Laboratorio 3 - SmallC: Semantica SOS
Ambiente, Memoria, Stato, Transizioni delle Dichiarazioni

Sommario: (4 -) 5 aprile, 2019

- Stato, Ambiente e Memoria
- Semantica SOS - Transizioni e Computazione
- Dichiarazioni: Le Transizioni
- Implementazione dello Stato
- OCaml: Blocco ed Eccezioni

SMallC: All'inizio di Ogni Sessione

- Caricare il file SmallC, distribuito per la sessione, sulla WAD.
  Verifica: Caricarlo sul TLE OCaml e Controllare Esecuzione Tests Precedenti.
- Copiarlo in SmallCXXX, con l'indice corretto.
- Seguire la Presentazione delle Attività della Sessione.
- Svogere le Attività modificando il file SmallC distribuito.

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:
  - **Memoria** $[\mathtt{loc}_1 \leftarrow \mathtt{Mv}_1, ..., \mathtt{loc}_k \leftarrow \mathtt{Mv}_k]$
    Implementazione della Memoria in Ocaml: Strutture Ausiliarie

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

  - **Stato** (ρ, μ)
    Implementazione in Ocaml:

- Prima di procedere con l'implementazione guadiamo la semantica che ci dice come devono essere usate nell'esecutore che dobbiamo realizzare

Semantica SOS di SmallC: Transizione e Computazione

- **Transizione**.
  - Esecuzione di un costrutto c nello stato σ della Macchina Astratta
  - La esprimiamo con:
    - $\langle c, \sigma \rangle \rightarrow \sigma'$, indicante ...
    - $\langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$, indicante ...
    - $\langle c_1, \sigma_1 \rangle \rightarrow \langle c'_1, \sigma'_1 \rangle, \cdots, \langle c_k, \sigma_k \rangle \rightarrow \langle c'_k, \sigma'_k \rangle \ / \ \langle c, \sigma \rangle \rightarrow \langle c', \sigma' \rangle$, k-premesse/1-conclusione, indicante ...

- **Compurazione** La sequenza $\sigma_1, ..., \sigma_k, ...$ degli stati effettivamente calcolati dalle transizioni usate nella valutazione/esecuzione del programma.

- **Notazione**.
  - (ρ, μ) stato con ambiente ρ (anche con apici/pedici) e memoria μ (anche con apici/pedici)
  - N (anche con pedici) intero, I (anche con pedici) identificatore, D (anche con pedici) valore denotabile
  - $[I/D] \circ \rho$ crea un nuovo ambiente che estende ρ con il binding [I/D]
  - $\rho(I)$ denotazione del binding di I in ρ, se esiste
  - $\perp_{mem}$ indefinito nei memorizzabili
  - allocate(μ, n) alloca n locazioni libere, in sequenza, a partire da loc, modificando μ in μ' e restituisce (loc, μ')
  - μ(loc) (loc deve essere una locazione giá allocata) restituisce il valore di loc
  - μ[loc $\leftarrow$ n] (loc deve essere giá allocata) modifica il valore di loc con il valore n

Dichiarazioni SmallC: Le Transizioni $\mathtt{Sem_{Dcl}}$

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

- Il sistema di regole $\mathtt{Sem_{Dcl}}$, sotto riportato, definisce il comportamento delle dichiarazioni SmallC durante la computazione dei Programmi.

$$\langle \mathtt{const\ I\ N}, (\rho, \mu)\rangle \rightarrow ([\mathtt{I/N}]\circ\rho, \mu)$$

$$\frac{\mathtt{allocate}(\mu, 1) = (\mathtt{loc}, \mu')}{\langle \mathtt{var\ I\ N}, (\rho, \mu)\rangle \rightarrow ([\mathtt{I/loc}]\circ\rho, \mu'[\mathtt{loc} \leftarrow \mathtt{N}])} \qquad \frac{\mathtt{allocate}(\mu, 1) = (\mathtt{loc}, \mu')}{\langle \mathtt{varN\ I}, (\rho, \mu)\rangle \rightarrow ([\mathtt{I/loc}]\circ\rho, \mu')}$$

$$\frac{\mathtt{allocate}(\mu, \mathtt{N}) = (\mathtt{loc}, \mu')}{\langle \mathtt{array\ I\ N}, (\rho, \mu)\rangle \rightarrow ([\mathtt{I/loc}]\circ\rho, \mu')}$$

$$\frac{}{\langle \mathtt{emptyDCL}, \sigma\rangle \rightarrow \sigma}$$

$$\frac{\langle \mathtt{d}_1, (\rho, \mu)\rangle \rightarrow (\rho', \mu')}{\langle \mathtt{d}_1; \mathtt{d}_2, (\rho, \mu)\rangle \rightarrow \langle \mathtt{d}_2, (\rho', \mu')\rangle}$$

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
  - **getEnv**(ρ, **ide**): (alias, $\rho(\texttt{ide})$) valore denotabile di un binding
  - **emptyEnv**(): crea un'ambiente senza bindings

Implementazione: Rappresentazione della Memoria

La rappresentazione stabilisce le strutture di dati con cui sarà rappresentato ogni valore Memoria e ogni valore Ambiente.

Useremo tipi algebrici

- **Memoria**.

```
type store = Store of int * (loc -> mval);;

(* Funzione di Astrazione ed Invariante di Rappresentazione per store:
1) Presentazione (o forma dei valori): [l1<-Mv1,...,lk<-Mvk] per k>=0
2) AF(c) = [] if c=Store(0,g)
   AF(c) = [l1<-Mv1,...,lk<-Mvk]
               if c=Store(k,g) and (perogni i\in[1..k], li=Loc(i-1) and (g li)=Mvi).
3) I(Store(n,g)) = 0≤n<=storeSize and
        (perogni i\in[1..n], li=Loc(i-1) and (g li) = Mval u and u\in [int]) and
        (perogni i\in[n+1..storeSize], li=Loc(i-1) and (g li) = Undef
*)
```

- **Ambiente**.

Implementazione: Rappresentazione dell'Ambiente

La rappresentazione stabilisce le strutture di dati con cui sarà rappresentato ogni valore Memoria e ogni valore Ambiente.

Useremo tipi algebrici

- **Memoria**.

- **Ambiente**.

```
type env = Env of ide list * (ide -> dval);;

(* Funzione di Astrazione ed Invariante di Rappresentazione per env:
   Modificata rispetto a SmallC: Includiamo anche il dominio ide list
   La presentazione non cambia. Cambiamenti nell'invariante e nelle
   operazioni (confronta definizione e loro uso)
1) Presentazione (o forma dei valori): [id1/D1,...,idk/Dk] per k>=0
2) AF(c) = [] if c = Env(l,g) and (perogni id, g(id) = Unbound)
   AF(c) = [id1/D1,...,idk/Dk]
         if c = Env(l,g) and (g(idi) = Di per i\in[1..k],
                              g(x) = Unbound altrimenti).
3) I(c) = (c = Env(l,g) => (List.mem x l) iff (g(x) != Umbound)
*)
```

OCaml: Nuovi Costrutti che Potremmo Usare

- **Blocco a Struttura di Espressione**. Con la seguente struttura EBNF:
  let `ide = exp {and ide = exp}` **in** `exp`

- **Eccezioni in Ocaml**.
  - **exception**
    Introduce Tipi Algebrici. Esempio
    ```
    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

SmallC: 3 Esercizi

Esercizio (1)

Nello spazio riservato ai "Tests: Semantic Structures", si mostri come dovrebbero essere usate le operazioni sulla memoria per scrivere un'espressione eMem che calcoli:

$[\mathtt{l}_1 \leftarrow \mathtt{m}_1, \mathtt{l}_2 \leftarrow \mathtt{m}_2]$ *dove:* $\mathtt{m}_1$ *sia il memorizzabile 12, ed,* $\mathtt{l}_2$ *sia la prima di 2 locazioni di un array contenente gli interi -18, 11, in tale ordine.*

Nel fare ciò si consideri il comportamento atteso dalle operazioni sulla memoria e non quello effettivo che, come indicato, è errato. Fatto ciò, si mostri:
(a) Quale memoria è calcolata valutando eMem;
(b) Si giustifichi il risultato ottenuto.

Esercizio (2)

Si correggano le definizioni delle operazioni sullo Store, modificando il file SmallC caricato in WAD. Quindi: (a) Si mostri la memoria calcolata dalla valutazione di eMem.

Esercizio (3)

Si correggano le definizioni delle operazioni su Env. Quindi:
(a) si fornisca un'espressione eEnv che calcoli:
$[\mathtt{id}_1/\mathtt{l}_1, \mathtt{id}_2/\mathtt{l}_2]$ *dove:* $\mathtt{id}_1$ *e* $\mathtt{id}_2$ *sia l'identificatore* `p1` *e* `mis`*, mentre le denotazioni siano proprio le locazioni introdotte negli esercizi precedenti.*

SMallC: Al termine di Ogni Sessione

- Riportare nel file SmallC.ml ogni progresso nel codice sviluppato durante la sessione.
- La prossima sessione:
  - Partiremo dalla Sintassi Astratta e dallo Stato
  - Semantica SOS delle dichiarazioni: Che Rivisiteremo Ancora
  - Implementeremo la Funzione Semantica per l'interpretazione [decodifica e modifica_dello_stato] delle dichiarazioni di SmallC