Laboratorio 4-5
(SmallC: Stato, Semantica SOS per Dcl)

Sommario: 13 aprile, 2018 -

- Sintassi Astratta: Presentazione dei Valori Astratti (Libreria/Modulo Printf do OCaml)
- Esercizi del 6 aprile: Cose da dire?
- Semantica SOS di SmallC. Stato: Definizione e Implementazione

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`

- Attività da svolgere su "oggi.ml": Equipaggiare Dcl (gli altri, poi...) con ...
  - Scrivere `toStringD`, `printD` per fornire come presentazione degli Alberi Astratti una corrispondente sintassi concreta
  - Usiamo operazioni del modulo Printf di OCaml.

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

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 ...

Semantica SOS di SmallC: Lo stato

- La definizione di **Stato**.

  - **Ambiente** per identificatori di costanti e di variabili (valori modificabili)
    - Lo rappresentiamo come sequenza finita di **bindings** separati da ","
      e racchiusi in parentesi quadre:
      $$[\mathtt{Ide}_1/\mathtt{Den}_1, ..., \mathtt{Ide}_k/\mathtt{Den}_k]$$

  - **Memoria** Statica per trattare variabili ed array a componenti modificabili di un solo tipo (interi)
    - Rappresentiamo la Memoria come sequenza finita di **words** separate da "," e racchiuse in parentesi quadre:
      $$[\mathtt{loc}_1 \leftarrow \mathtt{Mv}_1, ..., \mathtt{loc}_k \leftarrow \mathtt{Mv}_k]$$

  - **Stato** è rappresentato una coppia (ρ, μ), indicante Ambiente e Memoria.
    - Lo Stato è denotata con la variabile σ (anche con pedici)

Ocaml: Implementazione dello stato

- Implementazione dello **Stato**.
  - **Ambiente** $[\mathtt{Ide}_1/\mathtt{Den}_1, ..., \mathtt{Ide}_k/\mathtt{Den}_k]$
    Implementazione dell'Ambiente in Ocaml:
    — da dare oggi
  - **Memoria** $[\mathtt{loc}_1 \leftarrow \mathtt{Mv}_1, ..., \mathtt{loc}_k \leftarrow \mathtt{Mv}_k]$
    Implementazione dello Memoria in Ocaml (vedi Listing del 10/4)

```
type loc = Loc of int;;
type mval = Mval of int | Undef;;
(* Store:  Operations and Exceptions *)
let storeSize = 1000;;
```

  - **Stato** (ρ, μ)
    Implementazione in Ocaml:
    — da dare oggi
- Prima di procedere con l'implementazione guadiamo la semantica che ci dice come deve essere fatto l'esecutore che stiamo implementando

Implementazione: Memoria, Ambiente e loro operazioni

Se esaminiamo le transizioni, vediamo le operazioni sulla Memoria e sull'Ambiente richieste per descrivere il comportamento delle dichiarazioni (semantica SOS).

- **Memoria**.
  - **allocate**(μ, **n**): alloca `n words` (per interi) in sequenza
  - **upd**(μ, **loc**, **n**): (alias, $\mu[\texttt{loc} \leftarrow \texttt{n}]$) modifica il valore di una locazione
  - **getStore**(μ, **loc**): (alias, $\mu(\texttt{loc})$) fornisce il valore di una locazione
  - **emptyStore**(): crea uno store iniziale con `words` libere, a valore indefinito
- **Ambiente**.
  - **bind**(ρ, **ide**, **den**): (alias, $[\texttt{ide}/\texttt{den}] \circ \rho$) aggiunge un nuovo binding a ρ.
  - **getEnv**(ρ, **ide**): (alias, $\rho(\texttt{ide})$) valore denotabile di un binding
  - **emptyEnv**(): crea un'ambiente senza bindings

Per l'implementazione vedere il codice OCaml nel listing allegato all'attività di oggi