Funktory i abstrakcyjne typy danych

W przykładzie dotyczącym kolejek priorytetowych pokazana była możliwość ukrycia faktycznej reprezentacji (implementacji) typu, który jest zadeklarowany w strukturze. Możliwość taka jest kluczowa przy budowie abstrakcyjnych typów danych, dla ochrony przed błędami programisty (zamierzonymi lub nie).

Przypomnijmy teraz przykład funktora, który ze struktury opisującej typ z porządkiem tworzył strukturę reprezentującą zbiory elementów tego typu za pomocą list.

```
# type comparison = Less | Equal | Greater;;
type comparison = | Less | Equal | Greater

# module type ORDERED_TYPE =
    sig type t
        val cmp: t -> t -> comparison
    end;;
module type ORDERED_TYPE =
  sig type t
      val cmp : t -> t -> comparison
  end
```

```
# module Set =
    functor (Elt: ORDERED_TYPE) ->
      struct
        type element = Elt.t
        type set = element list
        let empty = []
        let rec add x s =
          match s with
            [] -> [x]
          | hd::tl ->
             match Elt.cmp x hd with
               Equal   -> s
             | Less    -> x :: s
             | Greater -> hd :: add x tl
        let rec member x s =
          match s with
            [] -> false
          | hd::tl ->
              match Elt.cmp x hd with
                Equal   -> true
              | Less    -> false
              | Greater -> member x tl
      end;;
```

```
module Set :
  functor(Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      type set = element list
      val empty : 'a list
      val add : Elt.t -> Elt.t list -> Elt.t list
      val member : Elt.t -> Elt.t list -> bool
    end
```

Stosując funktor `Set` do struktury implementującej typ z porządkiem, otrzymujemy strukturę z operacjami na zbiorach elementów tego typu:

```
# module OrderedString =
    struct
      type t = string
      let cmp x y =
         if x = y then Equal
                  else if x < y
                       then Less else Greater
    end;;
```

```
module OrderedString :
  sig
    type t =      string
    val cmp : 'a -> 'a -> comparison
  end

# module StringSet = Set(OrderedString);;
module StringSet :
  sig
    type element = OrderedString.t
    type set = element list
    val empty : 'a list
    val add : OrderedString.t
              -> OrderedString.t list
                      -> OrderedString.t list
    val member : OrderedString.t
              -> OrderedString.t list -> bool
  end
```

Jak widać z typu wypisanego przez kompilator, w sygnaturze wynikowej widoczna jest reprezentacja zbiorów (to, że są reprezentowane przez listy). Pozwala to na wykonywanie na nich wszystkich operacji, jakie można wykonać na listach. Może to prowadzić do błędów kilku rodzajow:

- złośliwe lub przypadkowe naruszenie integralności struktury adnych (np. przez wstawienie elementów do uporządkowanej listy w sposób, który nie zachowuje porządku);
- wykorzystanie zawartości struktury w sposób niezgodny z intencją autora;
- zaprojektowanie kodu, korzystającego ze struktury danych w sposób zależny od jej implementacji. Uniemożliwi to wymianę modułu implementacyjnego bez naruszenia poprawności programu.

Dla uniknięcia tego typu błędów potrzebujemy w naszym przykładzie możliwości ukrycia reprezentacji typu `set` przed modułami, które ze zbiorów korzystają. Można w tym celu przykroić strukturę wynikową funktora za pomocą odpowiednio dobranej sygnatury.

```
# module type SETFUNCTOR =
    functor (Elt: ORDERED_TYPE) ->
      sig
        type element = Elt.t   (* konkretny *)
        type set               (* abstrakcyjny *)
        val empty : set
        val add : element -> set -> set
        val member : element -> set -> bool
      end;;
module type SETFUNCTOR =
  functor(Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      type set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end

# module AbstractSet = (Set : SETFUNCTOR);;
module AbstractSet : SETFUNCTOR
```

```
# module AbstractStringSet =
              AbstractSet(OrderedString);;
module AbstractStringSet :
  sig
    type element = OrderedString.t
    type set = AbstractSet(OrderedString).set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end

# AbstractStringSet.add
               "gee" AbstractStringSet.empty;;
- : AbstractStringSet.set = <abstr>
```

Chcąc zapisać ten sam moduł bardziej elegancko, można nadać nazwę sygnaturze wynikowej funktora i w deklaracji użyć tej nazwy.

```
# module type SET =
    sig
      type element
      type set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end;;
module type SET =
  sig
    type element
    type set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end

# module WrongSet =
     (Set : functor(Elt: ORDERED_TYPE) -> SET);;
module WrongSet :
     functor(Elt : ORDERED_TYPE) -> SET
```

```
# module WrongStringSet =
            WrongSet(OrderedString);;
module WrongStringSet :
  sig
    type element =
           WrongSet(OrderedString).element
    type set = WrongSet(OrderedString).set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end

# WrongStringSet.add
           "gee" WrongStringSet.empty;;
Characters 19-24:
This expression has type string
  but is here used with type
  WrongStringSet.element =
                WrongSet(OrderedString).element
```

Sygnatura `SET` specyfikuje typ `element` jako abstrakcyjny, zatem zgubiona zostaje informacja o o równości między typem `element` w strukturze wynikowej funktora i typem `t` w jego argumencie.

W efekcie typ `WrongStringSet.element` nie jest tym samym co `string`, zatem nie mozna do niego zastosować operacji ze struktury `WrongStringSet`. Musimy w jakiś sposób zadeklarować, że typ `element` w sygnaturze `SET` jest tym samym, co typ `Elt.t`. Służy do tego deklaracja `with type`:

```
# module AbstractSet =
    (Set : functor(Elt: ORDERED_TYPE)
            -> (SET with type element = Elt.t));;
module AbstractSet :
  functor(Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      type set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end
```

Podobnie, jak w przypadku prostych struktur, możemy użyć alternatywnej składni:

```
module AbstractSet(Elt: ORDERED_TYPE)
           : (SET with type element = Elt.t) =
  struct ... end;;
```

Opisany mechanizm pozwala na bardzo ścisłą ochronę implementacji struktur danych, co pozwala na uniknięcie wielu rodzajów trudnych do wykrycia błędów. Zilustrujemy tę tezę przykładem.

```
# module NoCaseString =
    struct
      type t = string
      let cmp s1 s2 =
        OrderedString.cmp
          (String.lowercase s1)
          (String.lowercase s2)
    end;;
module NoCaseString :
  sig
    type t = string
    val cmp : string -> string -> comparison
  end
```

```
# module NoCaseStringSet =
          AbstractSet(NoCaseString);;
module NoCaseStringSet :
  sig
    type element = NoCaseString.t
    type set = AbstractSet(NoCaseString).set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end

# NoCaseStringSet.add
        "FOO" AbstractStringSet.empty;;
Characters 26-49:
This expression has type
  AbstractStringSet.set =
           AbstractSet(OrderedString).set
but is here used with type
  NoCaseStringSet.set =
            AbstractSet(NoCaseString).set
```

Typy `AbstractStringSet.set` i `NoCaseStringSet.set` nie są zgodne ze sobą i nie wolno wymiennie używać wartości tych typów.

Jest to zachowanie ze wszech miar właściwe i porządane: co prawda obie struktury danych dla zbiorów przechowują napisy, ale są oparte na różnych porządkach na tych napisach, zatem działanie obu struktur danych jest oparte na różnych niezmiennikach (porządek z uwzględnieniem rózróżnienia wielkich i małych liter lub bez uwzględnienia tychże). Stosowanie operacji ze struktury `AbstractStringSet` do wartości typu `NoCaseStringSet.set` mogłoby prowadzić do niepoprawnych wyników.

Moduły i rozłączna kompilacja

Wszystkie dotychczasowe przykłady pokazywały używanie systemu intereakcyjnego. Jednak podział na moduły jest ważny głównie w przypadku dużych (lub bardzo dużych) programów, które kompiluje się wsadowo. W ich przypadku zachodzi konieczność dzielenia programu na wiele plików, które powinny być kompilowane osobno.

W Camlu jednostki rozłącznej kompilacji są szczególnymi rodzajami sygnatur i struktur. Na jednostkę rozłącznej kompilacji a składają się:

- implementacja modułu, zawarta w pliku a`.ml`, który zawiera ciąg definicji i odpowiada zawartości konstrukcji składniowej `struct` ... `end`;
- interfejs modułu, zawarty w pliku a`.mli`, który zawiera ciąg deklaracji i odpowiada zawartości konstrukcji składniowej `sig` ... `end`.

Oba pliki razem definiują strukturę o nazwie A (nazwa taka jak plików, z pierwszą literą zamienioną na wielką) i następującej definicji:

```
module A: sig (* zawartość pliku a.mli *) end
      = struct (* zawartość pliku a.ml *) end;;
```

Oba pliki kompiluje się osobno poleceniem `ocamlc -c` (znaczenie parametru `-c` jest takie samo jak w przypadku kompilatora `cc`). Z pliku a`.mli` powstaje skompilowany plik a`.cmi`, a z a`.ml` powstaje skompilowany plik a`.cmo`. Pliki `.cmo` odpowiadające wszystkim modułom programu konsoliduje się w plik wykonywalny poleceniem `ocamlc`. Na przykład mając program złożony z dwóch modułów `aux` i `main`, należy wykonać następujące polecenia:

```
$ ocamlc -c aux.mli  # powstaje aux.cmi
$ ocamlc -c aux.ml   # powstaje aux.cmo
$ ocamlc -c main.mli # powstaje main.cmi
$ ocamlc -c main.ml  # powstaje main.cmo
$ ocamlc -o program aux.cmo main.cmo
                     # powstaje program
```

Uzyskany program zachowuje się tak, jakbyśmy wpisali w środowisku interakcyjnym następujące deklaracje:

```
module Aux: sig (* treść pliku aux.mli *) end
  = struct (* treść pliku aux.ml *) end;;
module Main: sig (* treść pliku main.mli *) end
  = struct (* treść pliku main.ml *) end;;
```

W szczególności moduł `Main` może odwoływać się do modułu `Aux`: definicje i deklaracje w plikach `main.ml` i `main.mli` mogą korzystać z definicji w pliku `aux.ml`, odwołując się do nich za pomocą notacji kropkowej (o ile te definicje są eksportowane z `aux.ml`).

Porządek plików `.cmo` przy konsolidacji jest istotny — w takiej kolejności są deklarowane moduły. W naszym przykładzie `Main` może się odwoływać do `Aux`, ale nie odwrotnie.

Na głównym poziomie programu (tj. w osobno kompilowanych plikach) mogą znajdować się tylko struktury, ale nie funktory ani typy modułów. Ponieważ jednak wszystkie one mogą być składowymi struktury, można je umieścić w strukturze i tę kompilować jako osobny plik.