A \to B$ is applied to an argument x of type A, it gives a result $(f\ x)$ of type B.
- In Haskell the notation to associate a type to a function is:

$$\begin{array}{rcl} square & :: & Integer \to Integer \\ fortytwo & :: & Integer \to Integer \end{array}$$

Notation

- Application is denoted by juxtaposition: $(f\ x)$
  Example: `(square 3)`
- To avoid writing too many brackets there are some conventions:
  - We will not write the outermost brackets:
    `square 3` instead of `(square 3)`
  - Application has precedence over other operations:
    `square 3 + 1` means `(square 3) + 1`
  - Application associates to the left:
    `square square 3` means `(square square) 3`

Syntax of Function Definitions

Functions are defined in terms of equations.

Examples

```
square x = x * x
min x y = if x <= y then x else y
```


We can also use *conditional equations* (aka *guarded equations*).

Examples


```
min x y
    | x <= y = x
    | x >  y = y
sign x
    | x <  0 = -1
    | x == 0 =  0
    | x >  0 =  1
```


The latter is equivalent to, but clearer than:

```
if x < 0 then -1 else if x == 0 then 0 else 1
```

Recursive Definitions

In the definition of a function `f` we can use the function `f`:

```
fact ::  Integer → Integer
fact n = if n==0 then 1 else n*(fact(n-1))
```

Recursive definitions are evaluated by simplification, as any other expression.

```
fact 0 →
if 0 == 0 then 1 else 0 * (fact (0 - 1)) →
if True then 1 else 0 * (fact (0 - 1)) → 1
```

Note the operational semantics of the conditional:

1. Evaluate first the condition.
2. If the result is True then evaluate only the expression in the left branch (`then`).
3. If the result is False then evaluate only the expression in the right branch (`else`).

Therefore the only reduction sequence for `fact 0` is the one shown above, and this program is terminating.

Recursive Definitions, continued

- However, if we start with

```
fact (-1)
```

the reduction sequence does not terminate.

- To avoid this problem we should write:

```
fact :: Integer → Integer
fact n
    | n > 0   = n * (fact (n - 1))
    | n == 0  = 1
    | n < 0   = error "negative argument"
```

`error` is a predefined function that takes a string as argument. When evaluated it causes immediate termination of the evaluator and displays the string.

Local Definitions

- As in mathematics, we can write:

```
f x = a + 1 where a = x/2
```

- or equivalently,

```
f x = let a = x/2 in a + 1
```


The words **let** and **where** are used to introduce local definitions, valid just on the right-hand side of the equation that we are writing.

We can write several local definitions:

```
f x = square (successor x) where
      square z = z * z ; successor x = x + 1
```

Arithmetic Functions

Arithmetic operations are also functions (primitives), used in infix notation: e.g. `3 + 4`

In Haskell we can use them in prefix notation if we enclose them in brackets: e.g `(+) 3 4`

`(+)` denotes the Curryfied version of `+`.

- `+ :: (Integer, Integer) → Integer`
- `(+) :: Integer → Integer → Integer`

Examples

- `3 - 1 - 2` should be read as `(3 - 1) - 2`
- `(+)1` is the successor function
- `(*)2` is the function that doubles its argument
- Application has priority:
  `square 1 + 4 * 2` should be read as
  `(square 1) + (4 * 2)`

Functional Composition

- Functions are the building blocks of functional languages.
- One way of combining functions is *composition*.
- Composition is itself a function (predefineds).

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(f . g) x = f (g x)
```


- We can only compose functions whose types match.

Example

```
square :: Integer -> Integer
quad = square . square
```

Types: General Concepts

Values are divided in classes, called *types*.
Each type is associated with a set of operations.

Predefined Types

- Basic data types: Booleans, Characters, Numbers.
  In Haskell: `Bool, Char, Int, Integer, Float ...`
- Structured types: Tuples, Strings, Lists.
  In Haskell `[Integer]` is the type of *lists* of integers.
- Function types: Integer $\rightarrow$ Integer, Integer $\rightarrow$ Float
  Example: `(Integer -> Integer) -> Integer`
  This is the type of a function that takes a function on integers as input and returns an integer.
  Convention: arrows associate to the right!

User-defined types

- The programmer can also define new types.
- Every valid expression must have a type. Expressions that cannot be typed are considered erroneous and are rejected by the compiler without evaluation (static typing).

Advantage of Statically Typed Languages

- Types help detecting errors at an early stage.
- Types help in the design of software since they are a simple form of specification.

A program that passes the type controls is not guaranteed to be correct, but it is free of type errors at runtime.

Polymorphism

Type systems can be

- *monomorphic:* every expression has at most one type, or
- *polymorphic:* some expressions have more than one type.

Example
Functional composition:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```


where `a`, `b`, `c` are *type variables*.

Type variables can be instantiated to different types in different contexts.
Therefore (.) is a polymorphic function.

Examples

1. `quad = square . square`

   `square :: Int → Int`

   Therefore

   `quad :: Integer → Int`

   using:

   `(.) :: (Int → Int) → (Int → Int) → (Int → Int)`

But we can also define `sqrt :: Integer → Float` and compose `sqrt` `square` using:

`(.) :: (Int → Float) → (Int → Int) → (Int → Float)`

Examples, continued

1. The `error` function is also polymorphic

```
error ::  String → a
fact ::  Integer → Integer
fact n
   n > 0  = n * (fact (n - 1))
   | n == 0  = 1
   | n < 0   = error "negative argument"
```

Here the function `error` is used with type

```
String → Integer
```

Polymorphism and Overloading

- Formally, the language of polymorphic types is defined as a set of *terms* built out of *type-variables* (`a`, `b`, `c`), and *type constructors* which are either atomic (Integer, Float, ...) or take arguments (e.g. $T_1 \rightarrow T_2$, [T]).

- A polymorphic type represents the set of its *instances*, obtained by substituting type variables by types.
- *Overloading* is a related notion (also called *ad-hoc polymorphism*) where several functions with different types share the same name.

Example (Overloading)

Arithmetic operations (such as addition) can be used both with integers or reals. But a polymorphic type such as `+ :: a → a → a` is too general: it allows addition to be used with characters.

There are several solutions to this problem:

1. Use different symbols for addition on integers and on reals.
   E.g. `+` and `+.` in Caml
2. Enrich the language of types:
   E.g. `+ :: (Integer → Integer → Integer) ∧ (Float → Float → Float)`
3. Define a notion of *type class*.
   E.g. in Haskell: `(+) :: Num a ⇒ a → a → a`
   (+) has type `a → a → a` where `a` is in the class `Num`.

Type Inference

Most modern functional languages do not require that the programmer provides the type for the expressions used. The compiler is able to *infer* a type, if one exists.

Intuitively:

- The expression is decomposed into smaller sub-expressions, and when a basic atomic expression is found, the information available is used (if it is a predefined constant) or otherwise it is assigned the most general type possible.
- The way that the different components are put together to form the expression indicates the constraints that the type variables must satisfy.

The type of an expression can be deduced from its components only.

Type Inference, continued

Example

With the definition:

```
square x = x * x
```

the expression[1]

```
square square 3
```


is rejected since

```
square :: Integer → Integer
```

and therefore its argument cannot be a function.

[1] This expression is equivalent to `(square square) 3` because application associates to the left.

Disjoint Sums

We can define several constructors in a type.

Example:

```
data Nat = Zero | Succ Nat
```

`Zero` and `Succ` are *constructors*. In this case they are not polymorphic, but we could define a type with polymorphic constructors.

Example:

```
data Seq a = Empty | Cons a (Seq a)
```


The constructors are `Empty` and `Cons` (polymorphic).

Constructors are used to build terms. What distinguishes a constructor from a function is that:

- There is no definition associated to a constructor.
- Constructors can be used in patterns.

Example:

```
isempty Empty = True
isempty (Cons x y) = False
```

Built-in Data Types

- Integer
- Double
- Bool: `True, False`
- Char: `'a', '2', ...`

You can find the type of an expression using `:type` (or just `:t`), for example:

```
Prelude> :t True
True :: Bool
Prelude> :t 5
5 :: (Num t) => t
Prelude> 5
5
Prelude> :t it
it :: Integer
Prelude> :t 'c'
'c' :: Char
```

Constructing data types: Pairs `(x,y)`

- Components do not need to have same type: `(2,True)`
- Built-in functions: `fst`, `snd` project the first and second component respectively.

```
Prelude> fst ('a',True)
'a'
Prelude> snd ('a',True)
True
```

- *n*-tuples: `(2,3,5)`, `(True,'c',42,(3,4))`, etc.
  Note that you can build also *n*-tuples from pairs: e.g., `((True,'c'),42)`

Exercises. Write functions to:

- extract `'c'` from `((True,'c'),42)` using only `fst` and `snd`.
- extract `'c'` from `(True,'c',42,(3,4))`

Lists

- A collection of things of the same type:
  Examples: `[1,2,3,7,9]`, `[True,False,True]`,
- Strings in Haskell are just: `[Char]`
- Built-in constructors and functions:
  `:`, `[]`, `head`, `tail`, `null`, `length`, ...

```
(:) :: a -> [a] -> [a]
length :: [a] -> Int
head :: [a] -> a
tail :: [a] -> [a]
null :: [a] -> Bool
```

- We can generate lists in a number of different ways:

```
1:2:3:4:[]
[1,2,3,4]
[1..4]
```

Lists: generation

- Cons and append. Try these:

```
6:[7,8]
[1,2,3] ++ [4,5]
'h':"ello"
"h" ++ "ello"
```

  What is the type of `(++)` ?

- Try these:

```
[1..10]
[10..1]
[5..11]
[3..]
head [3..]
```

- List comprehension. Try these:

```
[x*x | x <- [1,2,3]]
[x*x | x <- [1..10], even x]
[(x,y) | x<-[1..10], y<-[1..5]]
```

Lists: writing functions

We can write the built-in functions ourselves:

```
len [] = 0
len (h:t) = 1+len t
hd [] = error "list is empty"
hd (h:t) = h
tl [] = error "list is empty"
tl (h:t) = t
app [] x = x
app (h:t) x = h:(app t x)
```


- Here we are using *pattern matching*. Does the order of functions matter? What if we miss some cases?
- Exercise: can you think of a way of writing these functions without pattern matching?

Worked example

Write a function to sum all the elements of a list.
Example: `sum [1..10] = 55`
First try:


```
sum x = if null x then 0 else head x+sum (tail x)
```

Using guarded equations:


```
sum x
  | null x = 0
  | otherwise = head x + sum (tail x)
```


Using pattern matching:

```
sum [] = 0
sum (h:t) = h+sum t
```

Which way is better? Are there other ways?

Algebraic data types

- We can define our own data types:

```
data Bool = True | False
data Suit = Club | Diamond | Heart | Spade
data List a = nil | Cons a (List a)
```

- And we can write functions over these types using pattern matching:

```
not True = False
not False = True

head Nil = error "list is empty"
head (Cons x y) = x
```

Algebraic data types: Trees


```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

toList (Leaf a) = [a]
toList (Branch l r) = (toList l) ++ (toList r)
```


Example:

```
toList (Branch (Leaf 1) (Leaf 2))
[1,2]
```


Exercise: List to Tree? Insert in order?

Summary

- We have given a summary of many aspects of functional programming, and the syntax of Haskell through examples.
- In the following lectures we will look at many of the topics in these notes again in more detail, and discuss foundations, implementations and applications of functional programming.
- Try out examples, and also exercises, by experimenting with Haskell in the labs.
- Test functions, find out about built-in functions, and know how to find out the type of these functions
- Next: foundations of functional programming