Laboratorio 3
(SmallC: Alberi Astratti con Tipi Algebrici in OCaml)

Sommario: 6 aprile, 2018 -

- Ocaml: Tipi Algebrici o Concreti.
- Sintassi Astratta: Esprimiamo Dcl con Tipi Algebrici OCaml.
- Programmare con Tipi Algebrici: Pattern-Matching.
- Matching nelle operazioni di riconoscimento, visita e selezione.
- Tests di sviluppo
- SmallC: Lo Stato – Definizione e Implementazione.
- Ambiente e Memoria e Stato con Tipi Algebrici

Tipi Algebrici o Concreti

Definition (Tipi di Dato Algebrico)

I Tipi di Dato Algebrico sono collezioni di valori (anche strutturati), detti "termini", definiti per iniezione a partire da un insieme finito $\mathtt{G} = \{\mathtt{g_i} : \mathtt{T_{i_1}} \times ... \times \mathtt{T_{i_k}} \rightarrow \mathtt{T_G}\}$ di funzioni iniettive, dette "costruttori", di nome $\mathtt{g_i}$, e segnatura $\mathtt{T_{i_1}} \times ... \times \mathtt{T_{i_k}} \rightarrow \mathtt{T_G}$ dati, dove $\mathtt{k} \geq 0$ e $\mathtt{T_G}$ sia il tipo dei valori definiti con $\mathtt{G}$. Allora l'insieme di tali termini è

$$\mathcal{D}^{\mathtt{T_G}} \equiv \{\mathtt{g_i}(\mathtt{v_{i_1}}, ..., \mathtt{v_{i_k}}) \mid \mathtt{v_{i_j}} \in \mathcal{D}^{\mathtt{T_{i_j}}}\},$$

dove abbiamo indicato con $\mathcal{D}^{\mathtt{T}}$ l'insieme dei valori di tipo $\mathtt{T}$.

Proposition (Presentazione e Identità Sintattica)

Presentazione. *Un valore algebrico, o termine, ha sempre forma* $\mathtt{g(v1, ..., vk)}$, *dove* $\mathtt{g}$ *sia un costruttore di arità* $\mathtt{k} \geq 0$ *e* $\mathtt{v1, \ldots, vk}$ *valori di tipo atteso dalla segnatura di* $\mathtt{g}$.
Identità Sintattica $\equiv$. *Siano* $\mathtt{g(v1, ..., vn)}$ *e* $\mathtt{g'(v1', ..., vm')}$ *due valori di uno stesso tipo di dato algebrico. Allora,*

$$\mathtt{g(v1, ..., vn)} = \mathtt{g'(v1', ..., vm')} \quad \textit{sse } \mathtt{g} \equiv \mathtt{g'} \wedge \mathtt{n} = \mathtt{m} \wedge \mathtt{v1} = \mathtt{v1'} \wedge ... \wedge \mathtt{vn} = \mathtt{vm'}$$

- In Ocaml, i tipi algebrici sono introdotti con dichiarazioni di tipo della forma:

$$\begin{aligned} \mathtt{type\ T_G} = {} & \mathtt{g_1\ of\ (Texp_{1,1} \star ... \star Texp_{1,k_1})} \\ & \mid ... \\ & \mid \mathtt{g_n\ of\ (Texp_{n,1} \star ... \star Texpt_{n,k_n})} \end{aligned}$$

Dove: - ...

Ocaml: Tipi Algebrici o Concreti

- In Ocaml, i tipi algebrici sono introdotti con dichiarazioni di tipo della forma:

$$\begin{aligned}\texttt{type } \mathtt{T_G} = \mathtt{g_1} \texttt{ of } (\mathtt{Texp_{1,1}} \star \ldots \star \mathtt{Texp_{1,k_1}}) \\ \mid \ldots \\ \mid \mathtt{g_n} \texttt{ of } (\mathtt{Texp_{n,1}} \star \ldots \star \mathtt{Texpt_{n,k_n}})\end{aligned}$$

  - **Dove:**
    - $\mathtt{T_G}$ *è il nome del tipo e deve iniziare con un carattere* **minuscolo**
    - $\mathtt{g_1}, .., \mathtt{g_n}$ *sono i nomi dei costruttori e hanno primo carattere* **maiuscolo**
    - of $(\mathtt{Texp_{i,1}} \star \ldots \star \mathtt{Texp_{i,k_i}})$ *è omesso quando* $\mathtt{k_i} = 0$, *i.e.* $\mathtt{g_i}$ *ha arità 0*
    - of $(\mathtt{Texp_{i,1}} \star \ldots \star \mathtt{Texp_{i,k_i}})$ *definisce la tupla dei tipi degli argomenti*
    - $\mathtt{Texp_{i,j}}$ è una qualunque espressione di tipo Ocaml
    - l'ordine dei costruttori è inessenziale
    - Nel caso di tipi algebrici polimorfi:
      - $\mathtt{T_G}$ ha prefisso la lista $\texttt{'a}_1, \ldots, \texttt{'a}_r$ delle variabili generiche
      - $\mathtt{Texp_{i,j}}$ può contenere variabili in $\{\texttt{'a}_1, \ldots, \texttt{'a}_r\}$ come libere
  - **Ricorsiva** $\mathtt{T_G}$ può occorrere come tipo in $\mathtt{Texp_{i,k_i}}$
  - **Allocazione Dinamica** Ogni valutazione di $\mathtt{g_i}(\mathtt{v_{i,1}}, \ldots, \mathtt{v_{i,k_i}})$ alloca dinamicamente una struttura di costruttore $\mathtt{g_i}$ avente $\mathtt{k_i}$ componenti di valore $\mathtt{v_{i,1}}, \ldots, \mathtt{v_{i,k_i}}$ in tale ordine.

Sintassi Astratta: Le categorie Dcl, Exp, Cmd, Prog.

- Definiamo gli alberi astratti delle categorie a partire da Dcl.
- Usiamo Tipi algebrici di Ocaml.
- Usiamo il file "oggi.ml" allegato come listing:
- Carichiamo un'ultima volta il file nell'interprete interattivo di Ocaml:
  - Controllando che le ultime definizioni inserite nel file non abbiano errori
  - Correggendo gli eventuali errori
  - Aggiungendo al file codice per il "test del comportamento atteso"

- Test di comportamento atteso: Definizione di codice che quando eseguito verifica il comportamento di strutture critiche del sistema che abbiamo definito.
  - Dopo aver definito gli Alberi Astratti di SmallC
  - Dovremmo scrivere codice per esprimere, riconoscere, visitare e selezionare i vari componenti, i.e. sotto-alberi e foglie
  - Questo richiede di sapere programmare con valori di Tipi Algebrici.

Sintassi Astratta di SmallC

$$\texttt{Dcl} ::= [\texttt{const}]\ \texttt{Ide}\ \texttt{Num} \mid [\texttt{var}]\ \texttt{Ide}\ \texttt{Num} \mid [\texttt{array}]\ \texttt{Ide}\ \texttt{Num} \mid \texttt{Dcl}\ [\texttt{seqD}]\ \texttt{Dcl}$$

$$\begin{aligned}\texttt{Exp} ::= {} & [\texttt{val}]\ \texttt{Ide} \mid \texttt{Num} \mid \texttt{Ide}\ [\uparrow]\ \texttt{Num} \\ & \mid \texttt{Exp}\ [+]\ \texttt{Exp} \mid \texttt{Exp}\ [-]\ \texttt{Exp} \mid \texttt{Exp}\ [*]\ \texttt{Exp} \mid \texttt{Exp}\ [\texttt{div}]\ \texttt{Exp} \\ & \mid \texttt{Exp}\ [==]\ \texttt{Exp} \mid \texttt{Exp}\ [<]\ \texttt{Exp} \mid \texttt{Exp}\ [>]\ \texttt{Exp} \\ & \mid [\texttt{not}]\ \texttt{Exp} \mid \texttt{Exp}\ [\texttt{or}]\ \texttt{Exp} \mid \texttt{Exp}\ [\texttt{and}]\ \texttt{Exp}\end{aligned}$$

$$\begin{aligned}\texttt{Cmd} ::= {} & \texttt{Ide}\ [=]\ \texttt{Exp} \mid \texttt{Ide}\ \texttt{Exp}\ [\leftarrow]\ \texttt{Exp} \mid \texttt{Cmd}\ [\texttt{seqC}]\ \texttt{Cmd} \\ & \mid [\texttt{ifte}]\ \texttt{Exp}\ \texttt{Cmd}\ \texttt{Cmd} \mid [\texttt{if}]\ \texttt{Exp}\ \texttt{Cmd} \mid [\texttt{while}]\ \texttt{Exp}\ \texttt{Cmd}\end{aligned}$$

$$\texttt{Prog} ::= [\texttt{prog}]\ \texttt{Dcl}\ \texttt{Cmd} \mid [\texttt{progN}]\ \texttt{Cmd}$$

where:
– $\uparrow$ costruttore termine (AT) accesso valore componente array;
– $\leftarrow$ costruttore termine modifica valore componente array;

Esercizio (1)

Scrivere in Oggi.ml una definizione di Dcl con Tipi Algebrici. Caricare sotto OCaml e correggere eventuali errori.
Una volta finito procedere con le altre categorie, fino al completamento.

Programmare con Tipi Algebrici o Concreti

- Programmare con Tipi Algebrici richiede non solo saper:
  - **Definire tipi algebrici.**
    Esempio, alberi `('a,'b) gTree` in Ocaml
    `type('a,'b)gTree = E | L of 'a | R of ('b * ('a,'b) gTree list)`
  - **Esprimere valori** di un dato tipo algebrico.
    Esempio, l'albero `tree`:
    `let tree = R("root1", [L 1; L 2; R("root2", []); E])`
- anche saper:
  - **Visitare la Struttura**
    Esempio: l'albero `tree` è una foglia? Ha più di 1 sottoalbero?
  - **Accedere Componenti**.
    Esempio: se `tree` ha sottoalberi, qual'è il suo primo sottoalbero?
  - **Modificare Componenti**
    Esempio: Ocaml permette tipi algebrici a componenti modificabili
- Operazioni per Visita, Accesso e Modifica (quando ammesso) ottenuti:
  - Meccanismo di Pattern-Matching (Ocaml, e L funzionali)
  - Meccanismi ad hoc

Pattern-Matching

Pattern-Matching è un meccanismo per *visitare, accedere, "modificare"* valori e componenti di tipi algebrici o concreti

- È realizzato mediante:
  - Patterns per dato tipo algebrico
  - Operazione Match per coppie pattern-termine

- Sia $\mathtt{G} = \{\mathtt{g_i} : \mathtt{T_{i_1}} \times ... \times \mathtt{T_{i_k}} \rightarrow \mathtt{T_G}\}$ l'insieme dei costruttori di un Tipo Algebrico $\mathtt{T_G}$, e indichiamo con $\mathtt{X}^{\mathtt{T_G}}$ un insieme delle variabili di tipo $\mathtt{T_G}$
  - **Termini**. $\mathcal{D}^{\mathtt{T_G}} \equiv \{\mathtt{g_i}(\mathtt{v_{i_1}}, ..., \mathtt{v_{i_k}}) \mid \mathtt{v_{i_j}} \in \mathcal{D}^{\mathtt{T_{i_j}}}\}$
    Ad esempio: $\mathtt{R("root1", [L\ 1; L\ 2; R("root2", []); E])}$ è un termine di `('a,'b) gTree`, in Ocaml.
  - **Patterns** $\mathcal{P}^{\mathtt{T_G}} \equiv \{\mathtt{g_i}(\mathtt{v_{i_1}}, ..., \mathtt{v_{i_k}}) \mid \mathtt{v_{i_j}} \in \mathcal{P}^{\mathtt{T_{i_j}}}\} \cup \mathtt{X}^{\mathtt{T_G}}$
    Ad esempio: $\mathtt{R(x, [L\ 1; L\ 2; y; z])}$ è un pattern dove `x` è una variabile di tipo `string`, `y, z` di tipo `('a,'b) gTree`.

- Operazione Match: Dato $\mathtt{T_G}$,
  - $\mathtt{Match} : \mathcal{D}^{\mathtt{T_G}} \times \mathcal{P}^{\mathtt{T_G}} \rightarrow \mathtt{Sostituzione}$
    ....

Pattern-Matching in Ocaml

Pattern-Matching è un meccanismo per *visitare, accedere, "modificare"* valori e componenti di tipi algebrici o concreti

- Operazione Match: Dato $\mathtt{T_G}$,
  - $\mathtt{Match} : \mathcal{D}^{\mathtt{T_G}} \times \mathcal{P}^{\mathtt{T_G}} \rightarrow \mathtt{Sostituzione}$
  - Sostituzione è l'insieme dei legami di variabili-valori che rendono il pattern uguale al termine dato, oppure *fail* se tale insieme non esiste.
    Esempio:
    $\mathtt{Match(R("root1", [L\ 1; L\ 2; R("root2", []); E]), R(x, [L\ 1; L\ 2; y; z]))}$
    calcola la sostituzione $\{\mathtt{x = "root1", y = R("root2", []), z = E}\}$

- Ocaml incorpora l'operazione Match in un costrutto match_with.

- Costrutto match_with è un condizionale multi-way avente la seguente forma:

```
match term with
      | pattern_1 -> exp_1
      | ...
      | pattern_n -> exp_n
```

dove `term` è un valore di un tipo algebrico `T`, e $\mathtt{pattern_1, ..., pattern_n}$ sono `n` patterns per lo stesso tipo `T` che devono fornire una copertura per `T` (ovvero, per ogni termine di `T` deve esistere un pattern nella copertura e una sostituzione per esso che rende il pattern identico al termine)

calcola $\mathtt{exp_i}$ tale che `i` è il più piccolo indice per cui:

$$\mathtt{Match(term, pattern_i)} \neq \mathtt{fail}$$

Pattern-Matching in Ocaml: Esempi

Pattern-Matching è un meccanismo per *visitare, accedere, "modificare"* valori e componenti di tipi algebrici o concreti

- Costrutto match_with è un condizionale multi-way avente la seguente forma:

```
match term with
      | pattern_1 -> exp_1
      | ...
      | pattern_n -> exp_n
```

dove `term` è un valore di un tipo algebrico `T`, e $\texttt{pattern}_1, \ldots, \texttt{pattern}_n$ sono ...
calcola $\texttt{exp}_i$ tale che `i` è il più piccolo indice per cui:

$$\texttt{Match}(\texttt{term}, \texttt{patter}_i) \neq \texttt{fail}$$

Esempio (Calcolo della sostituzione)

Definiamo un tipo per la presentazioni delle sostituzioni:

```
type 'a substs = Sub of 'a | Fail;;
```

ed usiamolo nel match di termini di tipo ('a, 'b)gTree *definito prima.*

```
let tree = R("root1", [L 1; L 2; R("root2", []); E]);;
match tree with R(x, [L 1; L 2; y; z]) -> Sub(x, y, z) | w -> Fail;;
```

Cosa otteniamo in Ocaml?

Esempio (Predicati e Selettori di gTree)

Definiamo i predicati isE, isL, isR per riconoscere i valori delle 3 diverse strutture :

```
let isE tree = match tree with E -> true | _ -> false;;
let isL tree = match tree with L _ -> true | _ -> false;;
let isR tree = match tree with R _ -> true | _ -> false;;
```

Matching nelle op. di riconoscimento, visita e selezione.

Esercizio (2)

Scrivere in Oggi.ml le definizioni dei costruttori:

```
dcl mkArray(ide i, num s);
dcl mkSeqD(dcl l, dcl r); )
```

Si commenti l'utililità delle operazioni definite.

Esercizio (3)

Scrivere in Oggi.ml le definizioni dei predicati:

```
bool isConst(dcl d)
bool isVar(dcl d)
bool isSeqD(dcl d)
```

Si commenti l'utililità delle operazioni definite.

Esercizio (4)

Scrivere in Oggi.ml le definizioni dei selettori:

```
ide getIde(dcl d);
dcl getDclL(dcl d);
```

Si commenti l'utililità delle operazioni definite.

Tests di Sviluppo

- Test di comportamento atteso: Definizione di codice che quando eseguito verifica il comportamento di strutture critiche del sistema che abbiamo definito.
  - Dopo aver definito gli Alberi Astratti di SmallC
  - Dovremmo scrivere codice per esprimere, riconoscere, visitare e selezionare i vari componenti, i.e. sotto-alberi e foglie

Esercizio (5)

Scrivere in Oggi.ml:

(a) una funzione che calcola la somma degli interi che compaiono nelle dichiarazioni di variabile contenute in una dichiarazione SmallC. Completare per in caso di nessuna occorrenza.

(b) una funzione che calcola il numero di identificatori dichiarati in una dichiarazione SmallC.

(c) una funzione che controlla che un identificatore non abbia più dichiarazioni in una dichiarazione SmallC.

(e) una funzione per la stampa di una dichiarazione SmallC in accordo alla Sintassi Astratta data per le dichiarazioni.

(f) Altri tests ....