4 Identificatori, tipi e dichiarazioni

Una caratteristica presente in quasi tutti i linguaggi di programmazione (inclusi i linguaggi assembler) e' la possibilita' di assegnare un nome ad un oggetto (sia esso una variabile, costante o comando), ed utilizzare tale nome per riferire l'oggetto. Un'altro strumento importante per evitare al programmatore di gestire direttamente la rappresentazione dei dati e la gestione della memoria sono i tipi e le dichiarazioni di variabile. I tipi sono anche essenziali per effettuare i controlli statici che vanno sotto il nome di type-checking.

$$
\begin{array}{rcl}
u, x, X \in < Identifier > & ::= & \ldots \\
T \in < Type > & ::= & X \mid \\
& & e_1..e_2 \mid \\
& & \mathrm{array}\ X[e] \mid \\
& & \mathrm{record}\ x_i : X_i\ \mathrm{end} \mid \\
& & \mathrm{union}\ x : X\ \mathrm{of}\ e_i : (x_i : X_i)\ \mathrm{end} \mid \\
& & \uparrow X \\
d \in < Declaration > & ::= & d_1 ; d_2 \mid \\
& & \mathrm{type}\ X = T \mid \\
& & \mathrm{enum}\ X = (u_i) \mid \\
& & \mathrm{var}\ x : X \mid \\
& & \mathrm{const}\ u = e \\
v \in < Variable > & ::= & x \mid v[e] \mid v.x \mid v \uparrow \\
e \in < Expression > & ::= & u \mid v \mid op(e_i) \\
c \in < Command > & ::= & v := e \mid new(v) \mid \ldots \mid \\
& & d\ \mathrm{begin}\ c_i\ \mathrm{end} \mid \\
& & \mathrm{with}\ v\ \mathrm{do}\ c_i\ \mathrm{end}
\end{array}
$$

Per gli identificatori adottiamo la convenzione che u e' un identificatore di costante, x di variabile, e X di tipo. Tutti i tipi introdotti, a parte il tipo union, sono noti dal Pascal. Il tipo union corrisponde alla unione disgiunta di insiemi, ed e' un caso particolare dei record con variante del Pascal. Non tutti gli errori di tipo relativi ad union ed array sono identificabili a tempo di compilazione, percio' o il compilatore inserisce automaticamente dei controlli a run-time, o si lascia alla discrezione del programmatore (come fa' il C).

Si osservi che le dimensioni di un array sono noto solo a run-time, questo e' compatibile con una gestione a stack della memoria, ma non con una gestione statica. Un altro punto delicato e' la mutua ricorsione a livello di dichiarazioni. In Pascal e C solo i tipi puntatore e le procedure possono usare nella definizione nomi non ancora definiti. Le restrizioni imposte sulle definizioni ricorsive dei tipi sono forzate da scelte di tipo implementativo sulla gestione della memoria.

4.1 Semantica statica e type-checking

Consideriamo anzitutto una semantica statica che si limita ad effettuare il type-checking. I punti piu' delicati sono: la generativita' dei costruttori di tipo (ci siamo ispirati al Pascal invece che al C), ed il trattamento di definizioni mutuamente ricorsive. La sematica operazionale statica fa uso dei seguenti **domini semantici**

- $x \in Id$ identificatore, quindi Id coincide con $< Identifier >$
- $m, n \in N$ name counter per implementare la generativita dei tipi, e' importante distinguere tra identificatori e nomi di tipo, questi ultimi hanno uno status simile alle locazioni.
- $\rho \in Env^S = Id \stackrel{fin}{\to} type(N) \mid var(N) \mid const(N)$ ambiente statico
- $s \in State^S = N \times (N \stackrel{fin}{\to} enum \mid array(N) \mid record(Id \stackrel{fin}{\to} N) \mid union(Id, N, Id \stackrel{fin}{\to} N)) \mid ptr(N)$ stato statico

  Dato uno stato statico $s = (n, g)$, la componente n indica il valore del name counter, mentre g associa ai nomi di tipo una descrizione della struttura. $g(n) = enum$ indica che un tipo enumerazione (o piu' in generale un tipo non strutturato).

$X \stackrel{fin}{\to} Y$ e' l'insieme delle funzioni parziali da X ad Y con dominio finito. Introduciamo le seguenti operazioni che opera su funzioni parziali

- 0 e' la funzione ovunque indefinita
- $f(x)\downarrow$ significa che $f(x)$ e' definita
- $f \oplus g$ e' la funziona parziale t.c. $(f \oplus g)(x) = \begin{cases} g(x) \text{ se } g(x)\downarrow \\ f(x) \text{ altrimenti} \end{cases}$

  l'operazione $\oplus$ e' estesa agli stati statici nel modo seguente $(n, g) \oplus (n', g') = (n + n', g \oplus g')$
- $x \mapsto y$ e' la funziona parziale t.c. $(x \mapsto y)(x') = \begin{cases} y \text{ se } x' = x \\ 0(x') \text{ altrimenti} \end{cases}$

Lo stato statico (al contrario dello stato dinamico) non subisce aggiornamenti, ma solo estensioni. A livello di semantica statica questo permette di descrivere la modifica ad uno stato statico (n, g) indotta da una dichiarazione d come un incremento (n', g'), ovvero dopo la dichiarazione lo stato sara' $(n+n', g\oplus g')$. Percio' n' dice quanti nuovi nomi di tipo sono generati da d, mentre g' ne descrive la struttura. In particolare, i domini di g e g' sono disgiunti (per la precisione questo e' vero solo quando le dichiarazioni non sono ricorsive). Anche per l'ambiente adottiamo un trattamento simile, ovvero l'ambiente ρ' prodotto da d valutata in ambieste ρ e stato s ha come dominio gli identificatori dhichiarati in d, percio' l'ambiente dopo la dichiarazione e' dato da $\rho \oplus \rho'$. Le asserzioni per la sematica statica sono:

- $\rho \vdash s, T \Rightarrow s', m$ e $\rho \vdash s, d \Rightarrow s', \rho'$ in ambiente ρ e stato s la dichiarazione d ha un effetto incrementale s' su s e ρ' su ρ. In una diversa interpretazione delle asserzioni s' e ρ' sono stato ed ambiente statico dopo le modifiche indotte da d, quindi s' e' una estensione di s ed il dominio di ρ' include quello di ρ. La prima scelta e' preferibile, poiche' permette di trattare facilmente anche le dichiarazioni ricorsive e le dichiarazioni locali ad altre dichiarazioni.
- $\rho, s \vdash v \Rightarrow m$ e $\rho, s \vdash e \Rightarrow m$ il tipo di v/e in ambiente ρ e stato s e' m
- $\rho, s \vdash c \Rightarrow ok$ in ambiente ρ e stato s il comando c e' staticamente corretto

Le regole di inferenza per la semantica statica sono:

- $\rho \vdash s, T \Rightarrow s', m$

  $$(X)\ \frac{\rho(X) = type(m)}{\rho \vdash s, X \Rightarrow (0, 0), m}$$

  $(0, 0)$ indica che lo stato non subisce variazioni

  $$(\text{subrange})\ \frac{\rho, (n, g) \vdash e_i \Rightarrow m \quad (i = 1, 2)}{\rho \vdash (n, g), e_1..e_2 \Rightarrow (0, 0), m}\ g(m) = enum$$

  nel type-checking a tempo di compilazione si ignorano i bounds, poiche' il valore delle espressioni e' noto solo a run-time, e quindi il bound-checking puo' essere fatto solo a run-time.

  $$(\text{array})\ \frac{\rho, s \vdash e \Rightarrow nat \quad \rho(X) = type(m)}{\rho \vdash (n, g), \text{array } X[e] \Rightarrow (1, n \mapsto array(m)), n}$$

  Si genera un nuovo nome n per il tipo array, quindi si deve incrementare il contatore di 1, inoltre si verifica che il tipo di e sia nat (il tipo enumerativo predefinito dei numeri naturali).

  $$(\text{record})\ \frac{\rho(X_i) = type(m_i)}{\rho \vdash (n, g), \text{record } x_i{:}\, X_i \text{ end} \Rightarrow (1, n \mapsto record(x_i \mapsto m_i)), n}$$

  $$(\text{union})\ \frac{\begin{array}{c}\rho(X) = type(m) \quad \rho(X_i) = type(m_i) \\ \rho, (n, g) \vdash e_i \Rightarrow m\end{array}}{\rho \vdash (n, g), \text{union } x{:}\, X \text{ of } e_i{:}\, (x_i{:}\, X_i) \text{ end} \Rightarrow (1, n \mapsto union(x, m, x_i \mapsto m_i)), n}\ g(m) = enum$$

  Il tipo del tag x deve essere enumerativo. a compile-time non si puo' verificare se le espressioni e_i hanno valori distinti, e coprano tutti i possibili valori del tag (in Pascal cio' e' possibile perche' le e_i sono costanti).

  $$(\text{ptr})\ \frac{\rho(X) = type(m)}{\rho \vdash (n, g), \uparrow X \Rightarrow (1, n \mapsto ptr(m)), n}$$

- $\rho \vdash s, d \Rightarrow s', \rho'$

  (comp) $$\frac{\rho \vdash s, d_1 \Rightarrow s_1, \rho_1 \quad \rho \oplus \rho_1 \vdash s \oplus s_1, d_2 \Rightarrow s_2, \rho_2}{\rho \vdash s, d_1; d_2 \Rightarrow s_1 \oplus s_2, \rho_1 \oplus \rho_2}$$

  In Pascal si richiede che ρ_i abbiano domini disgiunti

  (type) $$\frac{\rho \vdash s, T \Rightarrow s', m}{\rho \vdash s, \mathrm{type}\ X = T \Rightarrow s', X \mapsto type(m)}$$

  Solo le dichiarazioni di tipo provocano una estensione dello stato statico

  (enum) $$\frac{}{\rho \vdash (n, g), \mathrm{enum}\ X = (u_i) \Rightarrow (1, n \mapsto enum), (X \mapsto type(n)) \oplus (u_i \mapsto const(n))}$$

  (var) $$\frac{\rho(X) = type(m)}{\rho \vdash s, \mathrm{var}\ x{:}\, X \Rightarrow (0,0), x \mapsto var(m)}$$

  (const) $$\frac{\rho, s \vdash e \Rightarrow m}{\rho \vdash s, \mathrm{const}\ u = e \Rightarrow (0,0), u \mapsto const(m)}$$

  A causa delle restrizioni imposte al tipo di una espressione (vedi soto), questa regola consente di definire solo costanti di tipo enumerativo/semplice.

- $\rho, s \vdash v \Rightarrow m$

  (x) $$\frac{\rho(x) = var(m)}{\rho, s \vdash x \Rightarrow m}$$

  (array) $$\frac{\rho, (n,g) \vdash v \Rightarrow m \quad \rho, (n,g) \vdash e \Rightarrow nat}{\rho, (n,g) \vdash v[e] \Rightarrow m'}\ g(m) = array(m')$$

  (record) $$\frac{\rho, (n,g) \vdash v \Rightarrow m}{\rho, (n,g) \vdash v.x_i \Rightarrow m_i}\ g(m) = record(x_i \mapsto m_i)$$

  il trattamento statico del caso $g(m) = union(x, m, x_i \mapsto m_i)$ e' analogo. E' pero' necessario aggiungere un controllo run-time per verificare che il valore del tag e' consistente con la componente selezionata.

  (ptr) $$\frac{\rho, (n,g) \vdash v \Rightarrow m}{\rho, (n,g) \vdash v \uparrow \Rightarrow m'}\ g(m) = ptr(m')$$

- $\rho, s \vdash e \Rightarrow m$

  (u) $$\frac{\rho(u) = const(m)}{\rho, s \vdash u \Rightarrow m}$$ (var) $$\frac{\rho, s \vdash_v v \Rightarrow m}{\rho, s \vdash_e v \Rightarrow m}\ g(m) = enum$$ (op) $$\frac{\rho, s \vdash e_i \Rightarrow m_i}{\rho, s \vdash op(e_i) \Rightarrow m}\ op{:}\, m_i \to m$$

  Le **espressioni possono essere solo di tipo enumerativo/semplice**, anche se non e' detto esplicitamente.

- $\rho, s \vdash c \Rightarrow ok$

  (:=) $$\frac{\rho, (n,g) \vdash v \Rightarrow m \quad \rho, (n,g) \vdash e \Rightarrow m}{\rho, (n,g) \vdash v{:}= e \Rightarrow ok}$$ (new) $$\frac{\rho, (n,g) \vdash v \Rightarrow ptr(m)}{\rho, (n,g) \vdash new(v) \Rightarrow ok}$$

  A causa delle restrizioni imposte al tipo di una espressione, si possono effettuare assegnamenti solo a variabili di tipo enumerativo/semplice.

  (block) $$\frac{\rho \vdash s, d \Rightarrow s', \rho' \quad \rho \oplus \rho', s \oplus s' \vdash c_i \Rightarrow ok}{\rho, s \vdash d\ \mathrm{begin}\ c_i\ \mathrm{end} \Rightarrow ok}$$

  (with) $$\frac{\rho, (n,g) \vdash v \Rightarrow m \quad \rho \oplus (x_i \mapsto type(m_i)), (n,g) \vdash c_i \Rightarrow ok}{\rho, (n,g) \vdash \mathrm{with}\ v\ \mathrm{do}\ c_i\ \mathrm{end} \Rightarrow ok}\ g(m) = record(x_i \mapsto m_i)$$

Per trattare le dichiarazioni mutuamente ricorsive occorre modificare solo la regola (block):

(block*) $$\frac{\rho \oplus \rho' \vdash (n, g \oplus g'), d \Rightarrow (n', g'), \rho' \quad \rho \oplus \rho', (n,g) \oplus (n', g') \vdash c_i \Rightarrow ok}{\rho, (n,g) \vdash d\ \mathrm{begin}\ c_i\ \mathrm{end} \Rightarrow ok}$$

L'idea chiave e' che una dichiarazione ricorsiva d deve essere valutata in un ambiente e stato gia' estesi con gli effetti di d (ad eccezione del name-counter). La nuova regola non modifica la decidibilita' del type-checking, perche' le derivazioni sono comunque guidate dalla sintassi.

4.2 Semantica dinamica e gestione della memoria a stack

Le informazioni di tipo si possono sfruttare per gestire la memoria, ovvero per stabilire quanto spazio allocare per una variabile, e per calcolare gli indirizzi fisici delle componenti di una variabile struturata. Inoltre Algol 60 e discendenti sono stati disegnati in modo da permettere una gestione a stack della memoria per le variabili temporanee dichiarate in un blocco, **purche' non si faccia uso dei tipi puntatore**. Anche il C segue tale approccio, pero' a differenza dell Algol 60 lascia molto a desiderare sugli aspetti di sicurezza degli accessi in memoria.
La sematica operazionale dinamica fa uso dei seguenti **domini semantici**

- $x \in Id$ identificatore
- $m, n \in N$ numeri naturali
- $t \in D$ descrittore di tipo, dove D e' definito induttivamente

  $D = enum(N) \mid range(NL, NU) \mid array(D, NS) \mid record(Id \stackrel{fin}{\to} (D \times NA)) \mid union(Id, D, Id \stackrel{fin}{\to} (D, NV))$

  dove $NL = NU = NS = NA = NV = N$. I numeri naturali hanno il seguente significato: $n \in N$ indica il numero di elementi di un tipo enumerativo, $m \in NL$ e $n \in NU$ indicano il limite inferiore e superiore di un tipo sottocampo, $n \in NS$ indica la dimensione di un array, $n \in NA$ indica l'indirizzo relativo della componente di un record, $n \in NA$ indica il valore del campo tag corrispondente ad una variante del tipo union.
- $\rho \in Env = Id \stackrel{fin}{\to} type(D) \mid var(N, D) \mid const(N)$ ambiente dinamico
- $s \in State = \{(n, M) | n \in N, M : n \to N\}$ stato dinamico

  n indica la dimensione della memoria, ed M ne descrive il contenuto. Una variabile strutturata corrisponde ad una sequenza di celle di memoria contigue, la lunghezza della sequenza e' determinata dal tipo della variabile.

Definiamo per induzione su $t \in D$ la funzione $|\_| : D \to N$ ed il predicato $\_ : \_ \subseteq N \times D$. $|t|$ e' il numero di celle di memoria necessarie per una variabile con descrittore di tipo t, mentre $m : t$ sse il valore m e' compatibile con il tipo t

$$\begin{aligned} |enum(n)| = |range(m, n)| &= 1 \\ |array(t, n)| &= n * |t| \\ |record(x_i \mapsto (t_i, n_i))| &= \Sigma_i |t_i| \\ |union(x, t, x_i \mapsto (t_i, n_i))| &= 1 + \max_i |t_i| \quad \text{(si suppone che } |t| = 1) \end{aligned}$$

$m : t \stackrel{\Delta}{\iff} (t = enum(n) \wedge m < n) \vee (t = range(m_1, m_2) \wedge m_1 \leq m \leq m_2)$
Le asserzioni per la sematica dinamica sono:

- $\rho, s \vdash T \Rightarrow t$ il descrittore del tipo T e' t
- $\rho \vdash s, d \Rightarrow s', \rho'$ la dichiarazione d ha un effetto incrementale ρ' su ρ e produce un nuovo stato s'
- $\rho \vdash v \Rightarrow m, t$ la prima locazione della variabile v e' m e t e' il suo descrittore di tipo
- $\rho, s \vdash e \Rightarrow m$ il valore della espressione (di tipo enumerativo/semplice) e e' m
- $\rho \vdash s, c \Rightarrow s'$ il comando c produce un nuovo stato s'

Le regole di inferenza per la semantica dinamica sono:

- $\rho, s \vdash T \Rightarrow t$

  $(X)\ \dfrac{\rho(X) = type(t)}{\rho, s \vdash X \Rightarrow t} \qquad \text{(subrange)}\ \dfrac{\rho, s \vdash e_i \Rightarrow m_i \quad (i = 1, 2)}{\rho, s \vdash e_1..e_2 \Rightarrow range(m_1, m_2)}$

  $\text{(array)}\ \dfrac{\rho, s \vdash e \Rightarrow m \quad \rho(X) = type(t)}{\rho, s \vdash \text{array } X[e] \Rightarrow array(t, m)}$

(record) $\dfrac{\rho(X_i) = type(t_i)}{\rho, s \vdash \text{record } x_i\colon X_i \text{ end} \Rightarrow record(x_i \mapsto (t_i, \Sigma_{j<i}|t_j|))}$

(union) $\dfrac{\rho(X) = type(t) \quad \rho(X_i) = type(t_i) \quad \rho, s \vdash e_i \Rightarrow m_i\colon t}{\rho, s \vdash \text{union } x\colon X \text{ of } e_i\colon (x_i\colon X_i) \text{ end} \Rightarrow union(x, t, x_i \mapsto (t_i, m_i))}\ |t| = 1$

- $\rho \vdash s, d \Rightarrow s', \rho'$

(comp) $\dfrac{\rho \vdash s, d_1 \Rightarrow s_1, \rho_1 \quad \rho \oplus \rho_1 \vdash s_1, d_2 \Rightarrow s_2, \rho_2}{\rho \vdash s, d_1; d_2 \Rightarrow s_2, \rho_1 \oplus \rho_2}$

(type) $\dfrac{\rho, s \vdash T \Rightarrow t}{\rho \vdash s, \text{type } X = T \Rightarrow s, X \mapsto type(t)}$

(enum) $\dfrac{}{\rho \vdash s, \text{enum } X = (u_i) \Rightarrow s, (X \mapsto type(enum(n))) \oplus (u_i \mapsto const(i))}$

Si suppone che $0 \leq i < n$ e che gli u_i siano distinti.

(var) $\dfrac{\rho(X) = type(t)}{\rho \vdash (n, M), \text{var } x\colon X \Rightarrow (n + |t|, M'), x \mapsto var(n, t)}$

Le dichiarazioni di variabile provocano una estensione dello stato, M' e' una qualsiasi estensione della memoria M, p.e. quella che assegna 0 alle nuove celle. Tutte le altre dichiarazioni non modificano lo stato.

(const) $\dfrac{\rho, s \vdash e \Rightarrow m}{\rho \vdash s, \text{const } u = e \Rightarrow s, u \mapsto const(m)}$

- $\rho, s \vdash v \Rightarrow l, t$

(x) $\dfrac{\rho(x) = var(l, t)}{\rho, s \vdash x \Rightarrow l, t}$ (array) $\dfrac{\rho, s \vdash v \Rightarrow l, array(t, m) \quad \rho, s \vdash e \Rightarrow m'}{\rho, s \vdash v[e] \Rightarrow l + |t| * m', t}\ m' < m$

Per gli array si controllo che l'indice sia minore della dimensione.

(record) $\dfrac{\rho, s \vdash v \Rightarrow l, record(f)}{\rho, s \vdash v.x \Rightarrow l + m, t}\ f(x) = (t, m)$ (union.tag) $\dfrac{\rho, s \vdash v \Rightarrow l, union(x, t, f)}{\rho, s \vdash v.x \Rightarrow l, t}$

(union) $\dfrac{\rho, s \vdash_e v.x' \Rightarrow m \quad \rho, s \vdash v \Rightarrow l, union(x', t', f)}{\rho, s \vdash v.x \Rightarrow l + 1, t}\ f(x) = (t, m)$

Si controlla la consistenza della selezione con il valore del tag.

- $\rho, s \vdash e \Rightarrow m$

(u) $\dfrac{\rho(u) = const(m)}{\rho, s \vdash u \Rightarrow m}$

(var) $\dfrac{\rho, (n, M) \vdash_v v \Rightarrow l, t}{\rho, (n, M) \vdash_e v \Rightarrow m}$ t semplice e $M(l) = m$

(op) $\dfrac{\rho, s \vdash e_i \Rightarrow m_i}{\rho, s \vdash op(e_i) \Rightarrow m}\ op(m_i) = m$

- $\rho \vdash s, c \Rightarrow s'$ Se si vieta l'allocazione dinamica della memoria (cioe' il comando $new(v)$), allora e' possibile adottare una gestione a stack della memoria (vedi la regola di inferenza per un blocco). In particolare, i comandi possono modificare la memoria M dello stato $s = (n, M)$, ma non possono allocarne di nuove (cioe' incrementare n).

(:=) $\dfrac{\rho, s \vdash v \Rightarrow l, t \quad \rho, s \vdash e \Rightarrow m\colon t}{\rho \vdash s, v := e \Rightarrow s \oplus (l \mapsto m)}$

(with) $\dfrac{\rho, s \vdash v \Rightarrow l, record(x_i \mapsto (t_i, m_i)) \quad \rho \oplus (x_i \mapsto var(l + m_i, t_i)) \vdash s, c \Rightarrow s'}{\rho \vdash s, \text{with } v \text{ do } c \text{ end} \Rightarrow s'}$

$$(\text{block})\ \frac{\begin{array}{c}\rho \vdash (n, M), d \Rightarrow (n+k, M'), \rho' \\ \rho \oplus \rho' \vdash (n+k, M'), c \Rightarrow (n+k, M'')\end{array}}{\rho \vdash (n, M), d \text{ begin } c \text{ end} \Rightarrow (n, M''\lceil n)}$$

$M''\lceil n$ e' la memoria M'', il cui dominio e' $n+k$, ristretta a n. Quindi alla fine del blocco si recupera lo spazio di memoria per le variabili temporanee.

4.3 Semantica dinamica ed gestione della memoria statica e ad heap

Una semplificazione della gestione della memoria (in assenza di procedure ricorsive) la si ottiene imponendo che i descrittori di tipo $t \in D$ siano calcolati a tempo di compilazione, ovvero senza accedere allo stato dinamico. Questo permette di adottare una allocazione statica delle variabili locali, poiche' la quantita' massima di memoria necessaria al programma e' calcolabile a tempo di compilazione. A livello di semantica dinamica questo vincolo e' forzato togliendo lo stato s dalle asserzioni $\rho, s \vdash T \Rightarrow t$. Le ripercussioni a livello di sintassi sono che le espressioni e utilizzabili in dichiarazioni di tipo (sottocampo, array e union) devono essere costanti, cioe' non dipendere dallo stato dinamico s.

Se invece si vuole aggiungere i tipi puntatore (per semplicita' ignoriamo i problemi connessi alle dichiarazioni di tipi mutuamente ricorsivi) e utilizzare il comando $new(v)$ per l'allocazione dinamica di memoria, siamo costretti ad gestire la memoria a heap. A livello di semantica dinamica le aggiunte o modificare da apportare sono:

- la definizione di D va estesa $D = \ldots \mid ptr(D)$, ed inoltre $|ptr(t)| = 1$ e $m\colon ptr(t)$ e' sempre vero
- (ptr) $\dfrac{\rho(X) = type(t)}{\rho, s \vdash \uparrow X \Rightarrow ptr(t)}$
- (ptr) $\dfrac{\rho, s \vdash v \Rightarrow l, ptr(t) \quad \rho, s \vdash_e v \Rightarrow m}{\rho, s \vdash v \uparrow \Rightarrow m, t}\ m \neq nil$
- (block) $\dfrac{\begin{array}{c}\rho \vdash s, d \Rightarrow s', \rho' \\ \rho \oplus \rho' \vdash s', c \Rightarrow s''\end{array}}{\rho \vdash s, d \text{ begin } c \text{ end} \Rightarrow s''}$

  (new) $\dfrac{\rho, (n, M) \vdash v \Rightarrow l, ptr(t)}{\rho \vdash (n, M) new(v) \Rightarrow (n + |t|, M' \oplus (l \mapsto n))}$

In pratica quello che succede nelle implementazioni e' che i tre tipi di gestione della memoria coesistono e si cerca di adottare per quanto possibile quello meno costoso a run-time: statico, a stack, a heap.