Tipi Astratti, Moduli: Implementazione

Sommario: 17 Aprile, 2018

- Presentazione di Valori Astratti (Modulo Printf di OCaml).
- Eccezioni ed Option/Maybe in Ocaml.
- Moduli: Implementazione e Activation Records.

Alberi Astratti: Presentazione dei Valori /1

- **Sintassi Astratta**. Implementata con Tipi Algebrici
  - **Costruttori Algebrici:**
    Ogni valore e' un albero costruito con i Costruttori Algebrici;
  - **Pattern Matching**
    Non servono operazioni: Semplici espressioni di Pattern-Matching per costruire, visitare, accedere componenti dei valori:
  - **Costruiamo i valori "dal basso":**

```
let d1 = Const("luigi",26);;
let d2 = Var("x",7);;
let d3 = SeqDcl(d1,d2);;
let d4 = Array("Tab",5);;
let d5 = Array("Vect",10));;
let d6 = SeqDcl(d4,d5);;
let d7 = SeqDcl(d3,d6);;
```

    Ma la "presentazione" di questi valori e' difficile da leggere.
    Il valore d7 si presenta così:

```
SeqDcl(
  SeqDcl(
    Const("luigi",26),Var("x",7)),
  SeqDcl(
    Array("Tab",5),Array("Vect",10))));;
```

Alberi Astratti: Presentazione dei Valori /2

- **Sintassi Astratta**. Implementata con Tipi Algebrici
  - **Costruttori Algebrici:**
  - **Pattern Matching**
  - **Costruiamo i valori "dal basso":**
    Ma la "presentazione" di questi valori e' difficile da leggere.
    quando confrontata al valore presentato in Sintassi Concreta:
    `luigi` $= 26$;
    `var x` $= 7$;
    `Tab`[5];
    `Vect`[10];
  - Tipi Algebrici: Operazioni per "presentare" i valori come stringhe e "stamparli"[1]
    Sia D un Tipo Algebrico da noi definito, allora:
    toStringD : `D` $\rightarrow$ `string`
    printD : `D` $\rightarrow$ `unit`

- **Tipi Astratti** Non hanno una propria presentazione
  - Scrivere `toStringD`, `printD` per fornirne (sempre) una.
  - Usiamo la Presentazione utilizzata nella funzione di astrazione AF
  - Usiamo operazioni del modulo Printf di OCaml.

[1] nel senso di scriverli in un file o supporto equivalente, come i buffer di stampa.

Applichiamo a Dcl di SmallC: Definiamo

```
type ide = string;;
type num = int;;
type dcl =
    Const of ide * num
    | Var of ide * num
    | VarN of ide
    | Array of ide * num
    | SeqDcl of dcl * dcl
    ;;

open Printf;;
let toStringI (i:ide) = sprintf "%s" i;;
let toStringN (n:num) = sprintf "%d" n;;
let rec toStringDcl = function
        Const(ide,num) ->
            let i = toStringI(ide) and n = toStringN(num) in
            sprintf "%s = %s;" i n
        | Var(ide,num) ->
            sprintf "var %s = %s;" (toStringI ide) (toStringN num)
        |_ -> "omitted"   (* correggere e completare *)
;;
let printDcl d = printf "\n%s\n\n" (toStringDcl d);;

let d1 = Const("luigi",26);;
toStringDcl d1;;
printDcl d1;;
let d2 = Var("x", 7);;
printDcl d2;;
let d3 = SeqDcl(d1, d2);;
printDcl d3;;
```

Applichiamo a Dcl di SmallC: Calcola

```
open Printf;;
let toStringI (i:ide) = sprintf "%s" i;;
let toStringN (n:num) = sprintf "%d" n;;
let rec toStringDcl = function
        Const(ide,num) ->
            let i = toStringI(ide) and n = toStringN(num) in
            sprintf "%s = %s;" i n
        | Var(ide,num) ->
            sprintf "var %s = %s;" (toStringI ide) (toStringN num)
        |_ -> "omitted"   (* correggere e completare *)
;;
let printDcl d = printf "\n%s\n\n" (toStringDcl d);;

let d1 = Const("luigi",26);;
toStringDcl d1;;
printDcl d1;;
let d2 = Var("x", 7);;
printDcl d2;;
let d3 = SeqDcl(d1, d2);;
printDcl d3;;


(* esecutore top-level mostra:
val d1 : dcl = Const ("luigi", 26)
- : string = "luigi = 26;"

luigi = 26;

- : unit = ()
val d2 : dcl = Var ("x", 7)

var x = 7;

- : unit = ()
val d3 : dcl = SeqDcl (Const ("luigi", 26), Var ("x", 7))

omitted
```

Applichiamo al Tipo Astratto Stack

```
module type STACK =
(* Uno 'a stack e' un valore [v1,...,vk] per k>0 ([] per k=0), contenente k valori
   di tipo generico 'a. Il valore vk e' l'ultimo aggiunto (mediante un'operazione
   push) ed il primo ad essere estratto (mediante un'operazione pop)
   REQUIRES: Solo valori di tipo 'a per i quali toString:'a->string sia definita.
*)
 sig type 'a stack
  exception Error;;
  val create_stack: unit -> 'a stack
  val push: 'a stack -> 'a -> 'a stack
  val top: 'a stack -> 'a
  val pop: 'a stack -> 'a stack
  val toString: ('a -> string) -> 'a stack -> string
  val printStack: ('a -> string) -> 'a stack -> unit
end;;

module Stack =
 (struct
  exception Error;;
  type 'a sk = E | SK of 'a sk * 'a
  type 'a stack = M of int * ('a sk)
  let create_stack = fun () -> M(0,E)
  let push (M(n,sk)) x =
        if n=100 then raise(Error)
        else M(n+1,SK(sk,x))
  let top (M(n,sk)) = match sk with
                E -> raise Error
                | SK(sk,v) -> v
  let pop (M(n,sk)) = match sk with
                E -> raise Error
                | SK(sk,v) -> M(n-1,sk)
  let toString toString (M(n,sk)) =
        let rec toStringSK x = match x with
                E -> ""
                | SK(sk,v) -> (let left = toStringSK sk in
                               if left = "" then toString v
                               else left^","^(toString v))
        in  "["^(toStringSK sk)^"]"
```

Applichiamo al Tipo Astratto Stack /2

```
 sig type 'a stack
  exception Error;;
  val create_stack: unit -> 'a stack
  val push: 'a stack -> 'a -> 'a stack
  val top: 'a stack -> 'a
  val pop: 'a stack -> 'a stack
  val toString: ('a -> string) -> 'a stack -> string
  val printStack: ('a -> string) -> 'a stack -> unit
end;;

module Stack =
 (struct
  exception Error;;
  type 'a sk = E | SK of 'a sk * 'a
  type 'a stack = M of int * ('a sk)
  let create_stack = fun () -> M(0,E)
  let push (M(n,sk)) x =
        if n=100 then raise(Error)
        else M(n+1,SK(sk,x))
  let top (M(n,sk)) = match sk with
                E -> raise Error
                | SK(sk,v) -> v
  let pop (M(n,sk)) = match sk with
                E -> raise Error
                | SK(sk,v) -> M(n-1,sk)
  let toString toString (M(n,sk)) =
        let rec toStringSK x = match x with
                E -> ""
                | SK(sk,v) -> (let left = toStringSK sk in
                               if left = "" then toString v
                               else left^","^(toString v))
        in  "["^(toStringSK sk)^"]"
  let printStack toStringEl stack =
        let open Printf in
        printf "\n%s\n\n" (toString toStringEl stack)
end:STACK);;
```

Applichiamo al Tipo Astratto Stack: Calcola

```
module Stack =
 (struct
  exception Error;;
  type 'a sk = E | SK of 'a sk * 'a
  type 'a stack = M of int * ('a sk)
  let create_stack = fun () -> M(0,E)
  let push (M(n,sk)) x =
        if n=100 then raise(Error)
        else M(n+1,SK(sk,x))
  let top (M(n,sk)) = match sk with
                E -> raise Error
                | SK(sk,v) -> v
  let pop (M(n,sk)) = match sk with
                E -> raise Error
                | SK(sk,v) -> M(n-1,sk)
  let toString toString (M(n,sk)) =
        let rec toStringSK x = match x with
                E -> ""
                | SK(sk,v) -> (let left = toStringSK sk in
                               if left = "" then toString v
                               else left^","^(toString v))
        in  "["^(toStringSK sk)^"]"
  let printStack toStringEl stack =
        let open Printf in
        printf "\n%s\n\n" (toString toStringEl stack)
end:STACK);;


(*
# open Stack;;
# let stack1 = push(push(push (create_stack())78)10)93;;
val stack1 : int Stack.stack = <abstr>
# let toStringI:int ->string = fun n -> let open Printf in
    sprintf "%d" n;;
val toStringI : int -> string = <fun>
# toString toStringI stack1;;
- : string = "[78,10,93]"
# printStack toStringI stack1;;

[78,10,93]
```

Esercizi di Laboratorio del 6/4

- **Eccezioni in Ocaml**.
  - **exception**
    introduce Tipi Algebrici
    `exception Err1 of string * string;;`
    i cui valori $\mathcal{D}^{\texttt{Err1}} \equiv \{\texttt{Err1(s1, s2)} | \forall \texttt{s1, s2} \in \mathcal{D}^{\texttt{string}}\} \sqsubset \mathcal{D}^{\texttt{exn}}$
  - **raise**
    introduce valori "exn" che interrompono la normale computazione
    `raise (Err1("sum", "illegal..."))`
  - **try**
    cattura alcune eccezioni e le risolve o risolleva
    `try sum d1 with (Err1("sum",_))-> v1 | ... | ...-> ...`

- **Particolari Anomalie**
  - Trattabili con un "valore aggiuntivo" al proprio tipo di dato
    `type 'a option = Some of 'a | None"`
  - Possono condurre a migliori forme di programma
  - e di controllo e soluzione dell'anomalia

- Attività da svolgere su "EsercizioLab3-2-5bis.ml": Definizione di sum ...

Applichiamo a sum su Dcl: Definiamo

```
(*************************************************************************)
(* Eserizio5a:                                                           *)
(* Scrivere una funzione che calcola la somma degli interi che oc-       *)
(* corrono nelle dichiarazioni di variabile contenute in una di-         *)
(* chiarazione SmallC.                                                   *)
(* Completare testo per il caso di nessuna occorenza.                    *)
(*************************************************************************)
```

```
(* soluzione *)
(* a: In caso di nessuna occorrenza restituiamo il valore 0 *)
let rec sum1 d = match d with
    Var (_,n) -> n
    | SeqDcl (d1,d2) -> (sum1 d1) + (sum1 d2)
    | _ -> 0
    ;;

(* INSODDISFATTI                                                         *)
(* Questa soluzione tratta nello stesso modo stati molto diversi         *)
(* quali: occorrenza di una variabile inizializzata a 0 e nessu-        *)
(* na occorrenza di variabili. Vogliamo distinguere.                     *)
```

Applichiamo a sum su Dcl: Eccezioni vs. Options

```
(* a meno di usare: try CriticalExp with p1 -> r1 |...|pn -> rn     *)
(*  ma qui e` faticoso perche` ..... vediamo cosa dovremmo scrivere *)

let rec sum d = match d with
    Var (_,n) -> n
    | SeqDcl (d1,d2) ->
         let v1 = try sum d1 with (Err1 _) -> -10 in
         if v1 = (-10) then sum d2
         else (let v2 = try sum d2 with (Err1 _) -> -10 in
               if v2 = (-10) then v1 else v1+v2)
    | _ -> raise (Err1("sum","No variabile in"))
    ;;



(* Una classe di Valori Algebrici con cui estendere ogni tipo 't   *)
(* 't option = None | Some of 't                                    *)
(* Utilizzabili per gestire eccezioni catturabili e trattabili     *)
(* come facciamo sotto, completando i puntini sopra,               *)

let rec sum3 d = match d with
    Var (_,n) -> Some n
    | SeqDcl (d1,d2) -> let (v1,v2) = (sum3 d1,sum3 d2) in
                (match (v1,v2) with
                    | (Some n1,Some n2) -> Some(n1+n2)
                    | (None,_) -> v2
                    | _ -> v1)
    | _ -> None
    ;;
```

Tipi Astratti: Implementazione

- Se e Cosa calcola il codice sotto?

```
module type CNR =
 sig
  type counter
  val max: int
  val mk: unit -> counter
  val reset: counter -> counter
  val get: counter -> int
  val inc: counter -> counter
end;;

module Cnr1 =
(struct
  type counter = int
  let max = 100
  let init = 25
  let mk() = init
  let reset _ = mk()
  let get c = c
  let inc c = c+1
end:CNR);;

type counter = int;;
let (c0:counter) = 0;;
let (c1:Cnr1.counter) = Cnr1.mk();;
let c01 = c0+1;;
let (inc:counter->counter) = fun c -> c+1;;
let c02 = inc(inc(inc c01));;
let c11 = c1 + 1;;
let c12 = Cnr1.inc(Cnr1.inc(Cnr1.inc(c1)));;
Cnr1.get c11 + c02 + get c12;;
```

- Com'è la struttura di Activation Records che ne controlla la computazione?
- Come l'implementazione garantisce l'inaccessibilità di stato e codice operazioni

Tipi Astratti: Implementazione. Funzioni up e down

- Il codice interno introduce le funzioni:
  - up : concrete → abstract – opacizza stato/valore concreto
  - down : abstract → concrete – mostra stato/valore concreto
  - $\forall v, \texttt{down}(\texttt{up}(v)) = v$ – Proprietà fondamentale

```
module Cnr1 =
(struct
  type counter = int
  let max = 100
  let init = 25
  let mk() = init
  let reset _ = mk()
  let get c = c
  let inc c = c+1
end:CNR);;
```

```
(* internal code with up and down functions
module Cnr1 =
(struct
  type counter = int
  let max = 100
  let init = 25
  let mk() = up(init)
  let reset _ = mk()
  let get c = down(c)
  let inc c = up(down(c)+1)
end:CNR);;
*)
```

- Analisi Statica controlla corrispondenza tra modulo CNR e modulo Cnr1
- Analisi Statica trova degli ERRORI - Ma noi "forziamo" l'esecuzione
- Si crea una struttura nonlocali (CS) e frame locali per il codice di Cnr1

——	Cnr1
CS	...
counter	$\texttt{counter}_{\texttt{def}}$
max	100
init	25
mk	$\texttt{mk}_{\texttt{code}}$
...	...
inc	$\texttt{inc1}_{\texttt{code}}$

Tipi Astratti: Implementazione. Esecuzione e AR

- Vediamo lo Stack degli AR durante l'esecuzione di:

```
module Cnr1 =
(struct
  type counter = int
  ...
end:CNR);;

type counter = int;;
let (c0:counter) = 0;;
let (c1:Cnr1.counter) = Cnr1.mk();;
let c01 = c0+1;;
let (inc:counter->counter) = fun c -> c+1;;
let c02 = inc(inc(inc c01));;
let c11 = c1 + 1;;
let c12 = Cnr1.inc(Cnr1.inc(Cnr1.inc(c1)));;
Cnr1.get c11 + c02 + get c12;;
```

- Sotto, lo Stack (a sinistra): Contiene 1 solo AR con accesso AR0
- AR0 è inserito come nonlocale (vedi CS) del Modulo Cnr1
- Notare assenza di valori modificabili nei denotabili (Linguaggi Funzionali)

——	AR0
CS	...
CD	...
Cnr1	Cnr1
counter	$\texttt{counter}_{\texttt{def}}$
c0	0
c1	
c01	
inc	$\texttt{inc2}_{\texttt{code}}$
c02	
c11	
c12	
Cnr1.mk()	

——	Cnr1
CS	AR0
counter	$\texttt{counter}_{\texttt{def}}$
max	100
init	25
mk	$\texttt{mk}_{\texttt{code}}$
...	...
inc	$\texttt{inc1}_{\texttt{code}}$

Tipi Astratti: Implementazione - Esecuzione di Cnr1.mk()

- La valutazione di Cnr1.mk() negli Intermedi di AR0 conduce alla
- Creazione di AR1 (dove va aggiunto il componete per l'indirizzo del valore di ritorno)

——	AR0
CS	...
CD	...
Cnr1	Cnr1
counter	$\texttt{counter}_{\texttt{def}}$
c0	0
c1	
c01	
inc	$\texttt{inc2}_{\texttt{code}}$
c02	
c11	
c12	
Cnr1.mk()	

——	AR1
CS	AR0
CD	AR0
counter	$\texttt{counter}_{\texttt{def}}$
max	100
init	25
mk	$\texttt{mk}_{\texttt{code}}$
...	...
inc	$\texttt{inc1}_{\texttt{code}}$
init up(init)	25 up(25)

——	Cnr1
CS	AR0
counter	$\texttt{counter}_{\texttt{def}}$
max	100
init	25
mk	$\texttt{mk}_{\texttt{code}}$
...	...
inc	$\texttt{inc1}_{\texttt{code}}$

Tipi Astratti: Implementazione - Ritorno dall'invoc. di mk

- AR0 è modificato con il valore calcolato dall'invocazione
- AR1 è rimosso

——	AR0
CS	...
CD	...
Cnr1	Cnr1
counter	$\texttt{counter}_{\texttt{def}}$
c0	0
c1	up(25)
c01	
inc	$\texttt{inc2}_{\texttt{code}}$
c02	
c11	
c12	
c0+1	

——	Cnr1
CS	AR0
counter	$\texttt{counter}_{\texttt{def}}$
max	100
init	25
mk	$\texttt{mk}_{\texttt{code}}$
...	...
inc	$\texttt{inc1}_{\texttt{code}}$

- Continuiamo con le successive valutazioni delle espressioni nei binding locali
- Interessante la valutazione dell'espressione c1+1 nel binding di c11

Tipi Astratti: Implementazione - binding di c01

- binding di c01 e valutazione dell'espressione c0+1

——	AR0
CS	...
CD	...
Cnr1	Cnr1
counter	$\texttt{counter}_{\texttt{def}}$
c0	0
c1	up(25)
c01	1
inc	$\texttt{inc2}_{\texttt{code}}$
c02	
c11	
c12	
c0+1	1

——	Cnr1
CS	AR0
counter	$\texttt{counter}_{\texttt{def}}$
max	100
init	25
mk	$\texttt{mk}_{\texttt{code}}$
...	...
inc	$\texttt{inc1}_{\texttt{code}}$

Tipi Astratti: Implementazione - binding di c02

- binding di c02 e valutazione dell'espressione inc c01

——	AR0
CS	...
CD	...
Cnr1	Cnr1
counter	$\texttt{counter}_{\texttt{def}}$
c0	0
c1	up(25)
c01	1
inc	$\texttt{inc2}_{\texttt{code}}$
c02	
c11	
c12	
inc(inc(inc c01)) inc(inc c01) inc c01 c01	1

——	AR2
CS	AR1
CD	AR1
c	1
c+1	2

——	Cnr1
CS	AR0
counter	$\texttt{counter}_{\texttt{def}}$
max	100
init	25
mk	$\texttt{mk}_{\texttt{code}}$
...	...
inc	$\texttt{inc1}_{\texttt{code}}$

Tipi Astratti: Implementazione - binding di c02 /2

- binding di c02 e valutazione dell'espressione inc(inc c01))

——	AR0
CS	...
CD	...
Cnr1	Cnr1
counter	$\texttt{counter}_{\texttt{def}}$
c0	0
c1	up(25)
c01	1
inc	$\texttt{inc2}_{\texttt{code}}$
c02	
c11	
c12	
inc(inc(inc c01)) inc(inc c01) inc c01	2

——	AR3
CS	AR1
CD	AR1
c	2
c+1	3

——	Cnr1
CS	AR0
counter	$\texttt{counter}_{\texttt{def}}$
max	100
init	25
mk	$\texttt{mk}_{\texttt{code}}$
...	...
inc	$\texttt{inc1}_{\texttt{code}}$

Tipi Astratti: Implementazione - binding di c02 /3

- binding di c02 e valutazione dell'espressione inc(inc(inc c01))

——	AR0
CS	...
CD	...
Cnr1	Cnr1
counter	$\texttt{counter}_{\texttt{def}}$
c0	0
c1	up(25)
c01	1
inc	$\texttt{inc2}_{\texttt{code}}$
c02	
c11	
c12	
inc(inc(inc c01)) inc(inc c01)	3

——	AR4
CS	AR1
CD	AR1
c	3
c+1	4

——	Cnr1
CS	AR0
counter	$\texttt{counter}_{\texttt{def}}$
max	100
init	25
mk	$\texttt{mk}_{\texttt{code}}$
...	...
inc	$\texttt{inc1}_{\texttt{code}}$

Tipi Astratti: Implementazione - binding di c02 /3

- binding di c02

——	AR0
CS	...
CD	...
Cnr1	Cnr1
counter	$\texttt{counter}_{\texttt{def}}$
c0	0
c1	up(25)
c01	1
inc	$\texttt{inc2}_{\texttt{code}}$
c02	4
c11	
c12	
inc(inc(inc c01))	4

——	Cnr1
CS	AR0
counter	$\texttt{counter}_{\texttt{def}}$
max	100
init	25
mk	$\texttt{mk}_{\texttt{code}}$
...	...
inc	$\texttt{inc1}_{\texttt{code}}$

- Completare con la valutazione dei bindings per c11 e c12
- Interessante la valutazione dell'espressione c1+1 nel binding di c11