5 Puntatori

Nel linguaggio che stiamo analizzando, frammento del linguaggio `C`, le *variabili* hanno quattro proprietà:

1. un *nome* (identificatore)
2. un *tipo* (`int`, `char`, ecc.)
3. un *valore* nello stato
4. un *indirizzo*

Nome, tipo ed indirizzo vengono fissati una volta per tutte al momento della dichiarazione della variabile, mentre il valore è l'unica proprietà modificabile. In particolare, nel modello semantico, l'indirizzo di una variabile corrisponde alla *locazione* di memoria associata nell'ambiente all'identificatore scelto per la variabile stessa al momento della sua dichiarazione.

Nel linguaggio è anche possibile *denotare* direttamente l'indirizzo di una variabile e *accedere* al suo valore non solo attraverso il suo nome, ma anche attraverso il suo indirizzo. Se `x` è un identificatore di variabile, `&x` denota il suo indirizzo. Inoltre, se `p` è un indirizzo, `*p` è il valore della variabile il cui indirizzo è `p`.

Il linguaggio permette di associare valori di tipo indirizzo alle variabili *puntatori*. Queste ultime sono variabili come tutte le altre, i cui valori però sono indirizzi. La dichiarazione di una variabile puntatore avviene premettendo al suo nome un asterisco. Ad esempio

```
int *p
```

è una dichiarazione di una variabile i cui valori possono essere solo indirizzi di variabili di tipo `int`. Si noti che, data una dichiarazione siffatta, la variabile dichiarata ha, come tutte le altre variabili:

1. un *nome*: `p`
2. un *tipo*: `int *`
3. un *valore*: imprecisato
4. un *indirizzo*: `&p`

I valori che può assumere `p` sono indirizzi di variabili di tipo `int`. Ad esempio, dopo la seguente sequenza di dichiarazioni e assegnamenti

```
int x;
int *p;
x = 10;
p = &x;
```

il valore della variabile puntatore `p` è l'indirizzo della variabile `x`. Diremo che `p` *punta* a `x` e che `x` *è puntata* da `p`. Nel modello che stiamo definendo avremo la seguente situazione:

ρ

`x`	ℓ_x
`p`	ℓ_p

μ

ℓ_x	`10`
ℓ_p	ℓ_x

Si noti che alla locazione ℓ_p non è associato un valore intero bensì una locazione, quella della variabile identificata da `x`.

Veniamo al trattamento formale di quanto visto intuitivamente. Per prima cosa estendiamo la sintassi del linguaggio per trattare i nuovi casi, aggiungendo le seguenti produzioni alle varie categorie sintattiche.

Estensioni sintattiche

```
Type ::=  Ptype
PType ::= Btype *

Exp ::= &Ide | *Ide

Com ::= *Ide = Exp;
```

In secondo luogo dobbiamo estendere il dominio dei frame memoria, in modo che il valore associato ad una locazione possa essere una locazione a sua volta.

Frame memoria estesi

Nel seguito un *frame memoria* sarà una funzione

$$\nu : \texttt{Loc} \to (\texttt{Val} \cup \texttt{Loc})_\perp$$

e di conseguenza l'insieme N dei frame memoria sarà

$$N = \{\nu \mid \nu : \texttt{Loc} \to (\texttt{Val} \cup \texttt{Loc})_\perp\}$$

Funzioni semantiche

Veniamo alle estensioni delle funzioni di interpretazione semantica $\mathcal{S}em_e$ e $\mathcal{S}em_d$ necessarie per trattare i nuovi casi. Prima di farlo, osserviamo che la funzione di interpretazione semantica $\mathcal{S}em_e$ deve ora prevedere anche le locazioni tra i possibili *valori* di una espressione. Dunque:

$$\mathcal{S}em_e : \texttt{Exp} \to P \to M \to \texttt{Val} \cup \texttt{Loc}$$

$\texttt{x} \in \texttt{Ide}$

$$\mathcal{S}em_e\ \texttt{\&x}\ \rho\ \mu = \rho\ \texttt{x}$$

$\texttt{id} \in \texttt{Ide}$

$$\mathcal{S}em_e\ \texttt{*id}\ \rho\ \mu = \mu(\mu(\rho\ \texttt{id}))$$

$$\mathcal{S}em_d\ [\texttt{T}\ *\texttt{p;}]\ \rho\ \mu = \langle \rho[^{\ell}/_{\texttt{p}}]^{add}, \mu[^{?}/_{\ell}]^{add} \rangle$$

dove $\ell = succloc\ \mu$

$$\mathcal{S}em_d\ [\texttt{T}\ *\texttt{p} = \texttt{E;}]\ \rho\ \mu = \langle \rho[^{\ell}/_{\texttt{p}}]^{add}, \mu[^{\ell'}/_{\ell}]^{add} \rangle$$

dove $\ell' = \mathcal{S}em_e\ \texttt{E}\ \rho\ \mu$

e $\quad \ell = succloc\ \mu$

Si noti che, mentre $\mathcal{S}em_e\ \texttt{x}\ \rho\ \mu = \mu(\rho\ \texttt{x})$, ovvero la semantica di un'espressione costituita da un identificatore è il *valore* associato all'identificatore nello stato, $\mathcal{S}em_e\ \texttt{\&x}\ \rho\ \mu = \rho\ \texttt{x}$ ad indicare che l'espressione `&x` sta per l'indirizzo (la locazione) associato all'identificatore nello stato. Inoltre, l'espressione `*x` comporta un doppio accesso alla memoria, al fine di reperire il valore *puntato* da `x`.

Veniamo all'accesso in *modifica* tramite puntatore.

$$\mathcal{S}em_c\ [\texttt{*id = E}]\ \rho\ \mu = \mu[^{v}/_{\mu(\rho(\texttt{id}))}]^{mod}$$

dove $v = \mathcal{S}em_e\ \texttt{E}\ \rho\ \mu$

Anche in questo caso si noti la differenza con $\mathcal{S}em_c\ [\texttt{id = E}]\ \rho\ \mu = \mu[^{v}/_{\rho(\texttt{id})}]^{mod}$

Esempi

Con l'introduzione dei puntatori abbiamo la possibilità di accedere e modificare variabili senza utilizzarne esplicitamente il nome. Questo può essere pericoloso e rendere meno leggibili, e quindi meno facilmente verificabili, i programmi che si scrivono.

```
int x = 10;  int y = 20;
int *p1 = &x; int *p2= &y;

*p1 = y + 1;
```

```
*p2 = *p2 + *p1;
x = *p2 + 1;
```

Quali sono i valori associati a `x` e `y` dopo l'esecuzione di questa sequenza?

Inoltre i puntatori sono variabili come tutte le altre. Il loro valore (che ricordiamo è un indirizzo) può variare durante l'esecuzione di un programma e quindi successivi accessi mediante l'operatore `*` possono causare la modifica di variabili diverse. Detto altrimenti, il medesimo assegnamento `*p = E` in punti diversi di un programma può comportare la modifica di variabili diverse, a seconda di quale è l'indirizzo associato a `p` al momento dell'esecuzione dell'assegnamento. Ad esempio:

```
int *p;  int x; int y;
...
p = &x;   *p = 10;    /* causa la modifica della variabile x */
...
p = &y;   *p = 10;    /* causa la modifica della variabile y */
```

Inoltre, è possibile associare ad un puntatore un indirizzo non solo attraverso l'operatore `&`, ma anche attraverso il normale assegnamento tra variabili. Ad esempio:

```
int *p;  int *p1; int y;
...
p = &y;
...
p1 = p;
```

Dopo l'ultimo assegnamento ci sono ben tre modi diversi per *accedere* sia in lettura che in modifica alla variabile `y`: `y`, `*p` e `*p1`!

5.1 Implementazione in CAML

```
(* ESTENSIONE DEI TIPI PER LA RAPPRESENTAZIONE DELLA SINTASSI *)

(* Espressioni *)
type exp = .....
           Deref of ide |
           Address of ide;;

(* Dichiarazioni *)
type dec = .....
           Pvar of ide |
           Pvar_init of ide * exp;;

(* Comandi *)
type com = ....
           Passign of ide * exp ;;
```

```
(* Estensione del tipo val *)

type val = ValN of int |
           ValB of bool |
           Mloc of mloc |
           Unknown ;;

(* Estensione delle funzioni di interpretazione semantica *)

let rec semexp e (a:amb list) (m:mem list) =
  match e with
    ...
    Deref i -> let (Def l) = search a i in
                  let (Def (Mloc l')) = search m l in
                     let (Def v) = search m l' in v      |
    Address i -> let (Def l) = search a i in Mloc l;;

let semd d (a:amb list) (m : mem list) = match d with
      ...
      Pvar x ->  add_stack a  x (Def (succloc m)),
                 add_stack m (succloc m) (Def Unknown) |
      Pvar_init(x,e) -> let v = (semexp e a m) in
                           add_stack a  x (Def (succloc m)),
                           add_stack m (succloc m) (Def v);;

let rec semc c (a:amb list) (m:mem list) = match c with
     ...
     Passign(x,e) -> let v = (semexp e a m) and (Def l) = (search a x) in
                          let (Def (Mloc l')) = search m l  in
                              update_stack m l' (Def v);;
```

6 Memoria dinamica e heap

6.1 Heap

Il linguaggio consente anche la allocazione *dinamica* di memoria, ovvero la possibilità di creare a tempo di esecuzione variabili (anonime) alle quali è possibile accedere solo attraverso il loro *indirizzo*. Tali variabili non vengono dichiarate con l'usuale meccanismo delle dichiarazioni, ma vengono create al bisogno. La componente dello stato preposta al rilascio della memoria per la gestione di tali variabili dinamiche viene detta *heap*: si tratta di una struttura che non viene gestita a pila. Lo heap è una parte della memoria dalla quale si va a prelevare lo spazio necessario a contenere una variabile al momento in cui il programma ne fa richiesta, a differenza della pila in cui l'allocazione ed il rilascio dello spazio avviene in maniera controllata e sistematica col meccanismo dei blocchi.

Le operazioni previste sullo heap sono due: la prima *new* consente di richiedere nuovo spazio per l'allocazione di una variabile dinamica e fornisce l'indirizzo dello spazio allocato, che deve essere associato ad una variabile di tipo puntatore per poterne fare uso; la seconda *free* consente

di rilasciare spazio precedentemente allocato tramite *new* per renderlo disponibile per successive allocazioni.

Formalmente lo heap è una funzione da locazioni in valori. Distinguiamo le locazioni nella pila dalle locazioni nello heap, indicando con $\texttt{Loc}^{\mu}$ le prime e con $\texttt{Loc}^{\zeta}$ le seconde, e indichiamo con $\texttt{Loc}$ l'insieme $\texttt{Loc}^{\mu} \cup \texttt{Loc}^{\zeta}$. Si noti che le locazioni in $\texttt{Loc}^{\mu}$ sono quelle che, nell'ambiente, vengono associate agli identificatori delle variabili che sono dichiarate nei blocchi. Con questa distinzione abbiamo dunque:

Frame ambiente Sono funzioni $\varphi : \texttt{Ide} \to \texttt{Loc}^{\mu}_{\perp}$

Frame memoria Sono funzioni $\nu : \texttt{Loc}^{\mu} \to (\texttt{Val} \cup \texttt{Loc})_{\perp}$

Heap Sono funzioni $\zeta : \texttt{Loc}^{\zeta} \to \texttt{Val}_{\perp}$.

Nel seguito indicheremo con $\mathcal{H}$ l'insieme degli heap, ovvero

$$\mathcal{H} = \{\zeta \mid \zeta : \texttt{Loc}^{\zeta} \to \texttt{Val}_{\perp}\}.$$

Come nel caso della memoria, supponiamo di disporre di una funzione

$$\mathit{newloc} : \mathcal{H} \longrightarrow \texttt{Loc}^{\zeta}$$

che, dato uno heap ζ, restituisce una locazione *libera* in ζ, ovvero disponibile per contenere una nuova variabile dinamica. Anche in questo caso, se $\mathit{newloc}(\zeta) = \ell$ allora $\zeta(\ell) = \perp$.

Inoltre, supponiamo di disporre di una funzione duale rispetto a *newloc*, che chiameremo *free* e che, data una locazione e uno heap, restituisce un nuovo heap in cui la locazione è resa di nuovo disponibile. Più precisamente

$$\mathit{free} : \texttt{Loc}^{\zeta} \to \mathcal{H} \to \mathcal{H}$$

e

$$\mathit{free}(\ell, \zeta)(y) = \begin{cases} \perp & \text{se } y = \ell \\ \zeta(y) & \text{altrimenti} \end{cases}$$

6.1.1 Implementazione CAML dello heap

```
(* Locazioni Heap *)
type hloc == int;;

(* VALORI *)
type val = ValN of int |
           ValB of bool |
           Mloc of mloc |
           Hloc of hloc |
           Unknown ;;

(* HEAP *)
type heap == hloc -> val bottom;;
```

```
let newloc (h: heap) =
   let rec findfree h n = match h n with
                      Bottom -> ((n:hloc), (add h n (Def Unknown))) |
                      _      -> findfree h (n+1)
   in (findfree h 0 : (hloc * heap));;

let free (l:hloc) (h : heap) =
       let g z = match z with
             l -> Bottom |
             _ -> h z
       in (g : heap);;
```

6.2 Estensioni sintattiche e semantiche

Le estensioni sintattiche per la gestione di variabili dinamiche sono le seguenti

```
Dec::= PType Ide = new;

Com ::= Ide = new; | free(Ide);
```

L'operazione `new` può essere utilizzata sia per inizializzare una variabile di tipo puntatore ad una nuova locazione nello heap, sia per attribuire una nuova locazione ad una variabile puntatore presente nello stato. L'operazione `free` consente di rilasciare memoria nello heap, quella la cui locazione è il valore del puntatore argomento di `free`.

Con l'introduzione dello heap lo stato è ora costituito da una terna: l'ambiente, la memoria e lo heap, di cui memoria e heap sono le parti modificabili per effetto dell'esecuzione dei comandi. Ciò comporta la necessità di dover rivedere tutte le funzioni di interpretazione semantica introdotte fino a questo momento in modo da includere la nuova componente dello stato.

- Comandi: $\mathcal{S}em_c : \texttt{Com} \to P \ \to M \ \to \mathcal{H} \to (M \ * \mathcal{H})$
- Espressioni: $\mathcal{S}em_e : \texttt{Exp} \to P \ \to M \ \to \mathcal{H} \to \texttt{Val} \cup \texttt{Loc}$
- Dichiarazioni: $\mathcal{S}em_d : \texttt{Dec} \to P \ \to M \ \to \mathcal{H} \to P \ * M \ * \mathcal{H}$

Nel dare la semantica delle nuove produzioni introdotte, vediamo anche come si modificano alcune delle regole semantiche definite in precedenza per le varie componenti del linguaggio, al fine di includere la nuova componente dello stato.

Espressioni

Pur non essendo previste nuove produzioni per la categoria sintattica `Exp`, le regole di $\mathcal{S}em_e$ vanno riviste per tener conto del nuovo stato, ma soprattutto va ridefinita la regola per l'espressione `*p`, dal momento che un puntatore può, nel nuovo modello, riferire una locazione nello heap o nella memoria.

Modifica regole esistenti

$$\frac{\texttt{id} \in \texttt{Ide}}{\mathcal{S}em_e\ \texttt{id}\ \rho\ \mu\ \zeta = \mu(\rho\ \texttt{id})}$$

$$\frac{\texttt{id} \in \texttt{Ide}}{\mathcal{S}em_e\ \texttt{\&id}\ \rho\ \mu\ \zeta = \rho\ \texttt{id}}$$

Per l'espressione `*p` abbiamo bisogno di distinguere due casi.

$$\mathcal{S}em_e\ \texttt{*id}\ \rho\ \mu\ \zeta = \mu(\mu(\rho\ \texttt{id}))$$

se $\mu(\rho(\texttt{id})) \in \texttt{Loc}^{\mu}$

$$\mathcal{S}em_e\ \texttt{*id}\ \rho\ \mu\ \zeta = \zeta(\mu(\rho\ \texttt{id}))$$

se $\mu(\rho(\texttt{id})) \in \texttt{Loc}^{\zeta}$

Dichiarazioni

Modifica regole esistenti

$$\mathcal{S}em_d\ [\texttt{T x;}]\ \rho\ \mu\ \zeta = \langle \rho[{}^{\ell}/_{\texttt{x}}]^{add}, \mu[{}^{?}/_{\ell}]^{add}, \zeta \rangle$$

dove $\ell = succloc\ \mu$

$$\mathcal{S}em_d\ [\texttt{T x = E;}]\ \rho\ \mu\ \zeta = \langle \rho[{}^{\ell}/_{\texttt{x}}]^{add}, \mu[{}^{v}/_{\ell}]^{add}, \zeta \rangle$$

dove $v = \mathcal{S}em_e\ \texttt{E}\ \rho\ \mu\ \zeta$

e $\ell = succloc\ \mu$

Nuova regola

La regola semantica per la nuova produzione di `Dec` è la seguente.

$$\mathcal{S}em_d\ [\texttt{T *x = new;}]\ \rho\ \mu\ \zeta = \langle \rho[{}^{\ell}/_{\texttt{x}}]^{add}, \mu[{}^{\ell'}/_{\ell}]^{add}, \zeta[{}^{?}/_{\ell'}]^{add} \rangle$$

dove $\ell = succloc\ \mu$

$\ell' = newloc\ \zeta$

Comandi

Modifica regole esistenti

Vediamo solo come si modificano alcune delle regole già viste, lasciando le altre per esercizio.

$$\mathcal{S}em_c\ [\texttt{id = E}]\ \rho\ \mu\ \zeta = \langle \mu[{}^{v}/_{\rho(\texttt{id})}]^{mod}, \zeta \rangle$$

dove $v = \mathcal{S}em_e\ \texttt{E}\ \rho\ \mu\ \zeta$

$$\mathcal{S}em_c\ \{\texttt{Dlist CList}\}\ \rho\ \mu\ \zeta = \langle \mu'', \zeta'' \rangle$$

dove $\mathcal{S}em_{dl}\ \texttt{Dlist}\ \omega.\rho\ \omega.\mu\ \zeta = \langle \varphi.\rho, \nu.\mu, \zeta' \rangle$

e $\mathcal{S}em_{cl}\ \texttt{Clist}\ \varphi.\rho\ \nu.\mu\ \zeta' = \langle \nu'.\mu'', \zeta'' \rangle$

Si noti che la regola per il blocco mette in evidenza la gestione a "pila" di ambiente e memoria.

Una modifica significativa riguarda l'assegnamento tramite puntatore, dal momento che nel nuovo modello un puntatore può riferire la memoria o lo heap (come per l'espressione `*p`).

Dunque abbiamo bisogno di due regole per i due casi.

$$\mathcal{S}em_c\ [\texttt{*id = E}]\ \rho\ \mu\ \zeta = \langle \mu[^{v}/_{\mu(\rho(\texttt{id}))}]^{mod}, \zeta \rangle$$

se $\mu(\rho(\texttt{id})) \in \texttt{Loc}^{\mu}$

dove $v = \mathcal{S}em_e\ \texttt{E}\ \rho\ \mu\ \zeta$

$$\mathcal{S}em_c\ [\texttt{*id = E}]\ \rho\ \mu\ \zeta = \langle \mu, \zeta[^{v}/_{\mu(\rho(\texttt{id}))}]^{mod} \rangle$$

se $\mu(\rho(\texttt{id})) \in \texttt{Loc}^{\zeta}$

dove $v = \mathcal{S}em_e\ \texttt{E}\ \rho\ \mu\ \zeta$

La regola semantica per il nuovo assegnamento in `Com` è la seguente.

$$\mathcal{S}em_c\ [\texttt{id = new}]\ \rho\ \mu\ \zeta = \langle \mu[^{\ell}/_{\rho(\texttt{id})}]^{mod}, \zeta[^{?}/_{\ell}]^{add} \rangle$$

dove $\ell = newloc\ \zeta$

Veniamo infine alla regola per il comando che consente di liberare memoria nello heap.

$$\mathcal{S}em_c\ [\texttt{free(id)}]\ \rho\ \mu\ \zeta = \langle \mu, free(\ell, \zeta) \rangle$$

dove $\ell = \mu(\rho(id))$

e $\ell \in \texttt{Loc}^{\zeta}$

6.2.1 Implementazione in CAML

```
(* Estensioni sintattiche *)

(* Dichiarazioni *)
type dec = ...
           Pnew of ide;;

(* Comandi*)
type com = ...
           Palloc of ide   |
           Free of ide;;
```

Riportiamo di seguito la definizione completa delle funzini CAML che implementano le funzioni di interpretazione semantica per le espressioni, le dichiarazioni ed i comandi.

```
(* Semantica delle espressioni *)

let rec semexp e (a:amb list) (m:mem list) (h : heap) =
  match e with
    Ide i -> (let (Def l) = search a i in
                  let (Def v) = search m l in v) |
    Num n -> ValN n |
    Bool b -> ValB b |
    BinExp (e1,o,e2) -> sembop o (semexp e1 a m h) (semexp e2 a m h)|
    UnExp (o,e1) -> semuop o (semexp e1 a m h) |
    Deref i -> let (Def l) = search a i in
                (match search m l with
                   Def (Mloc l') ->
                         let (Def v) = search m l' in v |
                   Def (Hloc l') -> let (Def v) = h l' in v) |
    Address i -> let (Def l) = search a i in (Mloc l);;

(* Semantica delle dichiarazioni *)
let semd d (a:amb list) (m : mem list) (h : heap)= match d with
      Var x -> ((add_stack a  x (Def (succloc m)),
             add_stack m (succloc m) (Def Unknown), h): (amb list * mem list * heap)) |
      Var_init(x,e) -> let v = (semexp e a m h) in
                   add_stack a  x (Def (succloc m)), add_stack m (succloc m) (Def v), h |
      Pvar x -> (add_stack a  x (Def (succloc m)),
              add_stack m (succloc m) (Def Unknown), h) |
      Pvar_init(x,e) -> let v = (semexp e a m h) in
                   (add_stack a  x (Def (succloc m)),
                        add_stack m (succloc m) (Def v), h) |
      Pnew x -> let (l,h')= newloc h in (add_stack a  x (Def (succloc m)),
             add_stack m (succloc m) (Def (Hloc l)), h');;

let rec semdl dl (a:amb list) (m : mem list) (h:heap)= match dl with
      []    -> (a,m,h) |
      d::ds -> let (a',m',h')=semd d a m h in semdl ds a' m' h';;


(* Semantica dei comandi *)
let rec semc c (a:amb list) (m:mem list) (h:heap) = match c with
     Assign(x,e) -> let v = (semexp e a m h) and (Def l) = (search a x) in
                  ((update_stack m l (Def v)), h : mem list * heap) |
     If_then_else(e,c1,c2) -> (match semexp e a m h with
                     ValB true -> semc c1 a m h |
                     ValB false -> semc c2 a m h ) |
     If_then(e,c1) -> (match semexp e a m h with
                     ValB true -> semc c1 a m h |
                     ValB false -> m, h ) |
     While(e,c1) -> (match semexp e a m h with
                     ValB false -> m, h |
```

```
                    ValB true -> let (m',h') = semc c1 a m h in
                                        semc (While(e,c1)) a m' h') |
     Block(dl,cl) -> let (a', m', h') = semdl dl (omega::a) (omega::m) h
                 in let ((fm :: m''), h'')= semcl cl a' m' h' in m'', h'' |
     Passign(x,e) -> let v = (semexp e a m h) and (Def l) = (search a x) in
                        (match search m l  with
                           Def (Mloc l') -> (update_stack m l' (Def v), h) |
                           Def (Hloc l') -> (m, update h l' (Def v)) ) |
     Palloc(x)    ->  let (Def l) = (search a x) and (l',h')= newloc h in
                         (update_stack m l (Def (Hloc l')), h') |
     Free(x)    ->  let (Def l) = (search a x) in
                         let (Def (Hloc l')) = search m l in
                              (m, update h l' Bottom)
and

semcl cl a m h = match cl with
     []    -> m,h |
     c::cs -> let (m',h') = semc c a m h in semcl cs a m' h' ;;
```

Tipizzazione

```
semexp : exp -> amb list -> mem list -> heap -> val
semd   : dec -> amb list -> mem list -> heap -> amb list * mem list * heap
semc   : com -> amb list -> mem list -> heap -> mem list * heap
```