

CS131: Programming Languages

Lun Liu

04.28.17

Midterm

- Wednesday, in class
- Sample Midterm available on CCLE

Today

- This week:
 - Type checking; Exceptions
 - Parametric Polymorphism
- Quick review
 - Very brief! Check out lecture notes and discussion notes for details and examples
- QA
- (Sample midterm)

Type Safety

- Types
 - Enforce abstractions
 - Identify errors
 - Enforce requirements on user-defined types (data structures)
- Static vs dynamic type checking
- Strongly typed vs weakly typed

Static vs Dynamic Type Checking

- Static vs dynamic type checking
 - When to perform type checking
 - Static -> compile time
 - Dynamic -> execution time
- Static: C, C++, Java, OCaml...
- Dynamic: Python, JavaScript, Ruby...

Strongly Typed vs Weakly Typed

- Strongly typed vs weakly typed
 - Can or cannot work around the type system
 - Strongly typed: A value has a type and that type cannot change
 - Weakly typed: Possible to change the type of variable
- Strongly typed: OCaml, Python...
- Weakly typed: C, C++...

Exceptions

```
exception NotFound of string
```

```
let rec lookupE k l = match l with  
    | [] -> raise (NotFound "key not in the  
list")  
    | (key, v) :: t ->  
        if key=k then v else lookupE k t
```

Exceptions

- **exception** keyword to create new exceptions
- **exception** <NAME> (**of** <PARAM>)
- Pros (Compared to `options`)
 - clear indication of an error
 - clean separation of error handling
 - allow callers to easily pass through the error if they can't handle it

Exceptions

- Exception catching syntax

try

dirty_code

with

Exception_1 -> Handle_1

| Exception_2 -> Handle_2

...

| Exception_N -> Handle_N

Exceptions (for fun 😊)

- Higher order library functions exit earlier on raising an exception within the argument function
 - Could be used to terminate `fold_left`, `map` etc earlier

Product of a List

- Use Recursion

```
let rec product l = match l with  
    | [] -> 1  
    | h::t -> h * (product t)
```

- Use fold

```
let product l = List.fold_left (fun acc e -> acc  
* e) 1 l
```

Product of a List

- More efficient way: early exit on a 0
- `Exit` is a built-in Exception

```
let product l =  
    try  
        List.fold_left (fun a x -> if x = 0 then  
            raise Exit else a * x) 1 l  
    with  
        Exit -> 0;;
```

Parametric Polymorphism

- “Generics”
- Can pass different types of arguments to one function
 - Type variable (e.g. ``a`) gets instantiated on each call to the function
- Contrast with overloading
 - Many functions with the same name

What have we learned so far?

- OCaml
 - Variables, functions, recursions, lists, tuples, pattern matching, user-defined types, options, exceptions
 - First-class functions
 - Higher-order functions
 - Currying
 - `map`, `fold_left`, `fold_right`...
- Scoping: static vs dynamic
- Type safety: static vs dynamic, strongly typed vs weakly typed
- Parametric polymorphism (vs static overloading)

Q&A

Backup

Sample Midterm

Problem 2: Type Checking

- Static Type Checking
 - Early error detections
 - Guarantees for all possible executions
 - Documentation
 - Efficiency
 - Enforce constraints of user-defined types
- Dynamic Type Checking
 - More flexible
 - Quick development
 - Relative concise code

Problem 3: Strongly typed

- Which of these things are implied by the fact that OCaml is strongly typed?
 - i. Each of a program's expressions is given a type at compile time.
→ static type checking
 - ii. A program cannot access memory that it did not allocate.
→ strongly typed: variable type cannot change
 - iii. A program can make use of parametric polymorphism.
→ not necessarily
 - iv. A program cannot terminate with an exception at run time.
→ well OCaml can...

Problem 3: Parametric Polymorphism

- Which of these things are implied by the fact that OCaml supports parametric polymorphism?
 - i. A single function can be passed arguments of different types.
→ this is the definition
 - ii. A function call cannot be completely typechecked until run time.
→ OCaml uses static type checking
 - iii. Two functions can have the same name.
→ iii is static overloading
 - iv. Functions can have other functions as arguments.
→ Not necessarily. This is because OCaml has first class functions

Problem 3: Static Scoping

- Which of these things are implied by the fact that OCaml supports static scoping?
 - i. A variable's value can never change after initialization.
→ this is the idea of functional programming
 - ii. Each variable usage can be bound to its associated declaration at compile time.
→ static scoping requires that when a function body is executed, we must use the environment that existed when that function was declared.
 - iii. Currying properly preserves the behavior of passing multiple arguments to a function.
 - iv. All memory can be allocated at compile time.

Currying and Static Scoping

- To make static scoping work, every function has to keep its static (lexical) environment with it.

```
(* env is [("add", (<fun>, []))] *)
# let x = 45;;
val x : int = 45
(* env is [("x", 45), ("add", (<fun>, []))] *)
# let add45 = add 45 //(function y -> y + x);;
val add45 : int -> int = <fun>
(* env is [("x", 45), ("add", (<fun>, [])), ("add45", (<fun>, [("x", 45)]))] *)
# let x = 12;;
val x : int = 12 (* env is [("x", 12), ("add", (<fun>, [])), ("add45",
(<fun>, [("x", 45)]))] *)
# add45 3;;
- : int = 48
```

Currying with Dynamic Scoping??

- If we have dynamic scoping, we are using the current env instead of a copy of the original env

```
(* env is [("add", (<fun>, []))] *)
# let x = 45;;
val x : int = 45
(* env is [("x", 45), ("add", (<fun>, []))] *)
# let add45 = add 45 // (function y -> y + x);;
val add45 : int -> int = <fun>
(* env is [("x", 45), ("add", (<fun>, [])), ("add45", <fun>)] *)
# let x = 12;;
val x : int = 12 (* env is [("x", 12), ("add", (<fun>, [])), ("add45", <fun>)] *)
# add45 3;;
- : int = 15
```

Problem 3: Static Scoping

- Which of these things are implied by the fact that OCaml supports static scoping?
 - i. A variable's value can never change after initialization.
→ this is the idea of functional programming
 - ii. Each variable usage can be bound to its associated declaration at compile time.
→ static scoping requires that when a function body is executed, we must use the environment that existed when that function was declared.
 - iii. Currying properly preserves the behavior of passing multiple arguments to a function.
→ With dynamic scoping, behavior of curried function depends on current env
 - iv. All memory can be allocated at compile time.
→ You still have to execute the program