FPV Tutorübung

Woche 10

OCaml: Modules

Manuel Lerchner

28.06.2023

T01: A Multitude of Map Modules

1. **StringPrintable** 2 von 2 Tests bestanden
   Implement a module `StringPrintable` of signature `Printable`.
2. **OrderedPrintable** 1 von 1 Tests bestanden
   Define a signature `OrderedPrintable` that extends the `Printable` signature by a `compare` function with the usual type.
3. **IntOrderedPrintable** 3 von 3 Tests bestanden
   Additionally to the `StringPrintable` now implement an `IntOrderedPrintable` module.
4. **BinaryTreeMap** 1 von 1 Tests bestanden
   Implement a functor `BinaryTreeMap` that realizes the `Map` signature and uses a binary tree to store *key-value*-pairs. The functor takes an ordered key printable and a value printable as arguments.
5. **IntIntMap** 5 von 5 Tests bestanden
   Use the `BinaryTreeMap` functor to define the module `IntIntMap` for *int-to-int* maps.
6. **IntStringMap** 5 von 5 Tests bestanden
   Use the `BinaryTreeMap` functor to define the module `IntStringMap` for *int-to-string* maps.

We model domains of printable values using modules with signature

```
module type Printable = sig
    type t
    val to_string : t -> string
end
```

```
module type Map = sig
    type key
    type value
    type t
    val empty : t
    val set : key -> value -> t -> t
    val get : key -> t -> value
    val get_opt : key -> t -> value option
    val to_list : t -> (key * value) list
    val to_string : t -> string
end
```

Where the functions from `Map` have the following semantics:

- `set key value m` updates the mapping, such that `key` is now mapped to `value`
- `get key m` retrieves the value for the given key and throws a `Not_found` exception if no such key exists in the map.
- `get_opt key m` retrieves the value for the given key or `None` if the key does not exist.
- `to_string m` produces a string representation for the mapping, e.g.: `"{ 1 -> \"x\", 5 -> \"y\" }"`
- `to_list m` returns a list containg all key-value tuples in the given mapping.

OCaml vs Java: Module Type

- Module types sind ähnlich zu Interfaces in Java
  - Kapselung von zusammengehörigen Daten / Funktionen

```ocaml
module type Animal = sig
  unit -> string
  val make_sound : unit → string
end
```

```java
public interface Animal {
    public String makeSound();
}
```

OCaml vs Java: Module

- Modules sind wie Klassen in Java, sie können Module types *implementieren*
- Typisierung entspricht *implements*

```ocaml
module  Cat : Animal = struct
  unit -> string
  let make_sound () = "Miau"
end

module Dog : Animal = struct
  unit -> string
  let make_sound () = "Woof"
end
```

```java
class Cat implements Animal {
    @Override
    public String makeSound() {
        return "Miau";
    }
}

class Dog implements Animal {
    @Override
    public String makeSound() {
        return "Woof";
    }
}
```

OCaml vs Java: Module Type with generic type

- Modules types mit eigenem Datentyp entsprechen generischen Interfaces
  - Mit zusätzlich get() Funktion

```ocaml
module type ListElement = sig
  type t
  unit -> t
  val get : unit -> t

end
```

```java
interface ListElement<T> {
    T get();
}
```

OCaml vs Java: Include Keyword

- The *include* keyword is similar to the *extend* keyword in Java

```ocaml
module type Animal = sig
  unit -> string
  val make_sound : unit → string
end

module type Mammal = sig
  include Animal
  unit -> string
  val give_birth : unit → string
end
```

```java
public interface Animal {
    public String makeSound();
}

interface Mammal extends Animal {
    public String giveBirth();
}
```

OCaml vs Java: Functors

- Functors are Similar to Generic classes, where the generic type has a *constraint*

```ocaml
module type Serializable = sig
  type t
  t -> string
  val serialize : t → string
end

module ListSerializer (T : Serializable ) = struct
  T.t list -> string
  let serialize_list (l:T.t list) =
    List.fold_left (fun acc x → acc ^ T.serialize x ^ "\n") "" l
end
```

```java
interface Serializable<T> {
    String serialize(T t);
    T get();
}

class ListSerializer<T extends Serializable> {
    public String serialize(T[] list) {
        String result = "";
        for (T t : list) {
            result += t.serialize(t.get()) + "\n";
        }
        return result;
    }
}
```

OCaml vs Java: Functor Example

```
module MyInteger : Serializable with type t = int = struct
  type t = int
  t -> string
  let serialize x = string_of_int x
end


module ListSerializer (T : Serializable ) = struct
  T.t list -> string
  let serialize_list (l:T.t list) =
    List.fold_left (fun acc x → acc ^ T.serialize x ^ "\n") "" l
end
```

```
module IntListSerializer = ListSerializer(MyInteger)
```

```
unit
let res =  print_string (IntListSerializer.serialize_list [1;2;3])

(*
1
2
3
*)
```

Nicht ganz, ListSerializer hat falsche Signatur

```java
class MyInteger implements Serializable<Integer> {
    Integer i;

    public MyInteger(Integer i) {
        this.i = i;
    }

    public String serialize(Integer i) {
        return i.toString();
    }

    public Integer get() {
        return i;
    }
}
```

```java
class ListSerializer<T extends Serializable> {    Seri
    public String serialize(T[] list) {
        String result = "";
        for (T t : list) {
            result += t.serialize(t.get()) + "\n";
        }
        return result;
    }
}
```

```java
class IntListSerializer extends ListSerializer<MyInteger> {
}
```

```java
IntListSerializer serializer = new IntListSerializer();

System.out.println(serializer.serialize(
        new MyInteger[] {
            new MyInteger(1),
            new MyInteger(2),
            new MyInteger(3)
        }));
// 1
// 2
// 3
```

Sharing Constraints

```ocaml
module type Inc = sig
  type t
  t -> t
  val inc : t → t
end

module HiddenIntInc : Inc = struct
  type t = int
  t -> t
  let inc x = x + 1
end
```

```java
interface Inc<T> {
    T inc(T t);
}

class HiddenIntInc implements Inc {
    public Integer inc(Integer t) {
        return t + 1;
    }
}
```

```
utop # HiddenIntInc.inc;;
- : HiddenIntInc.t -> HiddenIntInc.t = <fun>
─( 17:23:37 )─< command 12 >──────────────────────────────
utop # HiddenIntInc.inc 4;;
Error: This expression has type int but an expression was expected of type
         HiddenIntInc.t
─( 17:23:46 )─< command 13 >──────────────────────────────
```

Sharing Constraints

```ocaml
module type Inc = sig
  type t
  (* t -> t *)
  val inc : t → t
end

module ExposedIntInc : Inc with type t = int = struct
  type t = int
  (* t -> t *)
  let inc x = x + 1
end
```

```
utop # ExposedIntInc.inc;;
- : int -> int = <fun>
─( 17:25:08 )─< command 15 >───
utop # ExposedIntInc.inc 4;;
- : int = 5
─( 17:25:12 )─< command 16 >───
utop #
```

```java
interface Inc<T> {
    T inc(T t);
}

class ExposedIntInc implements Inc<Integer> {
    public Integer inc(Integer t) {
        return t + 1;
    }
}
```

Summary

- Ähnlichkeiten zwischen OCaml und Java
  - module type == Interface
  - module == Klasse
  - typisiertes Module == Klasse die Interface implementiert
  - Include keyoword in Module type == Interface extended anderes Interface
  - Functor == generische Klasse mit Constraint auf Generic
    - Wird „ausgeführt“ indem der generische Typ „festgelegt“ wird