Laboratorio 2 - Small20. Sintassi Astratta e Concreta: Scrittura, Rappresentazione e Stampa dei Programmi

Sommario: 03 Aprile, 2020

- Strumenti: OCaml - Cosa abbiamo visto.
- Tipi Algebrici: Perchè usarli nello sviluppo di Small20
- Vediamo il Listing LambdaAST del 31/03/20
- Valori Algebrici, Pattern e Pattern-Matching in OCaml
- Sintassi Astratta: Scrittura di Programmi Small20 mediante Generazione dei loro AST.
- Sintassi Concreta: Stampa di AST in S. Concreta di Small20
- Attività: 3 Esercizi
- Conclusioni: Alla prossima sessione

Strumenti. OCaml: Cosa abbiamo visto sull'Esecutore. Operiamo a Top Evaluation Level (Interpretation Cycle)

- Eseguiamo ocaml in una shell Unix ed entriamo in Top Evaluation Level (TEL)
- Ambiente Interattivo: Un blocco virtuale, Scope Statico, Un ciclo:
  - Input: # Termine T (i.e. Espressione, Programma) seguito da ;;
  - Valutazione di T
    - T è analizzato dal FrontEnd OCaml
    - T è Compilato in Bytecode OCaml, ottenendo T_B
    - T_B è Interpretato e valutato ad un valore V
  - output: *nome*: *tipo* = V
- Il ciclo input-valutazione-output si ripete:
  - Fintantochè T è OCaml-corretto e valutabile ad un valore;
  - Valutazioni Non Terminanti per termini divergenti;
  - control-c per interrompere esecuzione e tornare alla fase input del ciclo;
  - exit(n);; – con n intero per chiudere il ciclo ed uscire dall'Ambiente.

Strumenti. OCaml: Cosa abbiamo visto sul Linguaggio. Programmare in OCaml

- OCaml è un Linguaggio MultiParadigma:
  - sottoLinguaggio Funzionale (puro): Calcolo Funzionale
  - sottoLinguaggio Imperativo: stato, modificabili e sequenza;
  - sottoLinguaggio Object Oriented: Classi, Oggetti ed Ereditarietà;
  - integrati in accordo a specifiche regole di composizione dei costrutti
- Nel Laboratorio useremo solo il sottoLinguaggio Funzionale
- Abbiamo visto (in Laboratorio1 del 27/03, e a Lezione-Esercitazione del 31/03)
  - Operatori primitivi e binari, infissi usabili anche in forma prefissa.
  - Naming per valori: Costrutto "let ide = T"
  - Valori Primitivi Atomici per tipi: int, bool, char.
  - Linguaggio fortemente tipato: Ogni termine ha uno e un solo Tipo.
  - Il Tipo determina le operazioni applicabili ad un valore e l'uso ammesso.
  - Valori Primitivi Strutturati: string, tuple e liste.
  - Valori Funzione: Astrazioni fun, Naming, Monadiche e Curried.
  - Ricorsione e operatore rec
  - Naming per tipi: Costrutto "type ide = Y"
  - Tipi Algebrici o Variant: Costruttori di Valori e Unione Disgiunta

Strumenti. OCaml: Cosa dobbiamo ancora vedere.

- Visitare, Accedere e Manipolare Valori e Componenti di Tipi Algebrici
  - Programmare con i Tipi Algebrici.
  - Tipi Algebrici: Costruttori, Pattern
  - Tipi Algebrici: Pattern-Matching nei Linguaggi Applicativi
  - Costrutto **match-with** di OCaml per accesso, visita e manipolazione di valori algebrici
- Valori Funzione: Altre forme di Astrazione.
  - Astrazione function: Definizione per casi

Programmare con i Tipi Algebrici: Perché

- La Domanda: Perchè usare i Tipi Algebrici e Programmare con Essi?
  - I Tipi Algebrici forniscono un costrutto per definire nuovi tipi:
  - Ottenuti da unione (disgiunta) di altri tipi.

    $\texttt{type t} = \texttt{A of } t_1^A * \ldots * t_{n_A}^A \mid \texttt{B of } t_1^B * \ldots * t_{n_B}^B$
  - Banalmente estendibili e/o modificabili.

    $\texttt{type t} = \texttt{A of } t_1^A * \ldots * t_{n_A}^A \mid \texttt{B of } t_1^B * \ldots * t_{n_B}^B \mid \texttt{C of } t_1^C * \ldots * t_{n_C}^C$

    $\texttt{type t} = \texttt{A of } t_1^A * \ldots * t_{n_A}^A \mid \texttt{C of } t_1^C * \ldots * t_{n_C}^C$
  - Adatti a Tipi a valori atomici, strutturati, o misti

    $\texttt{type t} = \texttt{A} \mid \texttt{B} \mid \texttt{C}$

    $\texttt{type t} = \texttt{A of } t_1^A * \ldots * t_{n_A}^A \mid \texttt{C of } t_1^C * \ldots * t_{n_C}^C$

    $\texttt{type t} = \texttt{A of } t_1^A * \ldots * t_{n_A}^A \mid \texttt{C of } t_1^C * \ldots * t_{n_C}^C \mid \texttt{B} \mid \texttt{D}$
  - Perfetti per esprimere AST di Sintassi Astratta:
    - Ad ogni Costrutto un Costruttore esibente la specifica Stuttura ad Albero del Costrutto

      $\texttt{exp} = \texttt{Num} \mid \texttt{IDE} \mid \texttt{Plus of exp} * \texttt{expE} \mid \ldots$
    - Esaminiamo l'implementazione data in LAST per la Sintassi Astratta del λ-Calcolo.
    - Vogliamo procedere nello stesso modo con la Sintassi Astratta di Small20

Programmare con i Tipi Algebrici: LambdAST (31/03/20)

- LASTH.ml introduce i Tipi Algebrici utilizzati per gli AST del λ-Calcolo.
- LASTE.ml introduce funzioni Ocaml per equipaggiare tali tipi con operazioni specifiche per:
  - Costruire ogni AST esprimibile: Operatori con Prefisso 'mk'
  - Identificare la struttura di ogni AST: Operatori con Prefisso 'is'
  - Selezionare i componenti di ogni AST: Operatori con Prefisso 'get'
  - Ottenere una presentazione in Sintassi Astratta di ogni AST: printAST
  - Ottenere la presentazione in Sintassi Concreta di ogni AST: printCRT
- Main.ml mostra un banale uso di tale sistema: Costruzione dell'AST di un λ-termine e stampa della sua presentazione in Sintassi Concreta

Tipi Algebrici: Valori, Pattern e Pattern Matching

- Come visitare, accedere, "modificare", valori e componenti di tipi algebrici?

```
'a list = Cons of 'a * 'a list | Nil
```

  Un tipo algebrico[1] isomorfo[2] al tipo primitivo **'a list** di Ocaml.

- Pattern e Pattern Matching sono gli strumenti che i Linguaggi Funzionali forniscono per operare con valori strutturati, di tipi definiti con costruttori.[3]

- Tipi definiti con costruttori in Ocaml sono:
  + Tipi primitivi: liste, tuple
  + Tipi definiti a programma: Tipi (variante o) Algebrici.

- Valori (Algebrici) e Pattern, sono espressi:
  + Valori: Termini di Costruttori applicati a valori i tipo atteso[4]
  + Pattern: Termini di Costruttori come i Valori ma contenenti anche variabili (libere e non ripetute) al posto di sotto-termini.

---

1 definito in OCaml

2 mappare costruttore Cons su ::, costruttore Nil su []

3 funzioni iniettive che generano valori univocamente definiti dagli argomenti cui sono applicati

4 dalla signature del costruttore

Gli Strumenti: Costrutto match-with di Ocaml

- Valori e Pattern, sono espressi:

  + Valori: Termini di Costruttori applicati a valori i tipo atteso:
    Ad esempio: il termine `t` $\equiv$ `3::5::7::[]` è un valore.

  + Pattern: Termini come i Valori contenenti anche variabili (libere e non ...).
    Ad esempio: il termine `p` $\equiv$ `x::5::z` è un pattern.

- Pattern Matching e . Operazione Linguaggi Applicativi
  - match(t,p) calcola sostituzione ρ contenente i bindings u/x, con u, sottotermine di t, ed x, variabile di p, che rende p identico a t (i.e. t = ρ(p)), se un ρ esiste.
  - Nell'esempio, match(t,p) = [3/x , 7::[]/x]

- Pattern Speciali e Definizione
  match(t,x) = [t/x]
  match(t,_) = [t/?], ambiente con binding per anonima variabile _
  match(c(t1,...,tk),c(p1,...pk)) = $\rho_1 \circ ... \circ \rho_k$
  con ρ_1 = match(t1,p1),...,ρ_k = match(tk,pk)
  $\circ$ = composizione di sostituzioni: Per disgiunti X=x1,..,xk e Y=y1,..,yh, i.e X$\cap$Y = {}
  [u1/x1,..,uk/xk] $\circ$[v1/y1,..,vh/yh] = [u1/x1,..,uk/xk,v1/y1,..,vh/yh]

- Costrutto match in OCaml

  `match t with p1 -> t1 | ... | pn -> tn`

  calcola il valore dell'espressione ρ`(ti)` se esiste minimo `i<=n`, tale che:

  `t = ` ρ`(pi).`

Gli Strumenti: Astrazioni in Ocaml

```
(* Astrazione Ordinaria e costrutto match-with *)
let getIde = fun x ->
    match x with
        | Const v -> v
        | Var v -> v
        | Abs(v,_) -> v
        | _ -> raise Error
        ;;


let getIde x = match x with
        | Const v -> v
        | Var v -> v
        | Abs(v,_) -> v
        | _ -> raise Error
        ;;


(* Astrazione function: Definizione per casi *)
let getIde = function
        | Const v -> v
        | Var v -> v
        | Abs(v,_) -> v
        | _ -> raise Error
        ;;
```

Sintassi Astratta di Small20: Una grammatica di alberi AT

Type ::= [int] | [bool] | [void]
| [arr] Type Num | [mut] Type | [terr]
| [abs] Type TypeSeq
TypeSeq ::= Type [x] Type | Type | [e]

Dcl ::= [const] Type Ide Exp | [var] Type Ide Exp
| ~~[constN] Type Ide~~ | [varN] Type Ide
| [array] Type Ide Num

Exp ::= [val] Ide | Num | Bool | Ide [↑] Exp | $[-_1]$Exp
| Exp [+] Exp | Exp [−] Exp | Exp [∗] Exp | Exp [div] Exp
| Exp [==] Exp | Exp [<] Exp | Exp [>] Exp
| [not] Exp | Exp [or] Exp | Exp [and] Exp
| Exp [=] Exp | [emptyE]

Cmd ::= Cmd [seqC] Cmd
| [ifE] Exp Cmd Cmd | [ifT] Exp Cmd
| [for] Exp Exp Exp Cmd

Stm ::= Dcl | Exp | Cmd | Stm [seqM] Stm | [emptyStm]

Prog ::= [prog] Ide Stm

Una Sintassi Concreta di Small20: Una grammatica incompleta

`Type` ::= `Simple` | `void` | `Simple` [`Num`]
`Simple` ::= `int` | `bool`

Sintassi Concreta: Il Programma P0 di SmallC*del*2019

- Il programma P0 scritto nella Sintassi Concreta di SmallC dell'edizione 2019.
- Indentazioni, Spazi, e Nuova Linea possono essere arbitrariamente scelte (dallo autore) come nel Nuova Linea prima del simbolo "else" .

```
{
 x = 7;
 var y = 10;
 var z;
 A[12];
 y = x+y;
 if (x+y < 15) z = y;
 else z = 0;
 while z<12 {
        A[z] = y+z;
        z = z+1
        }
 }
```

Front-End di SMallC del 2019: Scrittura di P0

- Il programma P0 è scritto direttamente utilizzando le funzioni per la costruzione di AST del Front-End dell'interprete SMallC.

- In figura una sessione OCaml in cui usiamo le funzioni del Front-End SMallC prima per scrivere P0 in Sintassi Astratta e poi, per ottenere l'AST di P0.

```
# let d1 = Const("x",7) in
let d2 = Var("y",10) in
let d3 = VarN "z" in
let d4 = Array("A",12) in
let dd = SeqDcl(d1,SeqDcl(d2,SeqDcl(d3,d4))) in
let c1 = UpdVar("y",Plus(Val "x",Val "y")) in
let e21 = LT (Plus(Val "x",Val "y"),Num 15) in
let c22 = UpdVar("z",Val "y") in
let c23 = UpdVar("z",Num 0) in
let c2 = IfE(e21,c22,c23) in
let e31 = LT (Val "z", Num 12) in
let c321 = UpArray("A",Val "z",Plus(Val "y",Val "z")) in
let c322 = UpdVar("z",Plus(Val "z",Num 1)) in
let c32 = SeqCmd(c321,c322) in
let c3 = While(e31,c32) in
let cc = SeqCmd(c1,SeqCmd(c2,c3)) in
let p0 = Program(dd,cc) in
p0;;
- : prog =
Program
 (SeqDcl (Const ("x", 7),
   SeqDcl (Var ("y", 10), SeqDcl (VarN "z", Array ("A", 12)))),
 SeqCmd (UpdVar ("y", Plus (Val "x", Val "y")),
  SeqCmd
   (IfE (LT (Plus (Val "x", Val "y"), Num 15), UpdVar ("z", Val "y"),
     UpdVar ("z", Num 0)),
   While (LT (Val "z", Num 12),
    SeqCmd (UpArray ("A", Val "z", Plus (Val "y", Val "z")),
     UpdVar ("z", Plus (Val "z", Num 1)))))))
```

SMallC del 2019: Stampa del Programma P0

- Il programma P0 è scritto direttamente utilizzando le funzioni per la costruzione di AST del Front-End dell'interprete SMallC.

- printProg converte lo AST di P0 in una presentazione di P0 in Sintassi Astratta.

```
p0;;
- : prog =
Program
 (SeqDcl (Const ("x", 7),
   SeqDcl (Var ("y", 10), SeqDcl (VarN "z", Array ("A", 12)))),
 SeqCmd (UpdVar ("y", Plus (Val "x", Val "y")),
  SeqCmd
   (IfE (LT (Plus (Val "x", Val "y"), Num 15), UpdVar ("z", Val "y"),
     UpdVar ("z", Num 0)),
   While (LT (Val "z", Num 12),
    SeqCmd (UpArray ("A", Val "z", Plus (Val "y", Val "z")),
     UpdVar ("z", Plus (Val "z", Num 1)))))))
# printProg p0;;

{
 x = 7;
 var y = 10;
 var z;
 A[12];
 y = (x + y);
 if ((x + y) < 15) {
      z = y;
      }
 else  z = 0;
 while (z < 12) {
         A[z] = (y + z);
         z = (z + 1);
         }
}
```

SMall20: 4 Esercizi

Esercizio (1)

Giustificare la non corrispondenza tra la Sintassi Astratta e la Sintassi Concreta dei tipi. In particolare, a quale costrutto della Sintassi Concreta corrispondono gli alberi costruiti con:

1) `[terr]`
2) `[mut]`
3) `[abs]`

Esercizio (2)

Con quali definizioni Ocaml (del listing Small20 distibuito oggi) sono espressi gli alberi di esercizio1?

1)
2)
3)

Esercizio (3)

Una non corrispondenza tra Sintassi Astratta di Small20, nei lucidi di oggi, e Tipi Algebrici, nel listing di oggi, evidenzia un errore nelle dichiarazioni. Si dica quale sia a)la non corrispondenza e quale sia b)l'errore

a)
b)

Esercizio (4)

Completare il codice relativo a `toStringTye` *e* `toStringTseq`.

OCaml: Uso nel Laboratorio

- **OCamlIde.**
  (a) Integraded Development Environment per OCaml: Ne esistono diversi.
  (b) Offrono un sistema di Editing-Esecuzione-Mofifica-Versionamento
  (c) Non li useremo: Li suppliremo in modo manuale.

- **SmallC.ml.**
  (a) file residente sulla directory di lavoro, **WAD**, per tutta l'intera attività di Sviluppo;
  (b) distribuito con un Contenuto Iniziale, e da caricare oggi, sulla propria WAD;

- **Ad ogni sessione di lavoro.**
  (a) Small20.ml è copiato e la copia rinominata in SmallCxxx.ml, dove xxx è un progressivo a partire da 0;
  (b) Ocaml è avviato da WAD;
  (c) Caricamento di Small20.ml: `#use "Small20.ml";;`

- **Editing del Codice.**
  - TEL legge ed analizza carattere per carattere editato dopo il prompt #
  - Nessuna possibilità di correzione in linea;
    (a) tenere aperta una finestra di editing su Small20.ml;
    (b) editare il proprio codice su Small20.ml;
    (c) per l'interpretazione di quanto editato procedere con 1 o con 2:
      (1) copy and past del codice editato;
      (2) ri-caricamento del file.

SMall20: Al termine di ogni sessione

- Riportare nel file Small20.ml ogni progresso nel codice sviluppato durante la sessione.

- Nelle 2 prossime sessioni:

  - Sintassi: Completa nei Tipi Algebrici e nelle Operazioni [toString e print] per la presentazione AST in Sintassi Concreta.

  - Formalizzeremo la Semantica SOS delle dichiarazioni e degli Statements;

  - Vedremo un'implementazione dello Stato;

  - Implementeremo la Funzione Semantica per l'interpretazione [decodifica e modifica_dello_stato] delle dichiarazioni di Small20