튜토리얼

OCAML 일종 모듈과 재귀 모듈 프로그래밍

임현승
POSTECH PL lab

ROSAEC Center Workshop @ 파주
2011-06-27

발표순서

- OCaml 모듈 시스템의 특징
- 일종 모듈 (first-class modules)
- 재귀 모듈 (recursive modules)
- 결론 및 요약

OCaml 모듈 시스템의 특징: 모듈

- 프로그램을 구조화하고 이름 공간(namespace)을 부여

```
#   module Tree = struct
      type t = Leaf | Node of t * t
      let max x = ...
    end

#   module Forest = struct
      type t = Tree.t list
      let max x = ...
    end

#   ... Tree.max ... Forest.max ...
```

OCaml 모듈 시스템의 특징: 모듈 타입

- 프로그램의 기능을 타입 형태로 기술하고 재사용
- 정보 은닉(information hiding) 지원

```
#   module type FOREST = sig
      type t
      val complicated_f : t -> int
    end

#   module Forest : FOREST = struct
      type t = Tree.t list
      let helper_f x = ...
      let complicated_f x = ... Helper_f ...
    end

#   ... MyModule.helper_f ...                    (* 타입 오류 *)
```

OCaml 모듈 시스템의 특징: 추상 타입

- **추상 타입(abstract types)을 이용하여 데이터 추상화(data abstraction)를 효과적으로 지원**

```
# module type FOREST = sig
    type t                                    (* 추상 타입 선언 *)
    val nth : t -> int
  end

# module Forest : FOREST = struct
    type t = Tree.t list
    let nth l = List.nth l
  end

# let f (l : Forest.t) = ... List.nth l ...   (* 타입 오류 *)
```

OCaml 모듈 시스템의 특징: 모듈 함수

- 모듈 함수를 이용한 코드 재사용
- 예: 다양한 데이터 타입에 대한 집합 모듈

```
#   type comparison = Less | Equal | Greater
#   module type ORDERED_TYPE = sig
      type t
      val compare : t -> t -> compare
    end

#   module Set = functor (Elt : ORDERED_TYPE) ->   (* 모듈 함수 선언 *)
    struct
      type element = Elt.t
      type set = element list
      let empty = []
      ...
    end
#   module OrderedString = struct type t = string ... end
#   module StringSet = Set(OrderedString)          (* 모듈 함수 적용 *)
```

OCaml 모듈 시스템의 특징: second-class

- 코어 언어와 모듈 언어가 계층적으로 분리되어 있음
  - 모듈을 함수의 인자로 넘기거나 결과로 돌려줄 수 없음
  - 데이터 타입 정의에 모듈(타입)을 포함시키거나 조건문, 패턴매칭에서 모듈을 사용할 수 없음
  - 실행시간에 라이브러리 또는 플러그인 선택 사용 불가

발표순서

- OCaml 모듈 시스템의 특징

- 일종 모듈 (first-class modules)
  - 실행시간에 라이브러리 선택하기
  - 타입 안전성이 보장되는 플러그인
  - 다형 함수를 함수의 인자로 전달하기 (first-class polymorphism)
  - 모듈을 인자로 받거나 결과로 돌려주는 함수
  - 고차 존재 타입(higher-kinded existential types)을 이용해서 스트림 구현하기

- 재귀 모듈 (recursive modules)

- 결론 및 요약

일종 모듈 (first-class modules)

- OCaml 3.12.0 부터 지원
- 모듈을 코어 값으로 묶거나(pack) 코어 값을 모듈로 풀 수 있음(unpack)

- 사용방법

```
#   module type S = sig type t ... end      (* 패키지 타입 *)

#   type s = (module S)                     (* 모듈 타입 -> 코어 타입 *)

#   let x = (module struct ... end : S)     (* 모듈 값 -> 코어 값)

#   module M = (val x : S)                  (* 코어 값 -> 모듈 값)
```

실행시간에 라이브러리 선택하기

```
(* 라이브러리 명세 *)
#  module type DEVICE = sig ... end
#  let devices : (string, (module DEVICE)) Hashtbl.t = Hashtbl.create 17

(* 라이브러리 1 *)
#  module SVG = struct ... end
#  let _ = Hashtbl.add devices "SVG" (module SVG : DEVICE)

(* 라이브러리 2 *)
#  module PDF = struct ... end
#  let _ = Hashtbl.add devices "PDF" (module PDF : DEVICE)

(* 실행 중에 사용할 모듈 선택 *)
#  module Device =
     (val (try Hashtbl.find devices Sys.argv.(1)
           with Not_found -> prerr_endline "Unknown device"; exit 2)
        : DEVICE)
```

타입 안전성이 보장되는 플러그인

```
#  module type PLUGIN = sig
     type t                          (* 플러그인의 상태를 추상 타입으로 표현 *)
                                     (* 구현 시 레퍼런스 타입을 이용 *)
     val init : t
     val start : t -> unit
     val stop : unit -> t
   end
#  let plugins = ref ([] : (string * (module PLUGIN)) list)

#  let new_instance name =
     let module P = (val List.assoc name !plugins : PLUGIN) in
     object                                        (* P.t는 새로운 추상 타입 *)
       val mutable state = P.init
       method start = P.start state
       method stop = state <- P.stop ()
     end ;;
val new_instance : string -> < start : unit; stop : unit > = <fun>
```

다형 함수를 함수의 인자로 전달하기 (first-class polymorphism)

```
#  module type ID = sig val id : 'a -> 'a end
#  let f id =
     let module Id = (val id : ID) in
     (Id.id 1, Id.id true) ;;
val f : (module ID) -> int * bool = <fun>

#  f (module struct let id x = print_endline "Id!"; x end : ID) ;;
Id!
Id!
- : int * bool = (1, true)

(* 레코드나 객체의 다형성을 이용해서도 표현 가능 *)
#  type t = { id : 'a. 'a -> 'a }              (* 레코드 타입 선언 *)
#  let f r = (r.id 1, r.id true)
#  let _ = f { id = fun x -> print_endline "Id!"; x }
```

모듈을 인자로 받거나 결과로 돌려주는 함수

```
(* Set 모듈의 정렬 함수를 이용하여 리스트를 정렬하는 함수 *)
(* 모듈을 인자로 받음 *)
#  let sort (type s) set l =              (* t는 로컬 변수, 추상 타입 *)
     let module Set = (val set : Set.S with type elt = s) in
     Set.elements (List.fold_right Set.add l Set.empty)
val sort : (module Set.S with type elt = 'a) -> 'a list -> 'a list

(* 비교 함수를 받아서 새로운 모듈을 생성하는 함수 *)
#  let make_set (type s) cmp =
     let module S = Set.Make(struct
       type t = s
       let compare = cmp
     end) in
     (module S : Set.S with type elt = s)
val make_set : ('a -> 'a -> int) -> (module Set.S with type elt = 'a)
```

고차 존재 타입 (higher-kinded existential types)을 이용해서 스트림 구현하기

```
(* 고차 존재 타입을 이용한 스트림 연산 인터페이스 *)
∃(t :: * -> *){
   head : ∀α. α t → α,
   tail : ∀α. α t → α t,
   from : ∀α. (int → α) → α t,
   map  : ∀α β. (α → β) → α t → β t
 }

(* 스트림 연산 인터페이스 *)
#  module type StreamOps = sig
      type 'a t                              (* 스트림 타입, 추상 타입 *)
      val head : 'a t -> 'a
      val tail : 'a t -> 'a t
      val from : (int -> 'a) -> 'a t
      val map  : ('a -> 'b) -> 'a t -> 'b t
    end
```

스트림 연산 구현 예

```
# module type StreamOps = sig
    type 'a t                              (* 스트림 타입, 추상 타입 *)
    val head : 'a t -> 'a
    val tail : 'a t -> 'a t
    val from : (int -> 'a) -> 'a t
    val map  : ('a -> 'b) -> 'a t -> 'b t
  end

(* 스트림: 정수 i를 받아서 i번째 원소를 리턴하는 데이터 구조 *)
# module FunctionalStream : StreamOps = struct
    type 'a t =
    let head s =
    let tail s =
    let from f =
    let map f s =
  end
```

스트림 값 생성하기

```
(* 스트림 모듈의 타입 *)
# module type Stream = sig
    type elem                          (* 스트림 원소의 타입 *)
    include StreamOps                  (* 스트림 타입은 추상 타입 'a t *)
    val stream : elem t                (* 스트림 값 *)
  end

# type 'a stream = (module Stream with type elem = 'a)

(* 스트림 연산을 이용한 스트림 생성 함수 *)
# let mk_stream : 'a.(module StreamOps) -> (int -> 'a) -> 'a stream
    = fun (type s) ops f ->
      (module
       struct
         type elem = s
         include (val ops : StreamOps)
         let stream = from f
       end : Stream with type elem = s)
```

이 밖에도

- Polymorphic recursion
- Sieve of Eratosthenes
- Dynamically-sized arrays
- Leibniz equality
- Generalized algebraic data types (GADTs)
- Generic programming

발표순서

- OCaml 모듈 시스템의 특징
- 일종 모듈 (first-class modules)
- **재귀 모듈 (recursive modules)**
  - **재귀 데이터 구조 구현하기**
  - **프로그래밍 언어 인터프리터 설계하기**
  - **여러 모양으로 함수를 재귀 호출하기 (polymorphic recursion)**
  - **일종 모듈과 재귀 모듈을 이용하여 객체와 클래스 표현하기**
- 결론 및 요약

재귀 모듈 (recursive modules)

- OCaml 3.07 부터 실험적으로 지원
- 상호 참조하는 타입이나 함수를 서로 다른 모듈에 작성할 수 있음

- 사용 방법

```
(* 재귀 모듈은 타입을 항상 적어줘야 함 *)
#   module rec Tree : sig ... end = struct
              ... Forest.compare ...
    end
    and Forest : sig ... end = struct
              ... Tree.compare ...
    end
```

재귀 데이터 구조 구현하기

```
(* Tree 데이터 구조 *)
#  module rec Tree : sig
         type t = Leaf of string | Node of Forest.t
         val compare: t -> t -> int
       end
     = struct
         type t = Leaf of string | Node of Forest.t
         let compare t1 t2 =
           match (t1, t2) with
           | (Leaf s1, Leaf s2) -> Pervasives.compare s1 s2
           | (Leaf _, Node _) -> 1
           | (Node _, Leaf _) -> -1
           | (Node n1, Node n2) -> Forest.compare n1 n2
       end
   and Forest : Set.S with type elt = Tree.t
     = Set.Make(Tree)          (* 이미 구현되어 있는 Set 라이브러리를 이용 *)
```

프로그래밍 언어 인터프리터 설계하기

```
#  module rec Expr : sig
     type t
     val simpl : t -> t
   end = struct
     type t = Var of string | LetExpr of Bind.t * t | ...
     let rec simpl = function
       | Var s -> Var s
       | LetExpr (b, t) -> LetExpr(Bind.simpl b, simpl t)
       | ...
   end
   and Bind : sig
     type t
     val simpl : t -> (string * Expr.t) list
   end = struct
     type t = LetBind of string * Expr.t | ...
     let simpl = function
       | LetBind (v, e) -> [(v, Expr.simpl e)]
       | ...
   end
```

따로 컴파일 (separate compilation)

```
#  module type EXPR = sig
     type t
     val simpl : t -> t
   end
#  module type BIND = sig
     type t
     type s                          (* s는 모듈 함수를 적용할 때 Expr.t로 변경 *)
     val simpl : t -> (string * s) list
   end

#  module ExprFn (Bind : BIND) = struct ... end
#  module BindFn (Expr : EXPR) = struct ... end

#  module rec Expr : EXPR = ExprFn(Bind)
   and Bind : BIND with type s = Expr.t = BindFn(Expr)
```

여러 모양으로 함수를 재귀 호출하기 (polymorphic recursion)

- 포화 이진 트리(perfect binary tree)의 모든 leaves의 레이블의 합을 구하는 함수 작성

```
# type 'a perfect = Zero of 'a | Succ of 'a fork perfect
  and 'a fork = Fork of 'a * 'a

# let sump_module x =
    let module M = struct
      module rec SumP : sig
        val sump : ('a -> int) -> 'a perfect -> int
      end = struct
        let sump = fun f -> function
          | Zero x -> f x
          | Succ x ->
      end                (* - : 'a fork -> int, x : 'a fork perfect *)
    end in M.SumP.sump x
```

일종 모듈과 재귀 모듈을 이용하여 객체와 클래스 표현하기

- 기본 아이디어: 클래스를 모듈 함수로 번역
- 모듈함수는 생성할 객체를 받아 그 객체의 멤버함수를 리턴

```
#   module type POINT = sig                    (* Point 인터페이스 *)
      val get_x : unit -> int
      val move : int -> unit
    end

#   module PointF (X : sig end) = struct    (* Point 클래스 *)
      let x = ref 0
      let get_x () = !x
      let move d = x := !x + d
    end

  (* Point 생성자, 매번 적용될 때마다 새로운 state 생성 *)
#   let new_point () = (module PointF(struct end) : POINT)
```

Traits와 상속 표현하기

```
#   module type SHOW = sig                          (* Show 인터페이스 *)
      val show : unit -> string
    end

#   module ShowF (P : POINT) = struct               (* Show trait *)
      let show () = Printf.sprintf "x = %i" (P.get_x ())
    end

#   module type SHOWPOINT = sig
      include POINT
      include SHOW
    end

(* 상속: Show trait을 Point 클래스에 합침 *)
#   module ShowPointF (P : SHOWPOINT) = struct
      include PointF(P)
      include ShowF(P)
    end
```

Tying the recursive knot

```
(* ShowPoint 생성자, 재귀 모듈을 이용하여 Point와 Show 클래스를 연결시킴 *)
(* 모듈 함수 ShowPointF의 고정점을 취함 *)
#  let new_showpoint () =
     let module M = struct
       module rec SP : SHOWPOINT = ShowPointF(SP)
     end in (module M.SP : SHOWPOINT)

(* ShowPoint 객체 생성 *)
#  let sp = new_showpoint ()
#  let a = let module SP = (val sp : ShowPoint) in
     SP.move 2; SP.show ()
val a : string = "x = 2"
```

- **이와 유사하게 객체 지향 프로그래밍에서 사용하는 method overriding, virtual methods, private methods등의 기능을 재귀 모듈과 일종 모듈을 이용하여 표현 가능**

이 밖에도

- Expression problem
- Bootstrapped heaps
- A translation from a machine independent intermediate language to machine dependent ones

결론 및 요약

- 일종 모듈과 재귀 모듈을 이용하면
  - 많은 새롭고 우아한 프로그래밍 패턴 사용 가능
  - System $F\omega$ + recursion 만큼 강력한 프로그래밍 가능
  - 재밌음

Backup Slides

일반화된 대수적 자료형 표현하기 (generalized algebraic data types)

```
(* a type of witnesses for type equalities *)
#  module TypEq : sig
     type ('a, 'b) t                              (* type equation *)
     val refl: ('a, 'a) t
     val sym: ('a, 'b) t -> ('b, 'a) t
     val apply: ('a, 'b) t -> 'a -> 'b
   end = struct
     type ('a, 'b) t = ('a -> 'b) * ('b -> 'a)
     let refl = (fun x -> x), (fun x -> x)
     let sym (f, g) = (g, f)
     let apply (f, _) x = f x
   end
```

A parameterized algebraic data type

```
#  module rec Typ : sig          (* typ and pair are mutually recursive *)
     module type PAIR = sig
       type t and t1 and t2
       val eq: (t, t1 * t2) TypEq.t
       val t1: t1 Typ.typ
       val t2: t2 Typ.typ
     end

     type 'a typ =
       | Int of ('a, int) TypEq.t
       | String of ('a, string) TypEq.t
       | Pair of (module PAIR with type t = 'a)       (* ∃t1 ∃t2 ... *)
   end = Typ
```

Values of type 'a Typ.typ

```
# let int = Typ.Int TypEq.refl

# let str = Typ.String TypEq.refl

# let pair (type s1) (type s2) t1 t2 =
    let module P = struct
      type t = s1 * s2
      type t1 = s1
      type t2 = s2
      let eq = TypEq.refl
      let t1 = t1
      let t2 = t2
    end in
    let pair = (module P : Typ.PAIR with type t = s1 * s2) in
    Typ.Pair pair
```

A pretty printer for values of type 'a Typ.type

```
#  open Typ

(* exact annotation for polymorphic recursion *)
#  let rec to_string: 'a. 'a Typ.typ -> 'a -> string =
     fun (type s) t x ->                (* s is locally abstract *)
       match t with
       | Int eq -> string_of_int (TypEq.apply eq x)
       | String eq -> Printf.sprintf "%S" (TypEq.apply eq x)
       | Pair p ->
           let module P = (val p : PAIR with type t = s) in
           let (x1, x2) = TypEq.apply P.eq x in
           Printf.sprintf "(%s,%s)"
             (to_string P.t1 x1) (to_string P.t2 x2)
```

참고자료

- The Objective Caml system release 3.12: Documentation and user's manual. http://caml.inria.fr/pub/docs/manual-ocaml/index.html

- Alain Frisch and Jacques Garrigue. First-class modules and composable signatures in Objective Caml 3.12. ML Workshop 2010.

- Jeremy Yallop and Oleg Kiselyov. First-class modules: hidden power and tantalizing promises. ML Workshop 2010. 고급 예제 코드 다음 웹 페이지에서 열람 가능: http://okmij.org/ftp/ML/first-class-modules/