Functional Programming
WS 2007/08

Christian Sternagel[1] (VO + PS)
Friedrich Neurauter[2] (PS)
Harald Zankl[3] (PS)

Computational Logic
Institute of Computer Science
University of Innsbruck

5 October 2007

[1] christian.sternagel@uibk.ac.at
[2] friedrich.neurauter@uibk.ac.at
[3] harald.zankl@uibk.ac.at

Overview

Overview

Lecture

- LV-Nr. 703017
- VO 2
- http://cl-informatik.uibk.ac.at/teaching/ws07/fp/
- lecture notes are available from the .uibk.ac.at network
- office hours: TBA
- evaluation: written exam

Exercises

- LV-Nr. 703018
- PS 1
- two groups:

group 1	Christian	Friday 8:15–9:00	in HS 10
group 2	Harald	Friday 9:15–10:00	in HS 10

- office hours:

Christian	TBA
Harald	TBA

- online registration required before 12 am on October 12
- evaluation: 2 tests + weekly exercises + optional programming project
- exercises are starting on October 12

Schedule

week 1	October 5	week 8	December 7
week 2	October 12	week 9	December 14
week 3	October 19		
week 4	November 9	week 10	January 11
week 5	November 16	week 11	January 18
week 6	November 23	1st exam	January 25
week 7	November 30		

Overview

Parts

part I: Practice

lists, strings,
trees, sets,
combinator parsing,
. . .

part II: Theory

λ-calculus, induction,
type checking,
type inference,
. . .

interwoven

Overview

Some Mantras

- keep referential transparency
- do not introduce side effects
- do not depend on global state
- use functions as values
- use recursion

But what do they mean?

Examples

Mathematics

- if $a = x + x$
- and $b = x + x$
- then $a = b$

replacing equals by equals

Example (Java)

```
public class Example1 {
 public static int count = 0;
 public static int inc() { return ++count; }

 public static void main(String[] args) {
  int a = inc() + inc();
  int b = inc() + inc();
  System.out.println("a = " + a);
  System.out.println("b = " + b);
 }
}
```

- no referential transparency
- side effects
- depends on global state

Examples (cont'd)

Goal

- arbitrary function $f : \mathbb{N} \to \mathbb{N}$
- sequence $s = 1, 2, 3$
- $\text{map}(f, s) = f(1), f(2), f(3)$
- e.g., $f(x) = x + 2$
  - result $3, 4, 5$
- e.g., $f(x) = 1$
  - result $1, 1, 1$

Example (Java)

```
public class Example2 {
 interface Function { public int call(int i); }
 public static int[] map(Function f, int[] seq) {
  for (int i = 0; i < seq.length; i++) {
   seq[i] = f.call(seq[i]);
  }
  return seq;
 }
 public static void main(String[] args) {
  int[] res = map(new Function(){
    public int call(int i) { return i + 2; }
   }, new int[]{1, 2, 3});
  for (int s : res) { System.out.println(s); }
 }
}
```

- pass functions via detour of classes

Examples (cont'd)

Sum of first n positive naturals

$$\mathrm{sum}(n) = \sum_{i=1}^{n} i$$

Example (Recursive)

```
public class Example4 {
 public static int sum(int n) {
  return (n < 1) ? 0 : n + sum(n - 1);
 }

 public static void main(String[] args) {
  int n = new Integer(args[0]);
  System.out.println(sum(n));
 }
}
```

Example (Java)

```
public class Example3 {
 public static int sum(int n) {
  int res = 0;
  for (int i = 1; i <= n; i++) { res += i; }
  return res;
 }

 public static void main(String[] args) {
  int n = new Integer(args[0]);
  System.out.println(sum(n));
 }
}
```

- depends on state (res)

Examples (cont'd)

Example (Solutions in OCaml)

- map a function over a list

```
let rec map(f, ls) = match ls with
  | [] -> []
  | x :: xs -> f(x) :: map(f, xs)
;;
map((fun x -> x + 2), [1; 2; 3]);;
map((fun x -> 1), [1; 2; 4]);;
```

- sum of first *n* positive naturals

```
let rec sum(n) = if n < 1 then 0 else n + sum(n - 1);;
```

Overview

Basic Types

- bool (e.g., **true**, **false**)
- char (e.g., 'a', 'b', 'c', ..., 'A', 'B', 'C', ..., '0', '1', '2', ...)
- float (e.g., 0., 1e−3, 3.1415, ...)
- int (e.g., ..., ~−2, ~−1, 0, 1, 2, ...)
- string (e.g., "Hello, world!\n")
- unit (e.g., ())

Basic Operations

Comparison

- '=' equality test
- '<>' inequality test
- '<' smaller than
- '>' greater than
- '<=' smaller than or equal
- '>=' greater than or equal
- 'compare' comparison
- 'min' minimum of 2 values
- 'max' maximum of 2 values

Example

```
# 'c' <> 'h';;
- : bool = true
# compare "Letter A" "Letter A";;
- : int = 0
# compare "Letter A" "Letter B";;
- : int = -1
# compare "Letter B" "Letter A";;
- : int = 1
# max 1 2;;
- : int = 2
# min 1 2;;
- : int = 1
```

Basic Operations (cont'd)

Booleans

- '&&' logical and
- '||' logical or
- 'not' logical not

Note

A && B (A || B): if A is **false** (**true**) then B is not evaluated

Basic Operations (cont'd)

Integers

- '~−' unary negation
- 'succ' successor function ($x \mapsto x + 1$)
- 'pred' predecessor function ($x \mapsto x - 1$)
- '+' addition
- '−' subtraction
- '∗' multiplication
- '/' division
- '**mod**' remainder of division
- 'abs' absolute value
- 'max_int' greatest representable integer
- 'min_int' smallest representable integer

Basic Operations (cont'd)

Floating Point Numbers

- '~−.' unary negation
- '+.' addition
- '−.' subtraction
- '∗.' multiplication
- '/.' division
- '∗∗' exponentiation
- 'sqrt' square root
- 'truncate' drop decimal places
- ...

Basic Operations (cont'd)

Strings

- '^' string concatenation

Example

```
# "Hello" ^ ", world!";;
- : string = "Hello, world!"
```

Types

- basic types
- type variables ('a, 'b, 'c, . . .)
- tuple types (int $*$ float, 'a $*$ 'a, . . .)
- function types (int −> int, bool −> bool −> bool, . . .)
- user-defined types

User-Defined Types

Type Abbreviations

- new name for existing type
- **type** coord = int ∗ int

Algebraic Datatypes

- **type** direction = North | East | South | West
- **type** number = Int **of** int | Float **of** float
- **type** 'a mylist = Empty | List **of** 'a ∗ 'a mylist

Values (Instances of Types)

- tuples ((1, 2) : int * int)
- anonymous functions (**fun** x -> x + 1 : int -> int)
- functions (**let** succ x = x + 1)
- variants (instances of algebraic datatypes;
  List (1, Empty) : int mylist)

Recursive Functions

- functions calling themselves
- recall
  ```
  let rec sum n = if n < 1 then 0 else n + sum (n - 1)
  ```

Pattern Matching

- recall

```
let rec map(f, ls) = match ls with
  | [] -> []
  | x :: xs -> f(x) :: map(f, xs)
```

- pattern

$$p ::= x \mid c \mid C\ (p, \ldots, p) \mid p \textbf{ as } x \mid (p) \mid p \mid p$$

Currying

- function

```
let rec map(f, ls) = match ls with
  | [] -> []
  | x :: xs -> f(x) :: map(f, xs)
```

  has type ('a -> 'b) * 'a list -> 'b list

- compare to

```
let rec map f ls = match ls with
  | [] -> []
  | x :: xs -> f x :: map f xs
```

  of type ('a -> 'b) -> 'a list -> 'b list

Currying (cont'd)

- every function has just one argument
- how to define functions with more arguments (e.g., $x + y$)?
- either use tuples (`let add (x, y) = x + y`)
- or curried (`let add x = (fun y -> x + y)`)
- curried form is OCaml standard (e.g., `let f x y z =` b equals `let f x = (fun y -> (fun z ->` b`))`)