Note di Linguaggi di Programmazione (III anno)
Anno Accademico 1997-98

Eugenio Moggi

January 18, 1998

1 Richiami di Sintassi

Un programma viene in genere descritto mendiante una sequenza finita di caratteri (stringa). Nella fase di analisi lessicale (presente sia nei compilatori che negli interpreti) si effettuano alcuni controlli di tipo puramente sintattico e si produce una descrizione piu' astratta del programma, che va sotto il nome di albero astratto.

1.1 Linguaggi e grammatiche

Fissato un insieme (finito) T di simboli terminali, un **linguaggio** L e' un sottoinsieme di T^*, dove T^* e' l'insieme delle sequenze finite di elementi di T (stringhe su T). Una **grammatica libera da contesto** (Context Free Grammar) e' una quadrupla $G = (T, N, P, S)$ t.c.

- T e' un insieme (finito)
- N e' un insieme (finito) disgiunto da T
- P e' un sottoinsieme (finito) di $N \times (N + T)^*$
- S e' un elemento di N

Gli elementi di T sono detti simboli terminali e vengono indicati con lettere minuscole $a, b, \ldots$. Gli elementi di N sono detti simboli non-terminali e vengono indicati con lettere mauiscole $A, B, \ldots$. Gli elementi di P sono detti produzioni e vengono rappresentati da $A ::= \alpha$ con $A \in N$ e $\alpha \in (N + T)^*$. S e' detto non-terminale iniziale.

1.2 Linguaggio associato ad una CFG

Una CFG $G = (T, N, P, S)$ identifica un insieme di regole di inferenza per derivare asserzioni della forma $\alpha \in A$ dove $A \in N$ e $\alpha \in T^*$. Piu' precisamente, ad ogni produzione $A ::= \alpha$ (di *nome* p) associamo la regola di inferenza

$$(p)\ \frac{\beta_i \in A_i \quad 1 \le i \le n}{\alpha_0\beta_1\alpha_1 \ldots \beta_n\alpha_n \in A}$$

dove $\alpha = \alpha_0 A_1 \alpha_1 \ldots A_n \alpha_n$ con $n \ge 0$, $\alpha_i \in T^*$ e $A_i \in N$ (tale decomposizione di α in stringhe di terminali alternate a simboli non-terminali e' unica).

Date queste regole possiamo associare ad ogni non-terminale $A \in N$ un linguaggio $[\![A]\!]_G \subseteq T^*$ cosi' definito: $\alpha \in [\![A]\!]_G$ sse esiste una derivazione di $\alpha \in A$ mediante le regole di inferenza associate alle produzioni di G.

Il **linguaggio** associato alla CFG G e' $L_G \stackrel{\Delta}{=} [\![S]\!]_G \subseteq T^*$. Diciamo che una CFG e' **ambigua** se esiste una stringa $\alpha \in L_G$ che ammette due alberi di derivazioni distinti, ovvero due modi diversi di inferire $\alpha \in S$.

1.3 Formalismi per descrivere CFG

Il formalismo piu' noto per descrivere CFG e' la Backus Naur Form (BNF). Altri formalismi sono vari forme di BNF estesa, ed i diagrammi sintattici.

Nella BNF i simboli non-terminali (detti anche categorie sintattiche) sono indicati con `<Nome>`, dove `Nome` dovrebbe indicare in modo informale il tipo di entita' (p.e. Espressioni, Variabili, Comandi, Identificatori) descritte con le stringhe del linguaggio corrispondente a quel non-terminale. Il formalismo BNF consente anche di raggruppare le produzioni corrispondenti ad una certa categoria sintattica scrivendo $A ::= \alpha_1 \mid \ldots \mid \alpha_n$, invece delle n produzioni $A ::= \alpha_1, \ldots, A ::= \alpha_n$.

Nelle estensioni del formalismo BNF si tendono a fornire ulteriori construtti idicare ripetizione oppure opzione (in genere si usano le notazioni $\{\alpha\}$ e $[\alpha]$). Questi costrutti possono essere eliminati introducendo nuove categorie sintattiche e produzioni.

1.4 Sintassi astratta associata ad una CFG

Ad una CFG $G = (T, N, P, S)$ possiamo associare una segnatura many-sorted Σ_G che astrae dai dettagli sintattici concreti, ovvero dai simboli terminali:

- $Sort(\Sigma) = N$, cioe' le sort sono esattamente i simboli non-terminali
- per ogni produzione $A ::= \alpha_0 A_1 \alpha_1 \ldots A_n \alpha_n$ (di *nome* p) abbiamo un simbolo di operazione $p: A_1 \ldots A_n \to A$.

I termini chiusi $t \in T_A^{\Sigma_G}$ sono in corrispondenza biunivoca con gli alberi di derivazione di asserzioni della forma $\alpha \in A$.
A partire dalla CFG G possiamo definire una particolare Σ_G-algebra:

- la sort A e' interpretata dall'insieme $[\![A]\!]_G \subseteq T^*$
- l'operazione $p: A_1 \ldots A_n \to A$, corrispondente a $A ::= \alpha_0 A_1 \alpha_1 \ldots A_n \alpha_n$, e' interpretata dalla funzione $[\![p]\!](\beta_1, \ldots, \beta_n) = \alpha_0 \beta_1 \alpha_1 \ldots \beta_n \alpha_n$

In tale algebra l'interpretazione $[\![t]\!]$ di un termine chiuso $t \in T_A^{\Sigma_G}$ e' esattamente la stringa $\alpha \in T^*$ t.c. $\alpha \in A$ e' la conclusione dell'albero di derivazione corrispondente al termine t.

1.5 Esempi di CFG

Consideriamo varie CFG per descrivere semplici espressioni, che hanno come non-terminale iniziale e' E:

$$\begin{array}{ll}
G_0 & E ::= 0 \mid 1 \mid E + E \mid E * E \\
G_1 & E ::= 0 \mid 1 \mid (E + E) \mid (E * E) \\
G_2 & E ::= 0 \mid 1 \mid (E) \mid E + E \mid E * E \\
G_3 & E ::= F \mid F + E \\
 & F ::= A \mid A * E \\
 & A ::= 0 \mid 1 \mid (E)
\end{array}$$

A livello di sintassi astratta G_0 e G_1 sono uguali, ma G_0 e' ambigua (vi sono due derivazioni di $0 + 0 + 1$), mentre G_1 genera espressioni piene di parentesi. G_2 ha gli stessi problemi di ambiguita' di G_0, ma offre la possibilita' di risolvere l'ambiguita' facendo uso delle parentesi. A livello di sintassi astratta G_2 introduce una operazione unaria, di scarso significato semantico. G_3 e' una grammatica non ambigua che descrive lo stesso linguaggio di G_2 (non e' immediato dimostrare che $L_2 \subseteq L_3$, si consiglia di procedere per induzione sulla lunghezza di una stringa di terminali), pero' per evitare l'ambiguita' e limitare l'uso delle parentesi, si complica notevolmente la sintassi astratta.

1.6 Alberi sintattici ed alberi astratti

Fissata una CFG G, diciamo che un albero ordinato con etichette sui nodi e' un **albero sintattico** per G sse:

- le etichette sono elementi di $N + T$, cioe' o terminali o non-terminali
- la radice e' etichettata con S
- se un nodo e' etichettato da un terminale allora non ha figli
- se un nodo e' etichettatto da un non-terminale A e α e' la sequenza delle etichette dei suoi figli, allora $A ::= \alpha$ e' una produzione

Le foglie di un albero sintattico sono o terminali o non-terminali A t.c. $A ::=$ e' una produzione. Un albero sintattico identifica una stringa in T^* data dalle etichette terminali lette da sinistra a destra.
Sia Σ_G la segnatura (sintassi astratta) associata a G, diciamo che un albero ordinato con etichette sui nodi e' un **albero astratto** per G sse:

- le etichette sono operazioni di Σ_G
- la radice e' etichettata da una operazione la cui sort di output e' S
- se un nodo e' etichettato da una operazione $p: A_1, \ldots, A_n \to A$, allora ha esattamente n figli e l'etichetta dell'i-esimo figlio deve avere sort di output A_i.

Un albero astratto non e' altro che la rappresentazione di un termine chiuso $t \in T_S^{\Sigma_G}$.

2 Richiami di semantica

In genere si utilizzano due tipi di semantica per descrivere i linguaggi di programmazione: quella denotazionale e quella operazionale.

La sematica dei linguaggi di programmazione puo' essere usata in vari modi: come specifica rivolta al programmatore di cosa fa' un programma (in questo si pone come alternativa a descrizioni piu' informali quali i manuali), come strumento per validare regole di tipo logiche (p.e. logica di Hoare) usate nelle dimostrazioni di correttezza, come strumento per dimostrare la correttezza di una implementazione.

Ad un certo linguaggio e' possibile dare semantiche a diverso livello di dettagli e descritte in modi molto diversi. Tuttavia e' ragionevole aspettarsi che ci sia concordanza nel predire il *comportamento osservabile*. Per esempio nel caso di un programma sequenziale quello che ci interessa osservare e' il comportamento input/output (piu' in generale l'interazione con il mondo esterno, che ha senso anche per programmi che non terminano) piuttosto che lo stato della memoria interna. Un tipo particolarmente semplice di osservazione (che spesso e' sufficiente per definire una equivalenza osservazione non banale per *parti di programma*) e' osservare la terminazione.

2.1 Semantica Denotazionale

La semantica denotazionale definisce una interpretazione (a la Tarski) dei programmi in un dominio semantico (p.e. un insieme o una struttura matematica piu' complessa). La semantica denotazionale dei linguaggi di programmazione si sviluppo a partire dagli anni '70, utilizzando le strutture matematiche proposte da Dana Scott per modellare il λ-calcolo non tipato. Tale interpretazione e' data per induzione sulla struttura astratta dei programmi (alberi astratti), e spesso si definisce l'interpretazione solo per programmi *staticamente* corretti.

La semantica denotazionale si puo' pensare anche come una traduzione (definita per induzione sulla struttura dei programmi) del linguaggio di programmazione in un linguaggio matematico. Da questo punto di vista una semantica denotazionale e' come un copilatore che ha come linguaggio target un formalismo matematico invece di un linguaggio di programmazione (a piu' basso livello).

2.2 Semantica Operazionale

La semantica operazionale e' essenzialmente una descrizione di un interprete per il linguaggio su una macchina astratta (in genere non realizzabile fisicamente). Un modo elegante di dare la semantica operazionale e' tramite regole di inferenza. Il metodo va sotto il nome di Structured Operational Semantics (SOS) e fu proposto da Plotkin verso il 1980.

Al contrario della semantica denotazionale, quella operazionale non e' necessariamente definita per induzione sulla struttura astratta dei programmi, ma e' comunque conveniente manipolare i programmi in forma astratta.

2.3 Equivalenza denotazionale e osservazionale

In semantica denotazionale vi e' una nozione ovvia di **equivalenza denotazionale** indotta dall'interpretazione: due comandi/espressioni sono denotazionalmente equivalenti se hanno la stessa interpretazioni. In effetti tale equivalenza e' una congruenza, cioe' dato un qualsiasi comando $c[\_]$ *con un buco per un comando* e due comandi denotazionalmente equivalenti $c_1 =_d c_2$, allora $c[c_1] =_d c[c_2]$.

Un'altra nozione di equivalenza definita in termini delle osservazioni e' la cosidetta **equivalenza osservazionale**, diciamo che $c_1 =_o c_2$ sse per ogni programma $p[\_]$ con un buco per un comando e test osservabile t si ha che $p[c_1]$ passa il test t sse lo passa $p[c_2]$. Anche l'equivalenza osservazionale e' una congurenza. Si osservi che la definizione di $=_o$ e' sensibile non solo alle osservazioni che e' possibile fare su un programma, ma anche all'insieme dei programmi. Quindi estendendo il set dei programmi puo' succedere che due comandi che prima dell'estensione risultavano osservazionamente equivalenti diventino distinguibili.

3 Linguaggi imperativi

3.1 Un linguaggio RAM con goto

In questa sezione descriviamo un linguaggio di programmazione per una architettura RAM costituita da una memoria con infinite celle di tipo intero indirizzabili mediante interi, e nastri di input e output contenenti una sequenza finita di interi (tenderemo ad ignorare gli aspetti legati all'input/output). Al contrario dell'architettura von Neumann assumeremo che il programma si trovi in una memoria a parte, leggibile ma non modificabile.

$$
\begin{array}{rcl}
i \in < Natural > & ::= & \ldots \\
n \in < Integer > & ::= & \ldots \\
v \in < Variable > & ::= & n \mid \uparrow n \\
e \in < Expression > & ::= & n \mid op(v_1, v_2) \\
c \in < Command > & ::= & v{:=}\, e \mid read(v) \mid write(v) \mid \\
 & & \text{if } v > 0 \text{ goto } i \\
p \in < Program > & ::= & \mid c; p
\end{array}
$$

La semantica operazionale a piccoli passi verra' descritta in termini di transizioni di stato $s \in S = Z^* \times (Z \to Z) \times Z^*$ e program counter $i \in N$. Per semplicita' ci limitiamo a modellare la sola memoria, cioe' prenderemo $S = Z \to Z$.

- $s \vdash v \Rightarrow m$

  $(m)\ \dfrac{}{s \vdash m \Rightarrow m} \qquad (\uparrow m)\ \dfrac{}{s \vdash \uparrow m \Rightarrow s[m]}$

- $s \vdash e \Rightarrow n$

  $(n)\ \dfrac{}{s \vdash n \Rightarrow n} \qquad (op)\ \dfrac{s \vdash v_i m_i \quad i = 1,2}{s \vdash op(v_1, v_2) \Rightarrow n}\ n = op(s[m_1], s[m_2])$

- $p \vdash s, i \to s', i'$ dove $p = c_1; \ldots c_n$;

  $(:=)\ \dfrac{s \vdash v \Rightarrow m \quad s \vdash e \Rightarrow n}{p \vdash s, i \to s[m \mapsto n], i+1}$ se $c_i \equiv (v{:=}\, e)$

  $(\text{goto.true})\ \dfrac{s \vdash v \Rightarrow m}{p \vdash s, i \to s, i'}$ se $c_i \equiv$ if $v > 0$ goto i' e $s[m] > 0$

  $(\text{goto.false})\ \dfrac{s \vdash v \Rightarrow m}{p \vdash s, i \to s, i+1}$ se $c_i \equiv$ if $v > 0$ goto i' e $s[m] \leq 0$

Una funzione $s{:}\, Z \to Z$ non e' rappresentabile in modo finitario, un per avere asserzioni rappresentabili in modo *finitario* e' considerare funzioni *quasi ovunque* nulle. Una semantica dovrebbe anzitutto specificare il comportamento input/output di un programma, che e' l'unico effettivamente osservabile. Quindi e' ragionevole associare ad un programma p una funzione parziale $[\![p]\!]{:}\, Z^* \rightharpoonup Z^*$ che ignora lo stato della memoria. Poiche' nella semantica operazionale abbiamo ignorato l'input/output possiamo prendere come osservazione la terminazione: $p \downarrow$ sse eseguendo p a partire dalla prima istruzione con stato iniziale s_0 si termina. Piu' formalmente $p \downarrow$ sse $\exists s{:}\, S. p \vdash s_0, 1 \Rightarrow s$, dove s_0 e' uno stato prefissato (p.e. quello in cui tutte le celle di memoria contengono 0) e $p \vdash s, i \Rightarrow s'$ e' la semantica operazionale a grandi passi definita dalle regole

$$(\text{step})\ \dfrac{\begin{array}{c} p \vdash s, i \to s', i' \\ p \vdash s', i' \Rightarrow s'' \end{array}}{p \vdash s, i \Rightarrow s''} \qquad (\text{halt})\ \dfrac{p \vdash s, i \not\to}{p \vdash s, i \Rightarrow s}$$

3.2 Un linguaggio RAM strutturato

$$
\begin{array}{rcl}
n \in < Integer > & ::= & \ldots \\
v \in < Variable > & ::= & n \mid \uparrow v \mid v[e] \\
e \in < Expression > & ::= & n \mid v \mid op(e_1, e_2) \\
c \in < Command > & ::= & v{:=}\, e \mid read(v) \mid write(v) \mid \\
 & & \text{if } e > 0 \text{ then } sc_1 \text{ else } sc_2 \text{ end} \mid \\
 & & \text{while } e > 0 \text{ do } sc \text{ end} \mid \\
 & & \text{case } e \text{ of } sa \text{ end} \mid \\
 & & \text{loop } sc \text{ end} \mid \text{exit} \mid \text{return} \\
sc \in < Command - Sequence > & ::= & \mid c; sc \\
sa \in < Alternative - Sequence > & ::= & \text{default } sc \mid n{:}\, sc | sa \\
p \in < Program > & ::= & sc
\end{array}
$$

- $s \vdash v \Rightarrow m$

$$(m)\ \frac{}{s \vdash m \Rightarrow m} \qquad (\uparrow)\ \frac{s \vdash \uparrow v \Rightarrow m}{s \vdash \uparrow v \Rightarrow s[m]} \qquad ([\,])\ \frac{s \vdash v \Rightarrow m \quad s \vdash e \Rightarrow n}{s \vdash v[e] \Rightarrow m+n}$$

- $s \vdash e \Rightarrow n$

$$(n)\ \frac{}{s \vdash n \Rightarrow n} \qquad (v)\ \frac{s \vdash_V v \Rightarrow m}{s \vdash_E v \Rightarrow s[m]}$$

$$(op)\ \frac{s \vdash e_i n_i \quad i = 1,2}{s \vdash op(e_1, e_2) \Rightarrow n}\ n = op(n_1, n_2)$$

- $c \vdash s \Rightarrow s', r$ dove $r ::= ok \mid return \mid exit$ indica il tipo di terminazione

$$(:=)\ \frac{s \vdash v \Rightarrow m \quad s \vdash e \Rightarrow n}{x := v \vdash s \Rightarrow s[m \mapsto n], ok}$$

$$(\text{if.true})\ \frac{s \vdash e \Rightarrow n \quad sc_1 \vdash s \Rightarrow s', r}{\text{if } e > 0 \text{ then } sc_1 \text{ else } sc_2 \text{ end} \vdash s \Rightarrow s', r}\ n > 0$$

$$(\text{if.false})\ \frac{s \vdash e \Rightarrow n \quad sc_2 \vdash s \Rightarrow s', r}{\text{if } e > 0 \text{ then } sc_1 \text{ else } sc_2 \text{ end} \vdash s \Rightarrow s', r}\ n \leq 0$$

$$(\text{while.true})\ \frac{\begin{array}{c} s \vdash e \Rightarrow n \quad sc \vdash s \Rightarrow s', ok \\ \text{while } e > 0 \text{ do } sc \text{ end} \vdash s' \Rightarrow s'', r \end{array}}{\text{while } e > 0 \text{ do } sc \text{ end} \vdash s \Rightarrow s'', r}\ n > 0$$

$$(\text{while.true.fail})\ \frac{s \vdash e \Rightarrow n \quad sc \vdash s \Rightarrow s', r}{\text{while } e > 0 \text{ do } sc \text{ end} \vdash s \Rightarrow s', r}\ r \neq ok$$

$$(\text{while.false})\ \frac{s \vdash e \Rightarrow n}{\text{while } e > 0 \text{ do } sc \text{ end} \vdash s \Rightarrow s, ok}\ n \leq 0$$

$$(\text{case})\ \frac{s \vdash e \Rightarrow n \quad sa \vdash n, s \Rightarrow s', r}{\text{case } e \text{ of } sa \text{ end} \vdash s \Rightarrow s', r}$$

$$(\text{loop.ok})\ \frac{\begin{array}{c} sc \vdash s \Rightarrow s', ok \\ \text{loop } sc \text{ end} \vdash s' \Rightarrow s'', r \end{array}}{\text{loop } sc \text{ end} \vdash s \Rightarrow s'', r} \qquad (\text{loop.exit})\ \frac{sc \vdash s \Rightarrow s', exit}{\text{loop } sc \text{ end} \vdash s \Rightarrow s', ok}$$

$$(\text{loop.fail})\ \frac{sc \vdash s \Rightarrow s', r}{\text{loop } sc \text{ end} \vdash s \Rightarrow s', r}\ r = return$$

- $sc \vdash s \Rightarrow s', r$

$$(\,)\ \frac{}{\vdash s \Rightarrow s, ok}$$

$$(;)\ \frac{c \vdash s \Rightarrow s', ok \quad sc \vdash s' \Rightarrow s'', r}{c; sc \vdash s \Rightarrow s'', r} \qquad (;\text{fail})\ \frac{c \vdash s \Rightarrow s', r}{c; sc \vdash s \Rightarrow s', r}$$

- $sa \vdash n, s \Rightarrow s', r$

$$(\text{default})\ \frac{sc \vdash s \Rightarrow s', r}{\text{default } sc \vdash n, s \Rightarrow s', r} \qquad (\text{match})\ \frac{sc \vdash s \Rightarrow s', r}{n : sc | as \vdash n, s \Rightarrow s', r}$$

$$(\text{mismatch})\ \frac{as \vdash n, s \Rightarrow s', r}{m : sc | as \vdash n, s \Rightarrow s', r}\ m \neq n$$

Sui programmi RAM strutturati possiamo definire l'osservazione $p \downarrow$ come $\exists s, r. p \vdash s_0, 1 \Rightarrow s, r$, dove non discriminiamo tra i tre tipi di terminazione.

Sui programmi del linguaggio RAM strutturato e' sensato effettuare dei controlli statici per segnalare usi sospetti di certi costrutti che potrebbe essere sintomo di un errore logico. Per esempio, si potrebbe avvertire il programmatore che ha usato un "exit" fuori dal corpo di un loop (la semantica ci dice che cio' porta alla terminazione del programma), o che ha etichettatto due alternative di un "case" con la stessa constante (la semantica ci dice che solo la prima e' presa in considerazione).

Tali controlli non possono essere effettuati in fave di analisi lessicale (cioe' mediante usando una CFG), ma si possono descrivere con regole di inferenza *dirette dalla sintassi*, che permettono di decidere se un asserto e' derivabile o no. Questa proprieta' di decidibilita' contrasta con la impossibilita' di decidere se un asserto della forma $p \vdash s \Rightarrow s'$, r e' derivabile, ed e' la proprieta' cruciale che distingue la semantica statica da quella dinamica.

- $\vdash c \Rightarrow r$ dove $r \subseteq Error = \{exit, case\}$ indica il tipo di errori presenti in c

  $$(:=)\ \frac{}{\vdash x := v \Rightarrow}$$

  $$(\text{if})\ \frac{\vdash sc_i \Rightarrow r_i \quad i = 1,2}{\vdash \text{if } e > 0 \text{ then } sc_1 \text{ else } sc_2 \text{ end} \Rightarrow r_1 \cup r_2}$$

  $$(\text{while})\ \frac{\vdash sc \Rightarrow r}{\vdash \text{while } e > 0 \text{ do } sc \text{ end} \Rightarrow r}$$

  $$(\text{case})\ \frac{\vdash sa \Rightarrow r}{\vdash \text{case } e \text{ of } sa \text{ end} \Rightarrow r \cap Error}$$

  $$(\text{loop})\ \frac{\vdash sc \Rightarrow r}{\vdash \text{loop } sc \text{ end} \Rightarrow r \cap \{case\}}$$

  $$(\ )\ \frac{}{\vdash \ \Rightarrow \emptyset}$$

  $$(;)\ \frac{\vdash c \Rightarrow r \quad \vdash sc \Rightarrow r'}{\vdash c; sc \Rightarrow r \cup r'}$$

- $\vdash sa \Rightarrow r$ dove $r \subseteq Error \cup Z$

  $$(\text{default})\ \frac{\vdash sc \Rightarrow r}{\vdash \text{default } sc \Rightarrow r}$$

  $$(\text{case.ok})\ \frac{\vdash sc \Rightarrow r \quad \vdash sa \Rightarrow r'}{\vdash n : sc | as \Rightarrow r \cup r' \cup \{n\}}\ n \notin r'$$

  $$(\text{case.fail})\ \frac{\vdash sc \Rightarrow r \quad \vdash sa \Rightarrow r'}{\vdash n : sc | as \Rightarrow r \cup r' \cup \{case\}}\ n \in r'$$

3.3 Esercizi

1. aggiungere al linguaggio RAM strutturato il comando "continue" del C, che permette di interrompere il ciclo corrente e passare al prossimo ciclo. Estendere la semantica operazionale statica e dinamica.

2. modificare la semantica dei costrutti "case" ed "exit", cercando di mimare il comportamento dei costrutti "switch" e "break" del C.

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.

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

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).

$X \overset{fin}{\to} Y$ 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:

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

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

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

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

  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.

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

  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).

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

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

  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).

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

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

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

  In Pascal si richiede che ρ_i abbiano domini disgiunti

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

  Solo le dichiarazioni di tipo provocano una estensione dello stato statico

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

  (var) $\dfrac{\rho(X) = type(m)}{\rho \vdash s, \text{var } x\colon X \Rightarrow (0,0), x \mapsto var(m)}$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$m{:}t \stackrel{\Delta}{\iff} (t = enum(n) \land m < n) \lor (t = range(m_1, m_2) \land m_1 \leq m \leq m_2)$
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:

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

- $\rho \vdash s, c \Rightarrow s'$ 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).

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

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

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

$M''\lceil n$ 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 $D = \dots \mid ptr(D)$, ed inoltre $|ptr(t)| = 1$ e $m{:}\,ptr(t)$ e' sempre vero
- $(\text{ptr}) \ \dfrac{\rho(X) = type(t)}{\rho, s \vdash \uparrow X \Rightarrow ptr(t)}$
- $(\text{ptr}) \ \dfrac{\rho, s \vdash v \Rightarrow l, ptr(t) \quad \rho, s \vdash_e v \Rightarrow m}{\rho, s \vdash v \uparrow \Rightarrow m, t} \ m \neq nil$
- $(\text{block}) \ \dfrac{\begin{array}{c}\rho \vdash s, d \Rightarrow s', \rho' \\ \rho \oplus \rho' \vdash s', c \Rightarrow s''\end{array}}{\rho \vdash s, d \text{ begin } c \text{ end} \Rightarrow s''}$

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

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.

5 Procedure

Le procedure sono uno meccanismo fondamentale per strutturare i programmi. Esse sono uno strumento essenziale per supportare la metodologia di sviluppo top-down, facilitano il riutilizzo del codice, e facilitano una programmazione modulare (chi usa una procedure non ha bisogno di sapere come e' implementata, e viceversa). All'inizio consideriamo il caso semplificato delle procedure non ricorsive e senza parametri, descrivendo le estensioni da apportare alla semantica operazionale statica e dinamica.

$$\begin{array}{rcl} d \in < Declaration > & ::= & \ldots \mid \mathrm{proc}\ P = c \\ c \in < Command > & ::= & \ldots \mid P \end{array}$$

Per gli identificatori adottiamo la convenzione che P e' un identificatore di procedura.
Vi sono due modalita' di gestione dell'ambiente direttamente connesse alle procedure: ambiente statico, e ambiente dinamico. Nella gestione statica dell'ambiente una chiamata di procedura P equivale ad eseguire il corpo c nell'ambiente in cui P e' dichiarata (e nello stato al momento della chiamata). Nella gestione dinamica invece si utilizza ambiente (e stato) al momento della chiamata. La gestione statica e' preferibile, perche' chi scrive il corpo c della procedure P conosce l'ambiente al momento della dichiarazione, ma non e' detto che sappia in quali ambienti P verra' chiamata. Anzi lo spirito della programmazione modulare e' proprio quello di ridurre al minimo l'interazione tra chi scrive c e chi utilizza P. Inoltre la gestione dinamica invalida principi basilari, quali il fatto che la redenominazione dei nomi locali non abbia conseguenze osservabili.

5.1 Semantica delle procedure con scoping statico

Cosideriamo le aggiunte/modifiche alla semantica statica per trattare le procedure con gestione dell'ambiente statica.

- $\rho \in Env^S = Id \stackrel{fin}{\to} \ldots \mid proc$ ambiente statico
- $s \in State^S = \ldots$ stato statico, non e' direttamente rilevante
- Le nuove regole di inferenza per la semantica statica sono:

  $$(\mathrm{proc})\ \frac{\rho, s \vdash c \Rightarrow ok}{\rho \vdash s, \mathrm{proc}\ P = c \Rightarrow (0,0), P \mapsto proc} \qquad (\mathrm{call})\ \frac{\rho(P) = proc}{\rho, s \vdash P \Rightarrow ok}$$

Per trattare procedure mutuamente ricorsive a livello di semantica statica basta sostituire la regola (block) con (block*), come gia' visto per i tipi mutuamente ricorsivi.
Per la semantica operazionale dinamica le aggiunte/modifiche sono

- $\rho \in Env = Id \stackrel{fin}{\to} \ldots \mid proc(Env, Command)$ ambiente dinamico.

  Una coppia (ρ, c) e' detta chiusura, e permette di associare ad un identificatore P di procedure non solo il corpo della procedura, ma l'ambiente al momento della dichiarazione
- Le nuove regole di inferenza per la semantica dinamica sono:

  $$(\mathrm{proc})\ \frac{}{\rho \vdash s, \mathrm{proc}\ P = c \Rightarrow s, P \mapsto proc(\rho, c)} \qquad (\mathrm{call})\ \frac{\begin{array}{c}\rho(P) = proc(\rho', c) \\ \rho' \vdash s, c \Rightarrow s'\end{array}}{\rho \vdash s, P \Rightarrow s'}$$

Per trattare procedure mutuamente ricorsive a livello di semantica dinamica occorrono alcune modifiche alla nozione di chiusura ed alla gestione dell'ambiente

- $\rho \in Env = Id \stackrel{fin}{\to} \ldots \mid proc(Env, Env, Command)$ ambiente dinamico.

  Una terna (ρ, ρ', c) e' una chiusura in cui ρ e' la parte non ricorsiva dell'ambiente, mentre ρ' e' la parte dell'ambiente definito in modo ricorsivo, cioe' gli identificatori nel dominio di ρ' possono essere usati nel codominio.
- $(\mathrm{proc})\ \dfrac{}{\rho \vdash s, \mathrm{proc}\ P = c \Rightarrow s, P \mapsto proc(\rho, 0, c)}$

  La parte ricorsiva e' vuota perche' l'ambiente ricorsivo e' noto solo dopo aver esaminato tutta la dichiarazione.

In presenza di dichiarazioni mutuamente ricorsive e' necessario rivedere anche la regola di composizione di dichiarazioni (l'ordine di dichiarazione diventa irrilevante) e quella per un blocco (che deve inserire l'ambiente ricorsivo nelle chiusure)

$$\text{(comp*)}\ \frac{\begin{array}{c}\rho \vdash s, d_1 \Rightarrow s_1, \rho_1 \\ \rho \vdash s_1, d_2 \Rightarrow s_2, \rho_2\end{array}}{\rho \vdash s, d_1; d_2 \Rightarrow s_2, \rho_1 \oplus \rho_2}\ \rho_i \text{ con domini disgiunti}$$

se vi sono solo dichiarazioni di procedure, lo stato non cambia e quindi l'ordine di dichiarazione e' irrilevante

$$\text{(block)}\ \frac{\begin{array}{c}\rho \vdash s, d \Rightarrow s', \rho' \\ \rho \oplus Rec\ \rho' \vdash s', c \Rightarrow s''\end{array}}{\rho \vdash s, d \text{ begin } c \text{ end} \Rightarrow s''}$$

dove $(Rec\ \rho')(P) \stackrel{\Delta}{=} (\rho, \rho', c)$ se $\rho'(P) = (\rho, \_, c)$. Quindi $Rec\ \rho'$ informa le chiusure che l'ambiente ricorsivo e' ρ'

- $$\text{(call)}\ \frac{\begin{array}{c}\rho(P) = proc(\rho', \rho'', c) \\ \rho' \oplus Rec\ \rho'' \vdash s, c \Rightarrow s'\end{array}}{\rho \vdash s, P \Rightarrow s'}$$

  Anche in questo caso si deve informare le varie chiusure in ρ'' che l'ambiente ricorsivo e' ρ''

5.2 Semantica delle procedure con scoping dinamico

Lo scoping dinamico equivale ad una sostituzione testuale di c al posto di una chiamata di P (simile a cio' che avviene con la macro espansione), e quindi il significato degli identificatori usati in c (ma non dichiarati localmente a c) sono interpretati nell'ambiente del chiamante, che varia da chiamata a chiamata. Si osservi che quando c usa solo identificatori dichiarati localmente in c (o parametri formali), le due modalita' di gestione diventano equivalenti.

- $\rho \in Env^S = Id \stackrel{fin}{\rightarrow} \ldots \mid proc(Command)$ ambiente statico
- $s \in State^S = \ldots$ stato statico, non e' direttamente rilevante
- Le nuove regole di inferenza per la semantica statica sono:

  $$\text{(proc)}\ \frac{}{\rho \vdash s, \text{proc } P = c \Rightarrow (0,0), P \mapsto proc(c)} \qquad \text{(call)}\ \frac{\begin{array}{c}\rho(P) = proc(c) \\ \rho, s \vdash c \Rightarrow ok\end{array}}{\rho, s \vdash P \Rightarrow ok}$$

  Con lo scoping statico i controlli statici (type-checking) su c venivano effettuati una sola volta (nell'ambiente in cui P e' dichiarata), adesso vengono effettuati per ogni chiamata di P.

Per la semantica operazionale dinamica le aggiunte/modifiche sono

- $\rho \in Env = Id \stackrel{fin}{\rightarrow} \ldots \mid proc(Command)$ ambiente dinamico.
- Le nuove regole di inferenza per la semantica dinamica sono simili a quelle per la semantica statica

  $$\text{(proc)}\ \frac{}{\rho \vdash s, \text{proc } P = c \Rightarrow s, P \mapsto proc(c)} \qquad \text{(call)}\ \frac{\begin{array}{c}\rho(P) = proc(c) \\ \rho \vdash s, c \Rightarrow s'\end{array}}{\rho \vdash s, P \Rightarrow s'}$$

Con gestione dinamica dell'ambiente non sono richiesti particolari accorgimenti per trattare le definizioni ricorsive.

5.3 Procedure con parametri

La flessibilita' delle procedure e' notevolmente accresciuta se e' possibile parametrizzarle rispetto a dei parametri che possono essere istanziati in modo diverso da chiamata a chiamata. In effetti la parametrizzazione e' un concetto del tutto generale, che puo' essere utilmente applicato ad altre categorie sintattiche oltre ai comandi (espressioni, variabili, dichiarazioni). Vi e' anche molta discrezionalita' su cosa puo' essere utilizzato come parametro. Un liguaggio che offre buoni meccanismi di parametrizzazione facilita enormemente il riutilizzo del codice. Si consideri un algoritmo di ordinamento basato su confronti, p.e. mergesort o quicksort. I parametri naturali dell'algoritmo sono un insieme X con un ordine totale $\leq : X, X \rightarrow Bool$ ed una sequenza A costituita da n elementi di X. Quindi vorremo poter scrivere

```
proc mergesort (type X; fun le(X,X):Bool; const n; var A:X[n])
```

Addirittura potremo aspettarci che i parametri X ed n siano ricavati automaticamente dagli altri due. Da questo punto di vista sia il C che il Pascal lasciano abbastanza a desiderare, p.e. i parametri X ed n non sono consentiti (poiche' il C tratta gli array come puntatori e' possibile passare un array senza fornire le sue dimenzioni). Vi sono alcune estensioni del Pascal che consentono il parametro n, e vredremo che ML (ed altri linguaggi funzionali) consentono di inferire il parametro X.

Consideriamo il seguente linguaggio, dove abbiamo introdotto un meccanismo uniforme di parametrizzazione per comandi, espressioni e variabili. Inoltre per illustrare meglio la simmetria tra parametrizzazione e dichiarazione abbiamo adottato la keyword "newvar" per la dichiarazione di una nuova variabile (che richiede l'allocazione di memoria), ed utilizzato la keyword "var" per una nuova forma di dichiarazione che permette di associare un identificatore ad una variabile gia allocata.

$$
\begin{array}{l}
u, x, X, P, F, S \in < Identifier > ::= \ldots \\
T \in < Type > ::= X \mid \ldots \\
\begin{array}{l|l}
d \in < Declaration > ::= & fp \in < FormalParam > ::= \\
\text{type } X = T \mid & \\
\text{newvar } x{:}\, X \mid & \\
\text{var } x = v \mid & \text{var } x{:}\, X \mid \\
\text{const } u = e \mid & \text{const } u{:}\, X \mid \\
\text{proc } P(fp^*) = c \mid & \text{proc } P(fp^*) \mid \\
\text{fun } F(fp^*) = e \mid & \text{fun } F(fp^*){:}\, X \mid \\
\text{sel } S(fp^*) = v & \text{sel } S(fp^*){:}\, X
\end{array} \\
ap \in < ActualParam > ::= v \mid e \mid P \mid F \mid S \\
v \in < Variable > ::= \ldots \mid S(ap^*) \\
e \in < Expression > ::= \ldots \mid F(ap^*) \\
c \in < Command > ::= \ldots \mid P(ap^*)
\end{array}
$$

Per gli identificatori adottiamo la convenzione che P e' un identificatore di procedura, F di funzione, e S di selettore.

A livello di semantica statica e dinamica (con scoping statico) le principali modifiche riguardano l'ambiente. Per semplicita' ci limitiamo a trattare il caso di procedure con parametri, poiche' le altre forme di parametrizzazione sono del tutto analoghe. Per la semantica statica abbiamo:

- $\rho \in Env^S = Id \stackrel{fin}{\to} \ldots \mid proc(FD^*)$ ambiente statico, FD e' l'insieme dei descrittori ft di parametro formale $FD = var(N) \mid const(N) \mid proc(FD^*)$, dove N e' l'insieme dei nomi di tipo.
- $\rho, s \vdash fp \Rightarrow \rho'$ e' una asserzione che associa un ambiente ad una dichiarazione di parametro formale

  $$(\text{var})\ \frac{\rho(X) = type(m)}{\rho, s \vdash \text{var } x{:}\, X \Rightarrow x \mapsto var(m)} \qquad (\text{const})\ \frac{\rho(X) = type(m)}{\rho, (n, g) \vdash \text{const } u{:}\, X \Rightarrow u \mapsto const(m)}\ g(m) = enum$$

  $$(\text{proc})\ \frac{\rho, s \vdash fp_i \Rightarrow x_i \mapsto ft_i}{\rho, s \vdash \text{proc } P(fp_i) \Rightarrow P \mapsto proc(ft_i)}\ g(m) = enum$$

- Le nuove regole di inferenza per la semantica statica sono:

  $$(\text{proc})\ \frac{\begin{array}{c}\rho, s \vdash fp_i \Rightarrow x_i \mapsto ft_i \\ \rho \oplus (x_i \mapsto ft_i), s \vdash c \Rightarrow ok\end{array}}{\rho \vdash s, \text{proc } P(fp_i) = c \Rightarrow (0, 0), P \mapsto proc(ft_i)} \qquad (\text{call})\ \frac{\begin{array}{c}\rho(P) = proc(ft_i) \\ \rho, s \vdash ap_i{:}\, ft_i\end{array}}{\rho, s \vdash P(ap_i) \Rightarrow ok}$$

  dove $\rho, s \vdash ap{:}\, ft$ controlla la compatibilita' tra parametro formale ed attuale, ed e' un abbreviazione per: $\rho, s \vdash_v ap \Rightarrow m$ se $ft = var(m)$, $\rho, s \vdash_e ap \Rightarrow m$ se $ft = const(m)$, e $\rho(ap) = ft$ se $ft = proc(ft_i)$.

Per la semantica dinamica abbiamo:

- $\rho \in Env = Id \stackrel{fin}{\to} \ldots \mid proc(Id^*, Env, Env, Command)$ ambiente dinamico, dove adesso una chiusura specifica anche gli identificatori dei parametri formali
- Le nuove regole di inferenza per la semantica dinamica sono:

  $$(\text{proc})\ \frac{x_i = \text{identificatore dichiarato in } fp_i}{\rho \vdash s, \text{proc } P(fp_i) = c \Rightarrow s, P \mapsto proc(x_i, \rho, 0, c)} \qquad (\text{call})\ \frac{\begin{array}{c}\rho(P) = proc(x_i, \rho', \rho'', c) \\ \rho, s \vdash ap_i \Rightarrow a_i \\ \rho' \oplus (x_i \mapsto a_i) \oplus Rec\ \rho'' \vdash s, c \Rightarrow s'\end{array}}{\rho \vdash x, P(ap_i) \Rightarrow s'}$$

  dove $\rho, s \vdash ap \Rightarrow a$ calcola il parametro attuale da associare al corrispondente identificatore di parametro formale, p.e. si deve avere $\rho, s \vdash_v ap_i var(l, t)$ quando il parametro formale fp_i e' var $x{:}\, X$. Per semplicita' abbiamo ignorato alcuni controlli di tipo a run-time.

5.4 Esercizi

1. Supponiamo di non ammettere ne' parametri di tipo procedura ne' procedure ricorsive, ovvero una procedura P puo' chiamare una procedura Q solo se Q e' visibile secondo le regole dello scoping statico e la definizione di Q precede (testualmente) quella di P. Si faccia vedere che in fase di esecuzione sullo stack non vi possono essere 2 record di attivazione per la stessa procedura.

2. Usando il precedente risultato discutere la possibilita' di gestione statica della memoria nelle ipotesi che: il layout dei tipi sia determinato staticamente (e non vi siano puntatori), le procedure non siano ricorsive e non possano essere passate come parametro.

3. Far vedere con un esempio che il requisito di non avere procedure passate come parametro non e' eliminabile.

4. Applicare le regole della semantica operazionale dinamica per procedure ricorsive (con scoping statico) al seguente programma

   ```
   begin
   proc P=Q;
   proc Q=P;
   P
   end
   ```

   serve a familiarizzarsi con l'uso dell'albiente mutuamente ricorsivo

5. Applicare le regole della semantica operazionale dinamica con scoping statico e con scoping dinamico al seguente programma, e dire cosa viene stampato nei due casi

   ```
   begin
   var x:int;
   proc P=(x:=1);
   x:=0;
   begin
     var x:int;
     x:=0;
     P; write(x)
   end
   P;write(x)
   end;
   ```

6. (difficile) Rivedere la semantica operazione dinamica per il linguaggio che ammette dichiarazioni di procedure e procedure come parametro (per semplicita' non si ammetta la ricorsione) in modo da trattare correttamenta i controlli al momento della chiamata (evitare quei controlli che sono gia' fattibili a tempo di compilazione).

7. (difficile) Si vuole modificare la gestione dei tipi union in modo da migliorare gli aspetti legati alla sicurezza dei dati: il campo tag $v.x$ e' trattato come una costante (cioe' puo' essere letto ma non modificato in modo diretto), quando si effettua un assegnamento ad una variante $v.x_i$ si modifica di conseguenza il tag $v.x$, quando si legge una variante $v.x_i$ si controlla che il tag $v.x$ abbia il valore giusto. Spiegare come vanno modificate la semantica statica e dinamica.

6 Il lambda calcolo non tipato

Il λ-calcolo e' la teoria che ha ispirato i linguaggi funzionali, nello stesso modo in cui la logica matematica ha ispirato i linguaggi per la programmazione logica. In effetti, il λ-calcolo e' uno strumento teorico estremamente versatile per studiare i linguaggi di programmazione in generale.
Vedremo il λ-calcolo da tre punti di vista: come teoria equazionale, come teoria della semplificazione, come linguaggio di programmazione (descritto mediante una semantica operazionale).

6.1 La teoria equazionale

I termini del λ-calcolo puro (e non tipato) sono descritti dalla seguente BNF

$$M \in \Lambda \quad ::= \quad x \mid M_1 M_2 \mid \lambda x.M$$

Nelle applicazioni si tende ad arricchire il linguaggio con costanti e/o operazioni (di arieta' prefissata)

$$M \in \Lambda \quad ::= \quad x \mid c \mid op(\overline{M}) \mid M_1 M_2 \mid \lambda x.M$$

Per il momento ci limiteremo al λ-calcolo puro, che nonostante la sua semplicita' ha *la stessa potenza* di calcolo delle macchine di Turing. La teoria equazionale del λ-calcolo e' definita dai seguenti assiomi e regole di inferenza per derivare asserti della forma $M = N$

- regole che asseriscono che $=$ e' una relazione di equivalenza (cioe' riflessiva, simmetrica e transitiva)
- regole che asseriscono che $=$ e' una congruenza, cioe'

  $$(\mathrm{app})\ \frac{M_1 = M_1' \qquad M_2 = M_2'}{M_1 M_2 = M_1' M_2'} \qquad\qquad (\xi)\ \frac{M = M'}{(\lambda x.M) = (\lambda x.M')}$$

- l'assioma che permette di ridenominare le variabili legate

  (α) $(\lambda x.M) = (\lambda y.[y/x]M)$ $y \notin \mathrm{FV}(M)$

  questa proprieta' e' tipica delle variabili legate, p.e. nella logica del primo ordine l'analogo di tale assioma asserisce l'equivalenza logica delle formule$\forall x.\phi$ e $\forall x.([y/x]\phi)$
- l'assioma che permette di semplificare l'applicazione di una funzione ad un argomento

  (β) $(\lambda x.M)N = [N/x]M$

  questa e' lunica regola che ha un significato computazionale
- un'altro assioma, che pero' noi non considereremo, asserisce che ogni termine e' una funzione

  (η) $M = (\lambda x.Mx)$ $x \notin \mathrm{FV}(M)$

  tale assioma e' sensato per il λ-calcolo puro, tuttavia se si aggiungono al linguaggio costanti che non corrispondono a funzioni la regola η diventa poco plausibile.

Negli assiomi α e β si e' fatto uso dell'insieme $\mathrm{FV}(M)$ delle variabili libere e dell'operazione $[N/x]M$ di sostituzione. La definizione di queste nozioni richiede un po' di attenzione. poiche' i termini del λ-calcolo contengono variabili legate (al contrario dei termini algebrici). Le soluzioni adottate nel λ-calcolo sono facilmente adattabili a qualsiasi formalismo in cui le variabili possono essere legate.

Definition 6.1 *L'insieme* $\mathrm{FV}(M)$ *e' definitio per induzione sulla struttura di* M

$$\begin{aligned} \mathrm{FV}(x) &= \{x\} \\ \mathrm{FV}(M_1 M_2) &= \mathrm{FV}(M_1) \cup \mathrm{FV}(M_2) \\ \mathrm{FV}(\lambda x.M) &= \mathrm{FV}(M) - \{x\} \end{aligned}$$

L'operazione di sostituzione $[N/x]M$ *e' definita per induzione sulla dimensione* $|M|$ *di* M*, o piu' precisamente il numero di* λ *ed applicazioni in* M *(tale numero e' invariante per ridenominazione di variabili)*

$$\begin{aligned} [N/x]x &= N \\ [N/x]y &= y \ \mathit{se}\ x \not\equiv y \\ [N/x](M_1 M_2) &= ([N/x]M_1 [N/x]M_2) \\ [N/x](\lambda y.M) &= \lambda z.[N/x]([z/y]M) \ \mathit{dove}\ z \notin \mathrm{FV}(\lambda y.M) \cup \mathrm{FV}(N) \cup \{x\} \end{aligned}$$

Una definizione alternativa di sostituzione che evita per quanto possibile la ridenominazione delle variabili legate e'

$$\begin{aligned}
[N/x](\lambda x.M) &= \lambda x.M \\
[N/x](\lambda y.M) &= \lambda y.[N/x]M \text{ se } x \not\equiv y \text{ e } y \notin \mathrm{FV}(N), \text{ altrimenti} \\
[N/x](\lambda y.M) &= \lambda z.[N/x]([z/y]M) \text{ dove } z \notin \mathrm{FV}(\lambda y.M) \cup \mathrm{FV}(N) \cup \{x\}
\end{aligned}$$

L'ultimo caso nella definizione di sostituzione dice che prima di effettuare la sostituzione di x con N si deve ridenominare la variabile legata y con una *nuova* variabile z. Questa ridenominazione serve ad evitare l'effetto indesiderato che si avrebbe con la definizione naive $[N/x]\lambda y.M = \lambda y.[N/x]M$, ovvero legare le occorenze libere di y in N.
La definizione di sostituzione e' data in modo non-deterministico, poiche' possiamo scegliere qualsiasi $z \notin \mathrm{FV}(\lambda y.M) \cup \mathrm{FV}(N) \cup \{x\}$. E' possibile fissare un criterio di scelta deterministico, p.e. fissando un ordine totale sulle variabili. Nella pratica si tende ad ignorare il problema, questa mancanza di precisione non e' pericolosa purche' si tenga presente che quello che si sta manipolango non sono termini ma classi di equivalenza di termini modulo *α-conversione*. A tal proposito e' opportuno distinguere tre relazioni di congruenza sui λ-termini

- $M \equiv N$ e' l'uguaglianza sintattica
- $M \equiv_\alpha N$ e' l'α-conversione, cioe' la congruenza indotta da assiomi e regole di inferenza rimuovendo la regola β
- $M =_\beta N$ e' la β-conversione, cioe' la congruenza indotta da assiomi e regole di inferenza.

Scriviamo $\equiv_\alpha$ invece di $=_\alpha$ per sottolineare che l'α-conversione e' piu' opportuno considerarla come la corretta definizione di uguaglianza sintattica in presenza di variabili legate.

Proposition 6.2 *Se $M \equiv_\alpha M'$ e $N \equiv_\alpha N'$, allora $\mathrm{FV}(M) = \mathrm{FV}(M')$ e $[N/x]M \equiv_\alpha [N'/x]M'$.*

Lavorando modulo α-conversione e' sempre possibile effettuare una ridenominazione delle variabili legate in M in modo da evitare conflitti con le variabili libere di N, dopodiche' si puo' usare senza pericolo anche una definizione naive di sostituzione $[N/x]M$ (che non effettua la ridenominazione).

6.2 La nozione di riduzione

Introduciamo adesso una relazione $M \rightarrow_\beta N$ tra termini, detta β-riduzione, che formalizza l'idea che N e' ottenuto da M effettuando un passo di *semplificazione*.

Definition 6.3 *La relazione $\rightarrow_\beta$ su Λ e' definita dalle seguente regole di inferenza:* (β) $(\lambda x.M)N \rightarrow_\beta [N/x]M$

$$(\text{app.1})\ \frac{M_1 \rightarrow_\beta M_1'}{M_1 M_2 \rightarrow_\beta M_1' M_2} \qquad (\text{app.2})\ \frac{M_2 \rightarrow_\beta M_2'}{M_1 M_2 \rightarrow_\beta M_1 M_2'} \qquad (\xi)\ \frac{M \rightarrow_\beta M'}{(\lambda x.M) \rightarrow_\beta (\lambda x.M')}$$

Si noti che la relazione $\rightarrow_\beta$ e' decidibile (al contrario di $=_\beta$), e per ogni M esiste un numero finito di N t.c. $M \rightarrow_\beta N$. una proprieta' fondamentale della β-riduzione e' la **proprieta' Church-Rosser**.

Theorem 6.4 *Se $M \rightarrow^* M_1$ e $M \rightarrow^* M_2$ allora esiste M_3 t.c. $M_1 \rightarrow^* M_3$ e $M_2 \rightarrow^* M_3$.*

Si tenga present eche stiamo lavorando modulo α-conversione, quindi volendo essere piu' espliciti dovremo scrivere

$$\begin{tikzcd}
 & M \arrow[dl] & \equiv_\alpha & M' \arrow[dr] & \\
M_1 \arrow[dr] & & & & M_2 \arrow[dl] \\
 & M_3 & \equiv_\alpha & M_3' &
\end{tikzcd}$$

Tale proprieta' ci dice che nonostante la relazione di semplificazione sia non deterministica, due semplificazioni M_1 e M_2 di un termine M possono essere ulteriormente semplificate ad un termine comune M_3. Il seguente risultato e' una conseguenza immediata della proprieta' CR.

Theorem 6.5 *$M \rightarrow M'$ implica $M = M'$, inoltre $M_1 = M_2$ sse esiste M_3 t.c. $M_1, M_2 \rightarrow^* M_3$.*

Ovvero per dimostrare la β-convertibilita' di due termini e' sufficiente dimostrare che si riducono ad un stesso termine.

6.3 Semantiche operazionali

Se vogliamo pensare al λ-calcolo come un linguaggio di funzionale (minimale), la cosa piu' semplice da fare e' dare una semantica operazionale. Considereremo quattro tipi di semantica operazionale corrispondenti a due scelte indipendenti

	call-by-value	call-by-name
a piccoli passi	$\to_v$	$\to_n$
a grandi passi	$\Downarrow_v$	$\Downarrow_n$

Tali semantiche sono una versione embrionale di quelle effettivamente utilizzate per descrivere i linguaggi di programmazione. Tali semantiche sono date mediante regole di inferenza. Indicheremo con Λ_0 l'insieme dei λ-termini chiusi (cioe' senza variabili libere), e con Val_0 l'insieme delle λ-astrazioni chiuse. Inoltre chiameremo un $M \in \Lambda_0$ **programma** ed un e $V \in \mathsf{Val}_0$ **valore**.

- $\Downarrow_n \subset \Lambda_0 \times \mathsf{Val}_0$ semantica CBN a grandi passi

  (V) $V \Downarrow_n V$ (app) $\dfrac{M_1 \Downarrow_n \lambda x.P \quad [M_2/x]P \Downarrow_n V}{M_1 M_2 \Downarrow_n V}$

  $\Downarrow_v \subset \Lambda_0 \times \mathsf{Val}_0$ semantica CBV a grandi passi

  (V) $V \Downarrow_v V$ (app) $\dfrac{M_1 \Downarrow_v \lambda x.P \quad M_2 \Downarrow_v V \quad [V/x]P \Downarrow_v V'}{M_1 M_2 \Downarrow_v V'}$

- $\to_n \subset \Lambda_0 \times \Lambda_0$ semantica CBN a piccoli passi

  (β) $(\lambda x.M)N \to_n [N/x]M$ (app.1) $\dfrac{M_1 \to_n M_1'}{M_1 M_2 \to_n M_1' M_2}$

- $\to_v \subset \Lambda_0 \times \Lambda_0$ semantica CBV a piccoli passi

  (β_v) $(\lambda x.M)V \to_v [V/x]M$ (app.1) $\dfrac{M_1 \to_v M_1'}{M_1 M_2 \to_v M_1' M_2}$ (app.2) $\dfrac{M_2 \to_v M_2'}{V M_2 \to_v V M_2'}$ $V \in \mathsf{Val}_0$

Le principali differenze tra β-riduzione e le semantiche operazionali date sopra sono: le semantiche operazionali sono deterministiche, e non effettuano semplificazioni nel corpo di una λ-astrazione. La seconda caratteristica differenzia le semantiche operazioni dalle strategie di riduzione (che scelgono una tra tutte le possibili semplificazioni possibili). Si osservi che le sostituzioni $[N/x]M$ per definire le semantiche operazionali sono tali che N e' sempre un termine chiuso, quindi non sara' mai necessario effettuare ridenominazioni delle variabili legate in M.

Proposition 6.6 *Sia* $x \in \{v, n\}$*, cioe' CBV o CBN, allora*

- $M \to_x N$ *implica* $M \to_\beta N$
- $M \Downarrow_x V$ *sse* $M \to_x^* V \in \mathsf{Val}_0$

Nota che i valori sono invarianti per β-riduzione, cioe' $M \to_\beta N$ e $M \in \mathsf{Val}_0$ implicano $N \in \mathsf{Val}_0$. E' possibile correlare la semantica operazionale alla β-conversione. Per far cio' e' necessario introdurre la nozione di osservazione ed equivalenza osservazionale. Per semplicita' considereremo solo le semantiche operazionali a grandi passi.

Definition 6.7 *Data una semantica operazione* $\Downarrow_x \subset \Lambda_0 \times \mathsf{Val}_0$*, si dice che "$M$* **termina***" $M \Downarrow_x$ sse $\exists V{:}\mathsf{Val}_0.M \Downarrow_x V$. La* **equivalenza osservazionale** $\approx_x \subset \Lambda \times \Lambda$ *e' data da $M \approx_x N$ sse $C[M] \Downarrow_x \leftrightarrow C[N] \Downarrow_x$ per ogni contesto $C[\_]$ con un buco t.c. $C[M]$ e $C[N]$ sono programmi. In modo analogo si definisce il* **preordine osservazionale** *$M \leq_x N$ sse $C[M] \Downarrow_x \supset C[N] \Downarrow_x$ per ogni contesto $C[\_]$ con un buco t.c. $C[M]$ e $C[N]$ sono programmi.*

Il seguente risultato dice che la β-conversione e' *sound* per dimostrare l'equivalenza osservazionale $\approx_n$.

Theorem 6.8 *$M =_\beta N$ implica $M \approx_n N$.*

Un risultato simile per $approx_v$ si ottiene indebolendo la teoria equazionale, ovvero sostituendo la regola β con

$$(\beta_v) \quad (\lambda x.M)V = [V/x]M \quad V \text{ valore}$$

Dove un valore (che puo' avere anche variabili libere) e' una variabile o una λ-astrazione. Invece le due equivalenza osservazionali non sono fra loro confrontabili, p.e.:

- $\Omega \approx_v K\ I\ \Omega \not\approx_v I$
- $\Omega \not\approx_n K\ I\ \Omega \approx_n I$

dove $I \equiv \lambda x.x$, $K \equiv \lambda x, y.x$ e $\Omega \equiv (\lambda x.xx)(\lambda x.xx)$ e' l'esempio tipico di programma che non termina, infatti $\Omega \to_x \Omega$.

7 Lambda calcoli applicati

Per studiare i linguaggi di programmazione mediante il λ-calcolo e molto conveniente estendere l'insieme dei termini. Se Σ e' una segnatura per una teoria algebrica single-sorted, cioe' Σ_n e' l'insieme dei simboli di operazione n-arie, allora vi sono due modi estendere l'insieme dei termini:

- $M \in \Lambda_\Sigma ::= \ldots \mid op(M_{i:n})$ dove $op \in \Sigma_n$, questo e' il modo piu' ovvio di combinare le regole di formazione dei termini algebrici e dei λ-termini, oppure
- $M \in \Lambda_\Sigma ::= \ldots \mid c_{op}$ dove $op \in \Sigma_n$, cioe' trattare una operazione come una costante higher-order.
- Le due estensioni si possono tradurre l'una nell'altra, infatti la costante c_{op} la si puo' identificare con $\lambda vetx.op(\overline{x})$, dove $\overline{x}$ e' una n-upla di variabili distinte, e $op(\overline{M})$ equivale a $c_{op}\ \overline{M}$.

Sebbene le costanti higher-order permettano una descrizione piu' concisa della sintassi, cio' puo' creare delle complicazioni nel descrivere la semantica operazionale, e soprattutto la nozione di valore. Quindi prenderemo il primo approccio come quello preferenziale. Nel resto della sezione esemplificheremo come estendere la β-conversione, -riduzione e semantica operazionale (per brevita ci limitiamo a considerare la semantica CBV a grandi passi). Parleremo di $\beta\delta$-conversione, dove le δ-regole sono quelle che descrivono il comportamente delle operazioni/costanti aggiuntive.

7.1 Estensione per i booleani

- Le operazioni da aggiungere, con la loro arieta', sono $true, false$: 0 e if: 3
- Le regole equazionali da aggiungere sono

  (if.true) $if(true, M, N) = M$ (if.false) $if(false, M, N) = N$

  per ottenere le regole di riduzione le equazioni vanno orientate da sinistra a desstra
- Il sottoinsieme dei valori (per la semantica operazionale CBV) e' dato dalla BNF

  $V \in \mathsf{Val} ::= x \mid \lambda x.M \mid true \mid false$
- Le regole per la semantica operazionale CBV a grandi passi sono

  $$\text{(V)}\ V \Downarrow_v V \qquad \text{(app)}\ \frac{M_1 \Downarrow_v \lambda x.P \quad M_2 \Downarrow_v V \quad [V/x]P \Downarrow_v V'}{M_1 M_2 \Downarrow_v V'}$$

  $$\text{(if.1)}\ \frac{P \Downarrow_v true \quad M_1 \Downarrow_v V}{if(P, M_1, M_2) \Downarrow_v V} \qquad \text{(if.2)}\ \frac{P \Downarrow_v false \quad M_2 \Downarrow_v V}{if(P, M_1, M_2) \Downarrow_v V}$$

  ovvero quelle per il λ-calcolo puro (ma per l'insieme dei valori esteso), a cui vanno aggiunte le due regole per vautare un if.

Si osservi che nei λ-calcoli applicati si possono avere dei programmi (cioe' termini chiusi) che non sono valori, ma che non sono ulteriormente riducibili. Per esempio $true\ false$ o $if(I, true, I)$. Secondo la definizione che abbiamo dato di terminazione, tali programmi non hanno un valore, e quindi non terminano. Pero' volendo essere piu' precisi si dovrebbe distinguere tra programmi che non terminano (come Ω) e programmi che si bloccano in una situazione anomala (come $true\ false$). Questo tipo di distinzione e' importante quando si considerano i *type systems* e la loro capacita' di individuare a tempo di compilazione la possibilita' di errori a tempo di esecuzione.

7.2 Estensione per i naturali

- Le operazioni da aggiungere (oltre a quelle per i booleani) sono $zero$: 0 e $succ, pred, is0$: 1
- Le regole equazionali da aggiungere sono

  (is0.true) $is0(zero) = true$ (is0.false) $is0(succ(M)) = false$ (pred) $pred(succ(M)) = M$

  anche in questo caso le riduzioni vanno orientate da sinistra a destra
- Il sottoinsieme dei valori (per la semantica operazionale CBV) e' dato dalla BNF

  $V \in \mathsf{Val} ::= \ldots \mid succ^k(zero)$ con $k \geq 0$

- Le regole da aggiungere per la semantica operazionale CBV a grandi passi sono

(is0.1) $\dfrac{M \Downarrow_v zero}{is0(M) \Downarrow_v true}$ (is0.2) $\dfrac{M \Downarrow_v succ^{k+1}(zero)}{is0(M) \Downarrow_v false}$ (pred) $\dfrac{M \Downarrow_v succ^{k+1}(zero)}{pred(M) \Downarrow_v succ^k(zero)}$

Anche in questo caso si hanno programmi che che corrispondono a condizioni di errore, in particolare $pred(zero)$ e' della stessa natura della divisione per zero (quindi non individuabile a tempo di compilazione).

7.3 Estensione con operatore di punto fisso

L'operatore di punto fisso fornisce un modo semplice di trattare le definizioni rocorsive. Infatti se $x = P$ e' una definizione ricorsiva (dove x puo' occorre in P, allora il valore a cui x viene legato lo si puo' esprimere come $fix(\lambda x.P)$.

- L'unica operazionE da aggiungereE' fix: 1
- La regola equazionale da aggiungere e'

  (fix) $fix(M) = M\ fix(M)$

  anche in questo caso le riduzioni vanno orientate da sinistra a destra, anche se cio' puo' sembrare controintuitivo.
- Il sottoinsieme dei valori (per la semantica operazionale CBV) rimane invariato
- La regola da aggiungere per la semantica operazionale CBV a grandi passi e'

  (fix) $\dfrac{M \Downarrow_v \lambda x.P \quad [fix(\lambda x.P)/x]P \Downarrow_v V}{fix(M) \Downarrow_v V}$

Sebbene nel λ-calcolo sia possibile definire termini chiusi che si comportano come operatore di punto fisso, p.e.

$$\begin{array}{rcl} (\lambda f.V\ V) & \text{dove} & V \equiv (\lambda x.f\ x\ x) \\ Z\ Z & \text{dove} & Z \equiv \lambda f, x.f\ (x\ x\ f) \end{array}$$

in pratica risulta piu' conveniente introdurre fix come un operatore primitivo.

8 Calcoli tipati e sistemi di tipi

In questa sezione introdurremo vari calcoli tipati, essi possono essere dati in due versioni: usando termini senza informazione di tipo, oppure usando termini con informazione di tipo (vi e' un certo grado di discrezionalita' quanta informazione di tipo esplicitare). Prima di considerare calcoli specifici, introduciamo un po' di terminologia e notazione per un generico linguaggio tipato L.

- $\tau \in T(L)$ espressioni di tipo
- $M \in \Lambda_T(L)$ termini con informazioni di tipo, $M \in \Lambda(L)$ termini senza informazione di tipo

  in generale un indice T indichera' la versione di L che usa termini con informazioni di tipo
- $|\_|: \Lambda_T(L) \to \Lambda(L)$ funzione di cancellazione dell'informazione di tipo
- $\Gamma \vdash M: \tau$ typing judgement, dove $\Gamma: Var \stackrel{fin}{\to} T(L)$ e' un typing context e $\mathrm{FV}(M) \subseteq dom(\Gamma)$
- $V \in \mathsf{Val}(L) \subseteq \Lambda(L)$ valori aperti; $V \in \mathsf{Val}_0(L) = \mathsf{Val}(L) \cap \Lambda_0(L)$ valori chiusi
- $M \Downarrow_e R$ semantica operazionale CBV con err, dove $M \in \Lambda_0(L)$ e $R \in \mathsf{Val}_0(L) + \{err\}$.

  La semantica operazionale con err serve a descrivere in modo preciso la classe di errori che si vorrebbero identificare a tempo di compilazione (mediante il type-checking). In certi casi puo' essere conveniente usare una semantica operazionale con ambiente: $\rho \vdash M \Downarrow_e R$, dove $\rho: Var \stackrel{fin}{\to} \mathsf{Val}_0(L)$ ambiente e $\mathrm{FV}(M) \subseteq dom(\rho)$.
- Dato un sistema (di type-checking) per derivare typing judgements, diciamo che vale la **correttezza** del type-checking relativamente alla semantica operazionale con err: "se M ha un tipo, allora non genera err a tempo di esecuzione", cioe' $\emptyset \vdash M: \tau$ e $M \Downarrow R$ implicano $R \not\equiv err$.

Descriveremo anche la semantica semplice dei tipi, nella quale un tipo τ e' interpretato da un sottoinsieme $[\![\tau]\!] \subseteq \mathsf{Val}_0$ di valori. Tale semantica permette di definire la soddisfacibilita' di un typing judgement $\Gamma \models M: \tau$ in funzione della sola semantica operazione, ed analogamente permette una interpretazione ovvia della relazione di sub-typing $\tau_1 \subseteq \tau_2$, cioe' $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$, e quindi validare la soundness di regole di inferenza per il sub-typing.

8.1 I tipi semplici

I tipi semplici sono quelli che tipicamente si incontrano in linguaggio di programmazione come il Pascal o Algol, cioe' con parametri di tipo procedure ma senza *polimorfismo*.

- $\tau \in T ::= \overline{\tau}\ C \mid \tau_1 \to \tau_2$, dove C e' un costruttore di tipo, p.e. $bool$ o $\tau\ list$.
- $M \in \Lambda ::= x \mid c \mid \lambda x.M \mid M_1 M_2$, dove c sono delle costanti (higher-order), p.e. if, nil, $cons$, fix.

  $M \in \Lambda_T ::= x \mid c_{\overline{\tau}} \mid \lambda x{:}\tau.M \mid M_1 M_2$ e' l'insieme dei termini con informazioni di tipo, e la funzione $|\_|: \Lambda_T \to \Lambda$ di cancellazione della informazione di tipo e' data da: $|x| = x$, $|c_{\overline{\tau}}| = c$, $|\lambda x{:}\tau.M| = \lambda x.|M|$, $|M_1 M_2| = |M_1||M_2|$.
- Le regole di inferenza per derivare typing judgements con informazioni di tipo nei termini sono

  $$(\mathrm{x})\ \Gamma \vdash_T x{:}\Gamma(x) \qquad (\mathrm{c})\ \Gamma \vdash_T c{:}\tau_c \qquad (\to\mathrm{I})\ \frac{\Gamma, x{:}\tau_1 \vdash_T M{:}\tau_2}{\Gamma \vdash_T \lambda x{:}\tau_1.M{:}\tau_1 \to \tau_2} \qquad (\to\mathrm{E})\ \frac{\Gamma \vdash_T M_1{:}\tau_1 \to \tau_2 \quad \Gamma \vdash_T M_2{:}\tau_1}{\Gamma \vdash_T M_1 M_2{:}\tau_2}$$

  alcuni esempi di tipi per le costanti sono:

  - $T, F{:}bool$ e $if_\tau{:}bool \to \tau \to \tau \to \tau$
  - $nil_\tau{:}\tau\ list$, $cons_\tau{:}\tau \to \tau\ list \to \tau\ list$ e $case_{\tau,\sigma}{:}\tau\ list \to \sigma \to (\tau \to \tau\ list \to \sigma) \to \sigma$, dove $case$ corrisponde al pattern matching per il tipo $\tau\ list$
  - $fix_\tau{:}(\tau \to \tau) \to \tau$, che serve per modellare le definizioni ricorsive.

  La seguente regola di **sostitutivita'** risulta essere *ammissibile*, cioe' se la premessa e' derivabile anche la conclusione lo e', e quindi puo' essere aggiunta senza modificare l'insieme dei typing judgements derivabili

  $$(\mathrm{sub})\ \frac{\Gamma \vdash M{:}\tau \quad \Gamma, x{:}\tau \vdash M'{:}\tau'}{\Gamma \vdash [M/x]M'{:}\tau'}$$

  Le regole di inferenza per derivare typing judgements senza informazioni di tipo nei termini si ottengono dalle precedenti applicando la funzione di cancellazione $|\_|$ ai termini M che compaiono nella regola.

- La semantica operazionale CBV con *err* $M \Downarrow R$ estende quella CBV precedentemente introdotta con regole aggiuntive per trattare le situazioni di errore. La daremo per i termini senza informazione di tipo, poiche' la semantica operazione non fa' alcun uso di tale informazione. Per semplicita' consideriamo il linguaggio con le sole costanti per il tipo *bool*, quindi l'insieme dei valori e' dato da $V \in \mathsf{Val} ::= x \mid T \mid F \mid \lambda x.M$.

$$\text{(V)}\ V \Downarrow V \qquad \text{(app)}\ \frac{M_1 \Downarrow \lambda x.M \quad M_2 \Downarrow V \quad [V/x]M \Downarrow R}{M_1 M_2 \Downarrow R} \qquad \text{(if.T)}\ \frac{M \Downarrow T \quad M_1 \Downarrow R}{if(M, M_1, M_2) \Downarrow R}$$

$$\text{(app1)}\ \frac{M_1 \Downarrow \lambda x.M \quad M_2 \Downarrow err}{M_1 M_2 \Downarrow err} \qquad \text{(app2)}\ \frac{M_1 \Downarrow R \not\equiv \lambda x.M}{M_1 M_2 \Downarrow err} \qquad \text{(if1)}\ \frac{M \Downarrow R \not\equiv T, F}{if(M, M_1, M_2) \Downarrow err}$$

- La semantica di una espressione di tipo τ e' data da un sottoinsieme $[\![\tau]\!] \subseteq \mathsf{Val}_0$ definito per induzione su τ. La definizione di $[\![\tau]\!]$ fa' uso della semantica operazione con *err*, ma e' indipendente dai typing judgments.

  Dato $X \subseteq \mathsf{Val}_0$ definiamo $\widetilde{X} \stackrel{\Delta}{=} \{M \in \Lambda_0 | \forall R.M \Downarrow R \supset R \in X\}$. Intuitivamente $\widetilde{X}$ per l'insieme dei programmi $M \in \Lambda_0$ che in caso di terminazione producono un valore in X (e quindi non possono terminare con *err*).

  Data una semantica delle espressioni di tipo, diciamo che $\Gamma \models M{:}\,\tau$ vale $\stackrel{\Delta}{\iff} \forall \rho \in [\![\Gamma]\!].([\rho]M) \in \widetilde{[\![\tau]\!]}$, dove $[\![\Gamma]\!] \stackrel{\Delta}{=} \{\rho{:}\, Var \stackrel{fin}{\to} \mathsf{Val}_0 | \forall x \in \mathrm{DV}(\Gamma).\rho(x) \in [\![\Gamma(x)]\!]\}$ e $[\rho]M \in \Lambda_0$ e' l'istanza di M ottenuta sostituendo ogni $x \in \mathrm{FV}(M)$ con $\rho(x) \in \mathsf{Val}_0$.

  Diamo il corrispettivo semantico di *bool* e dei costruttori *list* e $\to$, che quindi agiscono su insiemi $X, Y \subseteq \mathsf{Val}_0$

  - $bool \stackrel{\Delta}{=} \{T, F\}$
  - $X \to Y \stackrel{\Delta}{=} \{\lambda x.M \in \mathsf{Val}_0 | \forall V \in X.[V/x]M \in \widetilde{Y}\}$
  - $X\ list \stackrel{\Delta}{=} \{[V_i|i{:}\,n] | \forall i{:}\,n.V_i \in X\}$, dove $[V_i|i{:}\,n]$ e' una abbreviazione per $cons(V_0, \ldots, cons(V_{n-1}, nil) \ldots)$

  La semantica dei tipi, oltre a spiegarne il significato, e' anche utile per stabilire la **soundness** di regole per derivare typing judgements (nello stesso modo in cui l'interpretazione della logica del primo ordine, permette di validare assioni e regole di inferenza). Una volta stabilita la soundness, cioe' $\Gamma \vdash M{:}\,\tau$ derivabile implica $\Gamma \models M{:}\,\tau$ vera, la correttezza discende come un facile corollario. Infatti, dalla definizione di $\Gamma \models M{:}\,\tau$ e' immediato dimostrare che $\emptyset \models M{:}\,\tau$ e $M \Downarrow R$ implicano $R \not\equiv err$.

Dimostriamo la correttezza delle regole di type-checking passando attraverso una dimostrazione di soundness. In seguito illustreremo una tecnica piu' diretta per dimostrare la correttezza (ma anche meno informativa).
Per semplicita' ci limiteremo a dimostrare la soundness delle regole ($\to$I) e ($\to$E).

- Per ($\to$I) dobbiamo dimostrare che $\Gamma, x{:}\,\tau_1 \models M{:}\,\tau_2$ implica $\Gamma \models \lambda x{:}\,\tau_1.M{:}\,\tau_1 \to \tau_2$. Per semplicita' supponiamo che Γ sia vuoto.

  Dalla definizione di $x{:}\,\tau_1 \models M{:}\,\tau_2$ abbiamo che $\forall V{:}\,[\![\tau_1]\!].[V/x]M \in \widetilde{[\![\tau_2]\!]}$, ma questa e' esattamente la definizione di $(\lambda x.M) \in [\![\tau_1]\!] \to [\![\tau_2]\!]$, che e' banalmente equivalente alla validita di $\lambda x{:}\,\tau_1.M{:}\,\tau_1 \to \tau_2 \models$.

- Per ($\to$E) dobbiamo dimostrare che $\Gamma \models M_1{:}\,\tau_1 \to \tau_2$ e $\Gamma \models M_2{:}\,\tau_1$ implicano $\Gamma \models M_1 M_2{:}\,\tau_2$. Anche in questo caso supponiamo che Γ sia vuoto.

  Espandendo la definizione di $\models M_1 M_2{:}\,\tau_2$, quello che dobbiamo dimostrare e': $M_1 M_2 \Downarrow R$ implica $R \in [\![\tau_2]\!]$. Facendo una analisi delle possibili regole per derivare $M_1 M_2 \Downarrow R$ si vede che l'unica regola possibile e' (app). Infatti se fosse (app2) si avrebbe $M_1 \Downarrow R_1 \not\equiv \lambda x.M$ che contraddice $\Gamma \models M_1{:}\,\tau_1 \to \tau_2$, e se fosse (app1) si avrebbe $M_2 \Downarrow err$ che contraddice $\Gamma \models M_2{:}\,\tau_1$. Quindi abbiamo $M_1 \Downarrow \lambda x.M$, $M_2 \Downarrow V$ e $[V/x]M \Downarrow R$. Ma per le assunzioni si ha che $\lambda x.M \in [\![\tau_1]\!] \to [\![\tau_2]\!]$ e $V \in [\![\tau_2]\!]$, e percio' $[V/x]M \in \widetilde{[\![\tau_2]\!]}$ e infine $R \in [\![\tau_2]\!]$.

8.2 I tipi semplici con variabili

Una estensione piccola, ma significativa, dei tipi semplici consiste nell'aggiunta delle variabili di tipo. Questa estensione fornisce un modo di esprimere il polimorfismo di un termine M assegnandogli un tipo τ con variabili di tipo. Sostituendo le variabili di tipo in τ con espressioni di tipo si ottengono altri tipi (piu' specifici) per M. Questa estensione ci permette di illustrare l'algoritmo W di type-inference dovuto ad Hindley-Milner in forma semplificata.

- $\tau \in T ::= \overline{\tau}\ C \mid \tau_1 \to \tau_2 \mid X$, dove C e' un costruttore di tipo (come illustrato in precedenza)

- Tutte le altre definizioni ricalcano nel modo ovvio quelle gia' date per i tipi semplici. In particolare, la definizioni di Λ e la semantica operazionale CBV $M \Downarrow R$ per i termini senza informazioni di tipo sono invariate, mentre i typing judgements $\Gamma \vdash M: \tau$ e le regole di type-checking sono uguali modulo l'estensione di T.

  La seguente regola di **sostitutivita'** per variabili di tipo risulta essere ammissibile

  $$(\text{Sub})\ \frac{\Gamma \vdash M: \tau}{[S]\Gamma \vdash M: [S]\tau}$$

  dove $S: TVar \stackrel{fin}{\to} T$ e' una sostituzione di variabili di tipo con espressioni di tipo, $[S]\tau$ e' l'istanza di τ ottenuta applicando la sostituzione S (si conviene che $S[X] = X$ se $X \notin dom(S)$) e $[S]\Gamma$ e' il typing context Γ' con lo stesso dominio di Γ t.c. $\Gamma'(x) = [S](\Gamma(x))$.

- Adesso la semantica $[\![\tau]\!]_\eta$ di una espressione di tipo τ dipendera' anche da un ambiente $\eta: TVar \stackrel{fin}{\to} \mathcal{P}(\mathsf{Val}_0)$ che fornisce l'interpretazione di tutte le variabili di tipo in $\mathrm{FV}(\tau)$, cioe' $\mathrm{FV}(\tau) \subseteq dom(\eta)$. Modulo cio' la definizione e' uguale al caso semplice, ovvero $[\![\tau_1 \to \tau_2]\!]_\eta = [\![\tau_1]\!]_\eta \to [\![\tau_2]\!]_\eta$, etc.

  Corrispondentemente la definizione di "$\Gamma \models M: \tau$ vale" richiede una ulteriore quantificazione sugli ambienti η, cioe' $\forall \rho \in [\![\Gamma]\!]_\eta.([\rho]M) \in \widetilde{[\![\tau]\!]}_\eta$ per ogni $\eta: TVar \stackrel{fin}{\to} \mathcal{P}(\mathsf{Val}_0)$ t.c. $\mathrm{FV}(\Gamma, \tau) \subseteq dom(\eta)$.

L'algoritmo W di Hindley-Milner puo' essere descritto nella seguente maniera

- input: $\Gamma: Var \stackrel{fin}{\to} T$ e $M \in \Lambda$ t.c. $\mathrm{FV}(M) \subseteq dom(\Gamma)$
- output: fail oppure una coppia (S, τ) dove $S: \mathrm{FV}(\Gamma) \to T$ e $\tau \in T$
- $W(\Gamma; M)$ fallisce sse non esistono S e τ t.c. $[S]\Gamma \vdash M: \tau$ sia derivabile
- altrimenti $W(\Gamma; M)$ restituisce una coppia (S, τ) t.c. $[S]\Gamma \vdash M: \tau$ e' derivabile, ed inoltre se $S': \mathrm{FV}(\Gamma) \to T$ e $[S']\Gamma \vdash M: \tau'$ e' derivabile, allora esiste S'' t.c. $S' = [S'']S$ e $\tau' = [S'']\tau$. Dove $[S_2]S_1$ e' la sostituzione S_3 con lo stesso dominio di S_1 t.c. $S_3(X) = [S_2](S_1(X))$.

Si osservi che per inferire il **tipo piu' generale** τ di M basta prendere $\Gamma: \mathrm{FV}(M) \hookrightarrow TVar$. L'algortimo Wprocede per induzione sulla struttura di M.

- $W(\Gamma; x) = (Id_{\mathrm{FV}(\Gamma)}, \Gamma(x))$, dove Id_U indica la sostituzione identica con dominio $U \subseteq_{fin} TVar$.
- $W(\Gamma; \lambda x.M) = (S|_{\mathrm{FV}(\Gamma)}, S(X) \to \tau)$ se $(S, \tau) = W(\Gamma, x: X; M)$, dove $X \in TVar$ e' *new*, $S|_U$ e' la restrizione di S al dominio U, ed inoltre si suppone che $x \notin dom(\Gamma)$ (se necessario si puo' effettuare una ridenominazione delle variabili legate). Per dare una definizione precisa di "X new" occorrerebbe passare all'algoritmo W un terzo parametro $U \subseteq_{fin} TVar$, per indicare le variabili sinora usate.
- $W(\Gamma; M_1M_2) = ([S_3][S_2]S_1, S_3(X))$ se $(S_1, \tau_1) = W(\Gamma; M_1)$, $(S_2, \tau_2) = W([S_1]\Gamma; M_2)$ e $S_3 = mgu([S_2]\tau_1, \tau_2 \to X)$, dove $X \in TVar$ e' *new* e $mgu(\tau, \tau')$ e' il **most general unifier** delle espressioni di tipi τ e τ'. La non esistenza di $mgu(\tau, \tau')$ e' il motivo per cui l'algoritmo W puo' fallire.
- $W(\Gamma; c) = (Id_{\mathrm{FV}(\Gamma)}, \tau_c)$, dove τ_c e' un **tipo piu' generale** per la costante c che usa solo varibili di tipo new, p.e. $\tau_{nil} = X\ list$ con X new. Questo e' uno dei casi in cui la coppia (S, τ) restituita da W non soddisfa la proprieta' $\mathrm{FV}(\tau) \subseteq \mathrm{FV}(S)$.

Consideriamo alcuni esempi di tipi inferiti da W per termini chiusi.

- $W(\emptyset; \lambda x.x) = (\emptyset, X \to X)$ con X new
- $W(\emptyset; (\lambda x.x)(\lambda x.x)) = (\emptyset, Y \to Y)$ con Y new, le due occorrenze di $(\lambda x.x)$ ricevono tipi diversi $X \to X$ e $Y \to Y$, ed infine si deve calcolare $S_3 = mgu(X \to X, (Y \to Y) \to Z)$, quindi il tipo calcolato e' $S_3(Z) = Y \to Y$.
- $W(\emptyset; \lambda x.xx)$ fallisce, infatti $W(x: X, x) = (Id, X)$ ma $mgu(X, X \to Z)$ fallisce.
- Come esercizio far vedere che (modulo i nomi delle variabili) W inferisce i seguenti tipi
  - $\lambda g, f, x.g(fx): (Y \to Z) \to (X \to Y) \to X \to Z$
  - $fix(\lambda x.x): X$
  - $cons(nil, nil): X\ list\ list$
  - $map = fix(\lambda m, f, l.case_{list}(l, nil, \lambda x, l.cons(fx, l))): (x \to Y) \to X\ list \to Y\ list$

8.3 Tipi polimorfi

L'aggiunta dei tipi polimorfi permette di tipare molti piu' termini $M \in \Lambda$. Tuttavia per tale estensione non esiste un analogo dell'algoritmo W, anzi e' stato dimostrato che e' indecidibile se $\Gamma \vdash M{:}\,\tau$ e' derivabile (ovviamente quando si usano i termini con informazioni di tipo la derivabilita' di $\Gamma \vdash_T M{:}\,\tau$ e' facilmente decidibile).

- $\tau \in T ::= \overline{\tau}\ C \mid \tau_1 \to \tau_2 \mid X \mid \forall X.\tau$, dove C e' un costruttore di tipo (come illustrato in precedenza)
- $M \in \Lambda_T ::= x \mid c \mid \lambda x{:}\,\tau.M \mid M_1 M_2 \mid \Lambda X.M \mid M[\tau]$, si osservi che le costanti polimorfe non hanno piu' bisogno di essere viste come famiglie di costanti indiciate da tipi. La definizione di $M \in \Lambda$ e' invariata, mentre la funzione di cancellazione dell'informazione di tipo e' estesa nel modo seguente: $|\Lambda X.M| = |M[\tau]| = |M|$.
- Vi sono due nuove regole di inferenza per derivare typing judgements (con informazioni di tipo nei termini)

  $$(\forall\mathrm{I})\ \frac{\Gamma \vdash_T M{:}\,\tau}{\Gamma \vdash_T \Lambda X.M{:}\,\forall X.\tau}\ X \notin \mathrm{FV}(\Gamma) \qquad (\forall\mathrm{E})\ \frac{\Gamma \vdash_T M{:}\,\forall X.\tau}{\Gamma \vdash_T M[\tau']{:}\,[\tau'/X]\tau}\ \tau' \in T$$

  inoltre vi e' un modo piu' naturale di tipare le costanti polimorfe:

  - $T, F{:}\,\mathit{bool}$ e $\mathit{if}{:}\,\forall X.\mathit{bool} \to X \to X \to X$
  - $\mathit{nil}{:}\,\forall X.X\ \mathit{list}$, $\mathit{cons}{:}\,\forall X.X \to X\ \mathit{list} \to X\ \mathit{list}$ e $\mathit{case}{:}\,\forall X,Y.X\ \mathit{list} \to Y \to (X \to X\ \mathit{list} \to Y) \to Y$
  - $\mathit{fix}{:}\,\forall X.(X \to X) \to X$

  Usando le nuove regole alcuni termini che prima non erano tipabili adesso lo divetano, p.e. $\lambda x.xx$ ha sia tipo $(\forall X.X \to X) \to X \to X$ che tipo $(\forall X.X) \to Y$ come si vede piu' chiaramente aggiungendo informazione di tipo

  $$\lambda x{:}\,(\forall X.X \to X).x[X \to X]\ (x[X]) \qquad \lambda x{:}\,(\forall X.X).x[X \to Y]\ (x[X])$$

  $\overline{X}{:}\,T \vdash \tau{:}\,T$, dove $\mathrm{FV}(\tau) \subseteq \overline{X}$

  $\overline{X}{:}\,T; \Gamma \vdash M{:}\,\tau$, dove $\mathrm{FV}(\Gamma, \tau) \subseteq \overline{X}$ e $\mathrm{FV}(M) \subseteq \mathit{dom}(\Gamma)$ (per termini senza tipi)
- La semantica operazionale CBV con $\mathit{err}\ M \Downarrow R$ e' invariata.
- La semantica $[\![\tau]\!]_\eta$ estende quella data per i tipi semplici con variabili di tipo per trattare il caso $\forall X.\tau$ dei tipi polimorfi $[\![\forall X.\tau]\!]_\eta = \cap\{[\![\tau]\!]_{\eta[X:=A]} | A \subseteq \mathsf{Val}_0\}$. Alternativamente possiamo dare la semantica dei tipi polimorfi usando il corrispettivo semantico di $\forall$

  - $\forall(F) = \cap\{F(A) | A \subseteq V_0\}$, dove $F{:}\,\mathcal{P}(\mathsf{Val}_0) \to \mathcal{P}(\mathsf{Val}_0)$.

  La definizione di "$\Gamma \models M{:}\,\tau$ vale" e' uguale a quella data nel caso di tipi semplici con variabili di tipo, e la dimostrazione di **soundness** si estende facilmente. Si osservi che in presenza di tipi polimorfi si hanno relazioni di sub-typing non banali, infatti $(\forall X.\tau) \subseteq [\tau'/X]\tau$ per qualunque τ e τ'.

8.4 I tipi di ML

Il sistema di tipi di ML aggiunge una forma ristretta di tipi polimorfi (detti schemi di tipo) senza pero' perdere le proprieta' dell'algoritmo W di type-inference.

- $\tau \in T ::= \overline{\tau}\ C \mid \tau_1 \to \tau_2 \mid X$ tipi

  $\sigma \in TS ::= \tau \mid \forall X.\sigma$ schemi di tipo

  Si osservi che il tipo polimorfo $(\forall X.X) \to Y$ non e' ne' un tipo ne' uno schema di tipo di ML
- $M \in \Lambda ::= x \mid c(\overline{M}) \mid \lambda x.M \mid M_1 M_2 \mid \mathrm{let}\ x = M_1\ \mathrm{in}\ M_2$

  Dal punto di vista della semantica operazionale il termine ($\mathrm{let}\ x = M_1\ \mathrm{in}\ M_2$) di ML puo' essere identificato con $(\lambda x.M_2)M_1$. Tuttavia le regole per il type-checking di ML permettono di assegnare un tipo a $\mathrm{let}\ x = M_1\ \mathrm{in}\ M_2$ anche quando $(\lambda x.M_2)M_1$ non e' tipabile.

  Dal punto di vista della semantica dei tipi di ML non vi e' niente da aggiungere rispetto a quanto visto in precedenza per i tipi polimorfi.
- Avendo distinto tra tipi e schemi di tipo, la definizione di typing judgement $\Gamma \vdash M{:}\,\tau$ richiede qualche precisazione. Supporremo che $\Gamma{:}\,\mathit{Var} \stackrel{fin}{\to} TS$ e $\tau \in T$, quindi le variabili nel typing context possono essere polimorfe, pero' ai termini assegnamo solo tipi. E' possibile considerare typing judgements $\Gamma \vdash M{:}\,\sigma$ che assegnato schemi di tipo a termini, pero' la versione piu' restrittiva permette di capire meglio l'estensione dell'algoritmo W

- Le regole di inferenza per derivare typing judgements senza informazioni di tipo nei termini sono

  (x) $\Gamma \vdash x{:}\,[S]\tau$ se $\forall \overline{X}.\tau = \Gamma(x)$ e $S{:}\,\overline{X} \to T$ (c) $\Gamma \vdash c{:}\,[S]\tau$ se $\forall \overline{X}.\tau = \sigma_c$ e $S{:}\,\overline{X} \to T$

  quando $\overline{X}$ e' vuoto si riottengono le regole usuali (x) e (c).

  $$(\to\text{I})\ \frac{\Gamma, x{:}\,\tau_1 \vdash M{:}\,\tau_2}{\Gamma \vdash \lambda x{:}\,\tau_1.M{:}\,\tau_1 \to \tau_2} \qquad (\to\text{E})\ \frac{\Gamma \vdash M_1{:}\,\tau_1 \to \tau_2 \quad \Gamma \vdash M_2{:}\,\tau_1}{\Gamma \vdash M_1 M_2{:}\,\tau_2}$$

  $$(\text{let})\ \frac{\Gamma \vdash M_1{:}\,\tau_1 \quad \Gamma, x{:}\,\forall\overline{X}.\tau_1 \vdash M_2{:}\,\tau_2}{\Gamma \vdash \text{let } x = M_1 \text{ in } M_2{:}\,\tau_2}\ \overline{X} = \text{FV}(\tau_1) - \text{FV}(\Gamma)$$

  Si osservi che $\sigma_c \in TS$, quindi le costanti di ML possono avere un tipo polimorfo. La differenza essenziale tra (let $x = M_1$ in M_2) e $(\lambda x.M_2)M_1$ e' che la regola (let) permette di assegnare ad x uno schema di tipo, mentre la regola $(\to\text{I})$ non lo consente.

L'algoritmo W di Hindley-Milner per ML va modificato per trattare i casi base x e c, e va' esteso per trattare il costrutto let-in. Le proprieta' dell'algortimo W precedentemente enunciate continuano a valere anche per l'estensione ad ML. Riportaimo solo i casi modificati e/o aggiunti.

- $W(\Gamma; x) = (Id_{\text{FV}(\Gamma)}, [S]\tau)$ se $\Gamma(x) = \forall\overline{X}.\tau$ e $S{:}\,\overline{X} \hookrightarrow TVar$ ridenomina le $\overline{X}$ con variabili new.

  $W(\Gamma; c)$ e' definito in modo analogo usando $\sigma_c = \forall\overline{X}.\tau$.

- $W(\Gamma; \text{let } x = M_1 \text{ in } M_2) = ([S_2]S_1, \tau_2)$ se $(S_1, \tau_1) = W(\Gamma; M_1)$ e $(S_2, \tau_2) = W([S_1]\Gamma, x{:}\,\forall\overline{X}.\tau_1; M_2)$, dove $\overline{X} = \text{FV}(\tau_1) - \text{FV}([S_1]\Gamma)$.

Anche con l'estensione ad ML si ha che $W(\emptyset; \lambda x.xx)$ fallisce. Pero' adesso possiamo assegnare il tipo $X \to X$ a let $x = \lambda y.y$ in xx, poiche' alla variabile x e' assegnato lo schema di tipo $\forall X.X \to X$.

8.5 Correttezza in presenza di fix e Subject Reduction

In presenza di constanti per dimostrare la soundness delle regole di type-checking relativamente alla semantica dei tipi occorre far vedere che $c \in [\![\tau_c]\!]$. In genere cio' richiede qualche aggiustamento alla definizione di Λ (p.e. non considerando $cons$ un termine ma un costruttore di termini $cons(M, N)$) o alla definizione di valore (p.e. considerando $cons$ e $cons\ V$ valori). Modulo questi aggiustamenti in genere non si presentano problemi ad estendere la dimostrazione di soundness. L'unica eccezione (tra gli esempi considerati) e' costituita dall'operatore di punto fisso fix. Infatti, se cerchiamo di dimostrare che $fix(\lambda x.M) \in X$ quando $(\lambda x.M) \in X \to X$ ci si accorge che avremo bisogno di assumere che cio' che vogliamo dimostrare (questo e' immediato quando si considera $fix(\lambda x.x)$. Un modo per evitare il problema e' considerare una interpretazione piu' restrittiva dei tipi, in cui $[\![\tau]\!]$ deve essere un *ideale*, un sottoinsieme di V_0 con proprieta' addizionali.

Se si vuole solo dimostrare la correttezza, si puo' evitare di considerare la semantica dei tipi (e la dimostrazione di soundness). La proprieta' che si deve dimostrare va sotto il nome di **Subject Reduction** (SR): $\vdash M{:}\,\tau$ e $M \Downarrow R$ implicano $\vdash R{:}\,\tau$ (e quindi $R \not\equiv err$. Tuttavia la demostrazione richiede una ipotesi induttiva (SR)+ piu' forte.

- dato $\rho{:}\,Var \overset{fin}{\to} \mathsf{Val}_0$ e $\Gamma{:}\,Var \to T$ (per semplicita' supponiamo di non avere variabili di tipo) diciamo che "$\rho{:}\,\Gamma$ vale" sse $\vdash \rho(x){:}\,\Gamma(x)$ vale per ogni $x \in dom(\Gamma)$.

- Ipotesi induttiva (SR)+: se $\Gamma \vdash M{:}\,\tau$, $\rho{:}\,\Gamma$ e $([\rho]M) \Downarrow R$ allora $\vdash R{:}\,\tau$.

- (SR)+ si puo' dimostra per induzione sulla derivazione di $([\rho]M) \Downarrow R$, case analysis sulla derivazione di $\Gamma \vdash M{:}\,\tau$), e sfruttando alcune regole ammissibili sui typing judgements, p.e. (sub).

Contents