Programming Languages: Recitation 11

Goktug Saatcioglu

04.18.2019

1 HW9 Problem 1

Rather than going over the whole implementation we go over the key ideas. For implementation details see the solutions available on the class Github repository.

- We begin by declaring the signature for our module. This requires the proper reading of the mathematical definitions given in the assignment specification. This is pretty straightforward as the signatures look like OCaml function type definitions. For example, we see that `count: MSet(U)  U  ` so we can use the signature `val count : 'u t -> 'u -> int`. The complete signature is given below.

```
module type MultisetType = sig
  type 'u t (** represents MSet(U) for arbitrary base sets U *)

  val count: 'u t -> 'u -> int
  val empty: 'u t
  val add: 'u t -> 'u -> 'u t
  val remove: 'u t -> 'u -> 'u t
  val union: 'u t -> 'u t -> 'u t
  val inter: 'u t -> 'u t -> 'u t
  val diff: 'u t -> 'u t -> 'u t
  val union: 'u t -> 'u t -> 'u t
  val sum: 'u t -> 'u t -> 'u t
  val equals: 'u t -> 'u t -> bool
end
```

- Now we have to decide on the implementation of `'u t` which will in turn define the implementation of our functions. Note that the way to read `'u t` is as a type `t` consisting of type `'u`. So if `t` is the multiset types then we get a multiset consisting of type `'u`. With this in mind we can then obtain a concrete implementation. There are many correct implementations but we will implement multisets as lists of pairs where the first element of the pair is of type `'u`, i.e. the base type of the multiset, and the second element of the pair is of type `int`, i.e. its multiplicity. So we get the definition `'u t = ('u * int) list`. Now the implementation is relatively straightforward and given below.

```
module Multiset : MultisetType = struct
  type 'u t = ('u * int) list

  let upsert op m x =
    let rec upsert res = function
      | (y, c) :: m1 as m ->
          let cmp = compare x y in
          if cmp = 0
          then List.rev res @ op c @ m1
          else if cmp < 0
          then List.rev res @ op 0 @ m
          else upsert ((y, c) :: res) m1
      | [] -> List.rev (op 0 @ res)
    in
    upsert [] m
```

```
let merge op m1 m2 =
  let rec merge res m1 m2 = match m1, m2 with
  | (x1, c1) :: m11, (x2, c2) :: m21 ->
      let cmp = compare x1 x2 in
      if cmp = 0
      then merge (op x1 c1 c2 @ res) m11 m21
      else if cmp < 0
      then merge (op x1 c1 0 @ res) m11 m2
      else merge (op x2 0 c2 @ res) m1 m21
  | [], (x2, c2) :: m21 ->
      merge (op x2 0 c2 @ res) [] m21
  | (x1, c1) :: m11, [] ->
      merge (op x1 c1 0 @ res) m11 []
  | [], [] -> List.rev res
  in
  merge [] m1 m2

let rec count m x = match m with
| (y, c) :: m1 ->
    let cmp = compare x y in
    if cmp = 0 then c else
    if cmp < 0 then 0 else
    count m1 x
| [] -> 0

let empty = []

let add m x =
  upsert (fun c -> [(x, c + 1)]) m x

let remove m x =
  upsert (fun c -> if c > 1 then [(x, c - 1)] else []) m x

let union m1 m2 =
  merge (fun x c1 c2 ->
    if c1 + c2 > 0
    then [(x, max c1 c2)]
    else []) m1 m2

let inter m1 m2 =
  merge (fun x c1 c2 ->
    if c1 > 0 && c2 > 0
    then [(x, min c1 c2)]
    else []) m1 m2

let diff m1 m2 =
  merge (fun x c1 c2 ->
    if c1 > c2
    then [(x, c1 - c2)]
    else []) m1 m2
```

```
  let sum m1 m2 =
    merge (fun x c1 c2 ->
      if c1 + c2 > 0
      then [(x, c1 + c2)]
      else []) m1 m2

  let equals m1 m2 = m1 = m2
end
```

- Notice how in the above we define a function `upsert` that acts as a sort of fold but for multisets. We can then pass appropriate operations to it such as `(fun c -> [(x, c + 1)])` to add to a multiset. We also define a function `merge` that merges two multisets according to some operation `op`. For example, if the operation is

```
(fun x c1 c2 ->
  if c1 > 0 && c2 > 0
  then [(x, min c1 c2)]
  else [])
```

  then we will end up taking the intersection of two multisets. Can you see why? It is because we only create a multiset entry if the element we are considering is in both multisets and then we make its multiplicty become the minimum of the two multiplicities.

2 HW9 Problem 2

Again, rather than going over the whole implementation we go over the key ideas. For implementation details see the solutions available on the class Github repository.

- We now have two different types where type `u` represents the type of the base set and type `t` represents the type of the multiset. In terms of the signature of our new module, we only need to make a minor change from the first part and replace all occurrences of `'u t` with `t`. This way we get the correct signature for our new model. The solution is given below.

```
module type MultisetType = sig
  type u (** represents base set U *)
  type t (** represents MSet(U) *)

  val count: t -> u -> int
  val empty: t
  val add: t -> u -> t
  val remove: t -> u -> t
  val union: t -> t -> t
  val inter: t -> t -> t
  val diff: t -> t -> t
  val union: t -> t -> t
  val sum: t -> t -> t
  val equals: t -> t -> bool

  val max_opt: t -> u option
end
```

- We now need to create a functor that takes an `OrderedType` and gives us a `MultiSetType`. We are free to decide on how to implement type `t` but the type of `u` must equal the type of the ordered type. Can

you explain why? The reason is that since we use ordered types as the argument for our functor then the type of our base set must be the interal type of the ordered types. We take the suggestion given in the assignment specification and make the base set be map of type `O.t` (where `O.t` is the internal type of the ordered type). This means that the type `t`, i.e. the multiset, should be the map we just defined with its values being of type `int` so that we can map `O.t`s to `int`s. Finally, to get the information `O.t` we expose it (as it is not available to our functor when we pass it) by using `with` statement. Bringing everything together, we get the following start to our implementation.

```
module Make(O: Map.OrderedType) : MultisetType with type u = O.t = struct
  type u = O.t module M = Map.Make(O)

  type t = int M.t
end
```

So we see that we use type `int Map.Make(O).t` to represnt multisets over `O.t`.

- To implement the functions, we assume that we do not store elements of multiplicty 0 in the maps. This way we can test for equality by using `Map.S.equal`. Furthermore, we use `Map.merge` which is a function in the `Map` module that takes a function and two maps and merges them accordingly. Looking at the signature of `merge` we see that we must use an option type. This `merge` function will play the same role as the `merge` function from part 1. We can also use the `update` and `find\_opt` functions from the `Map` module to implement the remaining function. The function `find\_opt` is better to use than `find` find raises an exception if a key is not in the map. However, `find\_opt` return `None` if the key is not in the map. Alternatively, we could have uses `mem` and if this is true then `find` too. The full implementation is given below.

```
module Make(O: Map.OrderedType) : MultisetType with type u = O.t = struct
  type u = O.t module M = Map.Make(O)

  type t = int M.t

  let count m x = match M.find_opt x m with
  | None -> 0
  | Some c -> c

  let empty = M.empty

  let add m x =
    M.update x (function None -> Some 1 | Some c -> Some (c + 1)) m

  let remove m x =
    M.update x (function Some c when c > 1 -> Some (c - 1) | _ -> None) m

  let union m1 m2 =
    M.merge (fun x a1 a2 -> match a1, a2 with
    | None, Some c | Some c, None -> Some c
    | Some c1, Some c2 -> Some (max c1 c2)
    | _ -> None) m1 m2

  let inter m1 m2 =
    M.merge (fun x a1 a2 -> match a1, a2 with
    | Some c1, Some c2 -> Some (min c1 c2)
    | _ -> None) m1 m2
```

```
  let diff m1 m2 =
    M.merge (fun x a1 a2 -> match a1, a2 with
    | Some c, None -> Some c
    | Some c1, Some c2 when c1 > c2 -> Some (c1 - c2)
    | _ -> None) m1 m2

  let sum m1 m2 =
    M.merge (fun x a1 a2 -> match a1, a2 with
    | Some c, None | None, Some c -> Some c
    | Some c1, Some c2 -> Some (c1 + c2)
    | _ -> None) m1 m2

  let equals m1 m2 = M.equal (=) m1 m2

  let max_opt m = match M.max_binding_opt m with
  | Some (x, _) -> Some x
  | None -> None
end
```

- The inclusion of the function `max\_opt` will be explained in part 3 but it simply find the largest element in `m` as long as `m` is not empty.

3 HW9 Problem 3

- We extend the definition of a multiset to include a function `compare` to get an `OrderedMultisetType`. This can simply be done by declaring a new signature and using the `include` statement. The solution is given below.

```
module type OrderedMultisetType = sig
  include Part2.MultisetType

  val compare: t -> t -> int
end
```

- Now we must implement the compare function such that if `m1 < m2` according to the Dershowitz-Manna ordering then we return -1. If `m1 = m2` then we return 0 and if `m1 > m2` then we return 1. Recall that the Dershowitz-Manna ordering is defined as `m1 < m2` if and only if `m1 != m2` and for every element $x \in U$ which occurs more often in `m1` than in `m2`, there exists an element $y \in U$ which occurs more often in `m2` than in `m1` and $x < y$. There are many correct ways to implement the `compare` function. However, for the sample solutions we wish to use an efficient as possible implementation which takes advantage of the fact that we have a total ordering. Recall that a total order is an order such that we have

  - Antisymmetry (i.e. if $a \leq b$ and $b \leq a$ then $a = b$).
  - Transitivity (i.e. if $a \leq b$ and $b \leq c$ then $a \leq c$).
  - Connexity (i.e. $a \leq b$ or $b \leq a$, so every element is related to every other element in some way.)

  This means that given `m1` and `m2`, we can take the difference between `m1` and `m2`, i.e. $m1 \setminus m2$, to get all the elements in `m1` that occur more often than the elements in `m2`. Furthermore, we can take the difference between `m2` and `m1`, i.e. $m2 \setminus m1$, to get all the elements in `m2` that occur more often than the elements in `m1`. Now it is enough to check the maximal elements of the two set differences and if the maximum element of the first difference is less than the maximum element of the second difference then we know that `m1 < m2`. Why do we not need to check any of the other elements in the differences? Recalling the definition of a total order given above, as long as the maximal elements satisfy the less

than relation we can relate all the other elements in the differences to the maximal elements and verify the Dershowitz-Manna ordering. Note that this is why defined the `max_opt` function in part 2. Finally, there are corner cases we need to check which are basically the case when either one of the differences is empty or both are empty. If both are empty the sets are equal and otherwise the one empty difference is less than the other difference. The solution is given below.

```
module Make(O: Map.OrderedType) : OrderedMultisetType with type u = O.t = struct
  include Part2.Make(O)

  let compare m1 m2 =
    let m1_diff_m2 = diff m1 m2 in
    let m2_diff_m1 = diff m2 m1 in
    match max_opt m1_diff_m2, max_opt m2_diff_m1 with
    | Some x, Some y -> O.compare x y
    | None, Some _ -> -1
    | Some _, None -> 1
    | None, None -> 0
end
```

4 V-Tables

- In last week's lecture we learned about v-tables and dynamic method dispatch. Let's practice creating v-tables.
- Consider the Scala program given below.

```
class Base(private var x: Char, var y: Char) {
  def m1(): String = "Base: m1()="+(x.toString)+"+"+(y.toString)
  def m2(): Int = {
    println("Base: m2()="+(x+y))
    x+y
  }
}
class Derived(var y0: Char, var z: Int) extends Base('a', y0) {
  def m3(): Int = m2() + y + z
  def m1(): String = "Derived:"+((m3()).toString)+(z.toString)
}
```

Remember that:

- Each data member can be accessed via a fixed offset from the base address of the data layout. The offset is determined by the number of bytes needed to represent a value of the type of that field.
- Subclass objects have the same memory layout as superclass objects with additional space for the subclass fields that succeeds the space for the superclass fields.
- Objects of type subtype of parent can be polymorphically operated on as if they were objects of type parent, since the offsets of the subclass fields are the same.
- Private fields are also included in the data layout because they contain instance-specific data. We can even indirectly access private fields with calls to the parent's methods.

How is the program above represented in the memory? We begin by obtaining the memory representation of `Base` and `Derived`. We see that `Base` has two local variables so we store the vptr followed by those variables. `Derived` has the two variables inherite from `Base` plus one more variable so we store the vptr followed by those variables. The solution is given below.

```
Base Instance:
0   -------------
   | vptr        |
8   -------------
   | value of x  |
9   -------------
   | value of y  |
10  -------------

Derived Instance:
0   -------------
   | vptr        |
8   -------------
   | value of x  |
9   -------------
   | value of y  |
10  -------------
   | value of z  |
14  -------------
```

Next, we need to show `Base` and `Derived`'s vtables. For `Base` the answer is pretty straightforward and we have pointers to `m1` and `m2`. Next, we see that `Derived` inherits overrides `m1`, inherits `m2` and implements a new method `m3`. Note that evem though `m3` is declared before `m1` we need `m1` to appear before `m3` in the vtable. The solution is given below.

```
Base vtable:
0   -------------
   | ptr. to m1  |
8   -------------
   | ptr. to m2  |
16  -------------

Derived Instance:
0   -------------
   | ptr. to m1  |
8   -------------
   | ptr. to m2  |
16  -------------
   | ptr. to m3  |
24  -------------
```

Finally, we have to figure out where these pointers point to. Since `m2` is shared by both method both pointers should point to `Base.m2`. Furthermore, only `Derived` uses `m3` so the `ptr. to m3` should point to `Derived.m3`. Finally, `Base.m1` is overriden by `Derived.m1` so we have two different implementations for these methods. The solution is given below.

```
Base vtable:
0   -------------
   | ptr. to m1  | --> [impl. of Base.m1]
8   -------------
   | ptr. to m2  | --> [impl. of Base.m2]
```

```
16  -------------          ^
                           |
Derived Instance:          |
0   -------------          |
   | ptr. to m1  | --> [impl. of Derived.m1]
8   -------------
   | ptr. to m2  | --------|
16  -------------
   | ptr. to m3  | --> [impl. of Derived.m3]
24  -------------
```

The complete picture is given below.

```
Base Instance:          Base vtable:
0    ------------   |-> 0   -------------
   | vptr        | -|      | ptr. to m1  | --> [impl. of Base.m1]
8    ------------       8   -------------
   | value of x  |         | ptr. to m2  | --> [impl. of Base.m2]
9    ------------       16  -------------        ^
   | value of y  |                               |
10   ------------                                |
                                                 |
Derived Instance:       Derived vtable           |
0   -------------   |-> 0   -------------        |
   | vptr        | -|      | ptr. to m1  | --> [impl. of Derived.m2]
8   -------------       8   -------------        |
   | value of x  |         | ptr. to m2  | ------|
9   -------------       16  -------------
   | value of y  |         | ptr. to m3  | --> [impl. of Derived.m3]
10  -------------       24  -------------
   | value of z  |
14  -------------
```

- Question: What happens if I add a call `m1()` just before Base.m2() returns its result `x+y` and then I execute the code `new Derived('a',0).m1()`? Answer: Stack overflow. What happens if I make Base.m1() private, remove the overrides keyword from Derived.m1() add the same call to Base.m2() and make the same call as before? Answer: The call now terminates because `m1()` is no longer dynamically dispatched (i.e. virtual).

5 Dynamic Dispatch

- Now that we know to think about data layouts and vtables let's practice answering questions about dynamic method dispatch.
- Consider the following class definitions.

```
class Base(var x: Int, private var y: Int) {
  def m1(s: String): Base = {
  println("Base.m1(String)")
    m2()
    this
```

```
  }
  def m2(): Unit = println("Base.m2()")
}
class Derived(var z: Base, var s: Int) extends Base(0,0) {
  override def m1(s: String): Base = {
    println("Derived.m2(String)")
    m2()
    this
  }
  override def m2(): Unit = println("Derived.m2(String)")
}
```

- Now consider the following piece of code.

```
val d: Base = new Derived(new Base(0,0), 0)
d.m1("Hello")
```

Answer the following questions. For each question point out which of the methods are virtual and which are non-virtual.

1. What is the static type of d? A: Base. What is the dynamic type of d? A: Derived.
2. What does this call print? A: "Derived.m1(String)" followed by "Derived.m2(String)".
3. What does this call print if I comment out Derived.m1? A: "Base.m1(String)" followed by "Derived.m2(String)".
4. What does this call print if I comment out Derived.m2? A: "Base.m1(String)" followed by "Base.m2()".
5. What does this call print if I comment out Derived.m1 and make both Base.m2 and Derived.m2 private? A: "Base.m1(String)" followed by "Base.m2()".
6. What does this call print if I change the declaration to `val d: Base = new Base(0,0)`? A: "Base.m1(String)" followed by "Base.m2()".
7. In the preceeding case, what is the static type of d? A: Base. What is the dynamic type of d? A: Base.

- Now consider the following piece of code.

```
val e = ((new Derived(new Base(0,0), 0)).m1("Hello")).m1("Hello")
```

Answer the following questions.

- Repeat questions 2-6 from above. What do we print now for each case? A: Nothing should change. What is the type of `e` for each case?
- Change the `this` at the end of Derived.m1 to `z`. What do I print out now?
- Bonus question: Consider another piece of code as continuation to the code above.

```
((new Derived(e, 0)).m1("Hello")).m1("Hello")
```

What happens in the case I return `this` for questions 2-6? What happens in the case I return `z` for questions 2-6?