Linguaggi

Sommario

Appunti del corso di Linguaggi, Anno Accademico 2007/2008.

Indice

1 Introduzione

1.1 Struttura del corso

- Macchine astratte, interpreti, compilatori, implementazioni miste.
- Semantica denotazionale ed operazionale.
  - Linguaggio di specifica-implementazione (*Ocaml*[1]).
- Tipi di dato, tipi di dato astratto.
- Espressioni e comandi.
- Ambiente, dichiarazioni, blocchi.
- Sottoprogrammi, regole di scoping, passaggio di parametri.
- Classi ed oggetti.
- Gestione dell'ambiente: implementazione.
- Gestione della memoria:implementazione.
- Ambiente globale, moduli, compilazione separata.
- Analisi statica, interpretazione astratta, esempi di analizzatori.
- Specializzazione di interpreti e generazione del codice attraverso valutazione parziale.
- Struttura della macchina intermedia: esempi.

1.2 Un po' di storia

- *Nascita*
  - **Macchina di Von Neumann** (macchina a programma memorizzato): possiede un interprete capace di fare eseguire il programma memorizzato, e quindi di implementare un qualunque algoritmo descrivibile nel "linguaggio macchina"[2].
- *Linguaggi macchina ad alto livello*
  - **Assembler**: assegna nomi simbolici ad operazioni e dati.
- *Anni 50*

[1] http://caml.inria.fr

[2] Un qualunque linguaggio macchina dotato di semplici operazioni primitive per effettuare la scelta e per iterare (o simili) è Turing-equivalente, cioè può descrivere tutti gli algoritmi.

  - **Fortran** e **Cobol**.
  - Notazioni ad alto livello orientate rispettivamente al calcolo scientifico (numerico) ed alla gestione dati (anche su memoria secondaria).
  - Astrazione procedurale (non veri e propri sottoprogrammi ma un'astrazione che sfrutta le caratteristiche del linguaggio macchina).
  - Nuove operazioni e strutture dati.

- *Anni 60*
  - Prime formalizzazioni sintattiche e risultati semantici basati sul λ-calcolo. Introduzione dell'ambiente. Vera astrazione procedurale con ricorsione. Argomenti procedurali e per nome.
  - **Algol60**: primo linguaggi imperativo di alto livello. Scoping statico. Gestione dinamica della memoria attraverso uno **stack**.
  - **Lisp**: primo linguaggio funzionale, direttamente ispirato al λ-calcolo. Scoping dinamico. Gestione dinamica della memoria con **heap** e **garbage collector**[3].
  - Questi due linguaggi danno origine a due filoni di sviluppo dei linguaggi: il filone **imperativo** ed il filone **logico**.

- *Fine degli anni 60*
  - **PL/1**: tentativo fallimentare di costruire un linguaggio "totalitario" che accorpasse i linguaggi esistenti.
  - **Simula67**: nasce il concetto di **classe**, estendendo Algol60.

- *Anni 70*
  - **Pascal**: estende Algol60 con la definizione dei **tipi**, l'uso dei **puntatori** e la gestione della memoria attraverso uno heap. Altamente portabile grazie ad una implementazione mista.

- *Il dopo-Pascal*
  - **C**: PASCAL + moduli + tipi astratti + eccezioni + semplice interfaccia per interagire con il sistema operativo.
  - **ADA**: secondo tentativo di linguaggio "totalitario". Come il C, ma con l'aggiunta di concorrenza e costrutti per la programmazione in tempo reale.
  - **C++**: C con oggetti e classi allocati su uno heap (senza garbage collector).

[3] Il successivo linguaggio di alto livello con garbage collector sarà Java.

- *Programmazione Logica*
  - **Prolog**: implementazione di una parte del calcolo dei predicati del primo ordine. Strutture flessibili e calcolo effettuato tramite un unico algoritmo (unificazione). Computazioni non-deterministiche. Memoria a heap con garbage collector.
  - **CLP** (Costraint Logic Programming): Prolog + calcolo su domini diversi con opportuni algoritmi di soluzione di vincoli.
- *Programmazione Funzionale*
  - **ML**: implementazione del λ-calcolo tipato. Definizione di nuovi tipi attraverso la ricorsione. Scoping statico. Semantica statica molto potente. Memoria a heap con garbage collector
  - **Haskell**: ML con regola di valutazione "lazy".
- *Java*
  - Caratteristiche del filone imperativo: essenzialmente molto vicino al C++.
  - Caratteristiche del filone logico: garbage collector.
  - Uso delle classi e dell'ereditarietà per ridurre il numero di meccanismi primitivi.
  - Implementazione mista, che ne facilita la portabilità.

2 Macchine astratte, linguaggi, interpretazione, compilazione

Introduzione ai concetti che saranno ampliati durante il corso, con particolare interesse nei confronti delle varie tipologie e metodologie di implementazione.

2.1 Macchina astratta

Una **macchina astratta** è una collezione di strutture dati ed algoritmi in grado di memorizzare ed eseguire programmi, composta da:

- Interprete
- Memoria (dati e comandi)
- Controllo
- Operazioni *primitive*

2.2 L'Interprete

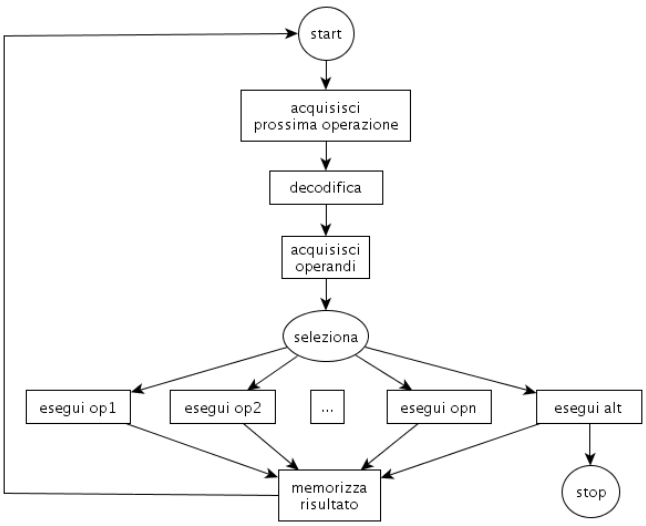

Figura 1: Schema di un Interprete.

2.3 Il componente Controllo

Strutture dati ed algoritmi per:

- Acquisire la successiva istruzione

- Gestire chiamate e ritorni ai sottoprogrammi
- Acquisire gli operandi e memorizzare i risultati
- Mantenere le associazioni nome - valore
- Gestire la memoria

2.4 Linguaggio macchina

Sia M una macchina astratta, allora indichiamo con L_M il **linguaggio macchina** di M, ovvero il linguaggio che ha come stringhe legali tutti i programmi interpretabili dall'interprete di M.
In quest'ottica i programmi non sono altro che i dati su cui opera l'interprete, e c'è corrispondenza tra i componenti di M ed i componenti di L_M.

2.5 Macchine astratte: implementazione

I componenti di una macchina astratta M sono realizzati mediante strutture dati ed algoritmi implementati nel linguaggio di una **macchina ospite** M_O, già esistente.
A questo punto, l'interprete di M può:

- Coincidere con l'interprete di M_O: per cui M è un'**estensione** di M_O.
- Essere diverso dall'interprete di M_O: per cui M è realizzata su M_O in modo **interpretativo**.

2.6 Dal linguaggio macchina alla macchina astratta

Supponiamo di voler implementare il linguaggio L su una macchina *fisica* M_O, realizzando dunque M_L, la macchina astratta di L.
In questo caso l'interprete di M_L è necessariamente diverso da quello di M_O, perché il linguaggio che vogliamo realizzare è di livello troppo alto per essere implementato come estensione della macchina fisica. Quindi:

- M_L è realizzata su M_O in modo interpretativo.
- L'implementazione di L si chiama **interprete**.
- Esiste una soluzione alternativa basate su tecniche di traduzione (**compilatore**).

2.7 Implementare un linguaggio

Supponiamo di voler implementare il linguaggio L, di alto livello, e quindi la sua macchina astratta M_L, su una generica macchina ospite M_O.
Esistono due casi (limite) per l'implementazione:

- **Interprete (puro)**: M_L viene realizzata su M_O in modo interpretativo.
  Questo implica una scarsa efficienza[4].

- **Compilatore (puro)**: i programmi di L sono tradotti in programmi equivalenti nel linguaggio macchina M_O.
  In questo caso M_L **non viene realizzata**, poiché i programmi tradotti sono eseguiti direttamente su M_O.
  Questo tipo di implementazione ha in se il problema dell'eccessiva dimensione del codice prodotto.

I due casi limite nella realtà non esistono quasi mai in questa forma.

2.8 La macchina intermedia

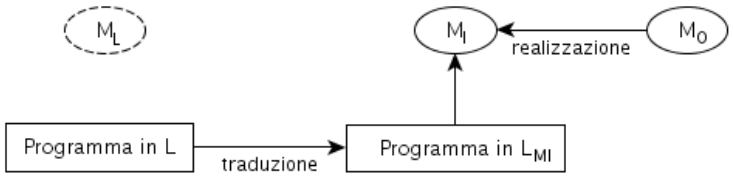

Figura 2: Realizzazione di una macchina intermedia.

Un'implementazione più efficiente rispetto ai casi puri appena descritti consiste nell'utilizzo di una **macchina intermedia**.
Con riferimento alla Figura 2, distinguiamo:

- L: linguaggio ad alto livello.
- M_L: macchina astratta di L.
- M_I: macchina intermedia.
- L_{M_I}: linguaggio intermedio.
- M_O: macchina ospite.

Il processo consiste quindi nella traduzione dei programmi da L al linguaggio intermedio L_{M_I} più la realizzazione, tramite interprete o compilatore, della macchina intermedia M_I su M_O.
E' bene notare che:

- Se $M_L = M_I$ allora abbiamo **interpretazione pura**.

[4]La Figura 1 (pag. 10) dà un'idea di come l'interprete rappresenti un costoso ciclo di decodifica.

- Se $M_O = M_I$ allora abbiamo **traduzione pura**.
  Questo caso è possibile solo nel caso in cui sia minima la differenza tra M_O ed M_L (es. $L = assembler$). In tutti gli altri casi abbiamo comunque una macchina intermedia, che eventualmente estende la macchina ospite (supporto runtime).

2.9 Supporto a tempo di esecuzione

Consideriamo, ad esempio, un linguaggio "antico", tipo FORTRAN, che non sia altro che una notazione "ad alto livello" per un linguaggio macchina.
Benché sia possibile tradurre completamente un programma in linguaggio macchina "puro", questa operazione produrrebbe centinaia di istruzioni facendo crescere a dismisura la dimensione del codice prodotto.
La soluzione consiste nell'inserire nel codice delle chiamate a routine (indipendenti dal particolare programma), caricate su M_O come parte del RTC.
Nei vari linguaggi ad alto livello questa situazione si presenta per quasi tutti i costrutti del linguaggio: meccanismi di controllo, routine e strutture dati.

2.10 Compilazione ed interpretazione mista

Nel caso del compilatore C, il codice prodotto è scritto in linguaggio macchina esteso con chiamate al rts, che contiene:

- Strutture dati: pila dei record di attivazione (ambiente, memoria, sottoprogrammi, ecc.); la memoria a heap (puntatori, ecc.).
- Sottoprogrammi che realizzano le operazioni sulle varie strutture dati.

Nel caso di un linguaggio come Java, si ha invece un'implementazione mista, poiché l'interprete della macchina intermedia M_I non coincide con quello di M_O. In questo caso esiste un ciclo di interpretazione del linguaggio intermedio L_{M_I} realizzato su M_O.
Questo consente di ottenere codice tradotto più compatto e favorisce la portabilità.

Confronto fra i due casi:

- Nella compilazione pura non c'è un livello intermedio di interpretazione, che causa inefficienza.
- Una buona implementazione mista consente la produzione di codice di dimensioni molto ridotte.
- Un'implementazione mista è più portabile.
- Il rts di un compilatore si ritrova quasi uguale nelle strutture dati e routine utilizzate dall'interprete del linguaggio intermedio.

2.11 Riassunto: famiglie di implementazioni

- Interprete puro
  - $M_L = M_I$
  - Interprete di L realizzato su M_O
  - Alcune implementazioni (vecchie) di linguaggi logici e funzionali (LISP, PROLOG)
- Compilatore
  - Macchina intermedia M_I realizzata per estensione sulla macchina ospite M_O con rts e nessun interprete (C, C++, PASCAL)[5].
- Implementazione mista
  - Traduzione dei programmi da L a L_{M_I}
  - I programmi L_{M_I} sono interpretati su M_O (Java, i "compilatori" per linguaggi funzionali e logici, alcune (vecchie) implementazioni di Pascal).

2.12 Implementazioni miste ed interpreti puri

L'implementazione mista presenta alcuni vantaggi rispetto all'interpretazione pura:

- La traduzione genera un linguaggio più facile da interpretare su una macchina ospite.
- E' possibile effettuare staticamente, a tempo di traduzione, analisi, verifiche ed ottimizzazioni che migliorano affidabilità ed efficienza.
- Varie proprietà interessanti, come inferenza e controllo dei tipi, controllo sull'uso dei nomi e loro risoluzione "statica".

2.13 Analisi statica

Alcuni linguaggi non permettono praticamente nessun tipo di analisi statica, come nel caso di LISP con scoping dinamico.
Altri linguaggi funzionali più moderni (ML) permettono di:

- Localizzare errori.
- Eliminare controlli a tempo di esecuzione (type checking dinamico nelle operazioni).

[5]Esistono implementazioni compilate di Java, ma non di PROLOG, LISP o ML, perché eccessivamente differenti dal linguaggio macchina.

- Semplificare alcune operazioni a tempo di esecuzione (es. trovare il valore denotato da un nome.

Vediamo alcune caratteristiche dell'analisi statica in **Java**:

- Fortemente tipato: il type checking può essere in gran parte effettuato dal traduttore, sparendo quindi dal byte-code generato.
- Le relazioni di **subtyping** permettono che un'entità abbia un tipo vero (*actual type*) ed un tipo apparente (*apparent type*), per cui:
  - Il tipo apparente è noto a tempo di traduzione.
  - Il tipo vero è noto solo a tempo di esecuzione.
  - E' garantito che il tipo apparente sia un supertype di quello vero.
- Alcune questioni legate ai tipi possono essere risolte solo a tempo di esecuzione:
  - Scelta del più specifico fra diversi metodi overloaded.
  - Casting.
  - Dispatching dei metodi.
- Controlli e simulazioni a tempo di esecuzione.

2.14 Quando la macchina ospite è "ad alto livello"

Consideriamo le caratteristiche della macchina ospite M_O utilizzata nell'implementazione del linguaggio L.
Il linguaggio L_{M_O} è il linguaggio in cui viene implementato, a seconda dei casi, l'interprete di L, oppure l'interprete della macchina intermedia M_I di L, oppure il supporto a tempo di esecuzione di L.
Se L_{M_O} è a sua volta un linguaggio ad alto livello, le implementazioni portano ad una stratificazione di macchine astratte[6], per cui, in altri termini, l'implementazione di L viene eseguita "sopra al supporto a tempo di esecuzione" che realizza M_O.
La situazione appena descritta è legata alla maggiore facilità di implementazione di un qualsiasi insieme di algoritmi e strutture dati in un linguaggio ad alto livello piuttosto che in linguaggio macchina, ma comporta una possibile perdita di efficienza legata alla stratificazione di macchine astratte.

La parte più critica di questo tipo di scelta riguarda il modo in cui l'implementazione tratta quei costrutti propri di L che hanno un corrispettivo diretto nel linguaggio di implementazione. Per evidenziare questo aspetto

[6] Questa caratteristica va tenuta in considerazione se si vuole ragionare sulle prestazioni dei programmi in esecuzione.

consideriamo una implementazione interpretativa di L realizzata in C, in cui, quindi, l'interprete realizzato (e quindi anche compilato dal compilatore C) gira sul supporto a tempo di esecuzione di C, con le sue strutture dati (pile, heap, ecc.).
Supponiamo che L abbia alcuni costrutti (astrazioni procedurali, ricorsione, allocazione dinamica su heap, ecc.).
L'interprete di L può valutare questi costrutti in diversi modi. I casi limite sono:

- Utilizzazione diretta del corrispondente costrutto di C (e quindi di quella parte del rts di C che lo realizza).
- Simulazione ex-novo del costrutto, con l'introduzione delle strutture dati necessarie (che potrebbero essere una replica molto simile di quelle del rts di C).

Nel primo caso lo strato aggiunto sopra a C è minore, anche se l'implementazione risultante non è necessariamente più efficiente.
Considerazioni molto simili si applicano alle altre classi di implementazioni.
Nel caso di implementazioni miste in cui la macchina intermedia è definita "a priori", come Java o Prolog, è comunque molto difficile riuscire a riutilizzare le strutture dati del supporto di C.

2.15 Semantica formale e macchine astratte

Gli aspetti sui quali ci concentriamo:

- **Semantica formale**: in forma eseguibile e con implementazione ad altissimo livello.
- **Implementazioni o macchine astratte**: interpreti e supporto a tempo di esecuzione.

Perché la semantica formale?

> Definizione precisa del linguaggio, indipendente dall'implementazione.
>
> - Il progettista la definisce.
> - L'implementatore la utilizza come specifica.
> - Il programmatore la utilizza per ragionare sul significato dei propri programmi.

Perché le macchine astratte?

> - Il progettista deve tenere conto delle caratteristiche possibili dell'implementazione.

- L'implementatore la realizza.
- Il programmatore la deve conoscere per utilizzare al meglio il linguaggio.

3 Un po' di algebra

L'algebra risulta necessaria per discutere la semantica denotazionale.
In particolare, nel seguito saranno trattati:

- Reticoli.
- Operatori su reticoli.
- Punti fissi e teoremi relativi.
- Calcolare un punto fisso.

3.1 Ordinamento parziale

Una relazione binaria su un insieme S è un **ordinamento parziale**, se gode delle proprietà **riflessiva**, **antisimmetrica** e **riflessiva**.

$(S, \leq)$ con $\leq$ ordinamento parziale su S

$$X \subseteq S, a \in S$$

- a è un **limite superiore** (upper bound) di X, se $\forall x \in X, x \leq a$
- a è un **limite inferiore** (lower bound) di X, se $\forall x \in X, a \leq x$
- a è un **minimo limite superiore** (least upper bound) di X, se
  - a è un limite superiore di X
  - $\forall b$ limite superiore di X, $a \leq b$
- a è un **massimo limite inferiore** (greatest lower bound) di X, se
  - a è un limite inferiore di X
  - $\forall b$ limite superiore di X, $b \leq a$

Il minimo limite superiore ed il massimo limite inferiore, se esistono, sono **unici** e si indicano con **lub(X)** e **glb(X)**.

3.2 Reticolo completo e minimo punto fisso

Un insieme parzialmente ordinato $(S, \leq)$ è un **reticolo completo** se $\forall X \subseteq S$ esistono $lub(X)$ e $glb(X)$ In questo caso indichiamo:

- $\top$ denota $lub(S)$ (massimo del reticolo)
- $\bot$ denota $glb(S)$ (minimo del reticolo)

$X \subseteq S$ è un **diretto**, se ogni sottoinsieme finito di X ha un limite superiore in X.

Dato un reticolo completo $(S, \leq)$ ed una funzione $\varphi : S \to S$, φ è un **operatore** su S.

- φ è **monotono**, se $\varphi(x) \leq \varphi(y)$, quando $x \leq y$.
- φ è **continuo**, se $\varphi(lub(X)) = lub\{\varphi(x) : x \in X\}, \forall X$ diretto di S.
- Se φ è continuo, è anche monotono.

Dato un reticolo completo $(S, \leq)$ ed un operatore $\varphi : S \to S$, $a \in S$ è il **minimo punto fisso** (least fixpoint, lfp) di φ se

- a è un punto fisso di φ, cioè $\varphi(a) = a$
- $\forall b$ punto fisso di φ, $a \leq b$

3.3 Teoremi

Teorema di Tarski:

> Se $(S, \leq)$ è un reticolo completo e $\varphi : S \to S$ è un operatore *monotono* su S, allora φ ha un minimo punto fisso $lfp(\varphi)$ e $lfp(\varphi) = glb\{x : \varphi(x) = x\}$

Il teorema afferma che $lfp(\varphi)$ esiste ed è il massimo limite inferiore dell'insieme dei punti fissi di φ.

Per ottenere una caratterizzazione costruttiva abbiamo bisogno di ipotesi più forti e dobbiamo ricorrere alle **potenze ordinali**.
Consideriamo i numeri ordinali[7] (rappresentati come insiemi, ciascuno dei quali contiene tutti i precedenti):

- Ordinali **finiti**:

$$0 = \varnothing; 1 = \{\varnothing\}; 2 = \{\varnothing\{\varnothing\}\} = \{0, 1\}; \ldots$$

[7] In matematica, i numeri ordinali costituiscono una estensione dei numeri naturali che tiene conto anche di successioni infinite. Nella teoria degli insiemi, i numeri naturali sono solitamente costruiti con gli insiemi, in modo tale che ogni numero naturale è l'insieme di tutti i numeri naturali più piccoli di esso:

$$0 = \varnothing(\textit{insieme vuoto}); 1 = \{0\} = \{\varnothing\}; 2 = \{0, 1\} = \{\varnothing, \{\varnothing\}\} \ldots$$

Visto in questo modo, ogni numero naturale è un insieme ben ordinato: l'insieme 4, per esempio, contiene gli elementi $0, 1, 2, 3$ che sono ovviamente ordinati in questo modo: $0 < 1 < 2 < 3$. Un numero naturale è più piccolo di un altro se e solo se è un elemento dell'altro.

- Ordinali **transfiniti**:

$$\omega = \{0, 1, 2, 3, \ldots\} : \ldots ; \alpha; \ldots ; \beta; \ldots \quad \alpha < \beta, se\ \alpha \in \beta$$

Consideriamo un reticolo completo $(S, \leq)$ ed un operatore φ *monotono* su S. Si ha:

$$\begin{aligned} &\varphi \uparrow 0 = \bot \\ &\varphi \uparrow (n+1) = \varphi(\varphi \uparrow n) \\ &\varphi \uparrow \omega = lub\{\varphi \uparrow n \mid n < \omega\} \\ &\varphi \uparrow (\alpha) = \varphi(\varphi \uparrow (\alpha - 1)) \end{aligned}$$

Teorema di Kleene:

> Se $(S, \leq)$ è un reticolo completo e $\varphi S \rightarrow S$ è un operatore *continuo* su S, allora $lfp(\varphi) = \varphi \uparrow \omega$

Quindi se l'operatore φ è continuo, il suo minimo punto fisso $lfp(\varphi) = \varphi \uparrow \omega = lub\{\varphi \uparrow n \mid n < \omega\}$ può essere calcolato come il lub dell'insieme di tutte le iterazioni finite di φ, a partire dal minimo del reticolo $\bot$.
Naturalmente non si può calcolare effettivamente perché sono necessarie ω iterazioni.

3.4 Uso del minimo punto fisso

Il minimo punto fisso serve a dare un significato alle definizioni ricorsive, ed in particolare alle funzioni parziali definite ricorsivamente.
Consideriamo, ad esempio, la definizione ricorsiva della funzione parziale fattoriale.

$$\texttt{fact} = \lambda \texttt{x}.\ \texttt{if x} = \texttt{0 then 1 else x} * \texttt{fact}(\texttt{x} - \texttt{1})$$

La funzione ricorsiva va intesa come la definizione di un operatore φ_{fact} (continuo) su `F` (funzionale)

$$\varphi_{\texttt{fact}} = \lambda \texttt{f}.\ \lambda \texttt{x}.\ \texttt{if x} = \texttt{0 then 1 else x} * \texttt{f}(\texttt{x} - \texttt{1})$$

di cui la funzione definita `fact` è il minimo punto fisso[8]

$$\texttt{fact} = \texttt{lfp}(\varphi_{\texttt{fact}})$$

$lfp(\varphi_{fact})$ può essere "calcolato", in accordo con il teorema di Kleene (vedi 3.3).

[8] Con questa definizione `fact` diventa iterativa e non più ricorsiva.

4 Elementi di semantica denotazionale ed operazionale

- Sintassi astratta e domini sintattici.
- Semantica denotazionale.
- Dalla semantica denotazionale a quella operazionale.
- Semantica e paradigmi.
- Semantica e supporto a tempo di esecuzione.
- Verso definizioni semantiche "eseguibili".

4.1 Sintassi e semantica

Come ogni sistema formale, un linguaggio di programmazione possiede una sintassi ed una semantica.
La *teoria dei linguaggi formali* fornisce formalismi di specifica e tecniche di analisi per trattare gli aspetti sintattici, mentre per quanto riguarda la semantica esistono diverse teorie. Nel seguito saranno descritte le due teorie più importanti, ovvero la **semantica denotazionale** e la **semantica operazionale**.

La semantica formale viene di solito definita su una rappresentazione dei programmi in **sintassi astratta** piuttosto che sulla rappresentazione sintattica concreta. Mentre in sintassi concreta i costrutti sono rappresentati come stringhe, la cui struttura importante dal punto di vista semantico è riconoscibile solo attraverso un'analisi sintattica, in sintassi astratta ogni costrutto è un'espressione (o albero) descritto in termini di applicazione di un operatore (con tipo ed arietà n) ad n operandi, che sono a loro volta espressioni.
La rappresentazione di un programma in sintassi astratta è "isomorfa" all'albero sintattico costruito dall'analisi sintattica.

4.2 Domini sintattici

La sintassi astratta è definita specificando i **domini sintattici**

- Nomi di dominio, con metavariabili relative.
- Definizioni sintattiche.

Esempio
Vediamo l'esempio di un frammento di linguaggio imperativo incompleto[9], assumendo l'esistenza del dominio sintattico `IDE` degli *identificatori*, con metavariabili $\mathtt{I}, \mathtt{I}_1, \mathtt{I}_2$.

- Domini:
  - `EXPR` (espressioni), con metavariabili $\mathtt{E}, \mathtt{E}_1, \mathtt{E}_2, \ldots$
  - `COM` (comandi), con metavariabili $\mathtt{C}, \mathtt{C}_1, \mathtt{C}_2, \ldots$
  - `DEC` (dichiarazioni), con metavariabili $\mathtt{D}, \mathtt{D}_1, \mathtt{D}_2, \ldots$
  - `PROG` (programma), con metavariablie `P`
- Definizioni sintattiche:
  - $\mathtt{E} ::= \mathtt{I} \mid \mathtt{val}(\mathtt{I}) \mid \mathtt{lambda}(\mathtt{I}, \mathtt{E}_1) \mid \mathtt{plus}(\mathtt{E}_1, \mathtt{E}_2) \mid \mathtt{apply}(\mathtt{E}_1, \mathtt{E}_2)$
  - $\mathtt{C} ::= \mathtt{ifthenelse}(\mathtt{E}, \mathtt{C}_1, \mathtt{C}_2) \mid \mathtt{while}(\mathtt{E}, \mathtt{C}_1) \mid \mathtt{assign}(\mathtt{I}, \mathtt{E}) \mid \mathtt{cseq}(\mathtt{C}_1, \mathtt{C}_2)$
  - $\mathtt{D} ::= \mathtt{var}(\mathtt{I}, \mathtt{E}) \mid \mathtt{dseq}(\mathtt{D}_1, \mathtt{D}_2)$
  - $\mathtt{P} ::= \mathtt{prog}(\mathtt{D}, \mathtt{C})$

4.3 Semantica denotazionale: domini

La semantica denotazionale associa ad ogni costrutto sintattico la sua **denotazione**, ovvero una funzione che ha come dominio e codominio opportuni **domini semantici**.
I domini semantici vengono definiti da **equazioni di dominio**: $nome - dominio = espressione - di - dominio$.
Le espressioni di dominio sono composte utilizzando i **costruttori di dominio**:

- **Enumerazione di valori**: $\mathtt{bool} = \{\mathtt{True}, \mathtt{False}\}; \mathtt{int} = \{1, 2, 3, \ldots\}$
- **Somma di domini**: $\mathtt{val} = [\mathtt{int} + \mathtt{bool}]$ (un elemento appartiene ad *int* oppure *bool*).
- **Iterazione finita di un dominio**: $\mathtt{listval} = \mathtt{val}*$ (un elemento è una sequenza finita, o lista, di elementi di *val*).
- **Funzioni tra domini**: $\mathtt{env} = \mathtt{IDE} \rightarrow \mathtt{val}$ (un elemento è una funzione da *IDE* a *val*).
- **Prodotto cartesiano di domini**: $\mathtt{stato} = \mathtt{env} * \mathtt{store}$ (un elemento è una coppia formata da un elemento di *env* ed uno di *store*).

Per ogni costruttore di dominio esiste un metodo per definire l'ordinamento parziale del dominio, a partire da quelle dei dei domini utilizzati, in modo da garantire che tutti i domini siano effettivamente reticoli completi.

[9] Mancano le costanti (ad esempio interi e booleani) più altre operazioni necessarie.

4.4 Domini semantici

Lo stato
In un qualunque linguaggio ad alto livello lo stato deve comprendere un dominio chiamato **ambiente** (environment), che modelli l'associazione tra identificatori e valori che questi possono denotare.

> $\texttt{env} = \texttt{IDE} \to \texttt{dval}$, con metavariabili $\rho, \rho_1, \rho_2, \ldots$
> Indichiamo con $[\rho/\texttt{I} \leftarrow \texttt{d}]$ l'ambiente $\rho' = \lambda\texttt{x}.\ \texttt{if x} = \texttt{I then d else } \rho\ \texttt{x}$

Il nostro frammento è imperativo ed ha la nozione di entità modificabile, cioè le variabili, che sono create con le dichiarazioni e modificate con gli assegnamenti. Non è quindi possibile modellare lo stato con un'unica funzione, perchè è possibile che identificatori diversi denotino la stessa locazione (*aliasing*), ed è dunque necessario un secondo componente dello stato che chiamiamo **memoria** (store).

> $\texttt{store} = \texttt{loc} \to \texttt{mval}$, con metavariabili $\sigma, \sigma_1, \sigma_2$
> Indichiamo con $[\sigma/\texttt{l} \leftarrow \texttt{m}]$ la memoria $\sigma' = \lambda\texttt{x}.\ \texttt{if x} = \texttt{l then m else } \sigma\ \texttt{x}$

I valori
Abbiamo introdotto, utilizzandoli ma non definendoli, due domini semantici di valori:

- `dval` (valori denotabili), dominio dei valori che possono essere denotati da un identificatore nell'ambiente.
- `mval` (valori memorizzabili), dominio dei valori che possono essere contenuti in una locazione di memoria.

E' necessario introdurre un terzo dominio semantico di valori:

- `eval` (valori esprimibili), dominio dei valori che possono essere ottenuti come semantica di un'espressione.

I tre domini sono in generale diversi anche se possono avere delle sovrapposizioni. In questi casi utilizzeremo delle *funzioni di trasformazione di tipo*. Per capire come sono fatti i domini analizzeremo il linguaggio, assumendo come predefiniti i domini *int* e *bool* (con le relative operazioni primitive) ed il dominio *loc* con una "funzione": $\texttt{newloc} : \to \texttt{loc}$ (che restituisce una nuova locazione).

Il dominio semantico `fun`
Il linguaggio definito prevede che le espressioni contengano l'astrazione (vedi 4.2), $\texttt{lambda}(\texttt{I}, \texttt{E}_1)$ rappresenta infatti una funzione, il cui corpo è l'espressione E_1 con parametro formale I.
La semantica di tale costrutto è una funzione del dominio `fun`

$$\texttt{fun} = \texttt{store} \to \texttt{dval} \to \texttt{eval}$$

il valore di tipo `dval` sarà il valore dell'argomento dell'applicazione. Come vedremo, il passaggio di parametri sarà effettuato nell'ambiente.

Caratteristiche dei domini semantici `eval`, `dval`, `mval`

- `eval`: deve contenere `int` e `bool` (predefiniti) e `fun`. Decidiamo che non deve contenere `fun`, benché la sintassi lo permetterebbe.
- `dval`: deve contenere `loc`. Decidiamo che contenga anche `int`, `bool` (costanti booleane) e `fun`.
- `mval`: deve contenere `int` e `bool`. Decidiamo che non contiene `loc` e `fun` benché la sintassi lo permetterebbe.

Quindi:

$$\texttt{eval} = [\texttt{int} + \texttt{bool} + \texttt{fun}]$$
$$\texttt{dval} = [\texttt{loc} + \texttt{int} + \texttt{bool} + \texttt{fun}]$$
$$\texttt{mval} = [\texttt{int} + \texttt{bool}]$$

4.5 Specifica della semantica denotazionale

Funzioni di valutazione semantica
Definiamo le funzioni di valutazione semantica con riferimento alla sintassi definita in precedenza (vedi 4.2):

E: $\texttt{EXPR} \to \texttt{env} \to \texttt{store} \to \texttt{eval}$
C: $\texttt{COM} \to \texttt{env} \to \texttt{store} \to \texttt{store}$
D: $\texttt{DEC} \to \texttt{env} \to \texttt{store} \to (\texttt{env} * \texttt{store})$
P: $\texttt{PROG} \to \texttt{env} \to \texttt{store} \to \texttt{store}$

Le funzioni di valutazione semantica assegnano un significato ai vari costrutti, con una definizione data sui casi della sintassi astratta.

Semantica delle espressioni

$$E(\texttt{I}) = \lambda\rho.\lambda\sigma.\ \texttt{dvaltoeval}(\rho(\texttt{I}))$$
$$E(\texttt{val(I)}) = \lambda\rho.\lambda\sigma.\ \texttt{mvaltoeval}(\sigma\ \rho(\texttt{I}))$$
$$E(\texttt{plus}(\texttt{E}_1, \texttt{E}_2)) = \lambda\rho.\lambda\sigma.\ (E(\texttt{E}_1)\ \rho\ \sigma) + (E(\texttt{E}_2)\ \rho\ \sigma)$$

Nel caso dell'operazione `plus`, ad esempio vediamo come la semantica di una espressione sia definita per composizione delle semantiche delle sue sottoespressioni (**composizionalità**).
Per dare la semantica dei costrutti riguardanti le funzioni definiamo:

$$\texttt{makefun}(\texttt{lambda}(\texttt{I}, \texttt{E}_1), \rho) = \lambda\sigma'.\lambda\texttt{d}.\ E(\texttt{E}_1)\ [\rho/\texttt{I} \leftarrow \texttt{d}]\ \sigma'$$
$$\texttt{applyfun}(\texttt{e}, \texttt{d}, \sigma) = \texttt{e}\ \sigma\ \texttt{d}$$

Definiamo quindi la semantica come:

$E(\texttt{lambda}(\texttt{I}, \texttt{E}_1)) = \lambda\rho.\lambda\sigma.\ \texttt{makefun}(\texttt{lambda}(\texttt{I}, \texttt{E}_1), \rho)$
$E(\texttt{apply}(\texttt{E}_1, \texttt{E}_2) = \lambda\rho.\lambda\sigma.\ \texttt{applyfun}(E(\texttt{E}_1)\ \rho\ \sigma, \texttt{evaltodval}(E(\texttt{E}_2)\ \rho\ \sigma), \sigma)$

Semantica dei comandi

$C(\texttt{assign}(\texttt{I}, \texttt{E})) = \lambda\rho.\lambda\sigma.\ [\sigma/\rho(\texttt{I}) \leftarrow \texttt{evaltomval}(E(\texttt{E})\ \rho\ \sigma)]$
$C(\texttt{cseq}(\texttt{C}_1, \texttt{C}_2)) = \lambda\rho.\lambda\sigma.\ C(\texttt{C}_2)\ \rho(C(\texttt{C}_1)\ \rho\ \sigma)$
$C(\texttt{ifthenelse}(\texttt{E}, \texttt{C}_1, \texttt{C}_2)) = \lambda\rho.\lambda\sigma.\ \texttt{if}\ E(\texttt{E})\ \rho\ \sigma\ \texttt{then}\ C(\texttt{C}_1)\ \rho\ \sigma\ \texttt{else}\ C(\texttt{C}_2)\ \rho\ \sigma$

Per poter dare una semantica composizionale del *while* abbiamo bisogno di un **punto fisso**.

$C(\texttt{while}(\texttt{E}, \texttt{C}_1)) = \lambda\rho.\lambda\sigma.\ (\mu\texttt{f}.\lambda\sigma'.\ \texttt{if}\ E(\texttt{E})\ \rho\ \sigma'\ \texttt{then}\ \texttt{f}(C(\texttt{C}_1)\ \rho\ \sigma')\ \texttt{else}\ \sigma')\ \sigma$

Dove la funzione $f : \mathit{store} \rightarrow \mathit{store}$ è il minimo punto fisso (μ) del funzionale, che, dopo essere stato calcolato, viene applicato allo store corrente σ.

Semantica delle dichiarazioni

$D(\texttt{var}(\texttt{I}, \texttt{E})) = \lambda\rho.\lambda\sigma.\ \texttt{let loc}\ =\ \texttt{newloc}()\ \texttt{in}\ ([\rho/\texttt{I} \leftarrow \texttt{loc}], [\sigma/\texttt{loc} \leftarrow E(\texttt{E})\ \rho\ \sigma])$
$D(\texttt{dseq}(\texttt{D}_1, \texttt{D}_2)) = \lambda\rho.\lambda\sigma.\ \texttt{let}\ (\rho', \sigma') = D(\texttt{D}_1)\rho\ \sigma\ \texttt{in}\ D(\texttt{D}_2)\ \rho'\ \sigma'$

Semantica dei programmi

$P(\texttt{prog}(\texttt{D}, \texttt{C})) = \lambda\rho.\lambda\sigma.\ \texttt{let}\ (\rho', \sigma') = D(\texttt{D})\ \rho\ \sigma\ \texttt{in}\ C(\texttt{C})\ \rho'\ \sigma'$

4.6 Caratteristiche della semantica denotazionale

In semantica denotazionale la semantica di un costrutto è definita per composizione delle semantiche dei suoi componenti (**composizionalità**).
La semantica assegna una denotazione al programma, attraverso le funzioni sui domini semantici, senza aver bisogno di conoscere "lo stato" in un particolare momento.
La costruzione della denotazione richiede un calcolo di **minimo punto fisso**, poiché le funzioni di valutazione semantica sono definite in modo ricorsivo (questo dipende appunto dalla loro definizione per composizione).
La semantica di un programma P è dunque:

$$P(\texttt{P}) \uparrow \omega$$

Nella **semantica denotazionale standard** si usa un terzo domino sintattico (oltre a `env` e `store`): le **continuazioni**.
Questo dominio consente il trattamento di costrutti "particolari", come i *salti*.

4.7 La semantica operazionale

Lo stile di rappresentazione tradizionale è quello dei **sistemi di transizione**, ovvero un insieme di regole che definiscono, attraverso funzioni di transizione, il modo in cui lo stato cambia per effetto dell'esecuzione dei vari costrutti.

Esempio (Comandi):
Configurazioni: triple $< COM, env, store >$
Relazione di transizione: $configurazione \rightarrow_{com} store$
Una regola di transizione:

$$\frac{< C_1, \rho, \sigma > \rightarrow_{com} \sigma_2 \qquad < C_2, \rho, \sigma_2 > \rightarrow_{com} \sigma_1}{< cseq(C_1, C_2), \rho, \sigma > \rightarrow_{com} \sigma_1}$$

Ecco la corrispondente funzione di valutazione semantica (in semantica denotazionale):

$$C(\texttt{cseq}(\texttt{C}_1, \texttt{C}_2)) = \lambda\rho.\lambda\sigma.\ C(\texttt{C}_2)\ \rho\ (C(\texttt{C}_1)\ \rho\ \sigma)$$

4.8 Relazioni o funzioni di transizione?

- Le transizioni si rappresentano più naturalmente attraverso le relazioni, che consentono inoltre di modellare transizioni nondeterministiche (programmazione logica e linguaggi concorrenti).
- Negli altri casi è possibile rappresentare le transizioni attraverso delle funzioni, simili alle funzioni di valutazione semantica dello stile denotazionale.

4.9 Il nostro stile di specifica della semantica operazionale

Per specificare la semantica operazionale utilizziamo lo stesso metalinguaggio della semantica denotazionale, lo stesso dominio sintattico e gli stessi domini semantici relativi allo stato.
Le differenze sostanziali, che sono quelle che distinguono concretamente i due stili, riguardano:

- Il "tipo" delle funzioni di valutazione semantica:
  (denotazionale) $C : \texttt{COM} \rightarrow \texttt{env} \rightarrow \texttt{store} \rightarrow \texttt{store}$ (*Ordine superiore*)
  (operazionale) $C : \texttt{COM} * \texttt{env} * \texttt{store} \rightarrow \texttt{store}$ (*Primo ordine*)

- I domini di valori "funzionali" (astrazioni funzionali):
  (denotazionale) $\texttt{fun} = \texttt{store} \to \texttt{dval} \to \texttt{eval}$
  (operazionale) $\texttt{fun} = \texttt{EXPR} * \texttt{env}$
- Eliminazione del calcolo dei punti fissi.

Tutti e tre i casi sono riconducibili all'eliminazione di domini funzionali di ordine superiore, che si ottiene utilizzando oggetti sintattico nei domini semantici e/o perdendo la composizionalità.

4.10 Specifica della semantica operazionale

Nel seguito vengono riportate le modifiche che interessano la semantica operazionale rispetto a quanto già definito in precedenza con la semantica denotazionale (vedi 4.5).

Domini e funzioni di valutazione

I domini:

$\texttt{env} = \texttt{IDE} \to \texttt{dval}$
$\texttt{store} = \texttt{loc} \to \texttt{mval}$
$\texttt{fun} = \texttt{EXPR} * \texttt{env}$
$\texttt{eval} = [\texttt{int} + \texttt{bool} + \texttt{fun}]$
$\texttt{dval} = [\texttt{loc} + \texttt{int} + \texttt{bool} + \texttt{fun}]$
$\texttt{mval} = [\texttt{int} + \texttt{bool}]$

Le funzioni di valutazione semantica al primo ordine.

E: $\texttt{EXPR} * \texttt{env} * \texttt{store} \to \texttt{eval}$
C: $\texttt{COM} * \texttt{env} * \texttt{store} \to \texttt{store}$
D: $\texttt{DEC} * \texttt{env} * \texttt{store} \to (\texttt{env} * \texttt{store})$
P: $\texttt{PROG} * \texttt{env} * \texttt{store} \to \texttt{store}$

Semantica delle espressioni

Rispetto al caso denotazionale, le differenze riguardano, anche, le definizioni di `makefun` e `applyfun`:

$\texttt{makefun}(\texttt{lambda}(\texttt{I}, \texttt{E}_1), \rho) = (\texttt{lambda}(\texttt{I}, \texttt{E}_1), \rho)$
$\texttt{applyfun}((\texttt{lambda}(\texttt{I}, \texttt{E}_1), \rho), \texttt{d}, \sigma) = E(\texttt{E}_1)\ [\rho/\texttt{I} \leftarrow \texttt{d}]\ \sigma$

Infine le definizioni vere e proprie:

$E(\mathtt{I}, \rho, \sigma) = \mathtt{dvaltoeval}(\rho(\mathtt{I}))$
$E(\mathtt{val}(\mathtt{I}), \rho, \sigma) = \mathtt{mvaltoeval}(\sigma\rho(\mathtt{I}))$
$E(\mathtt{plus}(\mathtt{E}_1, \mathtt{E}_2), \rho, \sigma) = E(\mathtt{E}_1, \rho, \sigma) \ + \ E(\mathtt{E}_2, \rho, \sigma)$
$E(\mathtt{lambda}(\mathtt{I}, \mathtt{E}_1), \rho, \sigma) = \mathtt{makefun}(\mathtt{lambda}(\mathtt{I}, \mathtt{E}_1), \rho)$
$E(\mathtt{apply}(\mathtt{E}_1, \mathtt{E}_2), \rho, \sigma) = \mathtt{applyfun}(E(\mathtt{E}_1, \rho, \sigma), \mathtt{evaltodval}(E(\mathtt{E}_2, \rho, \sigma)), \sigma)$

E' importante notare come, rispetto al caso denotazionale (vedi 4.5), si sia spostata l'"esecuzione" delle funzioni all'interno della semantica dell'applicazione piuttosto che nell'astrazione. Per fare questo si utilizza, ed anche questo differenzia notevolmente il caso operazionale da quello denotazionale, alla sintassi astratta, che ricorre all'interno della definizione semantica.

Semantica dei comandi

$C(\mathtt{assign}(\mathtt{I}, \mathtt{E}), \rho, \sigma) = [\sigma \ / \ \rho(\mathtt{I}) \leftarrow \mathtt{evaltomval}(E(\mathtt{E}, \rho, \sigma))]$
$C(\mathtt{cseq}(\mathtt{C}_1, \mathtt{C}_2)\rho, \sigma) = C(\mathtt{C}_2, \rho, C(\mathtt{C}_1, \rho, \sigma))$
$C(\mathtt{ifthenelse}(\mathtt{E}, \mathtt{C}_1, \mathtt{C}_2), \rho, \sigma) = \mathtt{if}\ E(\mathtt{E}, \rho, \sigma)\ \mathtt{then}\ C(\mathtt{C}_1, \rho, \sigma)\ \mathtt{else} C(\mathtt{C}_2, \rho, \sigma)$

Infine la semantica del `while`, che utilizza al posto del punto fisso un'operazione sulla sintassi, che fa perdere la composizionalità.

$C(\mathtt{while}(\mathtt{E}, \mathtt{C}_1), \rho, \sigma) = \mathtt{if}\ E(\mathtt{E}, \rho, \sigma)\ \mathtt{then}\ C(\mathtt{cseq}(\mathtt{C}_1, \mathtt{while}(\mathtt{E}, \mathtt{C}_1))\rho, \sigma)\ \mathtt{else}\ \sigma$

Semantica delle dichiarazioni

$D(\mathtt{var}(\mathtt{I}, \mathtt{E}), \rho, \sigma) = \mathtt{let}\ \mathtt{loc} = \mathtt{newloc}()\ \mathtt{in}\ ([\rho \ / \ \mathtt{I} \leftarrow \mathtt{loc}], [\sigma \ / \ \mathtt{loc} \leftarrow E(\mathtt{E}, \rho, \sigma)])$
$D(\mathtt{dseq}(\mathtt{D}_1, \mathtt{D}_2), \rho, \sigma) = \mathtt{let}(\rho', \sigma') = D(\mathtt{D}_1, \rho, \sigma)\ \mathtt{in}\ D(\mathtt{D}_2, \rho', \sigma')$

Semantica dei programmi

$P(\mathtt{prog}(\mathtt{D}, \mathtt{C}), \rho, \sigma) = \mathtt{let}(\rho', \sigma') = D(\mathtt{D}, \rho, \sigma)\ \mathtt{in}\ C(\mathtt{C}, \rho', \sigma')$

4.11 Caratteristiche della semantica operazionale

Innanzi tutto la semantica operazionale non è sempre operazionale, e non assegna una denotazione al solo programma ma piuttosto definisce come si raggiunge uno stato finale a partire dallo stato iniziale.
Pur essendo le funzioni di valutazione semantica ancora definite in modo ricorsivo, nel caso della semantica operazionale non c'è bisogno di calcolare punti fissi.

4.12 Semantica (denotazionale) e paradigmi

Linguaggi funzionali

- Un unico dominio sintattico: `EXPR`
- Un unico dominio semantico per lo stato[10]: `env`
- `dval` = `eval`
- I valori esprimibili contengono sempre `fun`
- Un'unica funzione di valutazione semantica: E : `EXPR` $\rightarrow$ `env` $\rightarrow$ `eval`

Linguaggi imperativi

- Tre domini sintattici: `EXPR`, `COM`, `DEC`
- Due domini semantici per lo stato: `env` e `store`
- `dval`, `eval`, `mval` sono generalmente diversi.
- I valori su cui si interpretano le astrazioni funzionali (`fun`) sono di solito denotabili[11], mentre le locazioni sono sempre denotabili[12].
- Tre funzioni di valutazione semantica:

$$E\text{: } \texttt{EXPR} \rightarrow \texttt{env} \rightarrow \texttt{store} \rightarrow \texttt{eval}$$
$$C\text{: } \texttt{COM} \rightarrow \texttt{env} \rightarrow \texttt{store} \rightarrow \texttt{store}$$
$$D\text{: } \texttt{DEC} \rightarrow \texttt{env} \rightarrow \texttt{store} \rightarrow (\texttt{env} * \texttt{store})$$

Linguaggi orientati ad oggetti

- Oltre ad `EXPR`, `COM`, `DEC` dichiarazione di classe.
- Tre domini semantici per lo stato: `env`, `store` ed un nuovo dominio `heap` per puntatori ed oggetti.
- `dval`, `eval`, `mval` sono generalmente diversi.
- I valori su cui si interpretano le astrazioni funzionali (`fun`) sono di solito solo denotabili, le locazioni sono sempre denotabili, gli oggetti di solito appartengono a tutti e tre i domini.

[10] Mancando lo store, non ci sono variabili e quindi manca il concetto di stato modificabile.
[11] Non è così nel nostro esempio.
[12] In alcuni casi possono essere anche esprimibili.

- Le funzioni di valutazione semantica prendono e restituiscono anche lo heap:

$$C\colon \texttt{COM} \to \texttt{env} \to \texttt{store} \to \texttt{heap} \to (\texttt{store} * \texttt{heap})$$

Supporto a runtime

Le differenze tra i linguaggi dei diversi paradigmi si riflettono in differenze nelle corrispondenti implementazioni. Tutte le caratteristiche importanti per progettare un interprete o un supporto a runtime possono essere ricavate ispezionando i soli domini semantici della semantica denotazionale.

5 OCAML

Contenuto del capitolo:

- Nucleo funzionale puro
  - Funzioni (ricorsive)
  - Tipi e pattern matching
  - Primitive utili: liste
  - Trascrizione della semantica denotazionale
- Componente imperativo
  - Variabili ed assegnamento
  - Primitive utili: arrays
- Moduli ed oggetti

5.1 Costrutti di base

Espressioni di base:

```
# 25;;
-:  int = 25
# 23 * 17;;
-:  int = 391
# if 2=3 then 23 * 17 else 15;;
-:  int = 15
# if 2=3 the 23 else true;;
This expression has type bool but is used with type int
```

Funzioni

```
# function x -> x + 1 ;;
-:  int -> int = <fun>
# (function x -> x + 1) 3 ;;
-:  int = 4
# function x -> x ;;
-:  'a -> 'a = <fun>
# function x -> function y -> xy ;;
-:  ('a -> 'b) -> 'a -> 'b = <fun>
# function (x,y) -> x+y;;
-:  int * int -> int = <fun>
# (function (x,y) -> x+y) (2,33);;
-:  int = 35
```

Let binding

```
# let x = 3;;
val x :  int = 3
# x;;
-:  int = 3
# let y=5 in x+y;;
-:  int = 8
# let f = function x -> x+1;;
val f:  int -> int = <fun>
# f 3;;
-:  int = 4
# let fact x = if x=0 then 1 else x*fact(x-1) ;;
Unbound value fact
```

Let rec binding

```
# let rec fact x = if x=0 then 1 else x*fact(x-1) ;;
val fact :  int -> int = <fun>
```

5.2 Tipi

Definiamo usando OCAML, i tipi del nostro linguaggio, prendendo come riferimento quanto detto in precedenza (vedi 4.2 e 4.4).

```
# type ide = string;;
# type expr = | Den of ide | Val of ide | Fun of ide * expr
   | Plus of expr * expr | Apply of expr * expr;;
# type eval = Int of int | Bool of bool | Efun of myfun
   | Unbound and myfun = eval -> eval;;
# type env = ide -> eval;;
# type com = Assign of ide * expr
   | Ifthenelse of expr * com list * com list
   | While of expr * com list;;
```

5.3 Liste

Le liste sono un utile tipo di dato primitivo offerto da OCAML. Le liste di OCAML sono dati **immutabili**, su di esse non sono quindi definite operazioni che aggiungono o rimuovono elementi.

```
# let l1 = [ 1; 2; 1 ];;
val l1 :  int list = [1;2;1]
# let l2 = 3::l1 ;;
val l2 :  int list = [3;1;2;1]
# let l3 = l1 @ l2 ;;
val l3 :  int list = [1;2;1;3;1;2;1]
```

```
# List.hd l3 ;;
-:  int =1
# List.tl l3 ;;
-:  int list = [2;1;3;1;2;1]
# List.lenght l3 ;;
-:  int = 7
```

5.4 Tipi e pattern matching

Attraverso il pattern matching è possibile definire le funzioni sul loro dominio, sfruttando il meccanismo dei tipi di ML per l'individuazione di eventuali errori.

```
# let rec sem e rho = match e with
   | Den i -> rho i
   | Plus(e1, e2) -> plus(sem e1 rho, sem e2 rho)
   | Fun(ide, e) -> Efun(function d -> sem e (bind(rho, i, d)))
   | Apply(e1, e2) -> match sem e1 rho with
      | Efun f -> f(sem e2 rho)
      | _ -> failwith(``wrong application'');;

val sem :  expr -> env -> eval = <fun>
```

5.5 Punti fissi

Vediamo come definire in OCAML una funzione che preveda il calcolo di un punto fisso. L'esempio più naturale è la semantica del *while*, come definita nel caso denotazionale.

```
# let rec semc c rho sigma = match c with
   | While(e, cl) ->
     let functional ((fi:store ->store)) = function s ->
       if sem e rho s = Bool(true)
        then fi(semcl cl rho s)
        else s in let rec ssfix =
          function x -> functional ssfix x in ssfix(sigma)
   | _ -> ...

val semc :  com -> env -> store -> store = <fun>
```

5.6 Variabili

Le variabili fanno parte del frammento imperativo di OCAML.

```
# let x = ref(3);;
val x :  int ref = {contents 3}
# !x;;
-:  int = 3
# x := 25;;
-:  unit = ()  (L'assegnamento è un comando che modifica lo stato)
# !x;;
```

```
-:  int = 25
# x := !x + 2; !x;;
-:  int = 27
```

5.7 Arrays

A differenza delle liste gli array di OCAML sono un tipo di dato **mutabile**.

```
# let a = [|1;2;3|];;
val a :  int array = [|1;2;3|]
# let b = Array.make 12 1;;
val b :  int array = [|1;1;1;1;1;1;1;1;1;1;1;1|]
# Array.length b;;
:- int = 12
# Array.get a 0;;
:- int = 1
# Array.set b 3 99;;
-:  unit = ()
# b;;
:- int array = [|1;1;1;99;1;1;1;1;1;1;1;1|]
```

5.8 Moduli

Definizione dell'interfaccia di un modulo:

```
# module type PILA =
sig
   type 'a stack
   val emptystack :  'a stack
   val push :  'a stack -> 'a -> 'a stack
   val pop :  'a stack -> 'a stack
   val top :  'a stack -> 'a
end;;
```

Implementazione del modulo:

```
# module SpecPila:  PILA =
struct
   type 'a stack = Empty | Push of 'a stack * 'a
   let emptystack = Empty
   let push p a = Push(p,a)
   let pop p = match p with
     | Push(p1,_) -> p1
     | Empty -> failwith(``Invalid operation 'pop' on empty stack'')
   let top p = match p with
     | Push(_,a) -> a
     | Empty -> failwith(``Invalid operation 'top' on empty stack'')
```

```
end;;
```

5.9 Classi ed oggetti

Definizione di una classe in OCAML:

```
# class point x_init = (* x_init è il parametro del costruttore *)
object
   val mutable x = x_init
   method get_x = x
   method move d = x <- x+d
end;;
```

Esempio di utilizzo degli oggetti:

```
# let p = new point(3);;
val p :  point = <obj>
# p#get_x;;
-:  int = 3
# p#move 33;;
-:  unit = ()
# p#get_x;;
-:  int = 36
```

5.10 Ereditarietà

Basandoci su quanto appena descritto, vediamo come si esprime l'ereditarietà in OCAML:

```
# class colored_point x (c:string) =
object
   inherit point x
   val c = c
   method color = c
end;;
```

Gli oggetti della classe così definita si costruiscono a partire dagli oggetti della superclasse (es. `# let cp = new colored_point 5 ‘‘red’’;;`) e su di essi è possibile richiamare i metodi di entrambe le classi.

5.11 Il linguaggio didattico

Per il nostro linguaggio didattico prendiamo le cose semanticamente importanti di OCAML, **escludendo**:

- Definizione di nuovi tipi e pattern matching
- Moduli
- Eccezioni

6 Dati

Dati, tipo di dati e strutture dati necessarie e caratteristiche dei linguaggi di programmazione.

- Descrittori, tipi, controllo ed inferenza di tipi.
- Specifica semantica ed implementazione di tipi di dato
  - Implementazioni "sequenziali" (pile non modificabili).
  - Implementazioni delle liste con heap.
  - Termini con pattern matching e unificazione.
  - Ambiente, tabelle.
  - Tipi modificabili (pile modificabili, S-espressioni).
  - Programmi come dati e metaprogrammazione.
- Meccanismi per la definizione di nuovi tipi.
- Astrazioni sui dati.

6.1 Tipi di dato di sistema e di programma

In una macchina astratta (e in una semantica) si possono vedere due classi di tipi di dato (o domini semantici)

- **Tipi di dato di sistema**: domini semantici che definiscono lo stato e strutture dati definite nella simulazione di costrutti di controllo.
- **Tipi di dato di programma**: domini corrispondenti ai tipi primitivi del linguaggio ed i tipi che possono essere definiti dall'utente (quando il linguaggio lo permette).

Tratteremo insieme le due classi, anche se il componente "dati" del linguaggio comprende solo i tipi di dato di programma.

6.2 Cos'è un tipo di dato

Un tipo di dato è una **collezione di valori**, rappresentati da opportune strutture dati, più un insieme di **operazioni** per manipolarli.
Ci interessano due livelli:

- Semantica: una specie di semantica algebrica "implementata" attraverso i meccanismi di OCAML per la definizione di nuovi tipi.
- Implementazione: altre implementazioni in OCAML, date in termini di strutture dati "a basso livello" (array).

6.3 I descrittori di dato

Immaginiamo di voler rappresentare una collezione di valori utilizzando quanto viene fornito da un linguaggio macchina (tipi numerici, caratteri, sequenze di celle di memoria, ecc.)
Qualunque valore è alla fine una stringa di bit che deve essere riconosciuta ed interpretata correttamente associandole (in via di principio) una struttura che contiene la descrizione del tipo (**descrittore di dato**), che viene usata quando si applica un'operazione al dato. Questo meccanismo consente di controllare che il tipo sia quello corretto per l'operazione, selezionare l'operatore giusto per eventuali operazioni overloaded, decodificare correttamente la stringa di bit.

6.4 Tipi a tempo di compilazione e a tempo di esecuzione

Distinguiamo tre casi, in cui l'informazione sul tipo è conosciuta a:

- **tempo di compilazione**: possibile eliminare i descrittori di tipi ed effettuare il type checking (statico) totalmente durante la compilazione (FORTRAN, ML).
- **tempo di esecuzione**: descrittori di tipo necessari per tutti i tipi di dato e type checking (dinamico) effettuato completamente a tempo di esecuzione (LISP, PROLOG).
- **parzialmente a tempo di compilazione**: i descrittori contengono solo l'informazione "dinamica", mentre il type checking è effettuato in parte a tempo di compilazione ed in parte a tempo di esecuzione (JAVA).

6.5 Tipi a tempo di compilazione: specifica o inferenza?

Generalmente l'informazione sul tipo viene fornita con delle asserzioni specifiche: nelle dichiarazioni di costanti e variabili, nelle dichiarazioni di procedura con i tipi dei parametri ed il tipo del risultato.
I alternativa (o in aggiunta) il linguaggio può essere dotato di un algoritmo di analisi statica che riesce ad inferire il tipo di ogni espressione, come nel caso dei linguaggi funzionali moderni (es. ML).

6.6 Tipi come valori esprimibili e denotabili

I tipi come valore esprimibile e denotabile[13] rappresentano un'importante strumento per la definizione di astrazioni sui dati, sempre più importante nei moderni linguaggi funzionali, imperativi ed object-oriented.

6.7 Semantica dei tipi di dato

Utilizzeremo per la semantica semplicemente OCAML, ed in particolare il meccanismo dei tipi varianti (costruttori, ecc.) per definire, per casi, un tipo (generalmente ricorsivo).

[13] Questa caratteristica manca nei linguaggi che ignorano i tipi, come LISP e PROLOG, e nei linguaggi antichi, come FORTRAN ed ALGOL, e viene introdotta da PASCAL.

6.8 Pila non modificabile

Interfaccia:

```
module type PILA =
sig
   type 'a stack
   val emptystack :  int * 'a -> 'a stack
   val push :  'a * 'a stack -> 'a stack
   val pop :  'a stack -> 'a stack
   val top :  'a stack -> 'a
   val empty :  'a stack -> bool
   val lungh :  'a stack -> int
   exception Emptystack
   exception Fullstack
end;;
```

Semantica:

```
module SemPila:  PILA =
struct
   type 'a stack = Empty of int | Push of 'a stack * 'a
   exception Emptystack
   exception Fullstack
   let emptystack (n,x) = Empty(n)
   let rec max = function
     | Empty n -> n
     | Push(p,a) -> max p
   let rec lungh = function
     | Empty n -> 0
     | Push(p,a) -> 1 + lungh(p)
   let push (a,p) = if lungh(p)=max(p)
      then raise Fullstack else Push(p,a)
   let pop = function
     | Push(p,a) -> p
     | Empty n -> raise Emptystack
   let top = function
     | Push(p,a) -> a
     | Empty(n) -> raise Emptystack
   let empty = function
     | Push(p,a) -> false
     | Empty(n) -> true
end;;
```

Diamo ora una semantica "isomorfa" ad una specifica in stile algebrico, escludendo i casi eccezionali.
Il tipo, ricorsivo, è definito per casi attraverso i costruttori e la semantica delle operazioni è definita da un'insieme di equazioni fra termini.

```
'a stack = Empty of int | Push of 'a stack * 'a
```

```
emptystack (n,x) = Empty(n)
lungh(Empty n) = 0
lungh(Push(p,a)) = 1 + lungh(p)
push(a,p) = Push(p,a) pop(Push(p,a)) = p
top(Push(p,a)) = a
empty(Empty n) = true
empty(Push(p,a)) = false
```

Implementazione della pila non modificabile:

```
module ImpPila:  PILA =
struct
   type 'a stack = Pila of ('a array) * int
   exception Emptystack
   exception Fullstack
   let emptystack(nm,x) = Pila(Array.create nm x, -1)
   let push(x,Pila(s,n)) = if n=(Array.length(s) - 1)
     then raise Fullstack
     else (Array.set s (n+1) x; Pila(s,n+1))
   let top(Pila(s,n)) = if n=(-1)
     then raise Emptystack
     else Array.get s n
   let pop(Pila(s,n)) = if n=(-1)
     then raise Emptystack
     else Pila(s,n-1)
   let empty(Pila(s,n)) = if n=(-1) then true else false
   let lungh(Pila(s,n)) = n
end;;
```

Il componente principale dell'implementazione è un array, che rappresenta la memoria fisica nel nostro ipotetico linguaggio macchina, sfruttando una classica implementazione sequenziale, utilizzabile anche per altri tipi di dato come le code.

6.9 Lista (non polimorfa)

Interfaccia:

```
module type LISTAINT =
sig
   type intlist
   val emptylist :  intlist
   val cons :  int * intlist -> intlist
   val tail :  intlist -> intlist
   val head :  intlist -> int
   val empty :  intlist -> bool
   val length :  intlist -> int
   exception Emptylist
end;;
```

Semantica

```
module SemListaint:  LISTAINT =
struct
   type intlist = Empty | Cons of int * intlist
   exception Emptylist
   let emptylist = Empty
   let rec length = function
     | Empty -> 0
     | Cons(n,l) -> 1 + length(l)
   let cons(n,l) = Cons(n,l)
   let tail = function
     | Cons(n,l) -> l
     | Empty -> raise Emptylist
   let head = function
     | Cons(n,l) -> n
     | Empty -> raise Emptylist
   let empty = function
     | Cons(n,l) -> false
     | Empty -> true
end;;
```

6.10 Lista e Pila, confronto e considerazioni

Lista e Pila appena descritte hanno semantiche molto simili. Dal punto di vista dell'implementazione, però, sono necessarie alcune considerazioni:

- L'implementazione tramite un unico array va bene per la coda, che è tipicamente un **dato di sistema** del quale esistono poche istanze, predefinite nell'implementazione del linguaggio.
- La lista è tipicamente un **dato utente**, del quale si possono generalmente costruire un numero arbitrario di istanze all'interno dei programmi.

Per questi motivi sarebbe utile adottare, per l'implementazione delle liste, una strategia che consenta l'utilizzo di un unico array (sequenziale) per rappresentare tante liste (**heap**).

6.11 Lista: implementazione a heap

Implementazione della lista non polimorfa attraverso uno heap (che **non** prevede la rimozione).

```
module ImpListaInt:  LISTAINT =
struct
   type intlist = int
   let heapsize = 100
   let heads = Array.create heapsize 0
   let tails = Array.create heapsize 0
   let next = ref(0)
```

```
  let emptyheap = let index = ref(0) in
    while !index<heapsize do
      Array.set tails !index (!index+1); index:=!index+1
    done;
    Array.set tails (heapsize-1)(-1); next:=0
  exception Fullheap
  exception Emptylist
  let emptylist = -1
  let empty l = if l=(-1) then true else false
  let cons(n,l) = if !next=(-1)
    then raise Fullheap else
    (
      let newpoint = !next in next:=Array.get tails !next;
      Array.set heads newpoint n;
      Array.set tails newpoint l;
      newpoint
    )
  let tail l = if empty l then raise Emptylist
    else Array.get tails l
  let head l = if empty l then raise Emptylist
    else Array.get heads l
  let rec length l = if l=(-1) then 0 else 1 + length(tail l)
end;;
```

6.12 Termini

Attraverso i Termini si rappresentano strutture ad albero composte da simboli. Ogni termine è

- Un simbolo di variabile.
- Un simbolo di funzione n-aria applicato ad n termini.

In ML i valori di un tipo definito per casi sono termini senza variabili

```
type intlist = Empty | Cons of int * intlist
```

mentre i pattern usati per selezionare i casi sono termini con variabili

```
let head = function
  | Cons(n,l) -> n
```

I termini con variabili rappresentano insiemi possibilmente infiniti di strutture dati. Il **pattern matching** può essere usato per definire "selettori", attraverso i quali le variabili del pattern vengono "legate" ai sottotermini che compongono una specifica struttura dati.
Per implementare il pattern matching si ricorre all'algoritmo di **unificazione** tra due termini con variabili, che calcola la sostituzione più generale che rende uguali i due termini (oppure fallisce). La sostituzione ottenuta è rappresentata da un'insieme di equazioni fra termini "in forma risolta" (*variabile = termine*).

6.13 Termini e sostituzioni

Definizione dell'interfaccia:

```
module type TERM =
sig
   type term
   type equationset
   exception MatchFailure
   exception UndefinedMatch
   exception UnifyFailure
   exception OccurCheck
   exception WrongSubstitution
   val unifyterms:  term * term -> equationset
   val matchterms:  term * term -> equationset
   val instantiate:  term * equationset -> term
end;;
```

Semantica

```
module Term:  TERM =
struct
   type term = Var of string | Cons of string * (term list)
   type equationset = (term * term) list

   exception MatchFailure
   exception UndefinedMatch
   exception UnifyFailure
   exception OccurCheck
   exception WrongSubstitution

   let rec ground = function
     | Var _ -> false
     | Cons(_ ,tl) -> groundl tl
   and groundl = function
     | [] -> true
     | t::tl1 -> ground(t) & groundl(tl1)

   let rec occurs (stringa, termine) = match termine with
     | Var st -> stringa = st
     | Cons(_ ,tl) -> occursl(stringa, tl)
   and occursl(stringa, tl) = match tl with
     | [] -> false
     | t::tl1 -> occurs(stringa, t) or occursl(stringa, tl1)

   let rec occursset(stringa, e) = match e with
     | [] -> false
     | (t1,t2)::e1 -> occurs(stringa, t1)
       or occurs(stringa, t2)
       or occursset(stringa, e1)
```

```
let rec rplact(t, v, t1) = match t with
  | Var v1 -> if v1=v then t1 else t
  | Cons(s, tl) -> Cons(s, rplactl(tl, v, t1))
and rplactl(tl, v, t1) = match tl with
  | [] -> []
  | t::tl1 -> rplact(t, v, t1) ::  rplactl(tl1, v, t1)

let rec replace(eqset, stringa, termine) = match eqset with
  | [] -> []
  | (s1, t1)::eqset1 ->
    (rplact(s1, stringa, termine), rplact(t1, stringa, termine))::
      replace(eqset1, stringa, termine)

let rec pairs = function
  | [], [] -> []
  | a::l1, b::l2 -> (a,b)::pairs(l1, l2)
  | _ -> raise UnifyFailure

let rec unify(eq1, eq2) = match eq1 with
  | [] -> eq2
  | (Var x, Var y)::eq11 -> if x=y then unify(eq11, eq2) else
    unify(replace(eq11, x, Var y), (Var x, Var y)::
     (replace(eq2, x, Var y)))
  | (t, Var y)::eq11 -> unify((Var y, t)::eq11, eq2)
  | (Var x, t)::eq11 -> if occurs(x, t) then
   raise OccurCheck else
    unify(replace(eq11, x, t), (Var x, t)::(replace(eq2, x, t)))
  | (Cons(x, xl), Cons(y, yl))::eq11 -> if not(x=y) then
    raise UnifyFailure else
    unify(pairs(xl, yl)eq11, eq2)

let unifyterms(t1, t2) = unify([(t1, t2)], [])

let rec pmatch(eq1, eq2) = match eq1 with
  | [] -> eq2
  | (Var x, t)::eq11 -> if occursset(x, eq11eq2) then
    raise UndefinedMatch else
    pmatch(eq11, (Var x, t)::eq2)
  | (Cons(x,xl), Cons(y,yl))::eq11 -> if not(x=y) then
    raise MatchFailure else
    pmatch(pairs(xl, yl)@eq11, eq2)
  | _ -> raise UndefinedMatch

let matchterms(pattern, termine) =
  if not(ground(termine)) then raise UndefinedMatch else
    pmatch([(pattern, termine)],[])

let rec instantiate(t, e) = match e with
```

```
    | [] -> t
    | (Var x, t1)::e1 -> instantiate(rplact(t, x, t1), e1)
    | _ -> raise WrongSubstitution

end;;
```

6.14 Ambiente

L'Ambiente (env) è un tipo polimorfo utilizzato nella semantica per mantenere l'associazione fra identificatori (stringhe) e valori di un opportuno tipo.
La specifica definisce il tipo dell'Ambiente come una funzione.

Interfaccia:

```
module type ENV =
sig
   type 't env
   val emptyenv :  't -> 't env
   val bind :  't env * string * 't -> 't env
   val bindlist :  't env * (string list) * ('t list) -> 't env
   val applyenv :  't env * string -> 't
   exception WrongBindlist
end;;
```

Semantica

```
module Funenv:ENV =
struct
   type 't env = string -> 't
   exception WrongBindlist
   let emptyenv(x) = function y -> x
   let applyenv(x, y) = x y
   let bind(r, l, e) =
     function lu -> if lu=l then e else applyenv(r, lu)
   let rec bindlist(r, il, el) = match (il, el) with
     | ([], []) -> r
     | i::il1, e::el1 -> bindlist (bind(r, i, e), il1, el1)
     | _ -> raise WrongBindlist
  end;;
```

Implementazione (utilizza le liste)

```
module Listenv:ENV =
struct
   type 't env = (string * 't) list
   exception WrongBindlist
   let emptyenv(x) = [('''',x)]
   let rec applyenv(x,y) = match x with
```

```
    | [(_ ,e)] -> e
    | (i1,e1) ::  x1 -> if y = i1 then e1 else applyenv(x1, y)
    | [] -> failwith(''wrong env'')
  let bind(r, l, e) = (l,e) ::  r
  let rec bindlist(r, il, el) = match (il,el) with
    | ([],[]) -> r
    | i::il1, e::el1 -> bindlist (bind(r, i, e), il1, el1)
    | _ -> raise WrongBindlist
end;;
```

6.15 Tipi di dato modificabili

Introduciamo tipi di dato modificabili, riconducendo la modificabilità a livello semantico alla nozione di variabile. Lo stato "modificabile" corrispondente sarà modellato con il dominio *store*.
Per l'implementazione facciamo ricorso a strutture dati modificabili come l'array.

6.16 Pila modificabile

Interfaccia

```
module type MPILA =
sig
  type 'a stack
  val emptystack :  int * 'a -> 'a stack
  val push :  'a * 'a stack -> unit
  val pop :  'a stack -> unit
  val top :  'a stack -> 'a
  val empty :  'a stack -> bool
  val lungh :  'a stack -> int
  val svuota :  'a stack -> unit
  val access :  'a stack * int -> 'a
  exception Emptystack
  exception Fullstack
  exception Wrongaccess
end;;
```

Semantica

```
module SemMPila:  MPILA =
struct
  type 'a stack = ('a SemPila.stack) ref
  exception Emptystack
  exception Fullstack
  exception Wrongaccess
  let emptystack (n, a) = ref(SemPila.emptystack(n, a) )
  let lungh x = SemPila.lungh(!x)
  let push (a, p) = p := SemPila.push(a, !p)
  let pop x = x := SemPila.pop(!x)
```

```
   let top x = SemPila.top(!x)
   let empty x = SemPila.empty !x
   let rec svuota x = if empty(x) then () else (pop x; svuota x)
   let rec faccess (x, n) = if n=0 then
     SemPila.top(x) else faccess(SemPila.pop(x), n-1)
   let access (x, n) = let nofpops = lungh(x) - 1 - n in
     if nofpops < 0 then
       raise Wrongaccess else faccess(!x, nofpops)
end;;
```

Implementazione

```
module ImpMPila:  MPILA =
struct
   type 'x stack = ('x array) * int ref
   exception Emptystack
   exception Fullstack
   exception Wrongaccess
   let emptystack(nm,(x:  'a)) = ((Array.create nm x, ref(-1)):
    'a stack)
   let push(x,((s,n):  'x stack)) = if !n=(Array.length(s)-1) then
     raise Fullstack else (Array.set s (!n +1) x; n := !n +1)
   let top(((s,n):  'x stack)) = if !n = -1 then
     raise Emptystack else Array.get s !n
   let pop(((s,n):  'x stack)) = if !n = -1 then
     raise Emptystack else n:= !n -1
   let empty(((s,n):  'x stack)) = if !n = -1 then true else false
   let lungh( (s,n):  'x stack) = !n
   let svuota (((s,n):  'x stack)) = n := -1
   let access (((s,n):  'x stack), k) = if not(k > !n) then
     Array.get s k else raise Wrongaccess
end;;
```

6.17 S-espressioni

Le S-espressioni sono le strutture dati fondamentali di LISP e rappresentano alberi binari con atomi (stringhe) modificabili come foglie.

Interfaccia

```
module type SEXPR =
sig
   type sexpr
   val nil :  sexpr
   val cons :  sexpr * sexpr -> sexpr
   val node :  string -> sexpr
   val car :  sexpr -> sexpr
   val cdr :  sexpr -> sexpr
```

```
   val null :  sexpr -> bool
   val atom :  sexpr -> bool
   val leaf :  sexpr -> string
   val rplaca:  sexpr * sexpr -> unit
   val rplacd:  sexpr * sexpr -> unit
   exception NotALeaf
   exception NotACons
end;;
```

Semantica

```
module SemSexpr:  SEXPR =
struct
   type sexpr = Nil| Cons of (sexpr ref) * (sexpr ref)|
       Node of string
   exception NotACons
   exception NotALeaf
   let nil = Nil
   let cons (x, y) = Cons(ref(x), ref(y))
   let node s = Node s
   let car = function Cons(x,y) -> !x
     | _ -> raise NotACons
   let cdr = function Cons(x,y) -> !y
     | _ -> raise NotACons
   let leaf = function Node x -> x
     | _ -> raise NotALeaf
   let null = function Nil -> true
     | _ -> false
   let atom = function Cons(x,y) -> false
     | _ -> true
   let rplaca = function (Cons(x, y), z) -> x := z
     | _ -> raise NotACons
   let rplacd = function (Cons(x, y), z) -> y := z
     | _ -> raise NotACons
end;;
```

L'implementazione (a heap) è simile a quella delle liste.

6.18 Programmi come dati

La caratteristica principale della macchina di Von Neumann consiste nel fatto che i programmi sono visti come un particolare tipo di dato, rappresentato nella memoria della macchina. Tale caratteristica consente, in linea di principio, che un qualunque programma possa operare su di essi.
Questo è sempre possibile nel linguaggio macchina, e nei linguaggi ad alto livello in cui la rappresentazione dei programmi è visibile nel linguaggio, ed in cui il linguaggio fornisca operazioni per manipolare programmi.
Dei linguaggi finora considerati solo LISP e PROLOG hanno le caratteristiche

appena indicate[14].

6.19 Metaprogrammazione

Un metaprogramma è un programma che opera su altri programmi (es. interpreti, analizzatori, compilatori, debuggers, ecc.).
La metaprogrammazione risulta utile soprattutto per definire, nel linguaggio stesso, strumenti di supporto allo sviluppo o estensioni del linguaggio.

6.20 Meccanismi per la definizione di tipi di dato

La programmazione di applicazioni consiste in gran parte nella definizione di nuovi tipi di dato, operazione possibile in qualunque linguaggio (compreso il linguaggio macchina).
Gli aspetti importanti nella definizione di un nuovo tipo di dato sono i seguenti:

- *Quanto costa?*
  Il costo della simulazione di un nuovo tipo di dato dipende dal repertorio di strutture dati fornite dal linguaggio (celle di memoria in linguaggio macchina, arrays in FORTRAN, strutture allocate dinamicamente e puntatori in PASCAL e C, s-espressioni in LISP, ecc.). In generale è comunque utile poter disporre di strutture dati statiche sequenziali, come arrays e records, e di un meccanismo per creare strutture dinamiche: tipi di dato dinamico (lista, termine, s-espressione) e allocazione esplicita con puntatori (Pascal, C, oggetti).

- *Esiste il tipo?*
  Nel caso di linguaggi come FORTRAN, ALGOL, LISP o PROLOG, anche realizzando un'implementazione di un dato particolare (es. liste) non abbiamo comunque a disposizione il tipo, dichiarando nuovi oggetti. Questo dipende dal fatto che i tipi non sono denotabili. In PASCAL, ML o Java i tipi sono invece denotabili, anche e con meccanismi diversi:

  - Dichiarazione di tipo: basato sui meccanismi forniti dal linguaggio per costruire espressioni i tipo: enumerazione, record, record ricorsivo in C; enumerazione prodotto cartesiano, iterazione, funzioni, ricorsione, ecc. in ML.
  - Dichiarazione di classe: il nuovo tipo è la classe ed i valori del nuovo tipo (oggetti) sono creati con un'operazione di istanziazione e non con una dichiarazione. La parte struttura dati degli oggetti è costruita da una serie di variabili di istanza allocati sulla heap.

- *Il tipo è astratto?*
  Un tipo astratto è un insieme di valori di cui non si conosce l'implementazione e che possono essere manipolati solo attraverso le operazioni associate. Sono astratti tutti i tipi primitivi forniti dal linguaggio. Per realizzare tipi di dato astratto sono necessari un meccanismo per assegnare un nome al nuovo tipo (dichiarazione di tipo o classe) più un meccanismo di "protezione" che renda

[14] Un programma LISP è visto come una s-espressione, mentre un programma PROLOG è rappresentato da un insieme di termini.

la rappresentazione visibile sollo alle operazioni primitive (variabili di istanza private in una classe, interfacce e moduli in ML e C).

7 Controllo di sequenza: espressioni e comandi

Contenuto del capitolo:

- Espressioni pure (senza blocchi e funzioni)
  - Regola di valutazione, operazioni strette e non strette
- Frammento di linguaggio funzionale
  - Semantica denotazionale
  - Semantica operazionale
  - Interprete iterativo
- Comandi puri (senza blocchi e sottoprogrammi)
  - Semantica dell'assegnamento
- Frammento di linguaggio imperativo
  - Semantica denotazionale
  - Semantica operazionale
  - Interprete iterativo

7.1 Espressioni in sintassi astratta

Per comporre le espressioni le rappresentiamo come alberi etichettati, formati da:

- *Nodi*: applicazioni di funzioni (operazioni primitive), i cui operandi sono i sottoalberi.
- *Figlie*: costanti o variabili (riferimenti ai dati)

Rappresentiamo in questo modo solo *espressioni pure*, che non contengono: definizione di funzione (λ-astrazione), applicazione di funzione, introduzione di nuovi nomi (blocchi).

7.2 Operazioni come funzioni

Le operazioni primitive sono generalmente *funzioni parziali*, indefinite per alcuni valori degli input (errori "hardware" o errori a runtime)[15].
Alcune operazioni primitive sono *funzioni non strette*.

> Una funzione è non stretta sul suo i-esimo operando, se ha un valore definito quando viene applicata ad una n-upla di valori, di cui l'i-esimo è indefinito.

[15] Nei linguaggi moderni questi casi provocano il sollevamento di eccezioni.

7.3 Espressioni: regole di valutazione

Due approcci possibili per la valutazione di un'espressione:

- **Regola interna**: prima di applicare l'operatore si valutano tutti i sottoalberi (sottoespressioni).
- **Regola esterna**: è l'operatore che richiede la valutazione dei sottoalberi se necessario.

Le due regole di valutazione possono dare semantiche diverse, in particolare se una delle sottoespressioni ha valore "indefinito" e l'operatore è non stretto.

Esempi

Condizionale

$$\texttt{ifthenelse}(=(\texttt{x},0),\texttt{y},/(\texttt{y},\texttt{x}))$$

Con la *regola interna* valuto tutti e tre gli operandi. Se x vale 0, la valutazione del terzo operando dà origine ad un errore, e l'intera espressione ha valore indefinito.
Con la *regola esterna* valuto solo il primo operando. Se x vale 0, valuto il secondo operando, mentre il terzo operando non viene mai valutato. L'intera espressione ha in questo caso un valore definito.

OR

$$\texttt{or}(\texttt{true},\texttt{expr1})$$

Con la *regola interna* valuto tutti e due gli operandi. Se la valutazione del secondo operando dà errore, l'intera espressione ha valore indefinito. Al di là della presenza di errori, la valutazione di *expr1* è comunque inutile.
Con la *regola esterna* viene valutato il primo operando. Se il primo operando vale *true* allora il risultato dell'intera espressione è *true*, altrimenti viene valutata *expr1*.

7.4 Regola esterna vs. regola interna

La *regola esterna*:

- E' sempre corretta.
- E' più complessa da implementare, perchè richiede che ogni operatore abbia un propria "politica".
- E' necessaria in pochi casi, per quanto riguarda le operazioni primitive (sono poche le operazioni primitive non strette).

La *regola interna*:

- Non è in generale corretta per le operazioni non strette.
- E' banale da implementare.

La soluzione più ragionevole consiste quindi nell'utilizzare:

- Regola interna per la maggior parte delle operazioni.
- Regola esterna per le poche primitive non strette.

7.5 Frammento funzionale: sintassi

Sintassi di un frammento di linguaggio funzionale:

```
type ide = string
type exp =
   | Eint of int
   | Ebool of bool
   | Den of ide
   | Prod of exp * exp
   | Sum of exp * exp
   | Diff of exp * exp
   | Eq of exp * exp
   | Minus of exp
   | Iszero of exp
   | Or of exp * exp
   | And of exp * exp
   | Not of exp
   | Ifthenelse of exp * exp * exp
```

7.6 Domini semantici (denotazionale)

Definizione dell'unico dominio semantico *eval*.

```
type eval =
   | Int of int
   | Bool of bool
   | Unbound
```

L'implementazione funzionale dell'ambiente è stata data in precedenza (vedi 6.14).

7.7 Operazioni primitive

Prima di definire la semantica del frammento funzionale, vediamo la semantica delle operazioni primitive su *eval*:

```
let typecheck (x, y) =
   match x with
     | ``int'' ->
       (match y with
         | Int(u) -> true
         | _ -> false)
     | ``bool'' ->
       (match y with
         | Bool(u) -> true
```

```
        | _ -> false)
    | _ -> failwith (``not a valid type'')

let minus x =
   if typecheck(``int'',x)
   then
     (match x with
       | Int(y) -> Int(-y) )
   else
     failwith (``type error'')

let iszero x =
   if typecheck(``int'',x)
   then
     (match x with
       | Int(y) -> Bool(y=0) )
   else
     failwith (``type error'')

let equ (x,y) =
   if typecheck(``int'',x) & typecheck(``int'',y)
   then
     (match (x,y) with
       | (Int(u), Int(w)) -> Bool(u = w))
   else failwith (``type error'')

let plus (x,y) =
   if typecheck(``int'',x) & typecheck(``int'',y)
   then
     (match (x,y) with
       | (Int(u), Int(w)) -> Int(u+w))
   else failwith (``type error'')

let diff (x,y) =
   if typecheck(``int'',x) & typecheck(``int'',y)
   then
     (match (x,y) with
       | (Int(u), Int(w)) -> Int(u-w))
   else failwith (``type error'')

let mult (x,y) =
   if typecheck(``int'',x) & typecheck(``int'',y)
   then
     (match (x,y) with
       | (Int(u), Int(w)) -> Int(u*w))
   else failwith (``type error'')

let et (x,y) =
   if typecheck(``bool'',x) & typecheck(``bool'',y)
```

```
   then
     (match (x,y) with
       | (Bool(u), Bool(w)) -> Bool(u & w))
   else failwith (``type error'')

  let vel (x,y) =
   if typecheck(``bool'',x) & typecheck(``bool'',y)
   then
     (match (x,y) with
       | (Bool(u), Bool(w)) -> Bool(u or w))
   else failwith (``type error'')

let non x =
   if typecheck(``bool'',x)
   then
     (match x with
       | Bool(y) -> Bool(not y) )
   else failwith (``type error'');;
```

7.8 Semantica denotazionale

Definizione della semantica denotazionale del frammento funzionale puro:

```
let rec sem (e:exp) (r:eval env) =
   match e with
     | Eint(n) -> Int(n)
     | Ebool(b) -> Bool(b)
     | Den(i) -> applyenv(r,i)
     | Iszero(a) -> iszero((sem a r) )
     | Eq(a,b) -> equ((sem a r) ,(sem b r) )
     | Prod(a,b) -> mult ( (sem a r), (sem b r))
     | Sum(a,b) -> plus ( (sem a r), (sem b r))
     | Diff(a,b) -> diff ( (sem a r), (sem b r))
     | Minus(a) -> minus( (sem a r))
     | And(a,b) -> et ( (sem a r), (sem b r))
     | Or(a,b) -> vel ( (sem a r), (sem b r))
     | Not(a) -> non( (sem a r))
     | Ifthenelse(a,b,c) ->
       let g = sem a r in
         if typecheck(bool,g) then
           (if g = Bool(true)
           then sem b r
           else sem c r)
         else failwith (``nonboolean guard'') ;;

val sem :  exp -> eval Funenv.env -> eval = <fun>
```

E' importante notare che `And` ed `Or` sono interpretati come funzioni *strette*, mentre

il condizionale `ifthenelse` è, ovviamente, una funzione *non stretta*.
Un'altra caratteristica interessante riguarda il fatto che la semantica denotazionale, come funzione di ordine superiore, può essere applicata anche al solo programma (senza specificare un ambiente). Ovvero:

```
val sem :  exp -> eval Funenv.env -> eval = <fun>
```

```
#sem (Prod(Sum(Eint 5,Eint 3),Diff(Eint 5,Eint 1)))(emptyenv Unbound);;
-:  eval = Int 32
#sem (Prod(Sum(Eint 5,Eint 3),Diff(Eint 5,Eint 1)));;
-:  eval Funenv.env -> eval = <fun>
#sem (Prod(Sum(Den ‘‘x’’,Eint 3),Diff(Den ‘‘y’’,Eint 1)));;
-:  eval Funenv.env -> eval = <fun>
```

7.9 Semantica operazionale

La semantica operazionale mantiene, in questo caso, gli stessi domini semantici della semantica denotazionale (non ci sono funzioni, che altrimenti andrebbero eliminate), e le stesse operazioni primitive. Cambia invece la funzione di valutazione semantica:

da

```
val sem :  exp -> eval Funenv.env -> eval = <fun>
```

a

```
val sem :  exp * eval Funenv.env -> eval = <fun>
```

Notiamo come nel caso denotazionale la funzione corrisponda ad una sorta di traduttore (compilazione), mentre nel caso operazionale si ha a che fare con un processo di interpretazione del programma.
Vediamo la semantica operazionale del frammento di linguaggio che stiamo considerando:

```
let rec sem (e:exp) (r:eval env) =
   match e with
     | Eint(n) -> Int(n)
     | Ebool(b) -> Bool(b)
     | Den(i) -> applyenv(r,i)
     | Iszero(a) -> iszero((sem a r) )
     | Eq(a,b) -> equ((sem a r) ,(sem b r) )
     | Prod(a,b) -> mult ( (sem a r), (sem b r))
     | Sum(a,b) -> plus ( (sem a r), (sem b r))
     | Diff(a,b) -> diff ( (sem a r), (sem b r))
     | Minus(a) -> minus( (sem a r))
     | And(a,b) -> et ( (sem a r), (sem b r))
```

```
    | Or(a,b) -> vel ( (sem a r), (sem b r))
    | Not(a) -> non( (sem a r))
    | Ifthenelse(a,b,c) ->
      let g = sem a r in
        if typecheck(bool,g) then
          (if g = Bool(true)
          then sem b r
          else sem c r)
        else failwith (``nonboolean guard'') ;;

val sem :  exp -> eval Funenv.env -> eval = <fun>
```

Come già detto, quindi, la semantica operazionale è in effetti un **interprete**, definito ricorsivamente sfruttando la ricorsione del metalinguaggio (linguaggio di implementazione)[16]. E' possibile comunque **eliminare la ricorsione** dall'interprete, attraverso l'iterazione, ottenendo una versione dell'interprete a più basso livello e più vicina ad una implementazione "reale".

7.10 Eliminare la ricorsione

La ricorsione può essere rimpiazzata con l'iterazione. Per farlo è necessario ricorrere alle pile, a meno di definizioni ricorsive con una struttura particolarmente semplice (tail recursion[17]).
Nel nostro linguaggio la struttura ricorsiva di *sem* ripropone quella del dominio sintattico delle espressioni (composizionalità). Per eliminare questo tipo di ricorsione dobbiamo ricorrere all'uso di pile, affidandoci alle seguenti considerazioni:

- La funzione ricorsiva ha due argomenti: *espressione* ed *ambiente*.
- La funzione ricorsiva calcola un *eval*.
- L'ambiente non viene mai modificato nelle chiamate ricorsive.
- L'informazione che deve essere memorizzata per simulare la ricorsione è dunque:
  - La (sotto)-espressione.
  - Il valore calcolato per la (sotto)-espressione.

Alla luce di quanto detto, utilizziamo le seguenti pile per simulare la ricorsione:

- `continuation`: una pila di espressioni etichettate che, ad ogni istante, contiene l'informazione su ciò che deve essere ancora valutato. Le etichette servono ad indicare se l'espressione è già stata valutata o meno.

[16] Questo implica il fatto che non possiamo, ad esempio, implementarlo in linguaggio macchina.

[17] In computer science, tail recursion (or tail-end recursion) is a special case of recursion in which the last operation of the function is a recursive call. Such recursions can be easily transformed to iterations. Replacing recursion with iteration, either manually or automatically, can drastically decrease the amount of stack space used and improve efficiency. This technique is commonly used with functional programming languages, where the declarative approach and explicit handling of state promote the use of recursive functions that would otherwise rapidly fill the call stack.

- `tempstack`: una pila di *eval* che, ad ogni istante, contiene i valori temporanei.

7.11 L'interprete iterativo

Innanzi tutto definiamo le strutture dati utilizzate dall'interprete.

```
type labeledconstruct =
   | Expr1 of exp
   | Expr2 of exp

let (continuation:  labeledconstruct stack) =
   emptystack(cframesize,Expr1(Eint(0)))

let (tempstack:  eval stack) =
   emptystack(tframesize,Unbound)
```

Expr1 ed *Expr2* sono le etichette che usiamo per indicare se un'espressione è da valutare o è stata valutata rispettivamente.
E l'interprete iterativo vero e proprio:

```
let sem ((e:exp), (rho:eval env)) =
   push(Expr1(e), continuation);
   while not(empty(continuation)) do (match top(continuation) with
     |Expr1(x) ->
       (pop(continuation);
       push(Expr2(x),continuation); (*1*)
       (match x with
         | Iszero(a) -> push(Expr1(a),continuation)
         | Eq(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
         | Prod(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
         | Sum(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
         | Diff(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
         | Minus(a) -> push(Expr1(a),continuation)
         | And(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
         | Or(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
         | Not(a) -> push(Expr1(a),continuation)
         | Ifthenelse(a,b,c) -> push(Expr1(a),continuation) (*2*)
         | _ -> ())) (*3*)
     |Expr2(x) ->
       (pop(continuation);
       (match x with
         | Eint(n) -> push(Int(n),tempstack)
         | Ebool(b) -> push(Bool(b),tempstack)
```

```
| Den(i) -> push(applyenv(rho,i),tempstack)
| Iszero(a) ->
  let arg=top(tempstack) in
    pop(tempstack);
  push(iszero(arg),tempstack)
| Eq(a,b) ->
  let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
      pop(tempstack);
      push(equ(firstarg,sndarg),tempstack)
| Prod(a,b) ->
  let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
      pop(tempstack);
      push(mult(firstarg,sndarg),tempstack)
| Sum(a,b) ->
  let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
      pop(tempstack);
      push(plus(firstarg,sndarg),tempstack)
| Diff(a,b) ->
  let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
      pop(tempstack);
      push(diff(firstarg,sndarg),tempstack)
| Minus(a) ->
  let arg=top(tempstack) in
    pop(tempstack);
    push(minus(arg),tempstack)
| And(a,b) ->
  let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
      pop(tempstack);
      push(et(firstarg,sndarg),tempstack)
| Or(a,b) ->
  let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
      pop(tempstack);
      push(vel(firstarg,sndarg),tempstack)
| Not(a) ->
  let arg=top(tempstack) in
    pop(tempstack);
    push(non(arg),tempstack)
```

```
        | Ifthenelse(a,b,c) ->
          let arg=top(tempstack) in
            pop(tempstack);
            if typecheck(bool,arg) then
              (if arg = Bool(true)
              then push(Expr1(b),continuation) (*4*)
              else push(Expr1(c),continuation))
            else failwith (type error))))
   done;
let valore= top(tempstack) in
pop(tempstack); valore;;
```

Commenti al codice:

1. Si cambia l'etichetta da *Expr1* a *Expr2*.
2. Si valuta la condizione.
3. In questo caso rientrano costanti, variabili denotate, booleani.
4. In base al risultato della valutazione della guardia si mette la giusta espressione sulla pila sintattica.

7.12 Effetti laterali, comandi ed espressioni pure

Prima di introdurre il frammento imperativo, consideriamo il ruolo ed il rapporto tra comandi ed espressioni nel linguaggio.
Assumiamo che nel linguaggio imperativo continuino ad esistere le espressioni, e che siano ben distinte dai comandi poiché la loro semantica non modifica in alcun modo lo store (non produce effetti laterali) e restituisce invece un valore (*eval*).
Una simile distinzione semantica, che noi forzeremo nel linguaggio didattico, non è ad esempio verificata dal linguaggio C, in cui ogni costrutto può modificare lo stato e restituire valori, ed è generalmente difficile da mantenere se si permette che i comandi possano occorrere all'interno delle espressioni (Java, ML).

7.13 Frammento imperativo: sintassi

Vediamo la sintassi del frammento imperativo con i soli comandi:

```
type ide = string
type exp =
   | Eint of int
   | Ebool of bool
   | Den of ide
   | Prod of exp * exp
   | Sum of exp * exp
   | Diff of exp * exp
   | Eq of exp * exp
   | Minus of exp
   | Iszero of exp
   | Or of exp * exp
   | And of exp * exp
```

```
   | Not of exp
   | Ifthenelse of exp * exp * exp
   | Val of exp

and com =
   | Assign of exp * exp
   | Cifthenelse of exp * com list * com list
   | While of exp * com list;;
```

La sintassi del frammento imperativo non è diversa da quella usata per il frammento funzionale puro, ad eccezione del nuovo dominio sintattico `com` che definisce la sintassi dei comandi, e del costrutto `Val`.
Utilizziamo le liste per rappresentare le sequenze di comandi (`Cifthenelse of exp * com list...`).

7.14 Domini semantici

Nel caso del linguaggio imperativo abbiamo bisogno, oltre che dell'ambiente, anche della memoria. Ai domini semantici dei valori si aggiungono le locazioni, che decidiamo non essere né esprimibili ne memorizzabili.
Abbiamo quindi tre distinti domini semantici: *eval*, *dval*, *mval*.
Definiremo operazioni di conversione tra i tre domini, ed una funziona di valutazione semantica (*semden*) che calcola un *dval* in vece che un *eval*.

7.15 Il dominio store

Il dominio *store* è simile all'ambiente (polimorfo).

```
module type STORE =
sig
   type 't store
   type loc
   val emptystore :  't -> 't store
   val allocate :  't store * 't -> loc * 't store
   val update :  't store * loc * 't -> 't store
   val applystore :  't store * loc -> 't
end;;

module Funstore:STORE =
struct
   type loc = int
   type 't store = loc -> 't
   let (newloc,initloc) =
     let count = ref(-1) in (*1*)
       (fun () -> count := !count +1;
         !count),
     (fun () -> count := -1)
   let emptystore(x) = initloc(); function y -> x (*2*)
   let applystore(x,y) = x y
   let allocate((r:  'a store) , (e:'a)) = let l = newloc() in
```

```
    (l, function lu -> if lu = l then e else applystore(r,lu))
  let update((r:  'a store) , (l:loc), (e:'a)) =
    function lu -> if lu = l then e else applystore(r,lu)
end;;
```

Commenti al codice:

1. `count` è una variabile "statica" condivisa tra `newloc` ed `initloc`.
2. `x` è il bottom.

7.16 Domini dei valori per il frammento imperativo

Definiamo i nuovi domini dei valori e le operazione di "conversione" da un tipo all'altro:

```
exception Nonstorable
exception Nonexpressible

type eval =
   | Int of int
   | Bool of bool
   | Novalue

type dval =
   | Dint of int
   | Dbool of bool
   | Unbound
   | Dloc of loc

type mval =
   | Mint of int
   | Mbool of bool
   | Undefined

let evaltomval e = match e with
   | Int n -> Mint n
   | Bool n -> Mbool n
   | _ -> raise Nonstorable

let mvaltoeval m = match m with
   | Mint n -> Int n
   | Mbool n -> Bool n
   | _ -> Novalue

let evaltodval e = match e with
   | Int n -> Dint n
   | Bool n -> Dbool n
   | Novalue -> Unbound

let dvaltoeval e = match e with
```

```
   | Dint n -> Int n
   | Dbool n -> Bool n
   | Dloc n -> raise Nonexpressible
   | Unbound -> Novalue
```

7.17 Semantica denotazionale

Ecco la semantica denotazionale completa per il frammento imperativo:

```
let rec sem (e:exp) (r:dval env) (s:  mval store) =
   match e with
     | Eint(n) -> Int(n)
     | Ebool(b) -> Bool(b)
     | Den(i) -> dvaltoeval(applyenv(r,i))
     | Iszero(a) -> iszero((sem a r s) )
     | Eq(a,b) -> equ((sem a r s) ,(sem b r s) )
     | Prod(a,b) -> mult ( (sem a r s), (sem b r s))
     | Sum(a,b) -> plus ( (sem a r s), (sem b r s))
     | Diff(a,b) -> diff ( (sem a r s), (sem b r s))
     | Minus(a) -> minus( (sem a r s))
     | And(a,b) -> et ( (sem a r s), (sem b r s))
     | Or(a,b) -> vel ( (sem a r s), (sem b r s))
     | Not(a) -> non( (sem a r s))
     | Ifthenelse(a,b,c) ->
       let g = sem a r s in
         if typecheck(``bool'',g) then
           (if g = Bool(true)
           then sem b r s
           else sem c r s)
         else failwith (``nonboolean guard'')
     | Val(e) -> match semden e r s with
       | Dloc n -> mvaltoeval(applystore(s, n))
     | _ -> failwith(``not a variable'')

and semden (e:exp) (r:dval env) (s:  mval store) = match e with
   | Den(i) -> applyenv(r,i)
   | _ -> evaltodval(sem e r s)

and semc (c:  com) (r:dval env) (s:  mval store) = match c with
   | Assign(e1, e2) ->
     (match semden e1 r s with
       | Dloc(n) -> update(s, n, evaltomval(sem e2 r s))
       | _ -> failwith (``wrong location in assignment''))
   | Cifthenelse(e, cl1, cl2) -> let g = sem e r s in
     if typecheck(``bool'',g) then
       (if g = Bool(true)
       then semcl cl1 r s
       else semcl cl2 r s)
```

```
    else failwith (nonboolean guard'')
  | While(e, cl) ->
    let functional ((fi:  mval store -> mval store)) =
      function sigma ->
        let g = sem e r sigma in
          if typecheck(``bool'',g) then
            (if g = Bool(true)
            then fi(semcl cl r sigma)
            else sigma)
          else failwith (``nonboolean guard'') in
    let rec ssfix = function x -> functional ssfix x in
      ssfix(s)

and semcl cl r s = match cl with
   | [] -> s
   | c::cl1 -> semcl cl1 r (semc c r s) ;;
```

Le funzioni definite hanno il seguente tipo:

```
val sem :  exp -> dval Funenv.env -> mval Funstore.store -> eval = <fun>

val semden :  exp -> dval Funenv.env -> mval Funstore.store -> dval =
<fun>

val semc :  com -> dval Funenv.env -> mval Funstore.store -> mval Funstore.store
= <fun>

val semcl :  com list -> dval Funenv.env -> mval Funstore.store -> mval
Funstore.store = <fun>
```

7.18 Semantica dell'assegnamento

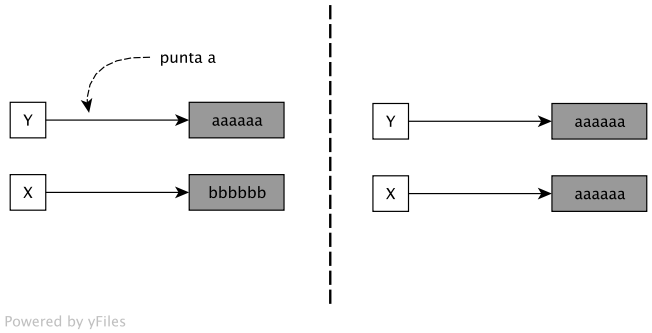

Figura 3: Assegnamento `X:=Y`

L'assegnamento è un'operazione che coinvolge sia la memoria che l'ambiente,

copiando un valore nella memoria, ma senza modificare l'ambiente. Quando i valori coinvolti nell'assegnamento sono strutture dati modificabili (s-espressioni in LISP, arrays in ML, oggetti in Java) il valore è in realtà un puntatore. L'effetto dell'assegamento in questi casi è che i valori sono in effetti condivisi tra X ed Y e che le modifiche dell'uno si ripercuotono sull'altro (l'assegnamento crea **aliasing**).

7.19 Semantica operazionale

Semantica operazionale completa per il frammento di linguaggio imperativo:

```
let rec sem ((e:exp), (r:dval env), (s:  mval store)) =
   match e with
     | Eint(n) -> Int(n)
     | Ebool(b) -> Bool(b)
     | Den(i) -> dvaltoeval(applyenv(r,i))
     | Iszero(a) -> iszero(sem(a, r, s))
     | Eq(a,b) -> equ(sem(a, r, s), sem(b, r, s))
     | Prod(a,b) -> mult (sem(a, r, s), sem(b, r, s))
     | Sum(a,b) -> plus (sem(a, r, s), sem(b, r, s))
     | Diff(a,b) -> diff (sem(a, r, s), sem(b, r, s))
     | Minus(a) -> minus(sem(a, r, s))
     | And(a,b) -> et (sem(a, r, s), sem(b, r, s))
     | Or(a,b) -> vel (sem(a, r, s), sem(b, r, s))
     | Not(a) -> non(sem(a, r, s))
     | Ifthenelse(a,b,c) ->
       let g = sem(a, r, s) in
         if typecheck(``bool'',g) then
           (if g = Bool(true)
           then sem(b, r, s)
           else sem(c, r, s))
         else failwith (``nonboolean guard'')
     | Val(e) -> match semden(e, r, s) with
     | Dloc n -> mvaltoeval(applystore(s, n))
     | _ -> failwith(``not a variable'')

and semden ((e:exp), (r:dval env), (s:  mval store)) = match e with
   | Den(i) -> applyenv(r,i)
   | _ -> evaltodval(sem(e, r, s))

and semc((c:  com), (r:dval env), (s:  mval store)) = match c with
   | Assign(e1, e2) ->
     (match semden(e1, r, s) with
       | Dloc(n) -> update(s, n, evaltomval(sem(e2, r, s)))
       | _ -> failwith (``wrong location in assignment''))
   | Cifthenelse(e, cl1, cl2) -> let g = sem(e, r, s) in
     if typecheck(``bool'',g) then
       (if g = Bool(true)
       then semcl(cl1, r, s)
       else semcl (cl2, r, s))
```

```
     else failwith (``nonboolean guard'')
   | While(e, cl) -> let g = sem(e, r, s) in (*1*)
     if typecheck(``bool'',g) then
       (if g = Bool(true)
       then semcl((cl @ [While(e, cl)]), r, s)
       else s)
     else failwith (``nonboolean guard'')

and semcl(cl, r, s) = match cl with
   | [] -> s
   | c::cl1 -> semcl(cl1, r, semc(c, r, s));;
```

Le funzioni definite hanno il seguente tipo:

```
val sem :  exp * dval Funenv.env * mval Funstore.store -> eval = <fun>

val semden :  exp * dval Funenv.env * mval Funstore.store -> dval = <fun>

val semc :  com * dval Funenv.env * mval Funstore.store -> mval Funstore.store
= <fun>

val semcl :  com list * dval Funenv.env * mval Funstore.store -> mval
Funstore.store = <fun>
```

Commenti al codice:

1. Eliminato il punto fisso, utilizzando la sintassi, non composizionale.

7.20 Eliminare la ricorsione

Per quanto riguarda le espressioni è necessario considerare il caso in cui il valore è un *dval*: aggiungiamo una pila di valori denotabili temporanei, e nuove etichette per le espressioni.
Per i comandi la ricorsione può essere sostituita con l'iterazione senza dover ricorrere a pile aggiuntive.
Il dominio dei comandi è "quasi" tail recursive: poiché non è mai necessario valutare i due rami del condizionale, che sintatticamente non è tail recursive, è possibile utilizzare la struttura sintattica (lista dei comandi) per mantenere l'informazione su ciò che deve essere ancora valutare. Per far questo è sufficiente un'unica cella, che possiamo "integrare" nella pila di espressioni etichettate.
Il valore restituito dalla funzione di valutazione semantica dei comandi è uno *store*, che può essere gestito come l'aggiornamento di una "variabile globale" di tipo *store*.

7.21 L'interprete iterativo

Definiamo le strutture dati usate dall'interprete iterativo:

```
type labeledconstruct =
   | Expr1 of exp
   | Expr2 of exp
```

```
   | Exprd1 of exp
   | Exprd2 of exp
   | Com1 of com
   | Com2 of com
   | Coml of labeledconstruct list

let (continuation:  labeledconstruct stack) =
   emptystack(cframesize,Expr1(Eint(0)))

let (tempstack:  eval stack) =
   emptystack(tframesize,Novalue)

let (tempdstack:  dval stack) =
   emptystack(tdframesize,Unbound)

let globalstore = ref(emptystore(Undefined))

let labelcom (dl:  com list) = let dlr = ref(dl) in
let ldlr = ref([]) in
   while not (!dlr = []) do
     let i = List.hd !dlr in
       ldlr := !ldlr @ [Com1(i)];
       dlr := List.tl !dlr
   done;
   Coml(!ldlr);;
```

In questo caso utilizziamo 7 diverse etichette per indicare: espressioni da valutare, espressioni valutate, espressioni da valutare che restituiscono un *dval*, espressioni valutate che restituiscono un *dval*, comandi da valutare, comandi valutati, lista di comandi.
Ecco l'interprete:

```
let itsem(rho) =
   (match top(continuation) with
    |Expr1(x) ->
       (pop(continuation);
       push(Expr2(x),continuation);
       (match x with
          | Iszero(a) -> push(Expr1(a),continuation)
          | Eq(a,b) -> push(Expr1(a),continuation);
             push(Expr1(b),continuation)
          | Prod(a,b) -> push(Expr1(a),continuation);
             push(Expr1(b),continuation)
          | Sum(a,b) -> push(Expr1(a),continuation);
             push(Expr1(b),continuation)
          | Diff(a,b) -> push(Expr1(a),continuation);
             push(Expr1(b),continuation)
          | Minus(a) -> push(Expr1(a),continuation)
          | And(a,b) -> push(Expr1(a),continuation);
```

```
          push(Expr1(b),continuation)
       | Or(a,b) -> push(Expr1(a),continuation);
          push(Expr1(b),continuation)
       | Not(a) -> push(Expr1(a),continuation)
       | Ifthenelse(a,b,c) -> push(Expr1(a),continuation)
       | Val(e) -> push(Exprd1(e),continuation) (*1*)
       | _ -> ()))
  |Expr2(x) ->
     (pop(continuation);
     (match x with
       | Eint(n) -> push(Int(n),tempstack)
       | Ebool(b) -> push(Bool(b),tempstack)
       | Den(i) ->
          push(dvaltoeval(applyenv(rho,i)),tempstack)
       | Iszero(a) ->
          let arg=top(tempstack) in
           pop(tempstack);
           push(iszero(arg),tempstack)
       | Eq(a,b) ->
          let firstarg=top(tempstack) in
           pop(tempstack);
           let sndarg=top(tempstack) in
            pop(tempstack);
            push(equ(firstarg,sndarg),tempstack)
       | Prod(a,b) ->
          let firstarg=top(tempstack) in
           pop(tempstack);
           let sndarg=top(tempstack) in
            pop(tempstack);
            push(mult(firstarg,sndarg),tempstack)
       | Sum(a,b) ->
          let firstarg=top(tempstack) in
           pop(tempstack);
           let sndarg=top(tempstack) in
            pop(tempstack);
            push(plus(firstarg,sndarg),tempstack)
       | Diff(a,b) ->
          let firstarg=top(tempstack) in
           pop(tempstack);
           let sndarg=top(tempstack) in
            pop(tempstack);
            push(diff(firstarg,sndarg),tempstack)
       | Minus(a) ->
          let arg=top(tempstack) in
           pop(tempstack);
           push(minus(arg),tempstack)
       | And(a,b) ->
          let firstarg=top(tempstack) in
           pop(tempstack);
```

```
              let sndarg=top(tempstack) in
                pop(tempstack);
                push(et(firstarg,sndarg),tempstack)
          | Or(a,b) ->
             let firstarg=top(tempstack) in
              pop(tempstack);
              let sndarg=top(tempstack) in
                pop(tempstack);
                push(vel(firstarg,sndarg),tempstack)
          | Not(a) ->
             let arg=top(tempstack) in
              pop(tempstack);
              push(non(arg),tempstack)
          | Ifthenelse(a,b,c) ->
             let arg=top(tempstack) in
              pop(tempstack);
              if typecheck(``bool'',arg) then
                (if arg = Bool(true)
                then push(Expr1(b),continuation)
                else push(Expr1(c),continuation))
              else failwith (``type error'')
          | Val(e) -> let v = top(tempdstack) in
             pop(tempdstack);
             (match v with
              | Dloc n ->
                 push(
                  mvaltoeval(applystore(!globalstore, n)),
                  tempstack)
              | _ -> failwith(``not a variable'')))))
    | _ -> failwith(``no more cases for semexpr''))

let itsemden(rho) =
   (match top(continuation) with
    |Exprd1(x) ->
       (pop(continuation); push(Exprd2(x),continuation);
       match x with
         | Den i -> ()
         | _ -> push(Expr2(x), continuation))
    |Exprd2(x) ->
       (pop(continuation); match x with
         | Den i -> push(applyenv(rho,i), tempdstack)
         | _ -> let arg = top(tempstack) in pop(tempstack);
            push(evaltodval(arg), tempdstack))
    | _ -> failwith(``No more cases for semden'') )

let itsemcl((rho:  dval env)) =
   let cl =
    (match top(continuation) with
     | Coml(dl1) -> dl1
```

```
  | _ -> failwith(``impossible in semdecl'')) in
if cl = [] then pop(continuation) else
  (let currc = List.hd cl in
  let newcl = List.tl cl in
   pop(continuation); push(Coml(newcl),continuation);
     (match currc with
      | Com1(Assign(e1, e2)) -> pop(continuation);
         push(Coml(Com2(Assign(e1, e2))::newcl),continuation);
         push(Exprd1(e1), continuation);
         push(Expr1(e2), continuation)
      | Com2(Assign(e1, e2)) ->
         let arg2 = evaltomval(top(tempstack)) in
          pop(tempstack);
          let arg1 = top(tempdstack) in
           pop(tempdstack);
           (match arg1 with
              | Dloc(n) ->
                globalstore := update(!globalstore, n, arg2)
              | _ -> failwith (``wrong location in assignment''))
      | Com1(While(e, cl)) ->
         pop(continuation);
         push(Coml(Com2(While(e, cl))::newcl),continuation);
         push(Expr1(e), continuation)
      | Com2(While(e, cl)) ->
         let g = top(tempstack) in
          pop(tempstack);
          if typecheck(``bool'',g) then
            (if g = Bool(true) then
               (let old = newcl in
               let newl =
                 (match labelcom cl with
                   | Coml newl1 -> newl1
                   | _ -> failwith(``impossible in while'')) in
               let nuovo
                  = Coml(newl @ [Com1(While(e, cl))] @ old) in
                  pop(continuation); push(nuovo,continuation))
             else ())
           else failwith (``nonboolean guard'')
      | Com1(Cifthenelse(e, cl1, cl2)) ->
         pop(continuation);
         push(Coml(Com2(Cifthenelse(e, cl1, cl2))::newcl),continuation);
         push(Expr1(e), continuation)
      | Com2(Cifthenelse(e, cl1, cl2)) ->
         let g = top(tempstack) in
          pop(tempstack);
          if typecheck(``bool'',g) then
            (let temp = if g = Bool(true) then
              labelcom (cl1) else labelcom (cl2) in
            let newl =
```

```
                (match temp with
                   | Coml newl1 -> newl1
                   | _ -> failwith(``impossible in cifthenelse'')) in
              let nuovo = Coml(newl @ newcl) in
                 pop(continuation); push(nuovo,continuation))
            else failwith (``nonboolean guard'')
        | _ -> failwith(``no more sensible cases in commands'') ))

let initstate() =
  svuota(continuation); svuota(tempstack)

let loop (rho) =
  while not(empty(continuation)) do
   let currconstr = top(continuation) in
    (match currconstr with
     | Expr1(e) -> itsem(rho)
     | Expr2(e) -> itsem(rho)
     | Exprd1(e) -> itsemden(rho)
     | Exprd2(e) -> itsemden(rho)
     | Coml(cl) -> itsemcl(rho)
     | _ -> failwith(``non legal construct in loop''))
  done

let sem (e,(r:  dval env), (s:  mval store)) =
  initstate();
  globalstore := s;
  push(Expr1(e), continuation);
  loop(r);
  let valore= top(tempstack) in
   pop(tempstack);
   valore

let semden (e,(r:  dval env), (s:  mval store)) =
  initstate();
  globalstore := s;
  push(Exprd1(e), continuation);
  loop(r);
  let valore= top(tempdstack) in
   pop(tempdstack);
   valore

let semcl (cl,(r:  dval env), (s:  mval store)) =
  initstate();
  globalstore := s;
  push(labelcom(cl), continuation);
  loop(r);
  !globalstore
```

Commenti al codice:

1. Questo è il caso in cui vogliamo che *sem* restituisca un *dval* anzichè un *eval*.

8 Blocchi ed ambiente in linguaggi funzionali ed imperativi

Contenuti del capitolo:

- Nomi ed associazioni: l'ambiente.
- Operazioni sulle associazioni (creazione, distruzione, disattivazione, riattivazione).
- Struttura a blocchi nei linguaggi funzionali (semantiche del *let*).
- Struttura a blocchi nei linguaggi imperativi (semantiche delle dichiarazioni e memoria locale).
- Digressione su meccanismi alternativi per la gestione statica dell'ambiente locale.

8.1 Nomi ed ambiente

Tutti i linguaggi ad alto livello utilizzano nomi per denotare entità di vario tipo: costanti, nomi delle operazioni primitive, identificatori di costanti, variabili e sottoprogrammi, parametri. formali). L'associazione tra nomi ed oggetti denotati, quando non è creata dall'implementazione del linguaggio (costanti, operazioni), è l'**ambiente**, che è ciò che maggiormente distingue i linguaggi macchina ed i linguaggi ad alto livello.
In un linguaggio *può* esistere un **ambiente globale**, che contenga associazioni comuni a diverse unità di programmi create dal programma principale ed esportate da un modulo, ed esiste *sempre* un **ambiente locale** con associazioni create attraverso due possibili meccanismi:

- Dichiarazione all'ingresso in un blocco.
- Passaggio di parametri in occasione di una chiamata a sottoprogramma.

Nel seguito del capitolo ci occuperemo unicamente delle associazioni locali create tramite dichiarazioni.

8.2 Operazioni sulle associazioni: ambiente locale dinamico e statico

Nel caso di ambiente locale **dinamico** abbiamo:

- *Creazione* di una associazione, tra nome ed oggetto denotato, all'ingresso di un blocco e suo *inserimento* nell'ambiente.
- *Distruzione* di una associazione all'uscita dal blocco in cui è stata creata, con la sua *rimozione* dall'ambiente.
- L'associazione è utilizzabile solo **all'interno** del blocco. Rientrando nello stesso blocco rieseguiamo le dichiarazioni e creiamo nuove associazioni.

Nel caso di ambiente locale **statico** abbiamo:

- *Attivazione* di una associazione all'ingresso del blocco, e sua *riattivazione* nell'ambiente.

- *Disattivazione* di una associazione all'uscita del blocco in cui è stata creata, con la sua *disattivazione* nell'ambiente.
- L'associazione è utilizzabile **quando è attiva**. Rientrando in uno stesso blocco non rieseguiamo le dichiarazioni, ma ci limitiamo a riattivare le associazioni precedenti.
- Le dichiarazioni vengono eseguite prima dell'inizio dell'esecuzione (compilatore, linker, loader) o alla prima esecuzione del blocco (o simili).

Tutti i linguaggi moderni utilizzano l'**ambiente locale dinamico**, con la possibilità, in alcuni casi, di trattare alcune associazioni in modo statico.
Nel linguaggio didattico assumiamo di avere l'ambiente locale dinamico, anche se vedremo anche cosa succede in caso di ambiente statico.

8.3 Dichiarazioni nei linguaggi imperativi: la memoria locale

Nei linguaggi imperativi l'esecuzione di una dichiarazione, effettuata un'unica volta o tante volte quante sono le esecuzioni del blocco, può provocare, oltre alla creazione dell'associazione, anche l'allocazione di **memoria locale**. In questo caso la memoria allocata segue l'evoluzione dell'ambiente locale, risultando anch'essa statica o dinamica. In caso di ambiente statico questo vuol dire che la memoria locale ad un blocco viene preservata tra due diverse esecuzioni del blocco, costituendo una sorta di stato interno al blocco stesso.

8.4 Il costrutto `let` nel linguaggio funzionale: sintassi

La nuova sintassi astratta del linguaggio funzionale, con l'aggiunta del costrutto `Let`:

```
type ide = string
type exp =
   | Eint of int
   | Ebool of bool
   | Den of ide
   | Prod of exp * exp
   | Sum of exp * exp
   | Diff of exp * exp
   | Eq of exp * exp
   | Minus of exp
   | Iszero of exp
   | Or of exp * exp
   | And of exp * exp
   | Not of exp
   | Ifthenelse of exp * exp * exp
   | Let of ide * exp * exp
```

L'utilizzo del costrutto *let* permette di cambiare l'ambiente in punti arbitrari all'interno di un'espressione, in modo che l'ambiente "nuovo" valga soltanto per la valutazione del "corpo" del **blocco**.
I blocchi così definiti possono essere annidati e l'ambiente locale di un blocco può

essere (in parte) visibile ed utilizzabile nel blocco più interno, come ambiente **non locale**.
In seguito vedremo come la presenza di blocchi porti ad una semplice gestione della memoria locale e come si sposi felicemente con la regola di scoping statico per la gestione dell'ambiente non locale.

8.5 Semantica denotazionale

La semantica denotazionale del linguaggio funzionale, con l'aggiunta del costrutto `Let`:

```
let rec sem (e:exp) (r:eval env) =
   match e with
     | Eint(n) -> Int(n)
     | Ebool(b) -> Bool(b)
     | Den(i) -> applyenv(r,i)
     | Iszero(a) -> iszero((sem a r) )
     | Eq(a,b) -> equ((sem a r) ,(sem b r) )
     | Prod(a,b) -> mult ( (sem a r), (sem b r))
     | Sum(a,b) -> plus ( (sem a r), (sem b r))
     | Diff(a,b) -> diff ( (sem a r), (sem b r))
     | Minus(a) -> minus( (sem a r))
     | And(a,b) -> et ( (sem a r), (sem b r))
     | Or(a,b) -> vel ( (sem a r), (sem b r))
     | Not(a) -> non( (sem a r))
     | Ifthenelse(a,b,c) ->
       let g = sem a r in
         if typecheck(bool,g) then
           (if g = Bool(true)
           then sem b r
           else sem c r)
         else failwith (``nonboolean guard'')
     | Let(i,e1,e2) -> sem e2 (bind (r,i,sem e1 r));;

val sem :  exp -> eval Funenv.env -> eval = <fun>
```

Commenti alla semantica del `Let`:

- L'espressione *e2*, che rappresenta il corpo del blocco, viene valutata nell'ambiente "esterno" r esteso con l'associazione tra il nome i ed il valore di *e1*.
- Le associazioni per nomi diversi da i sono comunque visibili durante la valutazione di *e2*, perchè presenti in r.

8.6 Semantica operazionale

La semantica operazionale del frammento funzionale con l'aggiunta dei blocchi:

```
let rec sem ((e:exp), (r:eval env)) =
  match e with
```

```
    | Eint(n) -> Int(n)
    | Ebool(b) -> Bool(b)
    | Den(i) -> applyenv(r,i)
    | Iszero(a) -> iszero(sem(a, r))
    | Eq(a,b) -> equ(sem(a, r), sem(b, r))
    | Prod(a,b) -> mult (sem(a, r), sem(b, r))
    | Sum(a,b) -> plus (sem(a, r), sem(b, r))
    | Diff(a,b) -> diff (sem(a, r), sem(b, r))
    | Minus(a) -> minus(sem(a, r))
    | And(a,b) -> et (sem(a, r), sem(b, r))
    | Or(a,b) -> vel (sem(a, r), sem(b, r))
    | Not(a) -> non(sem(a, r))
    | Ifthenelse(a,b,c) ->
       let g = sem(a, r) in
         if typecheck(``bool'',g) then
           (if g = Bool(true)
           then sem(b, r)
           else sem(c, r))
         else failwith (``nonboolean guard'')
    | Let(i,e1,e2) -> sem(e2, bind (r ,i, sem(e1, r))

val sem :  exp -> eval Funenv.env -> eval = <fun>
```

8.7 Eliminare la ricorsione

La funzione ricorsiva *sem* ha due argomenti (espressione ed ambiente) e calcola un risultato (un *eval*). In presenza del *let* l'ambiente viene modificato all'interno di alcune chiamate ricorsive di *sem* per poi essere "ripristinato" all'uscita dal blocco (ritorno della chiamata ricorsiva). Tale situazione può essere gestita attraverso una pila di ambienti su cui effettuare una *push* all'ingresso del blocco ed una *pop* all'uscita. Per poter mantenere la corretta interazione tra la pila degli ambienti e le pile già introdotte (pila sintattica e pila dei temporanei) è però necessario che ogni ambiente possa riferirsi ad una propria coppia di pile `continuation` e `tempstack`. Utilizzeremo dunque tre pile:

- `envstack`: pila di ambienti.
- `cstack`: pila di pile di espressioni etichettate.
- `tempvalstack`: pila di pile di *eval*.

Con l'ambiente dinamico ogni volta che si entra in un blocco si crea una nuova *attivazione*[18], che corrisponde, nell'implementazione iterativa, alla creazione di un nuovo **record di attivazione** che contiene tutte le informazioni caratteristiche della attivazione (ambiente, espressione da valutare e pila necessaria alla sua valutazione, pila dei valori temporanei). Mentre nelle implementazioni standard esiste un'unica *pila dei record di attivazione*, sulla quale un nuovo record viene inserito

[18] Il termine *attivazione* si riferisce solitamente ai sottoprogrammi. In effetti i blocchi non sono altro che un caso particolare di sottoprogramma, senza parametri e senza distinzione tra λ-astrazione ed applicazione.

all'ingresso in un blocco e rimosso all'uscita, noi utilizzeremo tre distinte pile gestite in modo "parallelo".

8.8 L'interprete iterativo

Definiamo le strutture dati utilizzate dall'interprete:

```
type labeledconstruct =
   | Expr1 of exp
   | Expr2 of exp

let (cstack:  labeledconstruct stack stack) =
   emptystack(stacksize,emptystack(1,Expr1(Eint(0))))

let (tempvalstack:  eval stack stack) =
   emptystack(stacksize,emptystack(1,Unbound))

let (envstack:  eval env stack) =
   emptystack(stacksize,emptyenv(Unbound))

let pushenv(r) = push(r,envstack)

let topenv() = top(envstack)

let svuotaenv() = svuota(envstack)

let popenv () = pop(envstack)
```

Definiamo ora la funzione `newframes` che crea un nuovo record di attivazione (frame):

```
let newframes(e,rho) =
   let cframe = emptystack(cframesize(e),Expr1(e)) in
   let tframe = emptystack(tframesize(e),Unbound) in
     push(Expr1(e),cframe);
     push(cframe,cstack);
     push(tframe,tempvalstack);
     pushenv(rho)
```

Infine, l'interprete iterativo:

```
let sem ((e:exp), (r:eval env)) =
  push(emptystack(1,Unbound),tempvalstack);
  newframes(e,r);
  while not(empty(cstack)) do
    while not(empty(top(cstack))) do (*1*)
      let continuation = top(cstack) in
      let tempstack = top(tempvalstack) in
      let rho = topenv() in
```

```
(match top(continuation) with
 |Expr1(x) ->
    (pop(continuation);
    push(Expr2(x),continuation);
    (match x with
       | Iszero(a) -> push(Expr1(a),continuation)
       | Eq(a,b) -> push(Expr1(a),continuation);
          push(Expr1(b),continuation)
       | Prod(a,b) -> push(Expr1(a),continuation);
          push(Expr1(b),continuation)
       | Sum(a,b) -> push(Expr1(a),continuation);
          push(Expr1(b),continuation)
       | Diff(a,b) -> push(Expr1(a),continuation);
          push(Expr1(b),continuation)
       | Minus(a) -> push(Expr1(a),continuation)
       | And(a,b) -> push(Expr1(a),continuation);
          push(Expr1(b),continuation)
       | Or(a,b) -> push(Expr1(a),continuation);
          push(Expr1(b),continuation)
       | Not(a) -> push(Expr1(a),continuation)
       | Ifthenelse(a,b,c) -> push(Expr1(a),continuation)
       | Let(i,e1,e2) -> push(Expr1(e1),continuation) (*2*)
       | _ -> ()))
 |Expr2(x) ->
    (pop(continuation);
    (match x with
       | Eint(n) -> push(Int(n),tempstack)
       | Ebool(b) -> push(Bool(b),tempstack)
       | Den(i) -> push(applyenv(rho,i),tempstack)
       | Iszero(a) ->
          let arg=top(tempstack) in
           pop(tempstack);
           push(iszero(arg),tempstack)
       | Eq(a,b) ->
          let firstarg=top(tempstack) in
           pop(tempstack);
           let sndarg=top(tempstack) in
            pop(tempstack);
            push(equ(firstarg,sndarg),tempstack)
       | Prod(a,b) ->
          let firstarg=top(tempstack) in
           pop(tempstack);
           let sndarg=top(tempstack) in
            pop(tempstack);
            push(mult(firstarg,sndarg),tempstack)
       | Sum(a,b) ->
          let firstarg=top(tempstack) in
           pop(tempstack);
           let sndarg=top(tempstack) in
```

```
                    pop(tempstack);
                    push(plus(firstarg,sndarg),tempstack)
              | Diff(a,b) ->
                 let firstarg=top(tempstack) in
                  pop(tempstack);
                  let sndarg=top(tempstack) in
                    pop(tempstack);
                    push(diff(firstarg,sndarg),tempstack)
              | Minus(a) ->
                 let arg=top(tempstack) in
                  pop(tempstack);
                  push(minus(arg),tempstack)
              | And(a,b) ->
                 let firstarg=top(tempstack) in
                  pop(tempstack);
                  let sndarg=top(tempstack) in
                    pop(tempstack);
                    push(et(firstarg,sndarg),tempstack)
              | Or(a,b) ->
                 let firstarg=top(tempstack) in
                  pop(tempstack);
                  let sndarg=top(tempstack) in
                    pop(tempstack);
                    push(vel(firstarg,sndarg),tempstack)
              | Not(a) ->
                 let arg=top(tempstack) in
                  pop(tempstack);
                  push(non(arg),tempstack)
              | Ifthenelse(a,b,c) ->
                 let arg=top(tempstack) in
                  pop(tempstack);
                  if typecheck(``bool'',arg) then
                    (if arg = Bool(true)
                    then push(Expr1(b),continuation)
                    else push(Expr1(c),continuation))
                  else failwith (``type error'')
              | Let(i,e1,e2) -> let arg=top(tempstack) in
                 pop(tempstack);
                 newframes(e2,bind(rho, i, arg)))))
  done; (*3*)
  let valore= top(top(tempvalstack)) in
    pop(top(tempvalstack));
    popenv();
    pop(cstack);
    pop(tempvalstack);
    push(valore,top(tempvalstack));
done; (*4*)
let valore = top(top(tempvalstack)) in
  pop(top(tempvalstack));
```

```
    pop(tempvalstack); valore
```

Commenti al codice:

1. In questo caso abbiamo due cicli: uno svuota `cstack`, l'altro svuota `top(cstack)`.
2. Il `Let` è non stretto. *e1* viene valutato in un ambiente, *e2* in un ambiente modificato.
3. Uscita dal blocco.
4. Uscita dall'interprete.

8.9 Blocchi in un linguaggio imperativo

In un linguaggio imperativo un blocco consiste in una **lista di dichiarazioni** seguita da una **lista di comandi**. La lista di comandi viene eseguita nello stato (ambiente e memoria) risultante dall'esecuzione della lista di dichiarazioni.
Il blocco è un comando che non modifica l'ambiente "esterno" e restituisce la memoria "esterna" (eventualmente) modificata dall'esecuzione dei comandi. Le associazioni locali (e le locazioni di memoria eventualmente ad esse associate) esistono solo all'interno del blocco.
Nel nostro linguaggio didattico introduciamo un nuovo costrutto di tipo espressione (simile al `ref` di ML) per realizzare le dichiarazioni di variabili.

8.10 Linguaggio imperativo con blocchi: domini sintattici

Aggiungiamo alla sintassi del frammento imperativo i costrutti `Let` e `NewLoc` come espressioni, e `Block` fra i comandi:

```
type ide = string
type exp =
   | Eint of int
   | Ebool of bool
   | Den of ide
   | Prod of exp * exp
   | Sum of exp * exp
   | Diff of exp * exp
   | Eq of exp * exp
   | Minus of exp
   | Iszero of exp
   | Or of exp * exp
   | And of exp * exp
   | Not of exp
   | Ifthenelse of exp * exp * exp
   | Val of exp
   | Let of ide * exp * exp
   | NewLoc of exp

and decl = (ide * exp) list

and com =
```

```
   | Assign of exp * exp
   | Cifthenelse of exp * com list * com list
   | While of exp * com list
   | Block of decl * com list;;
```

8.11 Semantica denotazionale

Ecco la semantica denotazionale del linguaggio imperativo con i blocchi:

```
let rec sem (e:exp) (r:dval env) (s:  mval store) =
  match e with
    | Eint(n) -> Int(n)
    | Ebool(b) -> Bool(b)
    | Den(i) -> dvaltoeval(applyenv(r,i))
    | Iszero(a) -> iszero((sem a r s) )
    | Eq(a,b) -> equ((sem a r s) ,(sem b r s) )
    | Prod(a,b) -> mult ( (sem a r s), (sem b r s))
    | Sum(a,b) -> plus ( (sem a r s), (sem b r s))
    | Diff(a,b) -> diff ( (sem a r s), (sem b r s))
    | Minus(a) -> minus( (sem a r s))
    | And(a,b) -> et ( (sem a r s), (sem b r s))
    | Or(a,b) -> vel ( (sem a r s), (sem b r s))
    | Not(a) -> non( (sem a r s))
    | Ifthenelse(a,b,c) ->
       let g = sem a r s in
        if typecheck(``bool'',g) then
          (if g = Bool(true)
          then sem b r s
          else sem c r s)
        else failwith (``nonboolean guard'')
    | Let(i,e1,e2) -> let (v, s1) = semden e1 r s in
       sem e2 (bind (r ,i, v)) s1
    | Val(e) -> let (v, s1) = semden e r s in
       (match v with
         | Dloc n -> mvaltoeval(applystore(s1, n))
         | _ -> failwith(``not a variable''))
    | _ -> failwith (``nonlegal expression for sem'')

and semden (e:exp) (r:dval env) (s:  mval store) = match e with
   | Den(i) -> (applyenv(r,i), s)
   | Newloc(e) -> let m = evaltomval(sem e r s) in
    let (l, s1) = allocate(s, m) in (Dloc l, s1)
   | _ -> (evaltodval(sem e r s), s)

and semdv dl r s =
  match dl with
    | [] -> (r,s)
    | (i,e)::dl1 -> let (v, s1) = semden e r s in
```

```
       semdv dl1 (bind(r, i, v)) s1

and semc (c:  com) (r:dval env) (s:  mval store) = match c with
   | Assign(e1, e2) -> let (v1, s1) = semden e1 r s in
      (match v1 with
       | Dloc(n) -> update(s1, n, (evaltomval(sem e2 r s)))
       | _ -> failwith (``wrong location in assignment''))
   | Cifthenelse(e, cl1, cl2) -> let g = sem e r s in
      if typecheck(``bool'',g) then
       (if g = Bool(true)
       then semcl cl1 r s
       else semcl cl2 r s)
      else failwith (``nonboolean guard'')
   | While(e, cl) ->
      let functional ((fi:  mval store -> mval store)) =
       function sigma ->
         let g = sem e r sigma in
          if typecheck(``bool'',g) then
           (if g = Bool(true)
           then fi(semcl cl r sigma)
           else sigma)
          else failwith (``nonboolean guard'') in
      let rec ssfix = function x -> functional ssfix x in
       ssfix(s)
   | Block(b) -> semb b r s

and semcl cl r s = match cl with
   | [] -> s
   | c::cl1 -> semcl cl1 r (semc c r s)

and semb (dl, cl) r s =
   let (r1, s1) = semdv dl r s in
    semcl cl r1 s1;;
```

Le funzioni definite hanno il seguente tipo:

```
val sem :  exp -> dval Funenv.env -> mval Funstore.store -> eval = <fun>

val semden :  exp -> dval Funenv.env -> mval Funstore.store ->
   dval * mval Funstore.store = <fun>

val semc :  com -> dval Funenv.env -> mval Funstore.store ->
   mval Funstore.store = <fun>

val semcl :  com list -> dval Funenv.env -> mval Funstore.store ->
   mval Funstore.store = <fun>

val semdv :  decl -> dval Funenv.env -> mval Funstore.store ->
   dval Funenv.env * mval Funstore.store = <fun>
```

```
val semb :  (decl * com list) -> dval Funenv.env ->
  mval Funstore.store -> mval Funstore.store = <fun>
```

E' importante notare che è cambiato il tipo della funzione `semden` (vedi 7.17). Notiamo inoltre come

```
          and semb (dl, cl) r s =
             let (r1, s1) = semdv dl r s in semcl cl r1 s1;;
```

valuti la lista dei comandi (cl) nell'ambiente "esterno" r esteso con la semantica delle dichiarazioni dl, e nella memoria "esterna" s anch'essa estesa. La memoria restituita contiene anche ciò che è stato prodotto nel blocco, ma le nuove locazioni risultano inaccessibili dall'ambiente "esterno" r.

8.12 Semantica operazionale

La semantica operazionale completa per il linguaggio imperativo con i blocchi:

```
let rec sem ((e:exp), (r:dval env), (s:  mval store)) =
  match e with
    | Eint(n) -> Int(n)
    | Ebool(b) -> Bool(b)
    | Den(i) -> dvaltoeval(applyenv(r,i))
    | Iszero(a) -> iszero(sem(a, r, s))
    | Eq(a,b) -> equ(sem(a, r, s), sem(b, r, s))
    | Prod(a,b) -> mult (sem(a, r, s), sem(b, r, s))
    | Sum(a,b) -> plus (sem(a, r, s), sem(b, r, s))
    | Diff(a,b) -> diff (sem(a, r, s), sem(b, r, s))
    | Minus(a) -> minus(sem(a, r, s))
    | And(a,b) -> et (sem(a, r, s), sem(b, r, s))
    | Or(a,b) -> vel (sem(a, r, s), sem(b, r, s))
    | Not(a) -> non(sem(a, r, s))
    | Ifthenelse(a,b,c) ->
       let g = sem(a, r, s) in
        if typecheck(``bool'',g) then
          (if g = Bool(true)
          then sem(b, r, s)
          else sem(c, r, s))
        else failwith (``nonboolean guard'')
    | Let(i,e1,e2) -> let (v, s1) = semden(e1, r, s) in
       sem(e2, bind (r ,i, v), s1)
    | Val(e) -> let (v, s1) = semden(e, r, s) in
       (match v with
        | Dloc n -> mvaltoeval(applystore(s1, n))
        | _ -> failwith(``not a variable''))
    | _ -> failwith (``nonlegal expression for sem'')

and semden ((e:exp), (r:dval env), (s:  mval store)) = match e with
```

```
   | Den(i) -> (applyenv(r,i), s)
   | Newloc(e) -> let m = evaltomval(sem(e, r, s)) in
     let (l, s1) = allocate(s, m) in (Dloc l, s1)
   | _ -> (evaltodval(sem(e, r, s)), s)

and semdv(dl, r, s) =
   match dl with
     | [] -> (r,s)
     | (i,e)::dl1 -> let (v, s1) = semden(e, r, s) in
        semdv(dl1, bind(r, i, v), s1)

and semc((c:  com), (r:dval env), (s:  mval store)) = match c with
   | Assign(e1, e2) -> let (v1, s1) = semden(e1, r, s) in
      (match v1 with
        | Dloc(n) -> update(s1, n, evaltomval(sem(e2, r, s)))
        | _ -> failwith (``wrong location in assignment''))
   | Cifthenelse(e, cl1, cl2) -> let g = sem(e, r, s) in
      if typecheck(``bool'',g) then
        (if g = Bool(true)
        then semcl(cl1, r, s)
        else semcl (cl2, r, s))
      else failwith (``nonboolean guard'')
   | While(e, cl) -> let g = sem(e, r, s) in
      if typecheck(``bool'',g) then
        (if g = Bool(true) then
          semcl((cl @ [While(e, cl)]), r, s)
        else s)
      else failwith (``nonboolean guard'')
   | Block(b) -> semb(b, r, s)

and semcl(cl, r, s) = match cl with
   | [] -> s
   | c::cl1 -> semcl(cl1, r, semc(c, r, s))

and semb ((dl, cl), r, s) =
   let (r1, s1) = semdv(dl, r, s) in
     semcl(cl, r1, s1)
```

Le funzioni definite hanno il seguente tipo (vedi 7.19):

```
val sem :  exp * dval Funenv.env * mval Funstore.store -> eval = <fun>

val semden :  exp * dval Funenv.env * mval Funstore.store ->
   (dval * mval Funstore.store) = <fun>

val semc :  com * dval Funenv.env * mval Funstore.store ->
   mval Funstore.store = <fun>

val semcl :  com list * dval Funenv.env * mval Funstore.store ->
```

```
  mval Funstore.store = <fun>

val semdv :  decl * dval Funenv.env * mval Funstore.store ->
  dval Funenv.env * mval Funstore.store = <fun>

val semb :  (decl * com list) * dval Funenv.env * mval Funstore.store
->
  mval Funstore.store = <fun>
```

8.13 Eliminare la ricorsione

Il dominio delle dichiarazioni è già iterativo (*tail recursive*)

```
type decl = (ide * exp) list
```

quindi, come per i comandi, si può utilizzare la struttura sintattica (lista di coppie) per mantenere le informazioni su ciò che deve essere ancora valutato. In sostanza ci basta un'unica cella per ogni attivazione, che può essere integrata nella pila "locale" dei costrutti sintattici etichettati.

8.14 Blocchi e record di attivazione

Il record di attivazione dell'interprete iterativo necessita in questo caso, oltre a quanto già discusso per il caso funzionale (vedi 8.7 e 8.8), anche di un riferimento alla memoria del blocco. Nell'implementazione iterativa, ad una attivazione corrisponde quindi la creazione di un nuovo *record di attivazione* con:

- Ambiente.
- Costrutto sintattico da valutare e pila necessaria per farlo.
- Pile per memorizzare i valori temporanei (*eval* e *dval*)
- Memoria

Implementiamo i record di attivazione attraverso le seguenti pile "parallele":

- `envstack`: pila di ambienti.
- `cstack`: pila di pile di costrutti sintattici etichettati.
- `tempvalstack`: pila di pile di *eval*.
- `tempdvalstack`: pila di pile di *dval*.
- `storestack`: pila di memorie.

E' necessaria una importante considerazione a proposito di ciò che succede all'uscita di un blocco.
Il ciclo dell'interprete iterativo tratta allo stesso modo tutti i costrutti che creano una nuova attivazione (`Let` e `Block`), "esportando" all'attivazioe precedente cose diverse nel momento in cui si esce dal blocco: *eval* per le espressioni e *store* per i comandi.
Il problema si ha sull'attivazione iniziale, che può anche corrispondere ad una dichiarazione, che deve restituire una coppia *(env * store)*.

E' quindi necessaria un'ulteriore informazione nel record di attivazione, che indichi il costrutto sintattico che l'ha originata. Di conseguenza aggiungiamo una **nuova pila** "parallela":

- `labelstack`: pila di costrutti sintattici etichettati.

8.15 Interprete iterativo

Definiamo innanzi tutto le strutture dati usate dall'interprete:

```
type labeledconstruct =
   | Expr1 of exp
   | Expr2 of exp
   | Exprd1 of exp
   | Exprd2 of exp
   | Com1 of com
   | Com2 of com
   | Coml of labeledconstruct list
   | Decl of labeledconstruct list
   | Dec1 of (ide * exp)
   | Dec2 of (ide * exp)

let (cstack:  labeledconstruct stack stack) =
   emptystack(stacksize,emptystack(1,Expr1(Eint(0))))

let (tempvalstack:  eval stack stack) =
   emptystack(stacksize,emptystack(1,Novalue))

let (tempdvalstack:  dval stack stack) =
   emptystack(stacksize,emptystack(1,Unbound))

let labelcom (dl:  com list) = let dlr =
   ref(dl) in let ldlr = ref([]) in
    while not (!dlr = []) do
      let i = List.hd !dlr in
        ldlr := !ldlr @ [Com1(i)];
        dlr := List.tl !dlr
    done;
    Coml(!ldlr)

let labeldec (dl:  (ide * exp) list) = let dlr = ref(dl) in
let ldlr = ref([]) in
   while not (!dlr = []) do
    let i = List.hd !dlr in
      ldlr := !ldlr @ [Dec1(i)];
      dlr := List.tl !dlr
   done;
   Decl(!ldlr)

let envstack = emptystack(stacksize,(emptyenv Unbound))
```

```
let storestack = emptystack(stacksize,(emptystore Undefined))

let pushenv(r) = push(r,envstack)

let topenv() = top(envstack)

let popenv () = pop(envstack)

let svuotaenv() = svuota(envstack)

let pushstore(s) = push(s,storestack)

let popstore () = pop(storestack)

let svuotastore () = svuota(storestack)

let topstore() = top(storestack)

let (labelstack:  labeledconstruct stack) =
  emptystack(stacksize,Expr1(Eint(0)))
```

Definiamo la funzione `newframes`, che crea un nuovo record di attivazione:

```
let newframes(ss,rho,sigma) =
  pushstore(sigma);
  pushenv(rho);
  let cframe = emptystack(cframesize(ss),Expr1(Eint 0)) in
  let tframe = emptystack(tframesize(ss),Novalue) in
  let dframe = emptystack(tdframesize(ss),Unbound) in
   push(tframe,tempvalstack);
   push(dframe,tempdvalstack);
   push(ss, labelstack);
   push(ss, cframe);
   push(cframe, cstack)
```

Definiamo ora l'interprete iterativo completo per il linguaggio imperativo con i blocchi:

```
let itsem() =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
   (match top(continuation) with
    |Expr1(x) ->
       (pop(continuation);
       push(Expr2(x),continuation);
       (match x with
```

```
        | Iszero(a) -> push(Expr1(a),continuation)
        | Eq(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
        | Prod(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
        | Sum(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
        | Diff(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
        | Minus(a) -> push(Expr1(a),continuation)
        | And(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
        | Or(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
        | Not(a) -> push(Expr1(a),continuation)
        | Ifthenelse(a,b,c) -> push(Expr1(a),continuation)
        | Let(i,e1,e2) -> push(Exprd1(e1),continuation)
        | Val(e) -> push(Exprd1(e),continuation)
        | Newloc(e) -> failwith (``nonlegal expression for sem'')
        | _ -> ()))
   |Expr2(x) ->
      (pop(continuation);
      (match x with
        | Eint(n) -> push(Int(n),tempstack)
        | Ebool(b) -> push(Bool(b),tempstack)
        | Den(i) -> push(dvaltoeval(applyenv(rho,i)),tempstack)
        | Iszero(a) ->
           let arg=top(tempstack) in
            pop(tempstack);
            push(iszero(arg),tempstack)
        | Eq(a,b) ->
           let firstarg=top(tempstack) in
            pop(tempstack);
            let sndarg=top(tempstack) in
              pop(tempstack);
              push(equ(firstarg,sndarg),tempstack)
        | Prod(a,b) ->
           let firstarg=top(tempstack) in
            pop(tempstack);
            let sndarg=top(tempstack) in
              pop(tempstack);
              push(mult(firstarg,sndarg),tempstack)
        | Sum(a,b) ->
           let firstarg=top(tempstack) in
            pop(tempstack);
            let sndarg=top(tempstack) in
              pop(tempstack);
              push(plus(firstarg,sndarg),tempstack)
        | Diff(a,b) ->
```

```
              let firstarg=top(tempstack) in
               pop(tempstack);
               let sndarg=top(tempstack) in
                pop(tempstack);
                push(diff(firstarg,sndarg),tempstack)
           | Minus(a) ->
              let arg=top(tempstack) in
               pop(tempstack);
               push(minus(arg),tempstack)
           | And(a,b) ->
              let firstarg=top(tempstack) in
               pop(tempstack);
               let sndarg=top(tempstack) in
                pop(tempstack);
                push(et(firstarg,sndarg),tempstack)
           | Or(a,b) ->
              let firstarg=top(tempstack) in
               pop(tempstack);
               let sndarg=top(tempstack) in
                pop(tempstack);
                push(vel(firstarg,sndarg),tempstack)
           | Not(a) ->
              let arg=top(tempstack) in
               pop(tempstack);
               push(non(arg),tempstack)
           | Ifthenelse(a,b,c) ->
              let arg=top(tempstack) in
               pop(tempstack);
               if typecheck(``bool'',arg) then
                 (if arg = Bool(true)
                 then push(Expr1(b),continuation)
                 else push(Expr1(c),continuation))
               else failwith (``type error'')
           | Val(e) -> let v = top(tempdstack) in
              pop(tempdstack);
              (match v with
                | Dloc n ->
                   push(mvaltoeval(applystore(sigma, n)), tempstack)
                | _ -> failwith(``not a variable''))
           | Let(i,e1,e2) -> let arg= top(tempdstack) in
              pop(tempdstack);
              newframes(Expr1(e2), bind(rho, i, arg), sigma)
           | _ -> failwith(``no more cases for semexpr'')))
     | _ -> failwith(``no more cases for semexpr''))

let itsemden() =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
```

```
  let rho = topenv() in
  let sigma = topstore() in
    (match top(continuation) with
      |Exprd1(x) -> (pop(continuation); push(Exprd2(x),continuation);
                  match x with
                    | Den i -> ()
                    | Newloc(e) ->
                       push(Expr1( e), continuation)
                    | _ -> push(Expr1(x), continuation))
      |Exprd2(x) -> (pop(continuation); match x with
                    | Den i -> push(applyenv(rho,i), tempdstack)
                    | Newloc(e) ->let m=evaltomval(top(tempstack)) in
                       pop(tempstack);
                       let (l, s1) = allocate(sigma, m) in
                        push(Dloc l, tempdstack);
                        popstore();
                        pushstore(s1)
                    | _ -> let arg = top(tempstack) in
                       pop(tempstack);
                       push(evaltodval(arg), tempdstack))
      | _ -> failwith(``No more cases for semden'') )

let itsemdecl () =
  let tempstack = top(tempvalstack) in
  let continuation = top(cstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  let dl =
    (match top(continuation) with
      | Decl(dl1) -> dl1
      | _ -> failwith(``impossible in semdecl'')) in
    if dl = [] then pop(continuation) else
      (let currd = List.hd dl in
      let newdl = List.tl dl in
       pop(continuation); push(Decl(newdl),continuation);
       (match currd with
          | Dec1( (i,e)) ->
             pop(continuation);
             push(Decl(Dec2((i, e))::newdl),continuation);
             push(Exprd1(e), continuation)
           | Dec2((i,e)) ->
             let arg = top(tempdstack) in
                pop(tempdstack);
              popenv();
              pushenv(bind(rho, i, arg))
          | _ -> failwith(``no more sensible cases for semdecl'')))

let itsemcl() =
```

```
let tempstack = top(tempvalstack) in
let continuation = top(cstack) in
let tempdstack = top(tempdvalstack) in
let rho = topenv() in
let sigma = topstore() in
let cl =
  (match top(continuation) with
    | Coml(dl1) -> dl1
    | _ -> failwith(``impossible in semcl'')) in
  if cl = [] then pop(continuation) else
    (let currc = List.hd cl in
    let newcl = List.tl cl in
      pop(continuation); push(Coml(newcl),continuation);
        (match currc with
           | Com1(Assign(e1, e2)) -> pop(continuation);
            push(Coml(Com2(Assign(e1, e2))::newcl),continuation);
            push(Exprd1(e1), continuation);
            push(Expr1(e2), continuation)
         | Com2(Assign(e1, e2)) ->
            let arg2 = evaltomval(top(tempstack)) in
             pop(tempstack);
             let arg1 = top(tempdstack) in
              pop(tempdstack);
              (match arg1 with
                 | Dloc(n) -> popstore();
                    pushstore(update(sigma, n, arg2))
                 | _ -> failwith (``wrong location in assignment''))
         | Com1(While(e, cl)) ->
            pop(continuation);
            push(Coml(Com2(While(e, cl))::newcl),continuation);
            push(Expr1(e), continuation)
         | Com2(While(e, cl)) ->
            let g = top(tempstack) in
             pop(tempstack);
             if typecheck(``bool'',g) then
               (if g = Bool(true) then
                  (let old = newcl in
                  let newl =
                    (match labelcom cl with
                      | Coml newl1 -> newl1
                      | _ -> failwith(``impossible in while'')) in
                    let nuovo =
                      Coml(newl @ [Com1(While(e, cl))] @ old) in
                       pop(continuation); push(nuovo,continuation))
                else ())
               else failwith (``nonboolean guard'')
         | Com1(Cifthenelse(e, cl1, cl2)) ->
            pop(continuation);
            push(Coml(Com2(Cifthenelse(e, cl1, cl2))::newcl),continuation);
```

```
            push(Expr1(e), continuation)
          | Com2(Cifthenelse(e, cl1, cl2)) ->
            let g = top(tempstack) in
             pop(tempstack);
             if typecheck(``bool'',g) then
              (let temp = if g = Bool(true) then
               labelcom (cl1) else labelcom (cl2) in
              let newl =
               (match temp with
                  | Coml newl1 -> newl1
                  | _ -> failwith(``impossible in cifthenelse'')) in
              let nuovo = Coml(newl @ newcl) in
                 pop(continuation); push(nuovo,continuation))
              else failwith (``nonboolean guard'')

           | Com1(Block((l1, l3))) ->
             newframes(labelcom(l3), rho, sigma);
             push(labeldec(l1),top(cstack));

          | _ -> failwith(``no more sensible cases in commands'') ))


let initstate() =
  svuota(cstack); svuota(tempvalstack); svuota(tempdvalstack);
  svuotaenv(); svuotastore(); svuota(labelstack)

let loop () =
  while not(empty(cstack)) do
   while not(empty(top(cstack))) do
    let currconstr = top(top(cstack)) in
     (match currconstr with
      | Expr1(e) -> itsem()
      | Expr2(e) -> itsem()
      | Exprd1(e) -> itsemden()
      | Exprd2(e) -> itsemden()
      | Coml(cl) -> itsemcl()
      | Decl(l) -> itsemdecl()
      | _ -> failwith(``non legal construct in loop''))
   done;
   (match top(labelstack) with
    | Expr1(_) -> let valore = top(top(tempvalstack)) in
       pop(top(tempvalstack));
       pop(tempvalstack); push(valore,top(tempvalstack));
       popenv(); popstore(); pop(tempdvalstack)
    | Exprd1(_) -> let valore = top(top(tempdvalstack)) in
       pop(top(tempdvalstack));
       pop(tempdvalstack); push(valore,top(tempdvalstack));
       popenv(); popstore(); pop(tempvalstack)
    | Decl(_) ->
```

```
        pop(tempvalstack); pop(tempdvalstack)
      | Coml(_) -> let st = topstore() in
        popenv();popstore(); popstore(); pushstore(st);
        pop(tempvalstack); pop(tempdvalstack)
      | _ -> failwith(``non legal label in loop''));
    pop(cstack);
    pop(labelstack)
  done

let sem (e,(r:  dval env), (s:  mval store)) =
  initstate();
  push(emptystack(tframesize(e),Novalue),tempvalstack);
  newframes(Expr1(e), r, s);
  loop();
  let valore= top(top(tempvalstack)) in
   pop(tempvalstack);
   valore

let semden (e,(r:  dval env), (s:  mval store)) =
  initstate();
  push(emptystack(tdframesize(e),Unbound),tempdvalstack);
  newframes(Exprd1(e), r, s);
  loop();
  let valore= top(top(tempdvalstack)) in
   pop(tempdvalstack);
   valore

let semcl (cl,(r:  dval env), (s:  mval store)) =
  initstate();
  pushstore(emptystore(Undefined));
  newframes(labelcom(cl), r, s);
  loop();
  let st = topstore() in
   popstore();
   st

let semdv(dl, r, s) =
  initstate();
  newframes(labeldec(dl), r, s);
  loop();
  let st = topstore() in
   popstore();
   let rt = topenv() in
    popenv();
    (rt, st)

let semc((c:  com), (r:dval env), (s:  mval store)) =
  initstate();
  pushstore(emptystore(Undefined));
```

```
   newframes(labelcom([c]), r, s);
   loop();
   let st = topstore() in
    popstore();
    st

let semb ((dl, cl), r, s) =
   initstate();
   pushstore(emptystore(Undefined));
   newframes(labelcom(cl), r, s);
   push(labeldec(dl), top(cstack));
   loop();
   let st = topstore() in
    popstore();
    st
```

8.16 Digressione sull'ambiente locale statico

I caso di ambiente locale statico, l'ambiente viene *attivato* all'ingresso di un blocco e *disattivato* all'uscita (non creato e distrutto).
Le dichiarazioni vengono quindi eseguite una sola volta (compilazione, caricamento o prima esecuzione del blocco) e le associazioni create sono utilizzabili solo quando sono *attive*.
Se il blocco contiene dichiarazioni di variabile anche la memoria locale al blocco viene preservata tra due diverse esecuzioni del blocco, costituendo una sorta di stato interno al blocco.
L'effetto concreto di tale meccanismo è rappresentato dal fatto che uscendo e rientrando in un blocco, lo ritroviamo così come l'avevamo lasciato, piuttosto che crearne uno nuovo ad ogni ingresso.
I linguaggi che fanno uso di ambiente locale statico sono:

- *FORTRAN*: pur non avendo blocchi, utilizza questa regola per gestire l'ambiente locale dei sottoprogrammi per cui la chiamata n-esima ad un sottoprogramma trova lo stato lasciato dalla chiamata (n-1)-esima. Questa caratteristica rende molto difficile definire la semantica dei sottoprogrammi.
- *ALGOL, PL/I, C*: alcune dichiarazioni locali (a blocchi e sottoprogrammi) possono essere dichiarate statiche (`static`, `own`, ecc.). Le altre associazioni sono trattate con ambiente dinamico.
- *Java*: le dichiarazioni `static` all'interno di una classe provocano la creazione di un ambiente che appartiene alla classe e non agli oggetti che la istanziano. Queste dichiarazioni sono eseguite una sola volta nel momento dell'esecuzione della dichiarazione della classe e possono costituire uno stato interno, comune a tutti gli oggetti della classe.

8.17 Ambiente locale statico: motivazioni ed implementazione

L'uso di ambiente locale statico trova la propria motivazione originale nella ricerca di una maggiore efficienza, anche se oggi si considera l'utilità di tale strumento in relazione alla possibilità di definire uno stato interno al blocco (magari condiviso), a discapito però della semplicità nella comprensione e definizione della semantica. Dal punto di vista implementativo, comunque, l'ambiente locale statico permette di non inserire all'interno dei record di attivazione le tabelle che realizzano le associazioni locali (necessarie altrimenti, come vedremo), consentendo la creazione a tempo di compilazione di ambiente e memoria locali (con il rischio di sprecare comunque molta memoria) e la conseguente eliminazione dei nomi.

9 Sottoprogrammi ed astrazioni funzionali in linguaggi funzionali

Contenuti del capitolo:

- Nascita dei sottoprogrammi: motivazioni, strumenti e prime implementazioni.
- Nozione "moderna" di astrazione funzionale.
- Funzioni nel linguaggio funzionale: astrazione, applicazione, regole di scoping.
- Semantica delle funzioni con scoping statico: denotazionale, operazionale, iterativa.
- Digressione sullo scoping dinamico.
- Confronto fra scoping statico e dinamico.

9.1 Le esigenze a cui si risponde con il sottoprogramma

Un sottoprogramma rappresenta in sostanza l'astrazione di una sequenza di istruzioni, e presenta i seguenti principali vantaggi:

- *Riduzione del "costo di programmazione"*: attraverso l'uso di *macro* e *macroespansione*.
- *Riduzione dell'occupazione di memoria*: attraverso il trasferimento del controllo dal programma principale verso un'unica copia del frammento, memorizzata separatamente, per poi riprendere il controllo quando l'esecuzione del frammento termina.
- *Astrazione via parametrizzazione*: possibilità di definire il frammento in modo parametrico, astraendo dall'identità di alcuni dati (possibile anche con le *macro* ed il *codice rientrante*).

9.2 Cosa fornisce l'hardware

A livello hardware viene utilizzata un'operazione primitiva di `return jump`.
Vediamo esattamente come funziona:

- Viene eseguita nel programma chiamante una `return jump a` memorizzata nella cella `b`.
- Il controllo viene trasferito alla cella `a` (*entry point della subroutine*).
- L'indirizzo dell'istruzione successiva (`b+1`) viene memorizzato in un posto noto, ad esempio nella cella (`a-1`) (*punto di ritorno*).
- Quando nella subroutine si esegue un'operazione di `return`, il controllo ritorna all'istruzione (del programma chiamante) memorizzata nel punto di ritorno.

9.3 Implementazione delle subroutine in FORTRAN

Vediamo come sono implementate le subroutine in FORTRAN, primo linguaggio che propone l'uso di (una sorta di) sottoprogrammi, sfruttando gli strumenti offerti dal linguaggio macchina.
In FORTRAN una subroutine è un pezzo di codice compilato a cui sono associati: una cella destinata a contenere (a tempo di esecuzione) i punti di ritorno relativi alle (possibili varie) chiamate, alcune celle destinate a contenere i valori di eventuali parametri, ambiente e memoria locali (come detto l'ambiente locale è statico).
Le subroutine in FORTRAN sono definite semplicemente attraverso la *copy rule statica* (**macroespansione**), ovvero rimpiazzando testualmente ogni chiamata con il codice corrispondente, operando opportunamente sui parametri e ricordandosi che le dichiarazioni vengono eseguite un'unica volta (ambiente locale statico).
Così come è definito il sottoprogramma non rappresenta semanticamente qualcosa di nuovo, ma è piuttosto uno strumento metodologico (astrazione).
I punti deboli dell'implementazione dei sottoprogrammi in FORTRAN sono rappresentati dall'incompatibilità con la ricorsione (si originerebbero programmi infiniti) e l'impossibilità di gestire più attivazioni presenti allo stesso tempo, a causa del punto di ritorno unico.
Il fatto che le subroutine FORTRAN siano concettualmente una cosa statica fa si che non esista il concetto di attivazione, e che l'ambiente locale sia necessariamente statico.

9.4 Verso una vera nozione di sottoprogramma

Ragionando in termini di attivazioni, come abbiamo fatto per i blocchi, la semantica dei sottoprogrammi può essere ancora definita da una *copy rule*, ma **dinamica**: ogni chiamata a sottoprogramma viene rimpiazzata a **tempo di esecuzione** da una copia del codice.
Il sottoprogramma è ora semanticamente qualcosa di nuovo e rende naturale l'uso della *ricorsione* e l'adozione della regola dell'*ambiente dinamico*.
Dal punto di vista implementativo ci aspettiamo dai sottoprogrammi:

- Un record di attivazione associato dinamicamente alle varie chiamate ai sottoprogrammi e contenente le stesse informazioni associate staticamente al codice compilato di FORTRAN (punto di ritorno, parametri, ambiente e memoria locale).
- Organizzazione dei record di attivazione in una pila, dato il comportamento LIFO dei sottoprogrammi, in maniera simile a quanto succedeva nell'interprete iterativo dei frammenti con blocchi[19].

Quanto detto ci consente dunque di definire un sottoprogramma "vero" (rispetto alla sola macroastrazione di FORTRAN) come:

- Astrazione procedurale (operazioni)
  - Astrazione di una sequenza di istruzioni.
  - Astrazione via parametrizzazione.

[19] I blocchi sono, in effetti, un caso particolare di sottoprogrammi.

- Luogo di controllo per la gestione di ambiente e memoria
  - Estensione del blocco
  - In assoluto, l'aspetto più interessante dei linguaggi, attorno a cui ruotano tutte le decisioni semantiche importanti.

9.5 Introduzione delle funzioni nel linguaggio funzionale: sintassi

Introduciamo nella sintassi del linguaggio funzionale l'astrazione e l'applicazione di funzione (ricorsiva e non), attraverso i costrutti `Fun`, `Appl` e `Rec`:

```
type ide = string
type exp =
   Eint of int
   | Ebool of bool
   | Den of ide
   | Prod of exp * exp
   | Sum of exp * exp
   | Diff of exp * exp
   | Eq of exp * exp
   | Minus of exp
   | Iszero of exp
   | Or of exp * exp
   | And of exp * exp
   | Not of exp
   | Ifthenelse of exp * exp * exp
   | Let of ide * exp * exp
   | Fun of ide list * exp (*1*)
   | Appl of exp * exp list (*2*)
   | Rec of ide * exp (*3*)
```

Commenti al codice:

1. `Fun`:
   - Rappresenta la λ-astrazione e quindi non prende come parametro il nome da assegnare alla funzione (si usa il `Let`).
   - Gli identificatori presi in ingresso rappresentano i parametri della funzione.
2. `Appl`: le espressioni in ingresso rappresentano i valori dei parametri della funzione.
3. `Rec`: per il momento ignoriamo questo costrutto.

Commenti sulle funzioni:

- In questo capitolo non ci occuperemo di definire le varie modalità per il passaggio dei parametri. Le espressioni parametro attuale sono valutate (*eval* oppure *dval*) ed i valori ottenuti sono legati nell'ambiente al corrispondente parametro formale.

- Con l'introduzione delle funzioni, il linguaggio funzionale è completo. Lo ritoccheremo solo per discutere alcune modalità di passaggio dei parametri.
- Un linguaggio funzionale reale (tipo ML) ha in più i tipi, il pattern-matching e le eccezioni.

9.6 Le regole di scoping

Dato il dominio *eval* così definito

```
type eval = Int of int | Bool of Bool | Unbound | Funval of efun
```

consideriamo le due seguenti possibilità per la definizione, in semantica denotazionale, della semantica di `Fun`.

```
type efun = eval list -> eval
let rec sem (e:exp) (r:eval env) = match e with
  ...
  | Fun(ii,aa) ->
   Funval(function d -> sem aa (bindlist(r,ii,d)))
```

Nel primo caso la definizione di *efun* e la corrispondente semantica dell'astrazione mostrano che il corpo della funzione viene valutato nell'ambiente ottenuto legando i parametri formali ai valori dei parametri attuali, che saranno noti al momento dell'applicazione, nell'ambiente $\boldsymbol{r}$ che è quello **in cui viene valutata l'astrazione**.
In questo caso siamo in presenza di **scoping statico** (lessicale): l'ambiente non locale della funzione è quello in cui viene valutata l'astrazione.

```
type efun = eval env -> eval list -> eval
let rec sem (e:exp) (r:eval env) = match e with
  ...
  | Fun(ii,aa) ->
   Funval(function x -> function d -> sem aa (bindlist(x,ii,d)))
```

Nel secondo caso la definizione di *efun* e la corrispondente semantica dell'astrazione mostrano che il corpo della funzione viene valutato nell'ambiente ottenuto legando i parametri formali ai valori dei parametri attuali, che saranno noti al momento dell'applicazione, nell'ambiente $\boldsymbol{x}$ che è quello **in cui avverrà l'applicazione**.
In questo caso siamo in presenza di **scoping dinamico**: l'ambiente non locale della funzione è quello esistente al momento in cui avviene l'applicazione.

Come vedremo in seguito, fra i due meccanismi quello migliore è lo **scoping statico**, in cui l'ambiente non locale della funzione è quello esistente al momento in cui viene valutata l'astrazione.
Rispetto allo scoping dinamico, infatti, lo scoping statico risulta più affidabile poiché fornisce la possibilità di effettuare analisi statiche (rilevazione di errori a tempo di compilazione) ed ottimizzazioni a livello di implementazione.
Nel linguaggio didattico adottiamo lo scoping statico, ma nel seguito avremo modo anche di discutere lo scoping dinamico e di confrontare i due meccanismi.

9.7 Semantica denotazionale

Definiamo innanzi tutto il dominio semantico *eval*:

```
type eval =
   | Int of int
   | Bool of bool
   | Unbound
   | Funval of efun
and efun = eval list -> eval
```

Notiamo come dalla definizione di *efun* si deduca che stiamo adottando la regola di scoping statico.
Passiamo ora alla definizione della semantica vera e propria.

```
let rec sem (e:exp) (r:eval env) =
   match e with
     | Eint(n) -> Int(n)
     | Ebool(b) -> Bool(b)
     | Den(i) -> applyenv(r,i)
     | Iszero(a) -> iszero((sem a r) )
     | Eq(a,b) -> equ((sem a r) ,(sem b r) )
     | Prod(a,b) -> mult ( (sem a r), (sem b r))
     | Sum(a,b) -> plus ( (sem a r), (sem b r))
     | Diff(a,b) -> diff ( (sem a r), (sem b r))
     | Minus(a) -> minus( (sem a r))
     | And(a,b) -> et ( (sem a r), (sem b r))
     | Or(a,b) -> vel ( (sem a r), (sem b r))
     | Not(a) -> non( (sem a r))
     | Ifthenelse(a,b,c) ->
        let g = sem a r in
         if typecheck(``bool'',g) then
           (if g = Bool(true)
           then sem b r
           else sem c r)
         else failwith (``nonboolean guard'')
     | Let(i,e1,e2) -> sem e2 (bind (r ,i, sem e1 r))
     | Fun(i,a) -> makefun(Fun(i,a), r)
     | Appl(a,b) -> applyfun(sem a r, semlist b r)
     | Rec(f,e) -> makefunrec (f,e,r)

and semlist el r = match el with
   | [] -> []
   | e::el1 -> (sem e r) ::  (semlist el1 r)

and makefun ((a:exp),(x:eval env)) = (*1*)
   (match a with
     | Fun(ii,aa) ->
              Funval(function d -> sem aa (bindlist (x, ii, d)))
     | _ -> failwith (``Non-functional object''))
```

```
and applyfun ((ev1:eval),(ev2:eval list)) =
  ( match ev1 with
     | Funval(x) -> x ev2
     | _ -> failwith (``attempt to apply a non-functional object''))

and makefunrec (i, Fun(ii, aa), r) = (*2*)
  let functional ff d =
   let r1 = bind(bindlist(r, ii, d), i, Funval(ff)) in
     sem aa r1 in
  let rec fix = function x -> functional fix x (*3*)
  in Funval(fix)
```

Commenti al codice:

1. La semantica relativa alle funzioni è praticamente tutta nella `makefun`. L'applicazione (`applyfun`) è decisamente più semplice.
2. Cerchiamo di capire perchè è stato necessario aggiungere dei costrutti apposta per gestire la ricorsione. In generale la definizione di una funzione ricorsiva è:

   $$\texttt{Let}(\texttt{i}, \texttt{e1}, \texttt{e2})$$

   in cui `i` è il nome della funzione (ricorsiva) ed `e1` è un'astrazione (`Fun(ii,aa)`) nel cui corpo (`aa`) c'è un'applicazione di `Den i`.
   Esaminando la semantica dei costrutti `Let`, `Fun` ed `Appl` vediamo che:

   - Il corpo (che include `Den i`) è valutato in un ambiente che è lo stesso in cui si valutano le espressioni `Let` e `Fun`, esteso con le associazioni per i parametri formali (`ii`).
   - Tale ambiente non contiene l'associazione tra la funzione ed il suo nome.
   - La semantica di `Den i` restituisce `Unbound`.

   Risulta quindi necessaria l'introduzione di un nuovo costrutto per "dichiarare" (come il `let rec` in ML) o per definire funzioni ricorsive (`Rec`).
3. La funzione `fix` ottenuta dal calcolo di punto fisso, valuta la semantica del corpo in un ambiente in cui è già stata inserita l'associazione tra nome e funzione.

9.8 Semantica operazionale

Nel passaggio da semantica denotazionale a semantica operazionale dobbiamo, oltre a cambiare il tipo di *sem* per portarla al primo ordine, modificare il dominio *efun*. In semantica denotazionale, infatti, *efun* è di ordine superiore poiché nell'astrazione costruiamo funzioni parametriche.

In semantica denotazionale: `efun = eval list -> eval`
In semantica operazionale: `efun = exp * eval env`

Per eliminare le funzioni da *efun* dobbiamo dunque differire la chiamata ricorsiva di *sem* al momento dell'applicazione, facendo in modo, al momento dell'astrazione, di

conservare l'informazione sintattica (codice della espressione `Fun`) "impaccandola" in una **chiusura** insieme all'ambiente (necessario perchè lo scoping è statico).
Dato che *efun* non è più funzionale devo anche cambiare il modo di calcolare il punto fisso.
Vediamo, innanzi tutto, come cambiano i domini semantici:

```
type eval =
   | Int of int
   | Bool of bool
   | Unbound
   | Funval of efun
and efun = exp * (eval env)
```

Definiamo quindi la semantica operazionale del linguaggio funzionale con le funzioni:

```
let rec sem ((e:exp), (r:eval env)) =
   match e with
     | Eint(n) -> Int(n)
     | Ebool(b) -> Bool(b)
     | Den(i) -> applyenv(r,i)
     | Iszero(a) -> iszero(sem(a, r))
     | Eq(a,b) -> equ(sem(a, r), sem(b, r))
     | Prod(a,b) -> mult (sem(a, r), sem(b, r))
     | Sum(a,b) -> plus (sem(a, r), sem(b, r))
     | Diff(a,b) -> diff (sem(a, r), sem(b, r))
     | Minus(a) -> minus(sem(a, r))
     | And(a,b) -> et (sem(a, r), sem(b, r))
     | Or(a,b) -> vel (sem(a, r), sem(b, r))
     | Not(a) -> non(sem(a, r))
     | Ifthenelse(a,b,c) ->
        let g = sem(a, r) in
          if typecheck(``bool'',g) then
            (if g = Bool(true)
            then sem(b, r)
            else sem(c, r))
          else failwith (``nonboolean guard'')
     | Let(i,e1,e2) -> sem(e2, bind (r ,i, sem(e1, r)))
     | Fun(i,a) -> makefun(Fun(i,a), r) (*1*)
     | Appl(a,b) -> applyfun(sem(a, r), semlist(b, r))
     | Rec(f,e) -> makefunrec (f,e,r)

and semlist(el, r) = match el with
   | [] -> []
   | e::el1 -> sem(e, r) ::  (semlist(el1, r))

and makefun ((a:exp),(x:eval env)) =
   (match a with
     | Fun(ii,aa) ->
```

```
            Funval(a,x)
    | _ -> failwith (``Non-functional object''))

and applyfun ((ev1:eval),(ev2:eval list)) =
   ( match ev1 with
      | Funval(Fun(ii,aa),r) -> sem(aa, bindlist( r, ii, ev2))
      | _ -> failwith (``attempt to apply a non-functional object''))

and makefunrec (i, e1, (r:eval env)) =
   let functional (rr:  eval env) = (*2*)
    bind(r, i, makefun(e1,rr)) in
   let rec rfix = function x -> functional rfix x
   in makefun(e1, rfix)
```

Commenti al codice:

1. In questo caso nel astrazione non facciamo altro che salvare la definizione della funzione (sintassi) e l'ambiente attuale (**chiusura**).
2. Questa volta il punto fisso lo calcoliamo sull'ambiente (che pure è una funzione). I due metodi usati sono entrambi validi.

9.9 Eliminare la ricorsione

Non servono strutture dati aggiuntive rispetto a quelle introdotte per la gestione dei blocchi; l'applicazione di funzione non fa altro che creare un nuovo frame anzichè fare una chiamata ricorsiva a *sem*.
La pila dei record di attivazione sarà quindi realizzata attraverso tre pile gestite "in parallelo":

- `envstack`: pila di ambienti.
- `cstack`: pila di pile di espressioni etichettate.
- `tempvalstack`: pila di pile di *eval*.

Introduciamo due nuove operazioni per inserire sulla pila sintattica una lista di espressioni etichettate (argomenti da valutare nell'applicazione) e per prelevare dalla pila dei temporanei una lista di *eval* (argomenti valutati nell'applicazione)

9.10 L'interprete iterativo

Definiamo innanzi tutto strutture dati ed operazioni usate dall'interprete iterativo:

```
type labeledconstruct =
   | Expr1 of exp
   | Expr2 of exp

let (cstack:  labeledconstruct stack stack) =
   emptystack(stacksize,emptystack(1,Expr1(Eint(0))))

let (tempvalstack:  eval stack stack) =
   emptystack(stacksize,emptystack(1,Unbound))
```

```
let (envstack:  eval env stack) =
   emptystack(stacksize,emptyenv(Unbound))

let pushenv(r) = push(r,envstack)

let topenv() = top(envstack)

let svuotaenv() = svuota(envstack)

let popenv () = pop(envstack)

let pushargs ((b:  exp list),continuation) = let br = ref(b) in
   while not(!br = []) do
     push(Expr1(List.hd !br),continuation);
     br := List.tl !br
   done

let getargs ((b:  exp list),(tempstack:  eval stack)) = let br = ref(b)
in
let er = ref([]) in
   while not(!br = []) do
     let arg=top(tempstack) in
       pop(tempstack); er := !er @ [arg];
       br := List.tl !br
   done;
   !er

let newframes(e,rho) =
   let cframe = emptystack(cframesize(e),Expr1(e)) in
   let tframe = emptystack(tframesize(e),Unbound) in
   push(Expr1(e),cframe);
   push(cframe,cstack);
   push(tframe,tempvalstack);
   pushenv(rho)

let makefun ((a:exp),(x:eval env)) =
      (match a with
      | Fun(ii,aa) ->
             Funval(a,x)
      | _ -> failwith (``Non-functional object''))

let applyfun ((ev1:eval),(ev2:eval list)) =
      ( match ev1 with
      | Funval(Fun(ii,aa),r) -> newframes(aa,bindlist(r, ii, ev2))
      | _ -> failwith (``attempt to apply a non-functional object''))

let makefunrec (i,e1,r) = let functional rr =
          bind(r, i, makefun(e1,rr)) in
```

```
          let rec rfix = function x -> functional rfix x
             in makefun(e1, rfix)
```

Come detto in precedenza, abbiamo aggiunto le funzioni `pushargs` e `getargs`. Definiamo adesso l'interprete iterativo per il linguaggio funzionale con le funzioni:

```
let sem ((e:exp), (r:eval env)) =
  push(emptystack(1,Unbound),tempvalstack);
  newframes(e,r);
  while not(empty(cstack)) do
   while not(empty(top(cstack))) do
    let continuation = top(cstack) in
    let tempstack = top(tempvalstack) in
    let rho = topenv() in
     (match top(continuation) with
      |Expr1(x) ->
         (pop(continuation);
         push(Expr2(x),continuation);
         (match x with
            | Iszero(a) -> push(Expr1(a),continuation)
            | Eq(a,b) -> push(Expr1(a),continuation);
              push(Expr1(b),continuation)
            | Prod(a,b) -> push(Expr1(a),continuation);
              push(Expr1(b),continuation)
            | Sum(a,b) -> push(Expr1(a),continuation);
              push(Expr1(b),continuation)
            | Diff(a,b) -> push(Expr1(a),continuation);
              push(Expr1(b),continuation)
            | Minus(a) -> push(Expr1(a),continuation)
            | And(a,b) -> push(Expr1(a),continuation);
              push(Expr1(b),continuation)
            | Or(a,b) -> push(Expr1(a),continuation);
              push(Expr1(b),continuation)
            | Not(a) -> push(Expr1(a),continuation)
            | Ifthenelse(a,b,c) -> push(Expr1(a),continuation)
            | Let(i,e1,e2) -> push(Expr1(e1),continuation)
            | Appl(a,b) -> push(Expr1(a),continuation);
              pushargs(b,continuation)
            | _ -> ()))
      |Expr2(x) ->
         (pop(continuation);
         (match x with
            | Eint(n) -> push(Int(n),tempstack)
            | Ebool(b) -> push(Bool(b),tempstack)
            | Den(i) -> push(applyenv(rho,i),tempstack)
            | Iszero(a) ->
               let arg=top(tempstack) in
                pop(tempstack);
                push(iszero(arg),tempstack)
```

```
| Eq(a,b) ->
   let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
     pop(tempstack);
     push(equ(firstarg,sndarg),tempstack)
| Prod(a,b) ->
   let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
     pop(tempstack);
     push(mult(firstarg,sndarg),tempstack)
| Sum(a,b) ->
   let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
     pop(tempstack);
     push(plus(firstarg,sndarg),tempstack)
| Diff(a,b) ->
   let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
     pop(tempstack);
     push(diff(firstarg,sndarg),tempstack)
| Minus(a) ->
   let arg=top(tempstack) in
    pop(tempstack);
    push(minus(arg),tempstack)
| And(a,b) ->
   let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
     pop(tempstack);
     push(et(firstarg,sndarg),tempstack)
| Or(a,b) ->
   let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
     pop(tempstack);
     push(vel(firstarg,sndarg),tempstack)
| Not(a) ->
   let arg=top(tempstack) in
    pop(tempstack);
    push(non(arg),tempstack)
| Ifthenelse(a,b,c) ->
   let arg=top(tempstack) in
    pop(tempstack);
    if typecheck(``bool'',arg) then
     (if arg = Bool(true)
```

```
                    then push(Expr1(b),continuation)
                    else push(Expr1(c),continuation))
                  else failwith (``type error'')
              | Fun(i,a) -> push(makefun(Fun(i,a),rho),tempstack)
              | Rec(f,e) -> push(makefunrec(f,e,rho),tempstack)
              | Let(i,e1,e2) -> let arg=top(tempstack) in
                 pop(tempstack); newframes(e2,bind(rho, i, arg))
              | Appl(a,b) -> let firstarg=top(tempstack) in
                 pop(tempstack);
                 let sndarg=getargs(b,tempstack) in
                  applyfun(firstarg,sndarg))))
  done;
  let valore= top(top(tempvalstack)) in
   pop(top(tempvalstack));
   popenv();
   pop(cstack);
   pop(tempvalstack);
   push(valore,top(tempvalstack));
done;
let valore = top(top(tempvalstack)) in
 pop(top(tempvalstack));
 pop(tempvalstack); valore
```

All'interprete definito manca ancora una vera e propria implementazione del dominio ambiente per poter essere un interprete reale. Nell'implementazione attuale abbiamo una pila di ambienti relativi alle varie attivazioni, in cui ogni ambiente è l'ambiente complessivo rappresentato da una funzione.
In una implementazione reale ogni attivazione dovrebbe avere l'ambiente locale, "implementato" al prim'ordine con una struttura dati, più un modo per poter risalire all'ambiente esterno visibile.
Troveremo una soluzione simile per il linguaggio imperativo con sottoprogrammi, dove il discorso riguarderà anche l'implementazione mediante strutture dati della memoria.
Vedremo tali implementazioni in seguito (vedi 13).

9.11 Digressione sullo scoping dinamico

In caso di scoping dinamico, come già detto, avremmo avuto:

```
type efun = eval env -> eval list -> eval
let rec sem (e:exp) (r:eval env) = match e with
  ...
  | Fun(ii,aa) ->
   Funval(function x -> function d -> sem aa (bindlist(x,ii,d)))
```

L'ambiente non locale delle funzione è quindi quello esistente al momento in cui avviene l'applicazione.
In questo caso cambiano (sia in semantica denotazionale che operazionale) `efun`, `makefun` e `applyfun`, e si semplifica il trattamento della ricorsione.

9.12 Semantica denotazionale con scoping dinamico

Il dominio semantico *eval* cambia in accordo con quanto detto:

```
type eval =
   | Int of int
   | Bool of bool
   | Unbound
   | Funval of efun
and efun = eval env -> eval list -> eval
```

Notiamo come dalla definizione di *efun* si deduca che stiamo adottando la regola di **scoping dinamico**.
Passiamo ora alla definizione della semantica vera e propria.

```
let rec sem (e:exp) (r:eval env) =
   match e with
     | Eint(n) -> Int(n)
     | Ebool(b) -> Bool(b)
     | Den(i) -> applyenv(r,i)
     | Iszero(a) -> iszero((sem a r) )
     | Eq(a,b) -> equ((sem a r) ,(sem b r) )
     | Prod(a,b) -> mult ( (sem a r), (sem b r))
     | Sum(a,b) -> plus ( (sem a r), (sem b r))
     | Diff(a,b) -> diff ( (sem a r), (sem b r))
     | Minus(a) -> minus( (sem a r))
     | And(a,b) -> et ( (sem a r), (sem b r))
     | Or(a,b) -> vel ( (sem a r), (sem b r))
     | Not(a) -> non( (sem a r))
     | Ifthenelse(a,b,c) ->
        let g = sem a r in
         if typecheck(``bool'',g) then
           (if g = Bool(true)
           then sem b r
           else sem c r)
         else failwith (``nonboolean guard'')
     | Let(i,e1,e2) -> sem e2 (bind (r ,i, sem e1 r))
     | Fun(i,a) -> makefun(Fun(i,a)) (*1*)
     | Appl(a,b) -> applyfun(sem a r, semlist b r, r)
     | Rec(f,e) -> makefunrec (f,e,r)

and semlist el r = match el with
   | [] -> []
   | e::el1 -> (sem e r) ::  (semlist el1 r)

and makefun ((a:exp)) = (*2*)
   (match a with
     | Fun(ii,aa) ->
        Funval(function x -> function d -> sem aa (bindlist (x, ii, d)))
     | _ -> failwith (``Non-functional object''))
```

```
and applyfun ((ev1:eval),(ev2:eval list),(r:eval env)) = (*3*)
  ( match ev1 with
     | Funval(x) -> x r ev2
     | _ -> failwith (``attempt to apply a non-functional object''))
       (*4*)
```

Commenti al codice:

1. Non passiamo più l'ambiente alla `makefun`.
2. La `makefun` non prende più in ingresso l'ambiente, che sarà invece preso al momento dell'applicazione e costruisce un *Funval* parametrico.
3. La `applyfun` prende in ingresso anche l'ambiente nel quale verrà valutata la funzione.
4. Rispetto allo scoping statico non è stato necessario aggiungere dei costrutti appositi (`Rec`) per le funzioni ricorsive. Per comprenderne il motivo prendiamo in considerazione la definizione di una funzione ricorsiva è:

   $$\texttt{Let}(\texttt{i}, \texttt{e1}, \texttt{e2})$$

   in cui `i` è il nome della funzione (ricorsiva) ed `e1` è un'astrazione (`Fun(ii,aa)`) nel cui corpo (`aa`) c'è un'applicazione di `Den i`.
   Esaminando la semantica dei costrutti `Let`, `Fun` ed `Appl` vediamo che:

   - Il corpo (che include `Den i`) è valutato in un ambiente che è lo stesso in cui si valuta la `Appl` ricorsiva, esteso con le associazioni per i parametri formali (`ii`).
   - Tale ambiente contiene già l'associazione tra la funzione ed il suo nome, perchè la `Appl` ricorsiva viene eseguita in un ambiente in cui ho già inserito, nell'ordine, le seguenti associazioni: nome della funzione (`i`), parametri formali della prima chiamata (`ii`).

   La ricorsione si ottiene dunque "gratuitamente" senza bisogno di un costrutto apposta.

9.13 Semantica operazionale con scoping dinamico

Come già fatto in caso di scoping statico, nel passaggio da semantica denotazionale ad operazionale dobbiamo, oltre che che cambiare il tipo della funzione *sem*, anche modificare il dominio *efun* con l'obiettivo di eliminare i valori di ordine superiore.

In semantica denotazionale: `efun = eval env -> eval list -> eval`
In semantica operazionale: `efun = exp`

Come in precedenza, la soluzione consiste nel differire la chiamata ricorsiva di *sem* al momento dell'applicazione, limitandoci a conservare l'informazione sintattica al momento dell'astrazione.
Non sono necessarie altre modifiche perchè non c'è trattamento speciale della ricorsione (calcolo del punto fisso).
Vediamo, dunque, come cambiano i domini semantici:

```
type eval =
   | Int of int
   | Bool of bool
   | Unbound
   | Funval of efun
and efun = exp
```

Definiamo infine la semantica operazionale con la regola di scoping dinamico:

```
let rec sem ((e:exp), (r:eval env)) =
   match e with
     | Eint(n) -> Int(n)
     | Ebool(b) -> Bool(b)
     | Den(i) -> applyenv(r,i)
     | Iszero(a) -> iszero(sem(a, r))
     | Eq(a,b) -> equ(sem(a, r), sem(b, r))
     | Prod(a,b) -> mult (sem(a, r), sem(b, r))
     | Sum(a,b) -> plus (sem(a, r), sem(b, r))
     | Diff(a,b) -> diff (sem(a, r), sem(b, r))
     | Minus(a) -> minus(sem(a, r))
     | And(a,b) -> et (sem(a, r), sem(b, r))
     | Or(a,b) -> vel (sem(a, r), sem(b, r))
     | Not(a) -> non(sem(a, r))
     | Ifthenelse(a,b,c) ->
        let g = sem(a, r) in
         if typecheck(``bool'',g) then
           (if g = Bool(true)
           then sem(b, r)
           else sem(c, r))
         else failwith (``nonboolean guard'')
     | Let(i,e1,e2) -> sem(e2, bind (r ,i, sem(e1, r)))
     | Fun(i,a) -> makefun(Fun(i,a))
     | Appl(a,b) -> applyfun(sem(a, r), semlist(b, r), r)

and semlist(el, r) = match el with
   | [] -> []
   | e::el1 -> sem(e, r) ::  (semlist(el1, r))

and makefun ((a:exp)) =
   (match a with
     | Fun(ii,aa) ->
        Funval(a)
     | _ -> failwith (``Non-functional object''))

and applyfun ((ev1:eval),(ev2:eval list),(r:eval env)) =
   ( match ev1 with
      | Funval(Fun(ii,aa)) -> sem(aa, bindlist( r, ii, ev2))
      | _ -> failwith (``attempt to apply a non-functional object''))
```

9.14 Interprete iterativo con scoping dinamico

L'interprete iterativo rimane praticamente uguale rispetto al caso dello scoping statico. Poiché cambiano `makefun` ed `applyfun`, infatti, cambia solo il modo di invocarle nella "seconda passata" dell'interprete.

```
let rec makefun ((a:exp)) =
   (match a with
     | Fun(ii,aa) ->
        Funval(a)
     | _ -> failwith (``Non-functional object''))

and applyfun ((ev1:eval),(ev2:eval list),(r:eval env)) =
   ( match ev1 with
      | Funval(Fun(ii,aa)) -> sem(aa, bindlist( r, ii, ev2))
      | _ -> failwith (``attempt to apply a non-functional object''))


let sem ((e:exp), (r:eval env)) =
   ...
     |Expr2(x) -> (pop(continuation); (match x with
       | ...
       | Fun(i,a) -> push(makefun(Fun(i,a)), tempstack)
       | ...
       | Appl(a,b) -> let firstarg=top(tempstack) in pop(tempstack);
         let sndarg=getargs(b,tempstack) in
           applyfun(firstarg,sndarg, rho)
       | ...
     done; ...
```

9.15 Scoping statico e dinamico

La differenza tra le due regole riguarda l'**ambiente non locale**, ovvero l'insieme di associazione che nel corpo di una funzione (o di un blocco) sono visibili, e quindi utilizzabili, pur appartenendo all'ambiente locali di altri blocchi o funzioni.

Per le funzioni con **scoping statico**

- L'ambiente non locale è quello in cui occorre l'astrazione funzionale, determinato dalla struttura sintattica di annidamento di blocchi (`Let`) ed astrazioni (`Fun` e `Rec`).
- Vengono "ereditate" tutte le associazioni, per nomi che non vengono ridefiniti, in blocchi e astrazioni più interni nella struttura sintattica.
- Un riferimento non locale al nome `x` nel corpo di un blocco o di una funzione `e` viene risolto con la (eventuale) associazione per `x` creata nel blocco o astrazione più interni fra quelli che sintatticamente "contengono" `e`.

Per le funzioni con **scoping dinamico**

- L'ambiente non locale è quello in cui occorre l'applicazione di funzione, determinato dalla struttura a runtime di valutazione di blocchi (`Let`) ed applicazioni (`Apply`).
- Vengono "ereditate" tutte le associazioni, per nomi che non vengono ridefiniti, in blocchi ed applicazioni successivi, nella sequenza di attivazioni a tempo di esecuzione.
- Un riferimento non locale al nome `x` nel corpo di un blocco o di una funzione `e` viene risolto con la (eventuale) associazione per `x` creata per ultima nella sequenza di attivazioni (a tempo di esecuzione).

In presenza del solo costrutto di blocco non c'è distinzione tra le due regole di scoping perchè non c'è distinzione tra definizione ed esecuzione (un blocco viene "eseguito" immediatamente quando lo si incontra).
Vediamo, nei due casi, che tipo di possibilità offre il linguaggio in termini di **analisi** ed **ottimizzazioni**:

In caso di **scoping statico**:

- Guardando la struttura sintattica siamo in grado di verificare se l'associazione per uno specifico nome esiste, identificare la dichiarazione (o il parametro formale) rilevanti e quindi conoscere l'eventuale *informazione sul tipo*.
- Determinare "staticamente" a tempo di compilazione: gli *errori di nome*, fare il controllo di tipo e, quindi, rilevare gli *errori di tipo*.
- Il compilatore può ottimizzare l'implementazione al prim'ordine dell'ambiente (che non abbiamo ancora visto) agendo sia sulla struttura dati che lo implementa, sia sull'algoritmo che permette di risalire da un nome all'entità denotata.

In caso di **scoping dinamico**:

- L'esistenza di una associazione per uno specifico nome (ed il tipo corrispondente) dipendono dalla particolare sequenza di attivazioni.
- Due applicazioni della stessa funzione, che utilizza un'associazione non locale, possono portare a risultati diversi, con l'effetto che gli errori di nome si possono rilevare solo a *tempo di esecuzione* e non è possibile fare il controllo dei tipi.
- Non sono possibili ottimizzazioni sull'ambiente a tempo di compilazione.

9.16 Linguaggi e regole di scoping

Lo scoping statico è decisamente migliore di quello dinamico, e lo dimostra anche il fatto che l'unico linguaggio moderno che adotta una regola di scoping dinamico è LISP. Questo particolare spiega alcune caratteristiche di LISP, come la scarsa attenzione ai tipi ed alla loro verificabilità.
Alcuni linguaggi, invece, *non hanno regole di scoping* ed utilizzano ambiente locale *oppure* globale: non ci sono, cioè, associazioni ereditate da altri ambienti locali. E' il caso di PROLOG, FORTRAN e JAVA.
Un'altra possibilità consiste nell'avere soltanto ambiente locale ed ambiente non locale con scoping statico, ma comporta problemi rispetto alla modularità ed alla

compilazione separata (PASCAL).
La **soluzione migliore** è rappresentata quindi da: ambiente locale, ambiente non locale con scoping statico ed ambiente globale basato su un meccanismo di moduli.

10 Sottoprogrammi in linguaggi imperativi

Contenut del capitolo:

- Introduzione delle procedure nel linguaggio imperativo: astrazione, chiamata, regole di scoping.
- Semantica delle procedure con scoping statico: denotazionale, operazionale, iterativa.
- Digressione sullo scoping dinamico e relative semantiche.

10.1 Introduzione delle procedure nel linguaggio

Il linguaggio imperativo include totalmente il linguaggio funzionale, inclusi i costrutti `Fun`, `Appl` e `Rec`, definiti nel capitolo precedente.
Aggiungiamo inoltre i costrutti:

- `Proc` (espressione): si usa soltanto nelle dichiarazioni di sottoprogrammi nei blocchi, per specificare sottoprogrammi che hanno come corpo un comando. La loro invocazione non provoca la restituzione di un valore, ma la modifica dello store.
- `Call` (comando): chiamata a sottoprogrammi.

Modifichiamo anche la sintassi usata per definire i blocchi (ma vale anche per i corpi delle procedure), in modo che vi siano due distinte liste di dichiarazioni (oltre alla lista di comandi): dichiarazioni di costanti e variabili (già viste), e dichiarazioni di funzioni e procedure (viste come un insieme di dichiarazioni mutuamente ricorsive). Definiamo dunque la sintassi aggiornata per il linguaggio imperativo:

```
type ide = string
type exp =
   | Eint of int
   | Ebool of bool
   | Den of ide
   | Prod of exp * exp
   | Sum of exp * exp
   | Diff of exp * exp
   | Eq of exp * exp
   | Minus of exp
   | Iszero of exp
   | Or of exp * exp
   | And of exp * exp
   | Not of exp
   | Ifthenelse of exp * exp * exp
   | Val of exp
   | Let of ide * exp * exp
   | Newloc of exp
   | Fun of ide list * exp
   | Rec of ide * exp
   | Appl of exp * exp list
   | Proc of ide list * block
```

```
and decl = (ide * exp) list * (ide * exp) list
and block = (ide * exp) list * (ide * exp) list * com list

and com =
   | Assign of exp * exp
   | Cifthenelse of exp * com list * com list
   | While of exp * com list
   | Block of block
   | Call of exp * exp list
```

Commenti:

- Come nel caso delle funzioni, le procedure hanno una lista di parametri che sono: identificatori nel costrutto di astrazione procedurale (`Proc`), espressioni nel costrutto di chiamata (`Call`).
- Come nel caso delle funzioni assumiamo la modalità standard di passaggio dei parametri: le espressioni parametro attuale sono valutate (*dval*) ed i valori sono legati nell'ambiente al corrispondente parametro formale.
- Con l'introduzione delle procedure il linguaggio imperativo è completo.

Un linguaggio imperativo reale ha in più i tipi, le eccezioni ed eventuali meccanismi come i puntatori (che vedremo nell'estensione orientata agli oggetti).

10.2 Semantica denotazionale

Decidiamo che, a differenza delle funzioni, i valori *proc* con cui interpretiamo le procedure, siano soltanto denotabili. Questo vuol dire che le procedure possono essere *dichiarate*, *passate come parametri*, essere *chiamate* attraverso il comando `Call`, ma *non possono essere restituite* come valore di una espressione.
Definiamo quindi i domini dei valori in semantica denotazionale e le funzioni di conversione:

```
exception Nonstorable
exception Nonexpressible

type eval =
   | Int of int
   | Bool of bool
   | Novalue
   | Funval of efun
and efun = (dval list) * (mval store) -> eval
and proc = (dval list) * (mval store) -> mval store

and dval =
   | Dint of int
   | Dbool of bool
   | Unbound
   | Dloc of loc
   | Dfunval of efun
```

```
   | Dprocval of proc

and mval =
   | Mint of int
   | Mbool of bool
   | Undefined

let evaltomval e = match e with
   | Int n -> Mint n
   | Bool n -> Mbool n
   | _ -> raise Nonstorable

let mvaltoeval m = match m with
   | Mint n -> Int n
   | Mbool n -> Bool n
   | _ -> Novalue

let evaltodval e = match e with
   | Int n -> Dint n
   | Bool n -> Dbool n
   | Novalue -> Unbound
   | Funval f -> Dfunval f

let dvaltoeval e = match e with
   | Dint n -> Int n
   | Dbool n -> Bool n
   | Dloc n -> raise Nonexpressible
   | Unbound -> Novalue
   | Dfunval f -> Funval f
```

Notiamo come il dominio *efun* cambi rispetto al caso funzionale, poiché il corpo della funzione viene valutato nell'ambiente ottenuto legando i parametri formali ai valori dei parametri attuali (noti al momento dell'applicazione) nell'ambiente in cui viene valutata l'astrazione (**scoping statico**) e *nello store esistente al momento dell'applicazione.*
Il dominio *proc* è del tutto simile, e cambia solo il codominio della dunzione (*store* invece di *eval*).
Definiamo ora le funzioni usate da *sem* per l'astrazione e l'applicazione di funzioni e procedure:

```
let rec makefun ((a:exp),(x:dval env)) =
   (match a with
     | Fun(ii,aa) ->
              Dfunval(function (d,s) -> sem aa (bindlist(x, ii, d)) s)
     | _ -> failwith (``Non-functional object''))

and makeproc ((a:exp),(x:dval env)) =
   (match a with
     | Proc(ii,b) ->
```

```
        Dprocval(function (d1,s1) ->
                semb b (bindlist(x, ii, d1)) s1)
    | _ -> failwith (``Non-procedural object''))

and applyfun ((ev1:dval),(ev2:dval list),s) =
   (match ev1 with
    | Dfunval(x) -> x (ev2,s)
    | _ -> failwith (``attempt to apply a non-functional object''))

and applyproc ((ev1:dval),(ev2:dval list),s) =
   (match ev1 with
    | Dprocval(x) -> x (ev2,s)
    | _ -> failwith (``attempt to apply a non-functional object''))

and makefunrec (i, Fun(ii, aa), r) =
   let functional ff (d,s1) =
    let r1 = bind(bindlist(r, ii, d), i, Dfunval(ff)) in
      sem aa r1 s1 in
   let rec fix = function x -> functional fix x
   in Funval(fix)
```

Non è stato necessario definire una funzione `makeprocrec` perchè le procedure sono automaticamente mutuamente ricorsive.
Definiamo quindi la funzione di valutazione semantica per il linguaggio imperativo con procedure:

```
and sem (e:exp) (r:dval env) (s:  mval store) =
   match e with
    | Eint(n) -> Int(n)
    | Ebool(b) -> Bool(b)
    | Den(i) -> dvaltoeval(applyenv(r,i))
    | Iszero(a) -> iszero((sem a r s) )
    | Eq(a,b) -> equ((sem a r s) ,(sem b r s) )
    | Prod(a,b) -> mult ( (sem a r s), (sem b r s))
    | Sum(a,b) -> plus ( (sem a r s), (sem b r s))
    | Diff(a,b) -> diff ( (sem a r s), (sem b r s))
    | Minus(a) -> minus( (sem a r s))
    | And(a,b) -> et ( (sem a r s), (sem b r s))
    | Or(a,b) -> vel ( (sem a r s), (sem b r s))
    | Not(a) -> non( (sem a r s))
    | Ifthenelse(a,b,c) ->
       let g = sem a r s in
        if typecheck(``bool'',g) then
          (if g = Bool(true)
          then sem b r s
          else sem c r s)
        else failwith (``nonboolean guard'')
    | Fun(i,a) -> dvaltoeval(makefun(Fun(i,a), r))
    | Appl(a,b) -> let (v1,s1) = semlist b r s in
```

```
       applyfun(evaltodval(sem a r s), v1, s)
    | Let(i,e1,e2) -> let (v,s1) = semden e1 r s in
       sem e2 (bind(r, i, v)) s
    | Rec(f,e) -> makefunrec (f,e,r)
    | Val(e) -> let (v, s1) = semden e r s in
       (match v with
         | Dloc n -> mvaltoeval(applystore(s1, n))
         | _ -> failwith(``not a variable''))
    | _ -> failwith (``nonlegal expression for sem'')

and semlist el r s = match el with
   | [] -> ([], s)
   | e::el1 -> let (v1, s1) = semden e r s in
    let (v2, s2) = semlist el1 r s1 in
      (v1 ::  v2, s2)

and semden (e:exp) (r:dval env) (s:  mval store) = match e with
   | Den(i) -> (applyenv(r,i), s)
   | Fun(i,e1) -> (makefun(e,r), s)
   | Proc(il,b) -> (makeproc(e,r), s)
   | Newloc(e) -> let m = evaltomval(sem e r s) in
    let (l, s1) = allocate(s, m) in (Dloc l, s1)
   | _ -> (evaltodval(sem e r s), s)

and semdv dl r s =
   match dl with
    | [] -> (r,s)
    | (i,e)::dl1 -> let (v, s1) = semden e r s in
       semdv dl1 (bind(r, i, v)) s1
and semdl (dl,rl) r s =
   let (r1, s1) = semdv dl r s in
    semdr rl r1 s1

and semdr rl r s = (*1*)
   let functional ((r1:  dval env)) =
    (match rl with
      | [] -> r
      | (i,e) ::  rl1 -> let (v, s2) = semden e r1 s in
       let (r2, s3) = semdr rl1 (bind(r, i, v)) s in
        r2) in
   let rec rfix = function x -> functional rfix x in
    (rfix, s)

and semc (c:  com) (r:dval env) (s:  mval store) = match c with
   | Assign(e1, e2) -> let (v1, s1) = semden e1 r s in
      (match v1 with
       | Dloc(n) -> update(s, n, (evaltomval(sem e2 r s)))
       | _ -> failwith (``wrong location in assignment''))
   | Cifthenelse(e, cl1, cl2) -> let g = sem e r s in
```

```
      if typecheck(``bool'',g) then
        (if g = Bool(true)
        then semcl cl1 r s
        else semcl cl2 r s)
      else failwith (``nonboolean guard'')
   | While(e, cl) ->
      let functional ((fi:  mval store -> mval store)) =
        function sigma ->
          let g = sem e r sigma in
            if typecheck(``bool'',g) then
              (if g = Bool(true)
              then fi(semcl cl r sigma)
              else sigma)
            else failwith (``nonboolean guard'') in
      let rec ssfix = function x -> functional ssfix x in
        ssfix(s)
   | Call(e1, e2) -> let (p, s1) = semden e1 r s in
     let (v, s2) = semlist e2 r s1 in
       applyproc(p, v, s2)
   | Block(b) -> semb b r s

and semcl cl r s = match cl with
   | [] -> s
   | c::cl1 -> semcl cl1 r (semc c r s)

and semb (dl, rdl, cl) r s =
   let (r1, s1) = semdl (dl,rdl) r s in
     semcl cl r1 s1
```

Le funzioni di valutazione semantica definite hanno il seguente tipo:

```
val sem :  exp -> dval env -> mval store -> eval = <fun>

val semden :  exp -> dval env -> mval store -> dval * mval store = <fun>

val semlist :  exp list -> dval env -> mval store ->
    (dval list) * mval store = <fun>

val semc :  com -> dval env -> mval store -> mval store = <fun>

val semcl :  com list -> dval env -> mval store -> mval store = <fun>

val semb :  (decl * com list) -> dval env -> mval store ->
    mval store = <fun>

val semdl :  decl -> dval env -> mval store ->
    dval env * mval store = <fun>

val semdv :  (ide * expr) list -> dval env -> mval store ->
```

```
   dval env * mval store = <fun>

val semdr :  (ide * expr) list -> dval env -> mval store ->
   dval env * mval store = <fun>
```

Commenti:

1. La semantica delle dichiarazioni mutuamente ricorsive è costruita mediante un punto fisso sull'ambiente (che deve essere una funzione e quindi non può essere astratto).

10.3 Semantica operazionale

Nel passaggio da semantica denotazionale a semantica operazionale cambiano, come al solito, i tipi di tutte le funzioni di valutazione sematica ed inoltre cambiano i domini *proc* ed *efun*:

efun diventa: `efun = exp * dval env`
proc diventa: `proc = exp * dval env`

Per eliminare la funzioni da `efun` e da `proc` dobbiamo differire le chiamate ricorsive di *sem* e *semb* al momento della chiamata, limitandoci, al momento dell'astrazione, ad "impaccare" sintassi (`Fun` o `Proc`) insieme all'ambiente corrente in una *chiusura* (*scoping statico*).
Ecco la semantica operazionale per il linguaggio imperativo con le proedure:

```
type eval =
   | Int of int
   | Bool of bool
   | Novalue
   | Funval of efun
and efun = exp * (dval env)
and proc = exp * (dval env)

let rec makefun ((a:exp),(x:dval env)) =
   (match a with
    | Fun(ii,aa) ->
        Dfunval(a, x)
    | _ -> failwith (``Non-functional object''))

and makeproc ((a:exp),(x:dval env)) =
   (match a with
    | Proc(ii,b) ->
        Dprocval(a, x)
    | _ -> failwith (``Non-procedural object''))

and applyfun ((ev1:dval),(ev2:dval list),s) =
   ( match ev1 with
      | Dfunval(Fun(ii,aa), x) -> sem(aa, bindlist(x, ii, ev2), s)
      | _ -> failwith (``attempt to apply a non-functional object''))
```

```
and applyproc ((ev1:dval),(ev2:dval list),s) =
   ( match ev1 with
      | Dprocval(Proc(ii,b),x) -> semb(b, bindlist(x, ii, ev2), s)
      | _ -> failwith (``attempt to apply a non-functional object''))

and makefunrec (i, e1, r) =
   let functional (rr:  dval env) =
    bind(r, i, makefun(e1,rr)) in
   let rec rfix = function x -> functional rfix x
   in dvaltoeval(makefun(e1, rfix))

and sem ((e:exp), (r:dval env), (s:  mval store)) =
   match e with
     | Eint(n) -> Int(n)
     | Ebool(b) -> Bool(b)
     | Den(i) -> dvaltoeval(applyenv(r,i))
     | Iszero(a) -> iszero(sem(a, r, s))
     | Eq(a,b) -> equ(sem(a, r, s), sem(b, r, s))
     | Prod(a,b) -> mult (sem(a, r, s), sem(b, r, s))
     | Sum(a,b) -> plus (sem(a, r, s), sem(b, r, s))
     | Diff(a,b) -> diff (sem(a, r, s), sem(b, r, s))
     | Minus(a) -> minus(sem(a, r, s))
     | And(a,b) -> et (sem(a, r, s), sem(b, r, s))
     | Or(a,b) -> vel (sem(a, r, s), sem(b, r, s))
     | Not(a) -> non(sem(a, r, s))
     | Ifthenelse(a,b,c) ->
        let g = sem(a, r, s) in
          if typecheck(``bool'',g) then
            (if g = Bool(true)
            then sem(b, r, s)
            else sem(c, r, s))
          else failwith (``nonboolean guard'')
     | Fun(i,a) -> dvaltoeval(makefun(Fun(i,a), r))
     | Appl(a,b) -> let (v1,s1) = semlist(b, r, s) in
        applyfun(evaltodval(sem(a, r, s)), v1, s)
     | Let(i,e1,e2) -> let (v,s1) = semden(e1, r, s) in
        sem(e2 ,bind(r, i, v), s)
     | Rec(f,e) -> makefunrec (f,e,r)
     | Val(e) -> let (v, s1) = semden(e, r, s) in
        (match v with
          | Dloc n -> mvaltoeval(applystore(s1, n))
          | _ -> failwith(``not a variable''))
     | _ -> failwith (``nonlegal expression for sem'')

and semlist(el, r, s) = match el with
   | [] -> ([], s)
   | e::el1 -> let (v1, s1) = semden(e, r, s) in
     let (v2, s2) = semlist(el1, r, s1) in
```

```
      (v1 ::  v2, s2)

and semden ((e:exp), (r:dval env), (s:  mval store)) = match e with
   | Den(i) -> (applyenv(r,i), s)
   | Fun(i,e1) -> (makefun(e,r), s)
   | Proc(il,b) -> (makeproc(e,r), s)
   | Newloc(e) -> let m = evaltomval(sem(e, r, s)) in
    let (l, s1) = allocate(s, m) in (Dloc l, s1)
   | _ -> (evaltodval(sem(e, r, s)), s)

and semdv(dl, r, s) =
   match dl with
     | [] -> (r,s)
     | (i,e)::dl1 -> let (v, s1) = semden(e, r, s) in
        semdv(dl1, bind(r, i, v), s1)

and semdl ((dl,rl), r, s) =
   let (r1, s1) = semdv(dl, r, s) in
    semdr(rl, r1, s1)

and semdr(rl, r, s) =
   let functional ((r1:  dval env)) =
    (match rl with
      | [] -> r
      | (i,e) ::  rl1 -> let (v, s2) = semden(e, r1, s) in
       let (r2, s3) = semdr(rl1, bind(r, i, v), s) in
        r2) in
   let rec rfix = function x -> functional rfix x in
    (rfix, s)

and semc((c:  com), (r:dval env), (s:  mval store)) = match c with
   | Assign(e1, e2) -> let (v1, s1) = semden(e1, r, s) in
      (match v1 with
       | Dloc(n) -> update(s, n, evaltomval(sem(e2, r, s)))
       | _ -> failwith (``wrong location in assignment''))
   | Cifthenelse(e, cl1, cl2) -> let g = sem(e, r, s) in
      if typecheck(``bool'',g) then
       (if g = Bool(true)
       then semcl(cl1, r, s)
       else semcl (cl2, r, s))
      else failwith (``nonboolean guard'')
   | While(e, cl) -> let g = sem(e, r, s) in
      if typecheck(``bool'',g) then
       (if g = Bool(true) then
         semcl((cl @ [While(e, cl)]), r, s)
       else s)
      else failwith (``nonboolean guard'')
   | Call(e1, e2) -> let (p, s1) = semden(e1, r, s) in
    let (v, s2) = semlist(e2, r, s1) in
```

```
      applyproc(p, v, s2)
   | Block(b) -> semb(b, r, s)

and semcl(cl, r, s) = match cl with
   | [] -> s
   | c::cl1 -> semcl(cl1, r, semc(c, r, s))

and semb ((dl, rdl, cl), r, s) =
   let (r1, s1) = semdl((dl, rdl), r, s) in
    semcl(cl, r1, s1)
```

Le funzioni di valutazione semantica definite hanno il seguente tipo:

```
val sem :  exp * dval env * mval store -> eval = <fun>

val semden :  exp * dval env * mval store -> dval * mval store = <fun>

val semlist :  exp list * dval env * mval store ->
   (dval list) * mval store = <fun>

val semc :  com * dval env * mval store -> mval store = <fun>

val semcl :  com list * dval env * mval store -> mval store = <fun>

val semb :  (decl * com list) * dval env * mval store ->
   mval store = <fun>

val semdl :  decl * dval env * mval store ->
   dval env * mval store = <fun>

val semdv :  (ide * expr) list * dval env * mval store ->
   dval env * mval store = <fun>

val semdr :  (ide * expr) list * dval env * mval store ->
   dval env * mval store = <fun>
```

10.4 Eliminare la ricorsione

Per eliminare la ricorsione non sono necessarie strutture dati diverse da quelle introdotte per gestire i blocchi. Come in quel caso, la chiamata a procedura crea un nuovo frame anzichè fare una chiamata ricorsiva a *semb*.
Utilizziamo quindi le seguenti pile, gestite in modo "parallelo":

- `envstack`: pila di ambienti.
- `cstack`: pila di pile di costrutti sintattici etichettati.
- `tempvalstack`: pila di pile di *eval*.
- `tempdvalstack`: pila di pile di *eval*.

- `storestack`: pila di memorie.
- `labelstack`: pila di costrutti sintattici etichettati

Utilizzaimo le due operazioni introdotte nel linguaggio funzionale per inserire nella pila sintattica una lista di espressioni etichettate (`pushargs`) e per prelevare dalla pila dei temporanei una lista di *eval* (`getargs`).

10.5 Interprete iterativo

Definiamo innanzi tutto strutture dati ed operazioni usate dall'interprete iterativo:

```
type labeledconstruct =
   | Expr1 of exp
   | Expr2 of exp
   | Exprd1 of exp
   | Exprd2 of exp
   | Com1 of com
   | Com2 of com
   | Coml of labeledconstruct list
   | Decl of labeledconstruct list
   | Rdecl of (ide * exp) list
   | Dec1 of (ide * exp)
   | Dec2 of (ide * exp)

let (cstack:  labeledconstruct stack stack) =
   emptystack(stacksize,emptystack(1,Expr1(Eint(0))))

let (tempvalstack:  eval stack stack) =
   emptystack(stacksize,emptystack(1,Novalue))

let (tempdvalstack:  dval stack stack) =
   emptystack(stacksize,emptystack(1,Unbound))

let labelcom (dl:  com list) = let dlr = ref(dl) in
let ldlr = ref([]) in
   while not (!dlr = []) do
     let i = List.hd !dlr in
       ldlr := !ldlr @ [Com1(i)];
       dlr := List.tl !dlr
   done;
   Coml(!ldlr)

let labeldec (dl:  (ide * exp) list) = let dlr = ref(dl) in
let ldlr = ref([]) in
   while not (!dlr = []) do
     let i = List.hd !dlr in
       ldlr := !ldlr @ [Dec1(i)];
       dlr := List.tl !dlr
   done;
   Decl(!ldlr)
```

```
let envstack = emptystack(stacksize,(emptyenv Unbound))

let storestack = emptystack(stacksize,(emptystore Undefined))

let pushenv(r) = push(r,envstack)

let topenv() = top(envstack)

let popenv () = pop(envstack)

let svuotaenv() = svuota(envstack)

let pushstore(s) = push(s,storestack)

let popstore () = pop(storestack)

let svuotastore () = svuota(storestack)

let topstore() = top(storestack)

let (labelstack:  labeledconstruct stack) =
  emptystack(stacksize,Expr1(Eint(0)))

let pushargs((b:exp list),(continuation:labeledconstruct stack)) =
  let br = ref(b) in
   while not(!br = []) do
    push(Exprd1(List.hd !br),continuation);
    br := List.tl !br
   done

let getargs ((b:  exp list),(tempstack:  dval stack)) =
  let br = ref(b) in
  let er = ref([]) in
   while not(!br = []) do
    let arg=top(tempstack) in
     pop(tempstack); er := !er @ [arg];
     br := List.tl !br
   done;
   !er

let newframes(ss,rho,sigma) =
  pushstore(sigma);
  pushenv(rho);
  let cframe = emptystack(cframesize(ss),Expr1(Eint 0)) in
  let tframe = emptystack(tframesize(ss),Novalue) in
  let dframe = emptystack(tdframesize(ss),Unbound) in
   push(tframe,tempvalstack);
   push(dframe,tempdvalstack);
```

```
    push(ss, labelstack);
    push(ss, cframe);
    push(cframe, cstack)

let makefun ((a:exp),(x:dval env)) =
   (match a with
    | Fun(ii,aa) ->
        Dfunval(a,x)
    | _ -> failwith (``Non-functional object''))

let makeproc ((a:exp),(x:dval env)) =
   (match a with
    | Proc(ii,b) ->
        Dprocval(a,x)
    | _ -> failwith (``Non-procedural object''))

let makefunrec (i, e1, r) =
   let functional (rr:  dval env) =
    bind(r, i, makefun(e1,rr)) in
   let rec rfix = function x -> functional rfix x
   in dvaltoeval(makefun(e1, rfix))

let applyfun ((ev1:dval),(ev2:dval list), s) =
   ( match ev1 with
      | Dfunval(Fun(ii,aa),r) ->
        newframes(Expr1(aa),bindlist(r, ii, ev2), s)
      | _ -> failwith (``attempt to apply a non-functional object''))

let applyproc ((ev1:dval),(ev2:dval list), s) =
   ( match ev1 with
      | Dprocval(Proc(ii,(l1, l2, l3)), x) ->
        newframes(labelcom(l3), bindlist(x, ii, ev2), s);
        push(Rdecl(l2), top(cstack));
        push(labeldec(l1),top(cstack))
      | _ -> failwith (``attempt to apply a non-functional object''))
```

Definiamo adesso l'interprete iterativo per il linguaggio imperativo con le procedure:

```
let itsem() =
   let continuation = top(cstack) in
   let tempstack = top(tempvalstack) in
   let tempdstack = top(tempdvalstack) in
   let rho = topenv() in
   let sigma = topstore() in
    (match top(continuation) with
     |Expr1(x) ->
        (pop(continuation);
        push(Expr2(x),continuation);
```

```
    (match x with
       | Iszero(a) -> push(Expr1(a),continuation)
       | Eq(a,b) -> push(Expr1(a),continuation);
          push(Expr1(b),continuation)
       | Prod(a,b) -> push(Expr1(a),continuation);
          push(Expr1(b),continuation)
       | Sum(a,b) -> push(Expr1(a),continuation);
          push(Expr1(b),continuation)
       | Diff(a,b) -> push(Expr1(a),continuation);
          push(Expr1(b),continuation)
       | Minus(a) -> push(Expr1(a),continuation)
       | And(a,b) -> push(Expr1(a),continuation);
          push(Expr1(b),continuation)
       | Or(a,b) -> push(Expr1(a),continuation);
          push(Expr1(b),continuation)
       | Not(a) -> push(Expr1(a),continuation)
       | Ifthenelse(a,b,c) -> push(Expr1(a),continuation)
       | Val(e) -> push(Exprd1(e),continuation)
       | Newloc(e) -> failwith (``nonlegal expression for sem'')
       | Let(i,e1,e2) -> push(Exprd1(e1),continuation)
       | Appl(a,b) -> push(Expr1(a),continuation);
          pushargs(b,continuation)
       | Proc(i,b) -> failwith (``nonlegal expression for sem'')
       | _ -> ()))
  |Expr2(x) ->
    (pop(continuation);
    (match x with
       | Eint(n) -> push(Int(n),tempstack)
       | Ebool(b) -> push(Bool(b),tempstack)
       | Den(i) -> push(dvaltoeval(applyenv(rho,i)),tempstack)
       | Iszero(a) ->
          let arg=top(tempstack) in
           pop(tempstack);
           push(iszero(arg),tempstack)
       | Eq(a,b) ->
          let firstarg=top(tempstack) in
           pop(tempstack);
           let sndarg=top(tempstack) in
            pop(tempstack);
            push(equ(firstarg,sndarg),tempstack)
       | Prod(a,b) ->
          let firstarg=top(tempstack) in
           pop(tempstack);
           let sndarg=top(tempstack) in
            pop(tempstack);
            push(mult(firstarg,sndarg),tempstack)
       | Sum(a,b) ->
          let firstarg=top(tempstack) in
           pop(tempstack);
```

```
    let sndarg=top(tempstack) in
      pop(tempstack);
      push(plus(firstarg,sndarg),tempstack)
| Diff(a,b) ->
   let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
      pop(tempstack);
      push(diff(firstarg,sndarg),tempstack)
| Minus(a) ->
   let arg=top(tempstack) in
    pop(tempstack);
    push(minus(arg),tempstack)
| And(a,b) ->
   let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
      pop(tempstack);
      push(et(firstarg,sndarg),tempstack)
| Or(a,b) ->
   let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
      pop(tempstack);
      push(vel(firstarg,sndarg),tempstack)
| Not(a) ->
   let arg=top(tempstack) in
    pop(tempstack);
    push(non(arg),tempstack)
| Ifthenelse(a,b,c) ->
   let arg=top(tempstack) in
    pop(tempstack);
    if typecheck(``bool'',arg) then
      (if arg = Bool(true)
      then push(Expr1(b),continuation)
      else push(Expr1(c),continuation))
    else failwith (``type error'')
| Val(e) -> let v = top(tempdstack) in
   pop(tempdstack);
   (match v with
    | Dloc n ->
       push(mvaltoeval(applystore(sigma, n)), tempstack)
    | _ -> failwith(``not a variable''))
| Fun(i,a) ->
   push(dvaltoeval(makefun(Fun(i,a),rho)),tempstack)
| Rec(f,e) -> push(makefunrec(f,e,rho),tempstack)
| Let(i,e1,e2) -> let arg= top(tempdstack) in
   pop(tempdstack);
   newframes(Expr1(e2), bind(rho, i, arg), sigma)
```

```
            | Appl(a,b) ->
               let firstarg=evaltodval(top(tempstack)) in
                pop(tempstack);
                let sndarg=getargs(b,tempdstack) in
                  applyfun(firstarg, sndarg, sigma)
            | _ -> failwith(``no more cases for semexpr''))))
      | _ -> failwith(``no more cases for semexpr''))

let itsemden() =
   let continuation = top(cstack) in
   let tempstack = top(tempvalstack) in
   let tempdstack = top(tempdvalstack) in
   let rho = topenv() in
   let sigma = topstore() in
     (match top(continuation) with
      |Exprd1(x) ->
         (pop(continuation); push(Exprd2(x),continuation);
         match x with
           | Den i -> ()
           | Fun(i,e) -> ()
           | Proc(il,b) -> ()
           | Newloc(e) -> push(Expr1( e), continuation)
           | _ -> push(Expr1(x), continuation))
      |Exprd2(x) ->
         (pop(continuation); match x with
           | Den i -> push(applyenv(rho,i), tempdstack)
           | Fun(i,e) -> push(makefun(x,rho),tempdstack)
           | Proc(il,b) -> push(makeproc(x,rho),tempdstack)
           | Newloc(e) ->let m=evaltomval(top(tempstack)) in
              pop(tempstack); let (l, s1) = allocate(sigma, m) in
                push(Dloc l, tempdstack);
                popstore();
                pushstore(s1)
           | _ -> let arg = top(tempstack) in pop(tempstack);
              push(evaltodval(arg), tempdstack))
      | _ -> failwith(``No more cases for demden'') )

let itsemdecl () =
   let tempstack = top(tempvalstack) in
   let continuation = top(cstack) in
   let tempdstack = top(tempdvalstack) in
   let rho = topenv() in
   let sigma = topstore() in
   let dl =
     (match top(continuation) with
      | Decl(dl1) -> dl1
      | _ -> failwith(``impossible in semdecl'')) in
     if dl = [] then pop(continuation) else
      (let currd = List.hd dl in
```

```
      let newdl = List.tl dl in
       pop(continuation); push(Decl(newdl),continuation);
       (match currd with
          | Dec1( (i,e)) ->
             pop(continuation);
             push(Decl(Dec2((i, e))::newdl),continuation);
             push(Exprd1(e), continuation)
           | Dec2((i,e)) ->
             let arg = top(tempdstack) in
                pop(tempdstack);
              popenv();
              pushenv(bind(rho, i, arg))
          | _ -> failwith(``no more sensible cases for semdecl''))))

let itsemrdecl() =
   let tempstack = top(tempvalstack) in
   let continuation = top(cstack) in
   let tempdstack = top(tempdvalstack) in
   let rho = topenv() in
   let sigma = topstore() in
   let rl = (match top(continuation) with
             | Rdecl(rl1) -> rl1
             | _ -> failwith(``impossible in semrdecl'')) in
    pop(continuation);
    let functional (rr:  dval env) =
     let pr = ref(rho) in
     let prl = ref(rl) in
      while not(!prl = []) do
       let currd = List.hd !prl in
        prl := List.tl !prl;
        let (i, den) =
          (match currd with
            |(j, Proc(il,b)) -> (j, makeproc(Proc(il,b),rr))
            |(j, Fun(il,b)) -> (j, makefun(Fun(il,b),rr))
            | _ -> failwith(``no more sensible cases ...''))  in
          pr := bind(!pr, i, den)
      done;
      !pr in
    let rec rfix = function x -> functional rfix x in
     popenv();
     pushenv(rfix)

let itsemcl() =
   let tempstack = top(tempvalstack) in
   let continuation = top(cstack) in
   let tempdstack = top(tempdvalstack) in
   let rho = topenv() in
   let sigma = topstore() in
   let cl =
```

```
(match top(continuation) with
  | Coml(dl1) -> dl1
  | _ -> failwith(``impossible in semcl'')) in
if cl = [] then pop(continuation) else
  (let currc = List.hd cl in
  let newcl = List.tl cl in
    pop(continuation); push(Coml(newcl),continuation);
      (match currc with
         | Com1(Assign(e1, e2)) -> pop(continuation);
          push(Coml(Com2(Assign(e1, e2))::newcl),continuation);
          push(Exprd1(e1), continuation);
          push(Expr1(e2), continuation)
       | Com2(Assign(e1, e2)) ->
          let arg2 = evaltomval(top(tempstack)) in
           pop(tempstack);
           let arg1 = top(tempdstack) in
            pop(tempdstack);
            (match arg1 with
               | Dloc(n) -> popstore();
                  pushstore(update(sigma, n, arg2))
               | _ -> failwith (``wrong location in assignment''))
       | Com1(While(e, cl)) ->
          pop(continuation);
          push(Coml(Com2(While(e, cl))::newcl),continuation);
          push(Expr1(e), continuation)
       | Com2(While(e, cl)) ->
          let g = top(tempstack) in
           pop(tempstack);
           if typecheck(``bool'',g) then
            (if g = Bool(true) then
               (let old = newcl in
               let newl =
                 (match labelcom cl with
                   | Coml newl1 -> newl1
                   | _ -> failwith(``impossible in while'')) in
                let nuovo =
                Coml(newl @ [Com1(While(e, cl))] @ old) in
                 pop(continuation); push(nuovo,continuation))
              else ())
             else failwith (``nonboolean guard'')
       | Com1(Cifthenelse(e, cl1, cl2)) ->
          pop(continuation);
          push(Coml(Com2(Cifthenelse(e, cl1, cl2))::newcl),continuation);
          push(Expr1(e), continuation)
       | Com2(Cifthenelse(e, cl1, cl2)) ->
          let g = top(tempstack) in
           pop(tempstack);
           if typecheck(``bool'',g) then
            (let temp = if g = Bool(true) then
```

```
                labelcom (cl1) else labelcom (cl2) in
              let newl =
                (match temp with
                   | Coml newl1 -> newl1
                   | _ -> failwith(``impossible in cifthenelse'')) in
              let nuovo = Coml(newl @ newcl) in
                 pop(continuation); push(nuovo,continuation))
            else failwith (``nonboolean guard'')
        | Com1(Call(e, el)) ->
           pop(continuation);
           push(Coml(Com2(Call(e, el))::newcl),continuation);
           push(Exprd1( e), continuation);
           pushargs(el, continuation)
        | Com2(Call(e, el)) ->
           let p = top(tempdstack) in
            pop(tempdstack);
            let args = getargs(el,tempdstack) in
              applyproc(p, args, sigma)
         | Com1(Block((l1, l2, l3))) ->
           newframes(labelcom(l3), rho, sigma);
           push(Rdecl(l2), top(cstack));
           push(labeldec(l1),top(cstack))
        | _ -> failwith(``no more sensible cases in commands'') ))

let initstate() =
  svuota(cstack); svuota(tempvalstack); svuota(tempdvalstack);
  svuotaenv(); svuotastore(); svuota(labelstack)

let loop () =
  while not(empty(cstack)) do
   while not(empty(top(cstack))) do
    let currconstr = top(top(cstack)) in
     (match currconstr with
      | Expr1(e) -> itsem()
      | Expr2(e) -> itsem()
      | Exprd1(e) -> itsemden()
      | Exprd2(e) -> itsemden()
      | Coml(cl) -> itsemcl()
      | Decl(l) -> itsemdecl()
      | Rdecl(l) -> itsemrdecl()
      | _ -> failwith(``non legal construct in loop''))
   done;
   (match top(labelstack) with
    | Expr1(_) -> let valore = top(top(tempvalstack)) in
       pop(top(tempvalstack));
       pop(tempvalstack); push(valore,top(tempvalstack));
       popenv(); popstore(); pop(tempdvalstack)
    | Exprd1(_) -> let valore = top(top(tempdvalstack)) in
       pop(top(tempdvalstack));
```

```
        pop(tempdvalstack); push(valore,top(tempdvalstack));
        popenv(); popstore(); pop(tempvalstack)
      | Decl(_) ->
        pop(tempvalstack); pop(tempdvalstack)
      | Rdecl(_) ->
        pop(tempvalstack); pop(tempdvalstack)
      | Coml(_) -> let st = topstore() in
        popenv();popstore(); popstore(); pushstore(st);
        pop(tempvalstack); pop(tempdvalstack)
      | _ -> failwith(``non legal label in loop''));
    pop(cstack);
    pop(labelstack)
  done

let sem (e,(r:  dval env), (s:  mval store)) =
  initstate();
  push(emptystack(tframesize(e),Novalue),tempvalstack);
  newframes(Expr1(e), r, s);
  loop();
  let valore= top(top(tempvalstack)) in
   pop(tempvalstack);
   valore

let semden (e,(r:  dval env), (s:  mval store)) =
  initstate();
  push(emptystack(tdframesize(e),Unbound),tempdvalstack);
  newframes(Exprd1(e), r, s);
  loop();
  let valore= top(top(tempdvalstack)) in
   pop(tempdvalstack);
   valore

let semcl (cl,(r:  dval env), (s:  mval store)) =
  initstate();
  pushstore(emptystore(Undefined));
  newframes(labelcom(cl), r, s);
  loop();
  let st = topstore() in
   popstore();
   st

let semdv(dl, r, s) =
  initstate();
  newframes(labeldec(dl), r, s);
  loop();
  let st = topstore() in
   popstore();
   let rt = topenv() in
    popenv();
```

```
      (rt, st)

let semdr(dl, r, s) =
   initstate();
   newframes(Rdecl(dl), r, s);
   loop();
   let st = topstore() in
    popstore();
    let rt = topenv() in
     popenv();
     (rt, st)

let semdl((dl,rl), r, s) =
   initstate();
   newframes(Rdecl(rl), r, s);
   push(labeldec(dl),top(cstack));
   loop();
   let st = topstore() in
    popstore();
    let rt = topenv() in
     popenv();
     (rt, st)

let semc((c:  com), (r:dval env), (s:  mval store)) =
   initstate();
   pushstore(emptystore(Undefined));
   newframes(labelcom([c]), r, s);
   loop();
   let st = topstore() in
    popstore();
    st

let semb ((dl, rdl, cl), r, s) =
   initstate();
   pushstore(emptystore(Undefined));
   newframes(labelcom(cl), r, s);
   push(Rdecl(rdl), top(cstack));
   push(labeldec(dl), top(cstack));
   loop();
   let st = topstore() in
    popstore();
    st
```

Come per il linguaggio funzionale, all'interprete manca ancora un'implementazione vera dell'ambiente (e dello *store*). Nell'implementazione attuale abbiamo una pila di ambienti ed una pila di memorie in cui ogni ambiente (ed ogni memoria) è l'ambiente (memoria) complessivo, rappresentato attraverso una funzione. In una implementazione reale ogni attivazione dovrebbe invece avere:

- L'ambiente locale, più un modo per reperire il resto dell'ambiente visibile.

- La memoria locale.
- Ambiente e momoria locali implementati al prim'ordine, con apposite strutture dati.

Vedremo in seguito delle implementazioni "realistiche" (vedi 14).

10.6 Digressione sullo scoping dinamico

Anche in questo caso rimangono valide le considerazioni fatte in precedenza a proposito dello scoping dinamico nel caso di linguaggio funzionale (vedi 9.11).
Vediamo soltanto i domini di funzioni e procedure in semantica denotazionale ed operazionale:

Semantica **denotazionale**:

- *Scoping statico*:
```
type efun = (dval list) * (mval store) -> eval
type proc = (dval list) * (mval store) -> mval store
```
- *Scoping dinamico*:
```
type efun = (dval env) -> ((dval list) * (mval store)) -> eval
type proc = (dval env) -> ((dval list) * (mval store)) -> mval store
```

Semantica **operazionale**:

- *Scoping statico*:
```
type efun = exp * (dval env)
type proc = exp * (dval env)
```
- *Scoping dinamico*:
```
type efun = exp
type proc = exp
```

11 Classi ed oggetti

Contenuti del capitolo:

- Dai sottoprogrammi alle classi: oggetti come attivaizoni permanenti, ambienti accessibili ovunque, entità con stato, strutture dati dinamiche.
- Ereditarietà (semplice) con annidamento di blocchi e sottoprogrammi: combinazione di modularità e scoping statico.
- Classi come tipi (semantica statica) ed ereditarietà come definizione di sottotipi.
- Estensione object-oriented del linguaggio imperativo: semenatica denotazionale, semantica operazionale, interprete iterativo.

11.1 Dai sottoprogrammi alle classi

Un sottoprogramma, oltre a definire un'astrazione procedurale, consente di gestire dinamicamente ambiente e memoria. La chiamata ad un sottoprogramma crea, infatti, un ambiente ad una memoria locale che esistono finchè l'attivazione non ritorna.
Se volessimo che ambiente e memoria creati fossero permanenti avremmo due possibilità:

- Adottare l'*ambiente locale statico*, di modo che ambiente e memoria, creati con la definizione della procedura, esistano (solo) per le diverse attivazioni della procedura.
- Definire un *meccanismo* che permetta di creare, al momento dell'attivazione, ambiente e memoria che siano **permanenti** (sopravvivano all'attivazione) e che siano **accessibili** ed **utilizzabili** da chiunque possieda il loro "manico" (l'oggetto che li contiene).

Ovviamente fra le due possibilità la seconda risulta preferibile, e ci porta alla definizione di un "nuovo" tipo di sottoprogramma a cui diamo il nome di **classe**.

Una classe, come un sottoprogramma, può avere dei *parametri* (a differenza di Java) e *contiene un blocco* (lista di dichiarazioni, lista di comandi).
L'**istanziazione** (attivazione) di una classe avviene attraverso il costrutto

`new`(`classe`, `parametri_attuali`)

che può avvenire in una *qualunque espressione* e che *restituisce un oggetto*.
Ambiente e memoria locali dell'oggetto sono creati dalla valutazione delle dichiarazioni: le dichiarazioni di costanti e variabili definiscono i **campi dell'oggetto** (se ci sono variabili, allora l'oggetto ha una memoria ed uno stato modificabile), mentre le dichiarazioni di funzioni e procedure ne definiscono i **metodi** (che vedono e possono modificare i campi, per la semantica dei blocchi). L'esecuzione della lista di comandi rappresenta l'**inizializzazione** dell'oggetto.
Abbiamo già detto che l'*oggetto* rappresenta un "manico" che permette di accedere ad ambiente e memoria locali, creati permanentemente. Per far questo utilizziamo l'operazione

`Field`(`oggetto`, `identificatore`)

che consente di accedere a campi e metodi di uno specifico oggetto. Nell'ambiente locale di un oggetto utilizzeremo il nome speciale `this` per denotare l'oggetto stesso.

11.2 Classi, oggetti e tipi di dato

Le classi sono un modo molto naturale per definire tipi di dato, ed in particolare tipi di dato con stato (modificabile), in cui:

- La rappresentazione dei *valori* del tipo è data dall'insieme campi dell'oggetto.
- Le *operazioni primitive* del tipo sono i metodi dell'oggetto.

La creazione di un oggetto corrisponde quindi alla creazione di un valore del tipo (se ci sono variabili, l'oggetto ha uno stato modificabile). Se i campi non sono accessibili dall'esterno (privati) allora il tipo di dato è *astratto*.

Anche sintatticamente la creazione degli oggetti assomiglia molto alla creazione dinamica di strutture dati (ad esempio in PASCAL e C) realizzata con operazioni come `new(tipo)`, che provoca l'allocazione dinamica di un valore di tipo `tipo`, con la restituzione di un puntatore a tale struttura. Tale meccanismo prevede l'esistenza di una memoria a **heap** (simile a quella usata per l'implementazione delle liste), che riprenderemo per implementare gli oggetti.
Le seguenti differenze distinguono, comunque, le strutture dati dinamiche dagli oggetti. Le strutture dati dinamiche:

- Hanno una semantica "ad hoc" non riconducibile a quella di blocchi e procedure.
- La loro rappresentazione non è realizzata con campi separati.
- Non hanno metodi.
- Non sono davvero permanenti, perchè esiste un'operazione che permette di distruggere la struttura dati (`dispose`).

11.3 Ereditarietà

Il concetto di ereditarietà non è un componente essenziale del costrutto classe-oggetto[20] ma si sposa bene con il concetto di oggetto, arricchendolo di caratteristiche molto importanti dal punto di vista delle metodologie di programmazione (riusabilità, estendibilità, astrazione insiemi di tipi di dati tra loro collegati).
Dal punto di vista dei tipi, l'ereditarietà permette l'introduzione di relazioni di **sottotipo**, con l'effetto di arricchire il sistema dei tipi del linguaggio e, d'altra parte, rendendo più complessa la semantica statica (inferenza di tipi e/o loro verifica).
A noi interessa riportare l'ereditarietà (semplice) ai concetti già visti legati all'ambiente.

Semanticamente la relazione di ereditarietà è simile a quella di annidamento tra blocchi e sottoprogrammi. Supponendo di avere

`c1` sottoclasse di `c2`

[20] Il concetto di ereditarietà nasce in contesti diversi e lontani, legati soprattutto alla creazione di tassonomie usate in rappresentazione della conoscenza.

Le *associazioni* esistenti in una istanziazione di `c1` sono tutte quelle generate dalla dichiarazioni di `c1`, più tutte quelle generate dalle dichiarazioni di `c2` che non sono state ridefinite in `c1`.
Di fatto è come se `c1` fosse sintatticamente dento `c2` con una regola di *scoping statico*, con il vantaggio di non dover mantere realmente l'inclusione a livello sintattico e quindi di poter compilare separatamente le due classi.
Un'altra caratteristica che distingue la relazione di ereditarietà tra classi da quella di annidamento tra blocchi e procedure, è rappresentata dal fatto che un'istanziazione di `c1` può esistere anche se non esiste già un'istanziazione di `c2` (questo non può succedere per le attivazioni di blocchi annidati), che di conseguenza viene creata dalla stessa classe `c1`.

11.4 Linguaggio object-oriented: sintassi

Aggiungiamo alle espressioni i costrutti `Field`, `New` e `This`, più la dichiarazione di classe `Class`:

```
type ide = string
type exp =
   | Eint of int
   | Ebool of bool
   | Den of ide
   | Prod of exp * exp
   | Sum of exp * exp
   | Diff of exp * exp
   | Eq of exp * exp
   | Minus of exp
   | Iszero of exp
   | Or of exp * exp
   | And of exp * exp
   | Not of exp
   | Ifthenelse of exp * exp * exp
   | Val of exp
   | Let of ide * exp * exp
   | Newloc of exp
   | Fun of ide list * exp
   | Rec of ide * exp
   | Appl of exp * exp list
   | Proc of ide list * block
   | Field of exp * ide
   | New of ide * exp list
   | This

and decl = (ide * exp) list * (ide * exp) list
and block = (ide * exp) list * (ide * exp) list * com list

and com =
   | Assign of exp * exp
   | Cifthenelse of exp * com list * com list
```

```
   | While of exp * com list
   | Block of block
   | Call of exp * exp list

type cdecl = Class of ide * ide list * (ide * ide list) * block
type prog = cdecl list * block
```

La dichiarazione di classe `Class` prende in ingresso: *nome della classe*, *parametri*, *nome della superclasse* e suoi *parametri*. Per indicare la radice della gerarchia delle classi useremo `Object`, senza parametri.
E' imprtante notare che le dichiarazioni di classe possono solo occorrere nell'ambiente globale: non c'è dunque annidamento fra classi (come in ML e diversamente da Java, dove è possibile in forme limitate).

11.5 Semantica denotazionale

Prima di inrodurre la semantica denotazionale vediamo in che modo definire i domini semantici, in base ai valori assegnati a classi ed oggetti.
In semantica denotazionale interpretiamo un oggetto su un domino di valori *obj* che sono semplicemente *ambienti*. I valori (denotabili, esprimibili e memorizzabili) con cui ci riferiamo agli oggetti sono invece di tipo *pointer*, definendo l'associazione tra *pointer* ed *obj* attraverso l'introduzione di un nuovo dominio semantico *heap*[21].
Le classi saranno invece interpretate attraverso i valori *eclass*, unicamente denotabili. Le classi possono quindi essere dichiarate, passate come parametri, utilizzate nell'espressione `New`, ma non restituite come valore di un'espressione.
Vediamo dunque la definizione dei domini semantici, delle funzioni di trasformazione e delle funzione connesse alla gestione del dominio *heap*:

```
exception Nonstorable
exception Nonexpressible

type pointer = int
type eval =
   | Int of int
   | Bool of bool
   | Novalue
   | Funval of efun
   | Object of pointer

and efun = (dval list) * (mval store) * heap ->
      eval * (mval store) * heap
and proc = (dval list) * (mval store) * heap ->
      (mval store) * heap
and eclass = dval list * (mval store) * heap ->
      eval * (mval store) * heap
and obj = dval env
and heap = pointer -> obj
```

[21] Se non ci fosse il costrutto `This` potremmo fare a meno del dominio *heap* (nelle semantiche), usando direttamente gli oggetti invece che i riferimenti ad essi.

```
and dval =
   | Dint of int
   | Dbool of bool
   | Unbound
   | Dloc of loc
   | Dfunval of efun
   | Dprocval of proc
   | Dobject of pointer
   | Classval of eclass

and mval =
   | Mint of int
   | Mbool of bool
   | Undefined
   | Mobject of pointer

let evaltomval e = match e with
   | Int n -> Mint n
   | Bool n -> Mbool n
   | Object n -> Mobject n
   | _ -> raise Nonstorable

let mvaltoeval m = match m with
   | Mint n -> Int n
   | Mbool n -> Bool n
   | Mobject n -> Object n
   | _ -> Novalue

let evaltodval e = match e with
   | Int n -> Dint n
   | Bool n -> Dbool n
   | Novalue -> Unbound
   | Funval f -> Dfunval f
   | Object n -> Dobject n

let dvaltoeval e = match e with
   | Dint n -> Int n
   | Dbool n -> Bool n
   | Dloc n -> raise Nonexpressible
   | Dprocval n -> raise Nonexpressible
   | Unbound -> Novalue
   | Dfunval f -> Funval f
   | Dobject n -> Object n
   | Classval n -> raise Nonexpressible

let (newpoint,initpoint) =
   let count = ref(-1) in
     (fun () -> count := !count +1;
```

```
      !count),
   (fun () -> count := -1)

let emptyheap () =
   initpoint(); ((function (x:  pointer) -> emptyenv Unbound):  heap)

let applyheap ((x:  heap), (y:pointer)) = x y

let allocateheap ((x:heap), (i:pointer), (r:obj)) =
   ((function j -> if j = i then r else x j):heap)
```

Oltre ad aver aggiunto classi ed oggetti (e puntatori) ad i domini corrispondenti, abbiamo definito, per la gestione del dominio *heap*, le funzioni `emptyheap`, `applyheap`, `allocateheap`.
Definiamo adesso le funzioni utilizzate per creare ed applicare funzioni e procedure, aggiungendo il necessario per la creazione e l'istanziazione delle classi:

```
let notoccur ((i:ide), l) = let vl = ref(l) in
let res = ref(true) in
   while not (!vl = []) do
     if i = List.hd(!vl) then (res := false; vl := [])
     else vl := List.tl(!vl)
   done;
   !res

let findone ((args:  ide list), (el:  dval list), (s:  ide )) =
   let vargs = ref(args) in
   let dargs = ref(el) in
   let valore = ref(Unbound) in
     while not(!vargs = []) do
       if s = List.hd !vargs then
         (vargs := []; valore := List.hd !dargs)
       else vargs := List.tl !vargs; dargs := List.tl !dargs
     done;
     !valore
let findsuperargs((args:ide list),(el:dval list),(sargs:ide list))=
   let vsargs = ref(sargs) in
   let res = ref([]) in
     while not(!vsargs = []) do
       let i = findone(args, el, List.hd(!vsargs)) in
         if i = Unbound then failwith(``problem with super parameters'')
         else res := !res @ [i]; vsargs := List.tl !vsargs
     done;
     !res

let localenv ((env1:dval env),(li:ide list),(envv:dval env)) =
   (function (j:ide) ->
     (if notoccur( j, li) & applyenv(envv,j) = Unbound then
        env1 j else Unbound):dval env)
```

```
let eredita ((env1:dval env) ,Object(n), (h:heap)) =
   let r = applyheap(h, n) in
    (function (i:ide) ->
     (if r i = Unbound then env1 i else r i):(dval env))

let rec makefun ((a:exp),(x:dval env)) =
   (match a with
    | Fun(ii,aa) ->
           Dfunval(function (d,s,h) ->
              sem aa (bindlist(x, ii, d)) s h)
    | _ -> failwith (``Non-functional object''))

and makeproc ((a:exp),(x:dval env)) =
   (match a with
    | Proc(ii,b) ->
       Dprocval(function (d1,s1, h1) ->
                semb b (bindlist(x, ii, d1)) s1 h1)
    | _ -> failwith (``Non-procedural object''))

and applyfun ((ev1:dval),(ev2:dval list),s, h) =
   ( match ev1 with
      | Dfunval(x) -> x (ev2,s, h)
      | _ -> failwith (``attempt to apply a non-functional object''))

and applyproc ((ev1:dval),(ev2:dval list),s, h) =
   ( match ev1 with
      | Dprocval(x) -> x (ev2,s, h)
      | _ -> failwith (``attempt to apply a non-functional object''))

and makefunrec (i, Fun(ii, aa), r) =
   let functional ff (d,s1, h1) =
    let r1 = bind(bindlist(r, ii, d), i, Dfunval(ff)) in
     sem aa r1 s1 h1 in
   let rec fix = function x -> functional fix x
   in Funval(fix)

and makeclass((c:  cdecl), r) = match c with
    Class(name, fpars, extends, (b1,b2,b3) ) ->
     Classval(function (apars, s, h) ->
      (match extends with
       | (``Object'',_) ->
          let i = newpoint() in
           (let (r2, s2, h2) =
              semdl (b1, b2)
               (bindlist(r, fpars @ [``this''],
                       apars @ [Dobject(i)])) s h in
            let (s3, h3) = semcl b3 r2 s2 h2 in
            let r3 = localenv( r2, fpars, r) in
```

```
              let newh = allocateheap(h3, i, r3) in
                (Object i, s3, newh ))
        | (super,superpars) ->
           let (v, s1, h1) =
            applyclass(applyenv(r,super),
               findsuperargs( fpars, apars, superpars), s, h) in
           let n = (match v with | Object n1 -> n1) in
           let (r2, s2, h2) = semdl (b1, b2)
            (bindlist(eredita( r, v, h1), fpars, apars)) s1 h1 in
           let (s3, h3) = semcl b3 r2 s2 h2 in
           let newh = allocateheap(h3, n, localenv( r2 ,fpars, r)) in
            (Object n, s3, newh)
      ))

and applyclass ((ev1:dval),(ev2:dval list), s, h) =
  ( match ev1 with
     | Classval(x) -> x (ev2, s, h)
     | _ -> failwith (``notaclass''))
```

Oltre ad aver aggiunto le funzioni `makeclass` ed `applyclass`, abbiamo definito:

- `localenv` che estrae da `env1` la funzione che contiene tutte le associazioni (non presenti in `envv`) che non riguardano i parametri formali della classe. Dato che le classi sono dichiarate tutte al top level, l'ambiente non locale dell'istanziazione(prima del passaggio dei parametri) contiene solo dichiarazioni di classi.
- `eredita` che eredita nell'ambiente contenuto nell'oggetto puntato da `n` le associazioni di `env1` non ridefinite.

Diamo quindi la definizione delle funzioni di valutazione sematica, in semantica denotazionale, per il linguaggio object-oriented:

```
and sem (e:exp) (r:dval env) (s:  mval store) (h:heap) =
  match e with
    | Eint(n) -> (Int(n),s,h)
    | Ebool(b) -> (Bool(b),s,h)
    | Den(i) -> (dvaltoeval(applyenv(r,i)),s,h)
    | Iszero(a) -> let (v1,s1,h1) = sem a r s h in (iszero(v1),s1,h1)
    | Eq(a,b) -> let (v1,s1,h1) = sem a r s h in
     let (v2,s2,h2) = sem b r s1 h1 in
      (equ(v1 ,v2),s2,h2)
    | Prod(a,b) -> let (v1,s1,h1) = sem a r s h in
     let (v2,s2,h2) = sem b r s1 h1 in
      (mult(v1 ,v2),s2,h2)
    | Sum(a,b) -> let (v1,s1,h1) = sem a r s h in
     let (v2,s2,h2) = sem b r s1 h1 in
      (plus(v1 ,v2),s2,h2)
    | Diff(a,b) -> let (v1,s1,h1) = sem a r s h in
     let (v2,s2,h2) = sem b r s1 h1 in
      (diff(v1 ,v2),s2,h2)
```

```
    | Minus(a) -> let (v1,s1,h1) = sem a r s h in (minus(v1),s1,h1)
    | And(a,b) -> let (v1,s1,h1) = sem a r s h in
     let (v2,s2,h2) = sem b r s1 h1 in
       (et(v1 ,v2),s2,h2)
    | Or(a,b) -> let (v1,s1,h1) = sem a r s h in
     let (v2,s2,h2) = sem b r s1 h1 in
       (vel(v1 ,v2),s2,h2)
    | Not(a) -> let (v1,s1,h1) = sem a r s h in (non(v1),s1,h1)
    | Ifthenelse(a,b,c) ->
       let (g,s1,h1) = sem a r s h in
        if typecheck(``bool'',g) then
          (if g = Bool(true)
          then sem b r s1 h1
          else sem c r s1 h1)
        else failwith (``nonboolean guard'')
    | Fun(i,a) -> (dvaltoeval(makefun(Fun(i,a), r)), s, h)
    | Appl(a,b) -> let (v1,s1,h1) = semden a r s h in
     let (v2,s2,h2) = semlist b r s1 h1 in
       applyfun(v1, v2, s2, h2)
    | Let(i,e1,e2) -> let (v, s1, h1) = semden e1 r s h in
       sem e2 (bind(r, i, v)) s1 h1
    | Rec(f,e) -> (makefunrec (f,e,r), s, h)
    | Val(e) -> let (v, s1, h1) = semden e r s h in
       (match v with
         | Dloc n -> (mvaltoeval(applystore(s1, n)), s1, h1)
         | _ -> failwith(``not a variable''))
    | New(i,ge) -> let (v, s1, h1) = semlist ge r s h in
       applyclass(applyenv(r,i), v, s1, h1)
    | This -> (dvaltoeval(applyenv(r,''this'')), s, h)
    | _ -> failwith (``nonlegal expression for sem'')

and semlist el r s h = match el with
   | [] -> ([], s, h)
   | e::el1 -> let (v1, s1, h1) = semden e r s h in
    let (v2, s2, h2) = semlist el1 r s1 h1 in
      (v1 ::  v2, s2, h2)

and semden (e:exp) (r:dval env) (s:  mval store) (h:heap) = match e with
   | Den(i) -> (applyenv(r,i), s, h)
   | Fun(i,e1) -> (makefun(e,r), s, h)
   | Proc(il,b) -> (makeproc(e,r), s, h)
   | Newloc(e) -> let (v, s1, h1) = sem e r s h in
    let m = evaltomval v in let (l, s2) = allocate(s1, m)
    in (Dloc l, s2, h1)
   | Field(e,i) ->
     (match sem e r s h with
      | (Object i1,s1,h1) -> let r1 = applyheap(h1, i1) in
       let field = applyenv(r1,i) in
         (field,s1,h1)
```

```
       | _ -> failwith(``notanobject''))
  | _ -> let (v, s1, h1) = sem e r s h in
     (evaltodval(v), s1, h1)

and semdv dl r s h =
  match dl with
    | [] -> (r,s,h)
    | (i,e)::dl1 -> let (v, s1, h1) = semden e r s h in
       semdv dl1 (bind(r, i, v)) s1 h1

and semdl (dl,rl) r s h =
  let (r1, s1, h1) = semdv dl r s h in
    semdr rl r1 s1 h1

and semdr rl r s h =
  let functional ((r1:  dval env)) =
    (match rl with
      | [] -> r
      | (i,e) ::  rl1 -> let (v, s2, h2) = semden e r1 s h in
       let (r2, s3, h3) = semdr rl1 (bind(r, i, v)) s h in
        r2) in
  let rec rfix = function x -> functional rfix x in
    (rfix, s, h)

and semc (c:  com) (r:dval env) (s:  mval store) (h:heap) = match c with
  | Assign(e1, e2) -> let (v1, s1, h1) = semden e1 r s h in
     (match v1 with
       | Dloc(n) -> let (v, s2, h2) = sem e2 r s1 h1 in
          (update(s2, n, evaltomval v), h2)
       | _ -> failwith (``wrong location in assignment''))
  | Cifthenelse(e, cl1, cl2) -> let (g, s1, h1) = sem e r s h in
     if typecheck(``bool'',g) then
      (if g = Bool(true)
      then semcl cl1 r s1 h1
      else semcl cl2 r s1 h1)
     else failwith (``nonboolean guard'')
  | While(e, cl) ->
     let functional((fi:(mval store)*heap -> (mval store)*heap)) =
      function (sigma, delta) ->
       let (g, s1, h1) = sem e r sigma delta in
        if typecheck(``bool'',g) then
          (if g = Bool(true)
          then fi(semcl cl r sigma delta)
          else (sigma,delta))
        else failwith (``nonboolean guard'') in
     let rec statefix = function x -> functional statefix x in
      (match statefix(s, h) with | (sfix, hfix) ->
        (sfix, hfix))
  | Call(e1, e2) -> let (p, s1, h1) = semden e1 r s h in
```

```
    let (v, s2, h2) = semlist e2 r s1 h1 in
     applyproc(p, v, s2, h2)
  | Block(b) -> semb b r s h

and semcl cl r s h = match cl with
  | [] -> s, h
  | c::cl1 -> let (s1,h1) = (semc c r s h ) in semcl cl1 r s1 h1

and semb (dl, rdl, cl) r s h =
  let (r1, s1, h1) = semdl (dl,rdl) r s h in
   semcl cl r1 s1 h1

and semclasslist cl ( r:  dval env ) =
  let functional (r1:  dval env) =
   (match cl with
     | [] -> r
     | Class(nome,x1,x2,x3)::cl1 ->
        semclasslist cl1 (bind(r,nome,
             makeclass(Class(nome,x1,x2,x3), r1)))) in
  let rec rfix = function i -> functional rfix i
  in rfix

and semprog (cdl,b) r s h = semb b (semclasslist cdl r) s h
```

Le funzioni definite hanno i seguenti tipi:

```
val sem :  exp -> dval env -> mval store -> heap ->
       eval * mval store * heap = <fun>

val semden :  exp -> dval env -> mval store -> heap ->
       dval * mval store * heap = <fun>

val semlist :  exp list -> dval env -> mval store -> heap ->
       (dval list) * mval store * heap = <fun>

val semc :  com -> dval env -> mval store -> heap ->
       mval store * heap = <fun>

val semcl :  com list -> dval env -> mval store -> heap ->
       mval store * heap = <fun>

val semb :  (decl * com list) -> dval env -> mval store ->
       heap -> mval store * heap = <fun>

val semdl :  decl -> dval env -> mval store -> heap ->
       dval env * mval store * heap = <fun>

val semdv :  (ide * expr) list -> dval env -> mval store ->
       heap -> dval env * mval store * heap = <fun>
```

```
val semdr :  (ide * expr) list -> dval env -> mval store ->
       heap -> dval env * mval store * heap = <fun>

val semclasslist :  cdecl list -> dval env -> dval env = <fun>

val semprog :  (cdecl list * block) -> dval env -> mval store ->
       heap -> mval store * heap = <fun>
```

Le dichiarazioni di classe sono trattate come mutuamente ricorsive: da ognuna di esse si vedono tutte le altre.

11.6 Semantica operazionale

Come al solito, nel passaggio da semantica denotazioale a semantica operazionale cambia il tipo di tutte le funzioni di valutazione.
Per eliminare i valori di ordine superiori, cambia inoltre il dominio *eclass*:

```
eclass = dval list * (mval store) * heap ->
      eval * (mval store) * heap
```

diventa:

```
eclass = cdecl * dval env
```

Per eliminare le funzioni da *eclass* è necessario spostare la "semantica della classe" dal momento della creazione (`makeclass`) a quello dell'istanziazione (`applyclass`). Al momento della creazione non resta altro da fare se non conservare l'informazione sintattica "impaccandola" in una chiusura insieme all'ambiente corrente (che contiene *tutte* le classi dichiarate nell'ambiente globale).
Vediamo quindi la semantica operazionale completa per il linguaggio imperativo object-oriented.

```
exception Nonstorable
exception Nonexpressible

type pointer = int
type eval =
   | Int of int
   | Bool of bool
   | Novalue
   | Funval of efun
   | Object of pointer

and efun = exp * (dval env)
and proc = exp * (dval env)
and eclass = cdecl * (dval env)

and obj = dval env
and heap = pointer -> obj
```

```
and dval =
   | Dint of int
   | Dbool of bool
   | Unbound
   | Dloc of loc
   | Dfunval of efun
   | Dprocval of proc
   | Dobject of pointer
   | Classval of eclass

and mval =
   | Mint of int
   | Mbool of bool
   | Undefined
   | Mobject of pointer

let evaltomval e = match e with
   | Int n -> Mint n
   | Bool n -> Mbool n
   | Object n -> Mobject n
   | _ -> raise Nonstorable

let mvaltoeval m = match m with
   | Mint n -> Int n
   | Mbool n -> Bool n
   | Mobject n -> Object n
   | _ -> Novalue

let evaltodval e = match e with
   | Int n -> Dint n
   | Bool n -> Dbool n
   | Novalue -> Unbound
   | Funval f -> Dfunval f
   | Object n -> Dobject n

let dvaltoeval e = match e with
   | Dint n -> Int n
   | Dbool n -> Bool n
   | Dloc n -> raise Nonexpressible
   | Dprocval n -> raise Nonexpressible
   | Unbound -> Novalue
   | Dfunval f -> Funval f
   | Dobject n -> Object n
   | Classval n -> raise Nonexpressible

let (newpoint,initpoint) =
   let count = ref(-1) in
     (fun () -> count := !count +1;
      !count),
```

```
   (fun () -> count := -1)


let emptyheap () = initpoint(); ((function (x:  pointer) ->
                              emptyenv Unbound):  heap)

let applyheap ((x:  heap), (y:pointer)) = x y

let allocateheap ((x:heap), (i:pointer), (r:obj)) =
   ((function j -> if j = i then r else x j):heap)

let notoccur ((i:ide), l) = let vl = ref(l) in
let res = ref(true) in
   while not (!vl = []) do
    if i = List.hd(!vl) then (res := false; vl := [])
    else vl := List.tl(!vl)
   done;
   !res

let findone ((args:  ide list), (el:  dval list), (s:  ide )) =
   let vargs = ref(args) in
   let dargs = ref(el) in
   let valore = ref(Unbound) in
    while not(!vargs = []) do
      if s = List.hd !vargs then
        (vargs := []; valore := List.hd !dargs)
      else vargs := List.tl !vargs; dargs := List.tl !dargs
    done;
    !valore

let findsuperargs ((args:ide list),(el:dval list),(sargs:ide list))=
   let vsargs = ref(sargs) in
   let res = ref([]) in
    while not(!vsargs = []) do
      let i = findone(args, el, List.hd(!vsargs)) in
        if i = Unbound then failwith(``problem with super parameters'')
        else res := !res @ [i]; vsargs := List.tl !vsargs
    done;
    !res

let localenv ((env1:dval env) ,(li:ide list), (envv:dval env)) =
   (function (j:ide) ->
    (if notoccur( j, li) & applyenv(envv,j) = Unbound then
       env1 j else Unbound):dval env)

let eredita ((env1:dval env) ,Object(n), (h:heap)) =
   let r = applyheap(h, n) in
    (function (i:ide) ->
      (if r i = Unbound then env1 i else r i):(dval env))
```

```
let rec makefun ((a:exp),(x:dval env)) =
   (match a with
     | Fun(ii,aa) ->
         Dfunval(a, x)
     | _ -> failwith (``Non-functional object''))

and makeproc ((a:exp),(x:dval env)) =
   (match a with
     | Proc(ii,b) ->
         Dprocval(a, x)
     | _ -> failwith (``Non-procedural object''))

and applyfun ((ev1:dval),(ev2:dval list),s, h) =
   ( match ev1 with
      | Dfunval(Fun(ii,aa), x) -> sem(aa, bindlist(x, ii, ev2), s, h)
      | _ -> failwith (``attempt to apply a non-functional object''))

and applyproc ((ev1:dval),(ev2:dval list),s, h) =
   ( match ev1 with
      | Dprocval(Proc(ii,b),x) -> semb(b, bindlist(x, ii, ev2), s, h)
      | _ -> failwith (``attempt to apply a non-functional object''))

and makefunrec (i, e1, r) =
   let functional (rr:  dval env) =
     bind(r, i, makefun(e1,rr)) in
   let rec rfix = function x -> functional rfix x
   in dvaltoeval(makefun(e1, rfix))

and makeclass((c:  cdecl), r) = Classval(c, r)

and applyclass ((ev1:dval),(apars:dval list), s, h) =
   ( match ev1 with
      | Classval(Class(name, fpars, extends, (b1,b2,b3) ),r) ->
         (match extends with
           | (``Object'',_) ->
              let i = newpoint() in
                (let (r2, s2, h2) =
                   semdl((b1, b2),
                        (bindlist(r, fpars @ [``this''],
                                apars @ [Dobject(i)])),
                        s, h) in
                  let (s3, h3) = semcl(b3, r2, s2, h2) in
                  let r3 = localenv( r2, fpars, r) in
                  let newh = allocateheap(h3, i, r3) in
                    (Object i, s3, newh ))
           | (super,superpars) ->
              let (v, s1, h1) =
                applyclass(applyenv(r,super),
```

```
                    findsuperargs(fpars,apars,superpars),s,h) in
            let n = (match v with | Object n1 -> n1) in
            let (r2, s2, h2) =
              semdl((b1, b2),
                   (bindlist(eredita(r,v,h1),fpars,apars)),s1,h1) in
            let (s3, h3) = semcl(b3, r2, s2, h2) in
            let newh = allocateheap(h3, n, localenv( r2 ,fpars, r)) in
              (Object n, s3, newh))
     | _ -> failwith(``not a class''))

and sem ((e:exp), (r:dval env), (s:  mval store), (h:  heap)) =
  match e with
    | Eint(n) -> (Int(n),s,h)
    | Ebool(b) -> (Bool(b),s,h)
    | Den(i) -> (dvaltoeval(applyenv(r,i)),s,h)
    | Iszero(a) -> let (v1,s1,h1) = sem(a, r, s, h)
      in (iszero(v1),s1,h1)
    | Eq(a,b) -> let (v1,s1,h1) = sem(a, r, s, h) in
      let (v2,s2,h2) = sem(b, r, s1, h1) in
        (equ(v1 ,v2),s2,h2)
    | Prod(a,b) -> let (v1,s1,h1) = sem(a, r, s, h) in
      let (v2,s2,h2) = sem(b, r, s1, h1) in
        (mult(v1 ,v2),s2,h2)
    | Sum(a,b) -> let (v1,s1,h1) = sem(a, r, s, h) in
      let (v2,s2,h2) = sem(b, r, s1, h1) in
        (plus(v1 ,v2),s2,h2)
    | Diff(a,b) -> let (v1,s1,h1) = sem(a, r, s, h)in
      let (v2,s2,h2) = sem(b, r, s1, h1) in
        (diff(v1 ,v2),s2,h2)
    | Minus(a) -> let (v1,s1,h1) = sem(a, r, s, h) in
        (minus(v1),s1,h1)
    | And(a,b) -> let (v1,s1,h1) = sem(a, r, s, h) in
      let (v2,s2,h2) = sem(b, r, s1, h1) in
        (et(v1 ,v2),s2,h2)
    | Or(a,b) -> let (v1,s1,h1) = sem(a, r, s, h) in
      let (v2,s2,h2) = sem(b, r, s1, h1) in
        (vel(v1 ,v2),s2,h2)
    | Not(a) -> let (v1,s1,h1) = sem(a, r, s, h) in
        (non(v1),s1,h1)
    | Ifthenelse(a,b,c) ->
       let (g,s1,h1) = sem(a, r, s, h) in
        if typecheck(``bool'',g) then
          (if g = Bool(true)
          then sem(b, r, s1, h1)
          else sem(c, r, s1, h1))
        else failwith (``nonboolean guard'')
    | Fun(i,a) -> (dvaltoeval(makefun(Fun(i,a), r)), s, h)
    | Appl(a,b) ->
       let (v1,s1,h1) = semden(a, r, s, h) in
```

```
       let (v2,s2,h2) = semlist(b, r, s1, h1) in
        applyfun(v1, v2, s2, h2)
    | Let(i,e1,e2) -> let (v, s1, h1) = semden(e1, r, s, h) in
       sem(e2 ,bind(r, i, v), s1, h1)
    | Rec(f,e) -> (makefunrec (f,e,r), s, h)
    | Val(e) -> let (v, s1, h1) = semden(e, r, s, h) in
       (match v with
        | Dloc n -> (mvaltoeval(applystore(s1, n)), s1, h1)
        | _ -> failwith(``not a variable''))
    | New(i,ge) -> let (v, s1, h1) = semlist(ge, r, s, h) in
       applyclass(applyenv(r,i), v, s1, h1)
    | This -> (dvaltoeval(applyenv(r,''this'')), s, h)
    | _ -> failwith (``nonlegal expression for sem'')

and semlist(el, r, s, h) = match el with
   | [] -> ([], s, h)
   | e::el1 -> let (v1, s1, h1) = semden( e, r, s, h) in
    let (v2, s2, h2) = semlist(el1, r, s1, h1) in
     (v1 ::  v2, s2, h2)

and semden(e, r, s, h) = match e with
   | Den(i) -> (applyenv(r,i), s, h)
   | Fun(i,e1) -> (makefun(e,r), s, h)
   | Proc(il,b) -> (makeproc(e,r), s, h)
   | Field(e,i) ->
      (match sem(e, r, s, h) with
       | (Object i1,s1,h1) -> let r1 = applyheap(h1, i1) in
        let field = applyenv(r1,i) in
          (field,s1,h1)
       | _ -> failwith(``notanobject''))
   | Newloc(e) -> let (v, s1, h1) = sem(e, r, s, h) in
    let m = evaltomval v in let (l, s2) = allocate(s1, m)
    in (Dloc l, s2, h1)
   | _ -> let (v, s1, h1) = sem(e, r, s, h) in
      (evaltodval(v), s1, h1)

and semdv(dl, r, s, h) =
   match dl with
    | [] -> (r,s,h)
    | (i,e)::dl1 -> let (v, s1, h1) = semden(e, r, s, h) in
       semdv(dl1, bind(r, i, v), s1, h1)

and semdl ((dl,rl), r, s, h) =
   let (r1, s1, h1) = semdv(dl, r, s, h) in
    semdr(rl, r1, s1, h1)

and semdr(rl, r, s, h) =
   let functional ((r1:  dval env)) =
    (match rl with
```

```
      | [] -> r
      | (i,e) ::  rl1 -> let (v, s2, h2) = semden(e, r1, s, h) in
       let (r2, s3, h3) = semdr(rl1, bind(r, i, v), s, h) in
        r2) in
  let rec rfix = function x -> functional rfix x in
    (rfix, s, h)

and semc((c:  com), (r:dval env), (s:  mval store), (h:heap)) =
  match c with
    | Assign(e1, e2) -> let (v1, s1, h1) = semden(e1, r, s, h) in
       (match v1 with
        | Dloc(n) -> let (v, s2, h2) = sem(e2, r, s1, h1) in
           (update(s2, n, evaltomval(v)),h2)
        | _ -> failwith (``wrong location in assignment''))
    | Cifthenelse(e, cl1, cl2) -> let (g,s1,h1) = sem(e,r,s,h) in
       if typecheck(``bool'',g) then
        (if g = Bool(true)
        then semcl(cl1, r, s1, h1)
        else semcl(cl2, r, s1, h1))
       else failwith (``nonboolean guard'')
    | While(e, cl) -> let (g, s1, h1) = sem(e, r, s, h) in
       if typecheck(``bool'',g) then
        (if g = Bool(true) then
          semcl((cl @ [While(e, cl)]), r, s1, h1)
        else (s1, h1))
       else failwith (``nonboolean guard'')
    | Call(e1, e2) -> let (p,s1,h1) = semden(e1,r,s,h) in
      let (v, s2, h2) = semlist(e2, r, s1, h1) in
       applyproc(p, v, s2, h2)
    | Block(b) -> semb(b, r, s, h)

and semcl(cl, r, s, h) =
  match cl with
    | [] -> s, h
    | c::cl1 -> let (s1,h1) = semc( c, r, s, h ) in
       semcl(cl1, r, s1, h1)

and semb ((dl, rdl, cl), r, s, h) =
  let (r1, s1, h1) = semdl((dl,rdl), r, s, h) in
    semcl(cl, r1, s1, h1)

and semclasslist (cl, ( r:  dval env )) =
  let functional (r1:  dval env) =
    (match cl with
      | [] -> r
      | Class(nome,x1,x2,x3)::cl1 ->
         semclasslist(cl1, bind(r,nome,
              makeclass(Class(nome,x1,x2,x3), r1)))) in
  let rec rfix = function i -> functional rfix i
```

```
  in rfix

and semprog ((cdl,b),r,s,h) = semb(b,semclasslist(cdl,r),s,h)
```

La semantica operazionale non è molto diversa da quella denotazionale se non per l'ormai banale scambio di funzionalità tra `makefun` ed `applyfun` (in modo da riportare i valori trattati al prim'ordine) e per le modifiche allle funzioni `makeclass` ed `applyclass` (per lo stesso motivo).

11.7 Eliminare la ricorsione

Per eliminare la ricorsione non servono strutture dati diverse da quelle già introdotte per gestire blocchi e procedure.
La istanziazione di classe crea un nuovo frame in cui valutare il corpo della classe, dopo eventualmente aver creato frames per le superclassi.
Servono, però, nuovi costrutti etichettati per le classi:

```
type labeledconstruct = ...
  | Ogg1 of cdecl
  | Ogg2 of cdecl
  | Ogg3 of cdecl
```

La pila dei record di attivazione sarà realizzata attraverso sei pile gestite "in parallelo": `envstack`, `cstack`,`tempvalstack`, `storestack`, `labelstack`.
La **heap** è globale ed è gestita da una variabile (di tipo heap) `currentheap`.

11.8 Interprete iterativo

Definiamo, come di consueto, strutture dati ed operazioni usate dall'interprete iterativo:

```
type labeledconstruct =
   | Expr1 of exp
   | Expr2 of exp
   | Exprd1 of exp
   | Exprd2 of exp
   | Com1 of com
   | Com2 of com
   | Coml of labeledconstruct list
   | Decl of labeledconstruct list
   | Rdecl of (ide * exp) list
   | Dec1 of (ide * exp)
   | Dec2 of (ide * exp)
   | Ogg1 of cdecl
   | Ogg2 of cdecl
   | Ogg3 of cdecl

let (cstack:  labeledconstruct stack stack) =
   emptystack(stacksize,emptystack(1,Expr1(Eint(0))))
```

```
let (tempvalstack:  eval stack stack) =
  emptystack(stacksize,emptystack(1,Novalue))

let (tempdvalstack:  dval stack stack) =
  emptystack(stacksize,emptystack(1,Unbound))

let labelcom (dl:  com list) = let dlr = ref(dl) in
let ldlr = ref([]) in
  while not (!dlr = []) do
   let i = List.hd !dlr in
     ldlr := !ldlr @ [Com1(i)];
     dlr := List.tl !dlr
  done;
  Coml(!ldlr)

let labeldec (dl:  (ide * exp) list) = let dlr = ref(dl) in
let ldlr = ref([]) in
  while not (!dlr = []) do
   let i = List.hd !dlr in
     ldlr := !ldlr @ [Dec1(i)];
     dlr := List.tl !dlr
  done;
  Decl(!ldlr)

let envstack = emptystack(stacksize,(emptyenv Unbound))

let storestack = emptystack(stacksize,(emptystore Undefined))

let pushenv(r) = push(r,envstack)

let topenv() = top(envstack)

let popenv () = pop(envstack)

let svuotaenv() = svuota(envstack)

let pushstore(s) = push(s,storestack)

let popstore () = pop(storestack)

let svuotastore () = svuota(storestack)

let topstore() = top(storestack)

let (labelstack:  labeledconstruct stack) =
  emptystack(stacksize,Expr1(Eint(0)))

let currentheap = ref(emptyheap())
```

```
let pushargs ((b:exp list),(continuation:labeledconstruct stack)) =
   let br = ref(b) in
    while not(!br = []) do
     push(Exprd1(List.hd !br),continuation);
     br := List.tl !br
    done

let getargs ((b:  exp list),(tempstack:  dval stack)) =
   let br = ref(b) in
   let er = ref([]) in
    while not(!br = []) do
     let arg=top(tempstack) in
       pop(tempstack); er := !er @ [arg];
       br := List.tl !br
    done;
    !er

let dlist ((b:  ide list),(r:  dval env)) =
   let br = ref(b) in
   let er = ref([]) in
    while not(!br = []) do
     let arg= applyenv(r, List.hd !br) in
       er := !er @ [arg];
       br := List.tl !br
    done;
    !er

let newframes(ss,rho,sigma) =
   pushstore(sigma);
   pushenv(rho);
   let cframe = emptystack(cframesize(ss),Expr1(Eint 0)) in
   let tframe = emptystack(tframesize(ss),Novalue) in
   let dframe = emptystack(tdframesize(ss),Unbound) in
    push(tframe,tempvalstack);
    push(dframe,tempdvalstack);
    push(ss, labelstack);
    push(ss, cframe);
    push(cframe, cstack)

let makefun ((a:exp),(x:dval env)) =
   (match a with
    | Fun(ii,aa) ->
        Dfunval(a,x)
    | _ -> failwith (``Non-functional object''))

let makeproc ((a:exp),(x:dval env)) =
   (match a with
    | Proc(ii,b) ->
        Dprocval(a,x)
```

```
    | _ -> failwith (``Non-procedural object''))

let makefunrec (i, e1, r) =
   let functional (rr:  dval env) =
    bind(r, i, makefun(e1,rr)) in
   let rec rfix = function x -> functional rfix x
   in dvaltoeval(makefun(e1, rfix))

let applyfun ((ev1:dval),(ev2:dval list), s) =
   ( match ev1 with
      | Dfunval(Fun(ii,aa),r) ->
         newframes(Expr1(aa),bindlist(r, ii, ev2), s)
      | _ -> failwith (``attempt to apply a non-functional object''))

let applyproc ((ev1:dval),(ev2:dval list), s) =
   ( match ev1 with
      | Dprocval(Proc(ii,(l1, l2, l3)), x) ->
         newframes(labelcom(l3), bindlist(x, ii, ev2), s);
         push(Rdecl(l2), top(cstack));
         push(labeldec(l1),top(cstack))
      | _ -> failwith (``attempt to apply a non-functional object''))

let makeclass((c:  cdecl), r) = Classval(c, r)

let applyclass ((ev1:dval),(apars:dval list), s, h) =
   ( match ev1 with
      | Classval(Class(name, fpars, extends, (b1,b2,b3) ),r) ->
         let j = newpoint() in
          newframes(Ogg1(Class(name, fpars, extends, (b1,b2,b3) )),
                  bindlist(r, fpars @ [``this''],
                        apars @ [Dobject(j)]), s)
      | _ -> failwith(``not a class''))
```

Definiamo adesso l'interprete iterativo per il linguaggio ad oggetti:

```
let itsem() =
   let continuation = top(cstack) in
   let tempstack = top(tempvalstack) in
   let tempdstack = top(tempdvalstack) in
   let rho = topenv() in
   let sigma = topstore() in
    (match top(continuation) with
     |Expr1(x) ->
        (pop(continuation);
        push(Expr2(x),continuation);
        (match x with
           | Iszero(a) -> push(Expr1(a),continuation)
           | Eq(a,b) -> push(Expr1(a),continuation);
              push(Expr1(b),continuation)
```

```
        | Prod(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
        | Sum(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
        | Diff(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
        | Minus(a) -> push(Expr1(a),continuation)
        | And(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
        | Or(a,b) -> push(Expr1(a),continuation);
           push(Expr1(b),continuation)
        | Not(a) -> push(Expr1(a),continuation)
        | Ifthenelse(a,b,c) -> push(Expr1(a),continuation)
        | Val(e) -> push(Exprd1(e),continuation)
        | Newloc(e) -> failwith (``nonlegal expression for sem'')
        | Let(i,e1,e2) -> push(Exprd1(e1),continuation)
        | Appl(a,b) -> push(Exprd1(a),continuation);
           pushargs(b,continuation)
        | Proc(i,b) -> failwith (``nonlegal expression for sem'')
        | New(i,ge) -> pushargs(ge, continuation)
        | _ -> ()))
   |Expr2(x) ->
      (pop(continuation);
      (match x with
        | Eint(n) -> push(Int(n),tempstack)
        | Ebool(b) -> push(Bool(b),tempstack)
        | Den(i) -> push(dvaltoeval(applyenv(rho,i)),tempstack)
        | Iszero(a) ->
           let arg=top(tempstack) in
            pop(tempstack);
            push(iszero(arg),tempstack)
        | Eq(a,b) ->
           let firstarg=top(tempstack) in
            pop(tempstack);
            let sndarg=top(tempstack) in
              pop(tempstack);
              push(equ(firstarg,sndarg),tempstack)
        | Prod(a,b) ->
           let firstarg=top(tempstack) in
            pop(tempstack);
            let sndarg=top(tempstack) in
              pop(tempstack);
              push(mult(firstarg,sndarg),tempstack)
        | Sum(a,b) ->
           let firstarg=top(tempstack) in
            pop(tempstack);
            let sndarg=top(tempstack) in
              pop(tempstack);
              push(plus(firstarg,sndarg),tempstack)
```

```
| Diff(a,b) ->
   let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
     pop(tempstack);
     push(diff(firstarg,sndarg),tempstack)
| Minus(a) ->
   let arg=top(tempstack) in
    pop(tempstack);
    push(minus(arg),tempstack)
| And(a,b) ->
   let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
     pop(tempstack);
     push(et(firstarg,sndarg),tempstack)
| Or(a,b) ->
   let firstarg=top(tempstack) in
    pop(tempstack);
    let sndarg=top(tempstack) in
     pop(tempstack);
     push(vel(firstarg,sndarg),tempstack)
| Not(a) ->
   let arg=top(tempstack) in
    pop(tempstack);
    push(non(arg),tempstack)
| Ifthenelse(a,b,c) ->
   let arg=top(tempstack) in
    pop(tempstack);
    if typecheck(``bool'',arg) then
     (if arg = Bool(true)
     then push(Expr1(b),continuation)
     else push(Expr1(c),continuation))
    else failwith (``type error'')
| Val(e) -> let v = top(tempdstack) in
   pop(tempdstack);
   (match v with
    | Dloc n ->
       push(mvaltoeval(applystore(sigma, n)),
            tempstack)
    | _ -> failwith(``not a variable''))
| Fun(i,a) ->
   push(dvaltoeval(makefun(Fun(i,a),rho)),tempstack)
| Rec(f,e) ->
   push(makefunrec(f,e,rho),tempstack)
| Let(i,e1,e2) -> let arg= top(tempdstack) in
   pop(tempdstack);
   newframes(Expr1(e2), bind(rho, i, arg), sigma)
| Appl(a,b) -> let firstarg=top(tempdstack) in
```

```
              pop(tempdstack);
              let sndarg=getargs(b,tempdstack) in
               applyfun(firstarg, sndarg, sigma)
          | New(i,ge) -> let arg=getargs(ge,tempdstack) in
              applyclass(applyenv(rho,i),arg,sigma,!currentheap)
          | This ->
              push(dvaltoeval(applyenv(rho,''this'')), tempstack)
          | _ -> failwith(``no more cases for semexpr''))
     | _ -> failwith(``no more cases for semexpr''))

let itsemden() =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
   (match top(continuation) with
    |Exprd1(x) ->
       (pop(continuation); push(Exprd2(x),continuation);
       match x with
         | Den i -> ()
         | Fun(i,e) -> ()
         | Proc(il,b) -> ()
         | Field(e,i) ->
            push(Expr1(e), continuation)
         | Newloc(e) -> push(Expr1( e), continuation)
         | _ -> push(Expr1(x), continuation))
    |Exprd2(x) ->
       (pop(continuation); match x with
         | Den i -> push(applyenv(rho,i), tempdstack)
         | Fun(i,e) -> push(makefun(x,rho),tempdstack)
         | Proc(il,b) -> push(makeproc(x,rho),tempdstack)
         | Field(e,i) -> let ogg = top(tempstack) in
            pop(tempstack);
            (match ogg with
               | Object i1 ->
                  let r1 = applyheap(!currentheap, i1) in
                  let field = applyenv(r1,i) in
                   push(field, tempdstack)
               | _ -> failwith(``notanobject''))
         | Newloc(e) ->let m=evaltomval(top(tempstack)) in
            pop(tempstack);
            let (l, s1) = allocate(sigma, m) in
             push(Dloc l, tempdstack);
             popstore();
             pushstore(s1)
         | _ -> let arg = top(tempstack) in pop(tempstack);
            push(evaltodval(arg), tempdstack))
     | _ -> failwith(``No more cases for demden'') )
```

```
let itsemdecl () =
   let tempstack = top(tempvalstack) in
   let continuation = top(cstack) in
   let tempdstack = top(tempdvalstack) in
   let rho = topenv() in
   let sigma = topstore() in
   let dl =
     (match top(continuation) with
       | Decl(dl1) -> dl1
       | _ -> failwith(``impossible in semdecl'')) in
     if dl = [] then pop(continuation) else
       (let currd = List.hd dl in
       let newdl = List.tl dl in
         pop(continuation); push(Decl(newdl),continuation);
         (match currd with
            | Dec1( (i,e)) ->
               pop(continuation);
               push(Decl(Dec2((i, e))::newdl),continuation);
               push(Exprd1(e), continuation)
             | Dec2((i,e)) ->
               let arg = top(tempdstack) in
                  pop(tempdstack);
                popenv();
                pushenv(bind(rho, i, arg))
           | _ -> failwith(``no more sensible cases for semdecl'')))

let itsemrdecl() =
   let tempstack = top(tempvalstack) in
   let continuation = top(cstack) in
   let tempdstack = top(tempdvalstack) in
   let rho = topenv() in
   let sigma = topstore() in
   let rl =
     (match top(continuation) with
       | Rdecl(rl1) -> rl1
       | _ -> failwith(``impossible in semrdecl'')) in
     pop(continuation);
     let functional (rr:  dval env) =
       let pr = ref(rho) in
       let prl = ref(rl) in
         while not(!prl = []) do
          let currd = List.hd !prl in
           prl := List.tl !prl;
           let (i, den) =
             (match currd with
               |(j, Proc(il,b)) -> (j, makeproc(Proc(il,b),rr))
               |(j, Fun(il,b)) -> (j, makefun(Fun(il,b),rr))
               | _ -> failwith(``no more sensible cases...''))  in
```

```
            pr := bind(!pr, i, den)
        done;
        !pr in
    let rec rfix = function x -> functional rfix x in
     popenv();
     pushenv(rfix)

let itsemcl() =
  let tempstack = top(tempvalstack) in
  let continuation = top(cstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  let cl =
    (match top(continuation) with
     | Coml(dl1) -> dl1
     | _ -> failwith(``impossible in semcl'')) in
    if cl = [] then pop(continuation) else
     (let currc = List.hd cl in
     let newcl = List.tl cl in
      pop(continuation); push(Coml(newcl),continuation);
       (match currc with
          | Com1(Assign(e1, e2)) -> pop(continuation);
           push(Coml(Com2(Assign(e1, e2))::newcl),continuation);
           push(Exprd1(e1), continuation);
           push(Expr1(e2), continuation)
        | Com2(Assign(e1, e2)) ->
           let arg2 = evaltomval(top(tempstack)) in
            pop(tempstack);
            let arg1 = top(tempdstack) in
             pop(tempdstack);
             (match arg1 with
                | Dloc(n) -> popstore();
                   pushstore(update(sigma, n, arg2))
                | _ -> failwith (``wrong location in assignment''))
        | Com1(While(e, cl)) ->
           pop(continuation);
           push(Coml(Com2(While(e, cl))::newcl),continuation);
           push(Expr1(e), continuation)
        | Com2(While(e, cl)) ->
           let g = top(tempstack) in
            pop(tempstack);
            if typecheck(``bool'',g) then
             (if g = Bool(true) then
                (let old = newcl in
                let newl =
                  (match labelcom cl with
                   | Coml newl1 -> newl1
                   | _ -> failwith(``impossible in while'')) in
```

```
                    let nuovo =
                    Coml(newl @ [Com1(While(e, cl))] @ old) in
                     pop(continuation); push(nuovo,continuation))
                 else ())
               else failwith (``nonboolean guard'')
          | Com1(Cifthenelse(e, cl1, cl2)) ->
             pop(continuation);
             push(Coml(Com2(Cifthenelse(e, cl1, cl2))::newcl),continuation);
             push(Expr1(e), continuation)
          | Com2(Cifthenelse(e, cl1, cl2)) ->
             let g = top(tempstack) in
              pop(tempstack);
              if typecheck(``bool'',g) then
               (let temp = if g = Bool(true) then
                labelcom (cl1) else labelcom (cl2) in
               let newl =
                (match temp with
                   | Coml newl1 -> newl1
                   | _ -> failwith(``impossible in cifthenelse'')) in
               let nuovo = Coml(newl @ newcl) in
                  pop(continuation); push(nuovo,continuation))
              else failwith (``nonboolean guard'')
          | Com1(Call(e, el)) ->
             pop(continuation);
             push(Coml(Com2(Call(e, el))::newcl),continuation);
             push(Exprd1( e), continuation);
             pushargs(el, continuation)
          | Com2(Call(e, el)) ->
             let p = top(tempdstack) in
              pop(tempdstack);
              let args = getargs(el,tempdstack) in
               applyproc(p, args, sigma)
           | Com1(Block((l1, l2, l3))) ->
             newframes(labelcom(l3), rho, sigma);
             push(Rdecl(l2), top(cstack));
             push(labeldec(l1),top(cstack))
          | _ -> failwith(``no more sensible cases in commands'') ))

let itsemobj() =
  let tempstack = top(tempvalstack) in
  let continuation = top(cstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
    (match top(continuation) with
      | Ogg1(Class(name, fpars, extends, (b1,b2,b3) )) ->
         pop(continuation);
         (match extends with
            | (``Object'',_) ->
```

```
              push(Ogg3(Class(name, fpars, extends, (b1,b2,b3) )),
                 continuation);
              push(labelcom(b3), top(cstack));
              push(Rdecl(b2), top(cstack));
              push(labeldec(b1),top(cstack))
           | (super,superpars) ->
              let lobj = applyenv(rho, ``this'') in
              let superargs = findsuperargs(fpars, dlist(fpars, rho),
                                    superpars) in
               push(Ogg2(Class(name, fpars, extends, (b1,b2,b3) )),
                  continuation);
               (match applyenv(rho, super) with
                | Classval(Class(snome,superfpars,sextends,sb),r) ->
                   newframes(Ogg1(Class
                                 (snome,superfpars,sextends,sb)),
                             bindlist(r,
                                    superfpars @ [``this''],
                                    superargs @ [lobj]),
                             sigma)
                | _ -> failwith(``not a superclass name'')))
     | Ogg2(Class(name, fpars, extends, (b1,b2,b3) )) ->
        pop(continuation);
        let v = top(tempstack) in
         pop(tempstack);
         let newenv = eredita(rho, v, !currentheap) in
          popenv();
          pushenv(newenv);
          push(Ogg3(Class(name,fpars,extends,(b1,b2,b3))),continuation);
          push(labelcom(b3), top(cstack));
          push(Rdecl(b2), top(cstack));
          push(labeldec(b1),top(cstack))
     | Ogg3(Class(name, fpars, extends, (b1,b2,b3) )) ->
        pop(continuation);
        let r = (match applyenv(rho,name) with
                 | Classval(_, r1) -> r1
                 | _ -> failwith(``not a class name'')) in
        let lobj = (match applyenv(rho, ''this'') with
                    | Dobject n -> n) in
        let newenv = localenv(rho, fpars, r) in
         currentheap := allocateheap (!currentheap, lobj, newenv);
         push(Object lobj, tempstack)
     | _ -> failwith(``impossible in semobj''))

let initstate() =
  svuota(cstack); svuota(tempvalstack); svuota(tempdvalstack);
  svuotaenv(); svuotastore(); svuota(labelstack)

let loop () =
  while not(empty(cstack)) do
```

```
    while not(empty(top(cstack))) do
     let currconstr = top(top(cstack)) in
       (match currconstr with
        | Expr1(e) -> itsem()
        | Expr2(e) -> itsem()
        | Exprd1(e) -> itsemden()
        | Exprd2(e) -> itsemden()
        | Coml(cl) -> itsemcl()
        | Decl(l) -> itsemdecl()
        | Rdecl(l) -> itsemrdecl()
        | Ogg1(e) -> itsemobj()
        | Ogg2(e) -> itsemobj()
        | Ogg3(e) -> itsemobj()
        | _ -> failwith(``non legal construct in loop''))
    done;
    (match top(labelstack) with
     | Expr1(_) -> let valore = top(top(tempvalstack)) in
        pop(top(tempvalstack));
        pop(tempvalstack); push(valore,top(tempvalstack));
        let st = topstore() in
          popenv();popstore(); popstore(); pushstore(st);
          pop(tempdvalstack)
     | Exprd1(_) -> let valore = top(top(tempdvalstack)) in
        pop(top(tempdvalstack));
        pop(tempdvalstack); push(valore,top(tempdvalstack));
        let st = topstore() in
          popenv();popstore(); popstore(); pushstore(st);
          pop(tempvalstack)
     | Decl(_) -> pop(tempvalstack); pop(tempdvalstack)
     | Rdecl(_) -> pop(tempvalstack); pop(tempdvalstack)
     | Ogg1(_) -> let valore = top(top(tempvalstack)) in
        pop(top(tempvalstack));
        pop(tempvalstack); push(valore,top(tempvalstack));
        let st = topstore() in
          popenv();popstore(); popstore(); pushstore(st);
          pop(tempdvalstack)
     | Coml(_) -> let st = topstore() in
        popenv();popstore(); popstore(); pushstore(st);
        pop(tempvalstack); pop(tempdvalstack)
     | _ -> failwith(``non legal label in loop''));
    pop(cstack);
    pop(labelstack)
  done

let sem (e,(r:  dval env), (s:  mval store), h) =
  initstate();
  currentheap := h;
  pushstore(emptystore(Undefined));
  push(emptystack(tframesize(e),Novalue),tempvalstack);
```

```
   newframes(Expr1(e), r, s);
   loop();
   let valore= top(top(tempvalstack)) in
    pop(tempvalstack);
    let st = topstore() in
     popstore();
     (valore, st, !currentheap)

let semden (e,(r:  dval env), (s:  mval store), h) =
   initstate();
   currentheap := h;
   pushstore(emptystore(Undefined));
   push(emptystack(tdframesize(e),Unbound),tempdvalstack);
   newframes(Exprd1(e), r, s);
   loop();
   let valore= top(top(tempdvalstack)) in
    pop(tempdvalstack);
    let st = topstore() in
     popstore();
     (valore, st, !currentheap)

let semcl (cl,(r:  dval env), (s:  mval store), h) =
   initstate();
   currentheap := h;
   pushstore(emptystore(Undefined));
   newframes(labelcom(cl), r, s);
   loop();
   let st = topstore() in
    popstore();
    (st, !currentheap)

let semdv(dl, r, s, h) =
   initstate();
   currentheap := h;
   newframes(labeldec(dl), r, s);
   loop();
   let st = topstore() in
    popstore();
    let rt = topenv() in
     popenv();
     (rt, st, !currentheap)

let semdr(dl, r, s, h) =
   initstate();
   currentheap := h;
   newframes(Rdecl(dl), r, s);
   loop();
   let st = topstore() in
    popstore();
```

```
    let rt = topenv() in
     popenv();
     (rt, st, !currentheap)

let semdl((dl,rl), r, s, h) =
  initstate();
  currentheap := h;
  newframes(Rdecl(rl), r, s);
  push(labeldec(dl),top(cstack));
  loop();
  let st = topstore() in
   popstore();
   let rt = topenv() in
    popenv();
    (rt, st, !currentheap)

let semc((c:  com), (r:dval env), (s:  mval store), h) =
  initstate();
  currentheap := h;
  pushstore(emptystore(Undefined));
  newframes(labelcom([c]), r, s);
  loop();
  let st = topstore() in
   popstore();
   (st, !currentheap)

let semb ((dl, rdl, cl), r, s, h) =
  initstate();
  currentheap := h;
  pushstore(emptystore(Undefined));
  newframes(labelcom(cl), r, s);
  push(Rdecl(rdl), top(cstack));
  push(labeldec(dl), top(cstack));
  loop();
  let st = topstore() in
   popstore();
   (st, !currentheap)

let semclasslist (cl, ( r:  dval env )) =
  let rcl = ref(cl) in
  let rrr = ref(r) in
  let functional rr =
   while not(!rcl = []) do
    let thisclass = List.hd !rcl in
     rcl := List.tl !rcl;
     match thisclass with
      | Class(nome,_,_,_) ->
         rrr := bind(!rrr, nome, makeclass(thisclass, rr))
   done;
```

```
    !rrr in
  let rec rfix = function i -> functional rfix i
  in rfix

let semprog ((cdl,b),r,s,h) = semb(b,semclasslist(cdl,r),s,h)
```

Oltre ad aver aggiunto le funzioni relative agli oggetti, abbiamo modificato le funzioni di valutazione semantica in modo che prendano la heap.
In `semclasslist` il funzionale è definito in modo iterativo, ma c'è comunque un calcolo di punto fisso.

12 Tecniche per il passaggio di parametri

Contenuti del capitolo:

- La tecnica base (nei vari paradigmi): passaggio per costante, per riferimento, di funzioni, procedure e oggetti.
- Altre tecniche:
  - Passaggio per nome: nuovi costrutti, nuovi domini, semantiche dei nuovi costrutti.
  - Argomenti funzionali à la LISP, con scoping dinamico.
  - Passaggio per valore, per risultato e per valore-risultato.

12.1 Passaggio dei parametri

Esistono due possibili meccanismi per condividere imformazioni tra sottoprogrammi diversi: l'utilizzo di *associazioni non locali* o *globali* (quando l'entità da condividere è sempre la stessa), e l'uso di *parametri* (quando l'entità da condividere cambia da attivazione ad attivazione).
Definiamo:

- **Argomenti formali**: lista dei nomi locali usati per riferire dati non locali.
- **Argomenti attuali**: lista di espressioni, i cui valori saranno condivisi.

Fra argomenti formali ed attuali c'è una corrispondenza posizionale[22].
Cos'è dunque il **passaggio di parametri**?

> Il binding, uno alla volta, nell'ambiente ρ tra il parametro formale (locale) ed il valore denotato ottenuto dalla valutazione dell'argomento attuale.

La *regola di scoping* influenza l'identità dell'ambiente (non locale) ρ, man non l'effetto su ρ del passaggio di parametri.
L'associazione per un nome locale viene creata dal passaggio invece che da una dichiarazione.

12.2 La tecnica base di passaggio

Vediamo il caso del passaggio di parametri alle procedure, considerando che il meccanismo è lo stesso anche per funzioni e classi.

In semantica *denotazionale*:

```
type proc = ((dval list) * (mval store)) -> mval store

let rec semden (e:exp) (r:  dval env) (s:  mval store) = match e with
   | Proc(i,b) -> (Dprocval(function (d,s1) ->
        sem b (bindlist (r,i,d)) s1), s)
```

[22] In alcuni linguaggi gli argomenti attuali possono anche essere meno degli argomenti formali.

```
let rec semc (c:com) (r:  dval env) (s:  mval store) = match c with
   | Call(e1,e2) -> let (p,s1) = semden e1 r s in let (v,s2) =
     semlist e2 r s1 in match p with
       | Dprocval (x) -> x (v, a2)
```

In semantica *operazionale*:

```
type proc = exp * dval env

let rec semden (e:exp) (r:  dval env) (s:  mval store) = match e with
   | Proc(i,b) -> (Dprocval(Proc(i,b), r) s)

let rec semc (c:com) (r:  dval env) (s:  mval store) = match c with
   | Call(e1,e2) -> let (p,s1) = semden (e1, r, s) in let (v,s2) =
     semlist(e2, r, s1) in match p with
       | Dprocval (Proc(i,b), x) -> semb(b, bindlist (x, i, v), s)
```

In entrambi i casi viene valutata la lista di argomenti nello stato corrente, ottenendo una lista di *dval* (*eval* nel linguaggio funzionale), per poi costruire un nuovo ambiente legando ogni identificatore nella lista dei formali `x` con il corrispondente valore della lista di *dval* `v` (identificatore e valore, ovviamente, devono avere lo stesso tipo).

12.3 Varie tecniche di passaggio

I valori che possono essere passati:

```
type dval =
   | Dint of int
   | Dbool of bool
   | Dloc of loc
   | Dfunval of efun
   | Dprocval of proc
```

A seconda del tipo del valore che viene passato si ottengono varie modalità note, tutte semanticamente equivalenti.

Passaggio per **costante**:

- Il parametro formale ha come tipo un valore tradizionale non modificabile.
- L'espressione corrispondente valuta ad un `Dint` o un `Dbool`.
- L'oggetto denotato non può essere modificato.
- Questo tipo di passaggio esiste in alcuni linguaggi imperativi ed in *tutti* i linguaggi funzionali[23].

Passaggio per **riferimento** (e di oggetti)

[23] Anche il passaggio ottenuto via pattern matching ed unificazione è quasi sempre un passaggio per costante.

- Il parametro formale ha come tipo un valore modificabile (locazione).
- L'espressione corrispondente valuta ad un `Dloc`.
- L'oggetto denotato può essere modificato dal sottoprogramma.
- Crea *aliasing*, poichè il parametro formale è un nuovo nome per una locazione già esistente.
- Produce *effetti laterali*, poiché le modifiche effettuate attraverso il parametro formale si ripercuotono all'esterno.
- Questo tipo di passaggio esiste in *quasi tutti* linguaggi imperativi .
- Il passaggio di oggetti si comporta esattamente allo stesso modo.

Passaggio di **funzioni**, **procedure** (e classi)

- Il parametro formale ha come tipo una funzione, una procedura o una classe.
- L'espressione corrispondente valuta ad un `Dfunval` o un `Dprocval`.
- L'oggetto denotato è una funzione in semantica denotazionale, o una chiusura in semantica operazionale e può essere ulteriormente passato come parametro o essere attivato (`Apply`, `Call`, `New`).
- Nei linguaggi imperativi ed orientati agli oggetti di solito anche le funzioni non sono esprimibili.
- In LISP (funzionale con scoping dinamico) gli argomenti funzionali si passano in un modo più complesso.

In aggiunta al meccanismo base già visto, esistono altre tecniche di passaggio che non coinvolgono solo l'ambiente (i passaggi per **valore** e **risultato** coinvolgono anche la memoria), non valutano il parametro attuale (passaggio per **nome**), cambiano il tipo del valore passato (argomenti funzionali in LISP).

12.4 Passaggio per nome

Con il passaggio per nome l'espressione passata in corrispondenza di un parametro `x` non viene valutata al momento del passaggio, ma si valuta ogni volta che (eventualmente) si incontra una occorrenza del parametro `x`.
Questo consente di definire sottoprogrammi (e funzioni) *non-stretti* su uno (o più di uno) dei loro argomenti. Come nella regola di valutazione esterna delle espressioni, l'attivazione può dare un risultato definito anche se l'espressione, se valutata, darebbe un valore indefinito (errore, eccezione, non terminazione), semplicemente perchè in una particolare esecuzione `x` non viene mai incontrato.
Dato che l'espressione si valuta nella *memoria corrente* quando viene incontrato il parametro formale, il passaggio per nome e per costante possono avere semantiche diverse anche nei sottoprogrammi stretti (se l'espressione contiene variabili che possono essere modificate, come non locali, dal sottoprogramma). Diverse occorrenze del parametro formale possono infatti dare valori diversi.

12.5 Passaggio per nome in semantica denotazionale

Come abbiamo già detto, l'espressione passata in corrispondenza di un parametro formale `x` non viene valutata al momento del passaggio, bensì quando viene incontrata un'occorrenza di `x`. Un'espressione non valutata è quindi una funzione

`mval store -> eval`

e la valutazione di `x` si effettua applicando la funzione denotata da `x` alla memoria corrente. In un linguaggio funzionale puro, una espressione non valutata è una funzione `unit -> eval` (nel linguaggio funzionale non ho lo *store*), e la valutazione dell'occorrenza di `x` si effettua ugualmente applicando la funzione denotata da `x` (a nulla).
A questo punto definiamo la semantica denotazionale del passaggio per nome, limitandoci a vedere novità e modifiche che il nuovo meccanismo richiede.
Aggiungiamo due nuovi costrutti al dominio sintattico delle espressioni: per passare un'espressione senza valutarla e per indicare le occorrenze del parametro formale per nome:

```
type exp = ...
   | Namexp of exp
   | Nameden of ide

type dval = Unbound
   | Dint of int
   | ...
   | Dnameval of namexp
and namexp = mval store -> eval
```

Abbiamo anche aggiunto un nuovo valore nel dominio *dval*.
Definiamo ora la funzione di valutazione semantica, in semantica denotazionale, per i nuovi costrutti:

```
let rec sem (e:exp) (r:dval env) (s:mval store) = match e with
   | ...
   | Nameden(i) ->match applyenv(r,i) with Dnameval(f) -> f s
   | ...
and semden (e:exp) (r:dval env) (s:mval store) = match e with
   | Namexp e1 -> (Dnameval(function s1 -> sem e1 r s1), s)
   | ...
```

Le funzioni di valutazione semantica definite hanno il seguente tipo:

```
val sem:  exp -> dval Funenv.env -> mval Funstore.store -> eval = <fun>

val semden:  exp -> dval Funenv.env -> mval Funstore.store ->
      dval * mval Funval.store = <fun>
```

12.6 Passaggio per nome in semantica operazionale

A differenza del caso denotazionale, non possiamo in semantica operazionale portarci dietro l'espressione non valutata in forma di funzione. Siamo infatti costretti, in questo caso come già altre volte, a sfruttare l'informazione sintattica, rappresentando l'espressione non valutata attraverso una chiusura

```
exp * dval env
```

La valutazione dell'occorrenza di un parametro formale `x` si effettua quindi valutando la chiusura denotata da `x` nella memoria corrente (espressione della chiusura, ambiente della chiusura, memoria corrente). Anche in un linguaggio funzionale puro una espressione non valutata è una chiusura, ed, in mancanza di *store*, si valuta l'espressione della chiusura nell'ambiente della chiusura.
Come fatto per la semantica denotazionale ci limitiamo a vedere le sole "novità" relative all'introduzione della tecnica di passaggio di parametri per nome.
Definiamo i domini sintattici:

```
type exp = ...
   | Namexp of exp
   | Nameden of ide

type dval = Unbound
   | Dint of int
   | ...
   | Dnameval of namexp
and namexp = exp * dval env
```

Rispetto al caso denotazionale cambia *namexp*, riportato al prim'oridine.
Definiamo quindi le funzioni di valutazione semantica, in semantica operazionale:

```
let rec sem ((e:exp),(r:dval env),(s:mval store)) = match e with
   | ...
   | Nameden(i) ->match applyenv(r,i) with
      Dnameval(e1,r1) -> sem(e1,r1,s)
   | ...
and semden ((e:exp),(r:dval env),(s:mval store)) = match e with
   | Namexp e1 -> (Dnameval(e1,r), s)
   | ...
```

Le funzioni definite hanno il seguente tipo:

```
val sem:  exp * dval Funenv.env * mval Funstore.store -> eval = <fun>

val semden:  exp * dval Funenv.env * mval Funstore.store ->
      dval * mval Funval.store = <fun>
```

12.7 Considerazioni sul passaggio per nome

Un'espressione passata per nome è chiaramante simile alla definizione di una funzione senza parametri, che viene applicata ogni volta che si incontra il parametro formale. Dal punto di vista della semantica, infatti, il passaggio per nome viene trattato esattamente con le stesse soluzioni introdotte per le funzioni: funzioni in semantica denotazionale, chiusura in semantica operazionale.
L'ambiente fissato (nella definizione della funzione o nella chiusura) è quello di passaggio (che per le espressioni è l'equivalente della definizione). Questo fa sì che, mentre la semantica delle funzioni è influenzata dalla regola di *scoping*, ciò non sia vero per le espressioni passate per nome, che vengono *comunque valutate nell'ambiente di passaggio*, anche con lo scoping dinamico!

Il passaggio per nome è previsto in linguaggi come ALGOL e LISP, ed è alla base dei meccanismi di valutazione lazy di linguaggi funzionali moderni come Haskell. In ML il passaggio per nome può essere simulato semplicemente passando funzioni senza argomenti.

12.8 Argomenti funzionali à la LISP

Come abbiamo detto, LISP ha lo *scoping dinamico*, quindi i domini delle funzioni sono:

- In *semantica denotazionale*: `type efun = eval env -> eval list -> eval`
- In *semantica operazionale*: `type efun = exp`

Un argomento formale `x` di tipo funzionale dovrebbe denotare un *eval* della forma `Funval(efun)`.
Se `x` viene successivamente applicato, il suo ambiente di valutazione dovrebbe correttamente essere quello del momento dell'applicazione. La semantica di LISP prevede invece che l'ambiente dell'argomento funzionale venga *congelato al momento del passaggio*. Questo porta alla necessità di definire due diversi domini per funzioni ed argomenti funzionali:

- In *semantica denotazionale*: `type funarg = eval list -> eval`
- In *semantica operazionale*: `type funarg = exp * eval env`

I domini degli argomenti funzionali sono quindi identici ai domini delle funzioni con *scoping statico*, ma l'ambiente rilevante (ad esempio, quello della chiusura) è quello del *passaggio* e non quello della definizione.
Quando una funzione viene passata come argomento di un'altra funzione (passando il nome della funzione, una lambda astrazione o un'espressione che restituisce una funzione), viene immediatamente "chiusa" con l'ambiente corrente: applicandola all'ambiente in semantica denotazionale, inserendo l'ambiente nella chiusura in semantica denotazionale.
A livello implementativo coesisteranno funzioni rappresentate dal solo codice ed argomenti funzionali rappresentati da chiusure (codice, ambiente).

In alcune implementazioni di LISP si segue la strada appena descritta anche per i valori restituiti da un'applicazione di funzione che restituisce un valore funzionale.

In questi casi, infatti, l'applucazione della funzione restituisce un valore *funarg* anzichè un *efun*. Anche questa scelta complica notevolmente l'implementazione del linguaggio: in termini di implementazione dell'ambiente ed in relazione alla generazione dei problemi legati allo scoping statico (retention, vedi 13.3), senza averne i vantaggi (virificabilità, ottimizzazioni).

12.9 Passaggio per valore

Il **passaggio per valore** ha a che fare con i valori *modificabili* e non esiste quindi nei linguaggi funzionali.
Nel passaggio per valore il parametro attuale è un *valore* di tipo t mentre il parametro formale `x` è una *variabile* di tipo t. L'argomento e parametro formale hanno quindi tipi diversi e non è possibile fare il passaggio con un'operazione di `bind` nell'ambiente.
Di fatto `x` è il nome di una variabile locale alla procedura che semanticamente viene creata prima del passaggio. Il passaggio diventa quindi l'assegnamento del valore dell'argomento alla locazione denotata dal parametro formale.
Se implementato correttamente, il passaggio non coinvolge la memoria, non viene creato *aliasing* e non ci sono effetti laterali anche se il valore denotato dal parametro formale è modificabile (non è così nel passaggio per costante) ed, infine, permette il passaggio di informazione *solo dal chiamante al chiamato*.

12.10 Passaggio per valore-risultato

Rispetto al passaggio per valore, consente di trasmettere anche l'informazione all'indietro *dal sottoprogramma chiamato al chiamante*, evitando gli effetti laterali del passaggio per riferimento.
In questo caso, sia il parametro formale `x` che il parametro attuale `y` sono variabili di tipo t, con `x` variabile locale al sottoprogramma chiamato.
Al momento della chiamata del sottoproggramma viene effettuato il passaggio *per valore* (`x:=!y`), copiando il valore della locazione (esterna) denotata da `y`, nella locazione (locale) denotata da `x`.
Al momento del ritorno del sottoprogramma si effettua l'assegnamento inverso (`y:=!x`).

Il passaggio per valore-risultato ha un effetto simile al passaggio per riferimento (trasmissione nei due sensi tra chiamante e chiamato) senza però creare aliasing. La variabile locale contiene infatti una copia del valore della variabile non locale, di modo che, durante l'esecuzione della procedura, le due variabili denotano locazioni distinte. Solo al momento del ritorno la variabile non locale riceve il valore di quella locale.
Nel passaggio per riferimento, invece, viene creato aliasing ed i due nomi denotano esattamente la stessa locazione.

13 Implementazione dell'ambiente nel linguaggio funzionale

Contenuti del capitolo:

- Ambiente locale dinamico con scoping statico:
  - Cosa serve (catena statica).
  - Un problema con le funzioni esprimibili: la retention).
  - Implemenazione.
  - Come cambia l'interprete iterativo.
- Ottimizzazioni eseguibili durante la compilazione: traduzione dei riferimenti ed eliminazione dei nomi.
- (Digressione) Ambiente locale dinamico con scoping dinamico:
  - Implementazione standard.
  - Cenni ad altre implementaizioni.
  - Eventuali ottimizzazioni.

13.1 Ambiente locale (dinamico) e non locale

Per ogni attivazione (entrata in un blocco o applicazione di funzione) abbiamo attualmente l'intero ambiente implementato come funzione. In accordo con la semantica dell'ambiente locale dinamico possiamo inserire nel record di attivazione una tabella che implementa il solo ambiente locale, più il necessario per reperire l'ambiente non locale, in accordo con la regola di scoping.
Quando l'attivazione termina (uscita dal blocco o ritorno dell'applicazione di funzione) possiamo eliminare l'ambiente locale (e cose eventuamente associate) insieme a tutte le altre informazioni contenute nel record di attivazione.
Vediamo più nel dettaglio come andiamo ad implementare l'ambiente.
L'**ambiente locale** contiene: nel caso del blocco (`Let`) una sola associazione, nel caso dell'applicazione (`Apply`) tante associazioni quanti sono i parametri.
Rappresentiamo quindi l'ambiente locale con una coppia di array: l'array dei *nomi* e l'array dei *valori denotati*.
Dato che la pila dei record di attivazione è realizzata con varie pile paralelle, la pila di ambienti `envstack` è rimpiazzata (per ora) da due pile di ambienti locali:

- `namestack`: pila di array di identificatori.
- `evalstack`: pila di array di valori denotati.

In ogni istante le pile rappresentano la sequenza di ambienti locali corrispondenti alla catena di attivazioni (**catena dinamica**).
Se adottiamo lo **scoping dinamico** non abbiamo bisogno di altro, visto che se un identificatore manca nell'ambiente locale può essere ricercato negli ambienti locali che lo precedono nella pila. In questo modo il primo ambiente (eventualmente) trovato identifica l'associazione corretta, perchè è l'ultima creata nel tempo.
Con lo **scoping statico**, invece, abbiamo bisogno di un meccanismo differente per poter gestire l'ambiente esterno. Nel caso delle funzioni, infatti, l'ambiente non

locale giusto non è quello che precede quello locale sulla pila, ma quello che esisteva al momento dell'astrazione. Tale ambiente ci è noto a tempo di esecuzione, perchè è contenuto nella chiusura (che è la semantica della funzione che applichiamo).

A questo punto è necessario domandarsi se è possibile essere certi che l'ambiente contenuto nella chiusura esista sempre sulla pila nel momento in cui si applica la funzione. Benchè la risposta reale sia "non necessariamente", è facile convincersi che la risposta sia "sì" se ci limitiamo ad applicare funzioni reperite attraverso il loro nome (funzioni denotate). Per la semantica del `Let` siamo infatti sicuri che l'applicazione di un `Den ide` può essere eseguita solo se è visibile (e quindi contenuto nella pila) l'ambiente ρ che contiene `ide`. L'ambiente contenuto nella chiusura può essere: ρ stesso se la funzione è ricorsiva, oppure quello che precede ρ nella pila (cioè quello in cui è valutato il `Let`) se la funzione non è ricorsiva. L'implementazione dovrà garantire che l'ambiente contenuto nella chiusura sia comunque presente nella pila.

A differenza di quanto avviene in caso di scoping dinamico, con lo scoping statico gli ambienti locali visibili come non locali formano (quasi sempre) una sottosequenza rispetto a quanto rappresentato dalle pile nella loro totalità (la sequenza degli ambienti locali, corrispondente alla catena di attivazioni). Tale sottosequenza riproduce a run time la struttura statica di annidamento fra blocchi e funzioni.
Se l'attivazione corrente è un **blocco**, l'ambiente locale giusto è quello precedente sulla testa della pila.
Se, invece, l'attivazione corrente è relativa all'applicazione di una **funzione di nome** `f`, deve esserci sulla pila una attivazione del blocco o applicazione in cui sia stata dichiarata `f`: se l'applicazione era relativa ad un nome locale tale attivazione precede quella di `f` sulla pila, se invece l'applicazione era relativa ad un nome non locale ci possono essere in mezzo altre attivazioni che non ci interessano.
Se, infine, l'attivazione corrente è relativa all'applicazione di una **funzione senza nome**, ottenuta dalla valutazione di un'espressione (le funzioni sono esprimibili), dobbiamo fare in modo che sulla pila ci sia anche l'ambiente delle chiusura.

13.2 Catena dinamica e catena statica

Abbiamo già detto che in ogni istante le pile rappresentano la sequenza di ambienti locali corrispondenti alla catena di attivazioni (**catena dinamica**).
Gli ambienti locali visibili come non locali secondo la regola di scoping statico formano (quasi) una sottosequenza di quella contenuta nella pila. Per ricostruire la struttura statica e determinare correttamente l'ambiente non locale, si associa ad ogni record di attivazione un puntatore (**catena statica**) al giusto ambiente locale sulla pila: quello precedente sulla pila se entro in un *blocco*, quello contenuto nella chiusura se applico una *funzione*.
Il tipo *ambiente* ha ora come implementazione un semplice intero (anche nelle chiusure), indice negli array che realizzano le pile.
Aggiungiamo quindi una nuova pila nell'implementazione dell'ambiente:

- `slinkstack`: pila di (puntatori ad) ambienti.

13.3 Funzioni esprimibili e retention

In un linguaggio funzionale di ordine superiore, come il nostro, è possibile che la valutazione di un'applicazione di funzione

`Appl(e1,e2)`

ritorni una funzione, cioè un valore (chiusura) del tipo `Funval(e,`ρ`)` in cui

- `e`: è un'espressione di tipo `Fun`.
- ρ: è l'ambiente in cui la funzione è stata costruita, cioè l'ambiente contenuto nel frame dell'applicazione `Appl(e1,e2)`, che serve per risolvere eventuali riferimenti non locali di `e`.

Quando la valutazione dell'applicazione `Appl(e1,e2)` ritorna, normalmente si dovrebbero poppare tutte le pile, comprese quelle che realizzano l'ambiente. In questo caso però non è possibile eseguire il pop sulle piole relative all'ambiente perchè l'ambiente ρ, attualmente sulla testa della pila, è utilizzato dentro la chiusura che si trova nella pila dei valori temporanei. In tale situazione è quindi necessario ricorrere alla **retention**, in cui l'ambiente locale viene conservato:

- Le teste delle pile che ralizzano l'ambiente vengono mantenute ed etichettate come `Retained` in un'ulteriore pila parallela `tagstack`.
- La computaizone continua in uno stato in cui l'ambiente corrente non è necessariamente quello in testa alle pile.
- Gli ambienti conservati vengono prima o poi eliminati, riportando le pile di ambienti nello stato normale.

13.4 Realizzazione dell'ambiente

Riassumendo quanto detto, utilizziamo le seguenti pile per l'implementazione dell'ambiente, al posto di `envstack`:

- `namestack`: pila di array di identificatori.
- `evalstack`: pila di array di valori denotati.
- `slinkstack`: pila di ambienti.
- `tagstack`: pila di etichette per la retention.

Vediamo l'implementazione delle pile:

```
type 't env = int

let (currentenv:  eval env ref) = ref(0)
let namestack = emptystack(stacksize,[| ``dummy'' |])
let evalstack = emptystack(stacksize,[| Unbound |])
let slinkstack = emptystack(stacksize, !currentenv)
type tag = Retained| Standard
let tagstack = emptystack(stacksize, Standard)
```

Vediamo ora come cambiano le operazioni già utilizzate in precedenza per la gestione dell'ambiente:

```
let applyenv ((x:  eval env), (y:  ide)) =
   let n = ref(x) in
   let den = ref(Unbound) in
    while !n > -1 do
      let lenv = access(namestack,!n) in
      let nl = Array.length lenv in
      let index = ref(0) in
        while !index < nl do
          if Array.get lenv !index = y then
            (den := Array.get (access(evalstack,!n)) !index;
            index := nl)
          else index := !index + 1
        done;
        if not(!den = Unbound) then n := -1
        else n := access(slinkstack,!n)
    done;
    !den

let bind ((r:eval env),i,d) =
   push(Array.create 1 i,namestack);
   push(Array.create 1 d,evalstack);
   push(r,slinkstack);
   push(Standard,tagstack);
   (lungh(namestack):  eval env)

let bindlist(r, il, el) =
   let n = List.length il in
   let ii = Array.create n ``dummy'' in
   let dd = Array.create n Unbound in
   let ri = ref(il) in
   let rd = ref(el) in
   let index = ref(0) in
    while !index < n do
      let i = List.hd !ri in
      let d = List.hd !rd in
        ( ri := List.tl !ri;
          rd := List.tl !rd;
          Array.set ii !index i;
          Array.set dd !index d;
          index := !index + 1)
    done;
    push(ii, namestack);
    push(dd,evalstack);
    push(r,slinkstack);
    push(Standard,tagstack);
    (lungh(namestack):  eval env)

let emptyenv(x) = currentenv := -1; svuota(namestack);
```

```
  svuota(evalstack); svuota(slinkstack); svuota(tagstack);
```

E' importante notare come questa volta l'operazione `applyenv` ritrovi l'associazione non locale seguendo all'indietro i *puntatori di catena statica*.
Aggiungiamo ora alcune operazioni che permettono la gestione (parziale) della *retention*:

```
let retained (n:eval env) = access(tagstack,n) = Retained

let retain () =
  setta(tagstack, !currentenv, Retained);
  let cont = ref(lungh(namestack)) in
   while !cont > -1 & retained(!cont)
   do cont := !cont - 1 done;
   currentenv := !cont

let to_be_retained (v:eval) = match v with
  | Funval (e, r1) -> not(r1 < !currentenv)
  | _ -> false (*1*)

let retainorpopenv (r, valore) =
  if (topenv() = r) then () else
   if !currentenv < lungh(namestack) then retain() else
    if to_be_retained(valore) then retain() else
     (popenv();
     let index = ref(!currentenv) in
      while !index > r do
       if retained(!index) then
        (currentenv := !currentenv - 1;
         index := !index -1)
       else (index := r) done )

let collectretained (r) = let index =
  ref(lungh(namestack)) in
  while !index > r do
   pop(namestack); pop(evalstack);
   pop(slinkstack); pop(tagstack);
   index := !index -1 done;
  currentenv := lungh(namestack)
```

Commenti al codice:

1. Si potrebbero recuperare alcuni frames retained.

13.5 Novità nell'interprete iterativo

Il fatto che l'ambiente sia un intero (puntatore nelle pile degli ambienti) ci obbliga innanzi tutto a modificare l'implementazione delle funzioni ricorsive, eliminando il calcolo del **punto fisso**, necessario a determinare l'ambiente da inserire nella chiusura. Possiamo infatti inserire nella chiusura direttamente la prossima posizione

nelle pile degli ambienti, dove verrà inserita esattamente l'associazione per il nome della funzione ricorsiva. L'ambiente corrispondente all'indice utilizzato non esiste ancora, ma ci sarà quando la chiusura sarà utilizzata.
La funzione `makefunrec`, diventa:

```
makefunrec (f, (a:exp), (x:eval env)) =
   makefun(a, ((lungh(namestack) +1):eval env))
```

Un'altra novità nelll'interprete iterativo è rappresentata dalla gestione della **retention**:

```
let sem ((e:exp), (r:eval env)) =
   push(emptystack(1,Unbound),tempvalstack);
   newframes(e,r);
   while not(empty(cstack)) do
    while not(empty(top(cstack))) do
      ...
    done;
    let valore= top(top(tempvalstack)) in
     pop(top(tempvalstack));
     pop(cstack);
     pop(tempvalstack);
     push(valore,top(tempvalstack));
     retainorpopenv (r, valore)
   done;
   collectretained(r);
   let valore = top(top(tempvalstack)) in
    pop(tempvalstack); valore
```

13.6 Interprete iterativo

Di seguito è riportato il codice completo per l'interprete iterativo del linguaggio funzionale, con l'implementazione dell'ambiente appena descritta:

```
type 't env = int

and eval =
   | Int of int
   | Bool of bool
   | Unbound
   | Funval of efun

and efun = exp * (eval env)

type labeledconstruct =
   | Expr1 of exp
   | Expr2 of exp
```

```
let (cstack:  labeledconstruct stack stack) =
  emptystack(stacksize,emptystack(1,Expr1(Eint(0))))

let (tempvalstack:  eval stack stack) =
  emptystack(stacksize,emptystack(1,Unbound))

let (currentenv:  eval env ref) = ref(0)

let namestack = emptystack(stacksize,[| ``dummy'' |])

let evalstack = emptystack(stacksize,[| Unbound |])

let slinkstack = emptystack(stacksize, !currentenv)

type tag = Retained| Standard

let tagstack = emptystack(stacksize, Standard)

let applyenv ((x:  eval env), (y:  ide)) =
  let n = ref(x) in
  let den = ref(Unbound) in
    while !n > -1 do
      let lenv = access(namestack,!n) in
      let nl = Array.length lenv in
      let index = ref(0) in
        while !index < nl do
          if Array.get lenv !index = y then
            (den := Array.get(access(evalstack,!n))!index;
            index := nl)
          else index := !index + 1
        done;
        if not(!den = Unbound) then n := -1
        else n := access(slinkstack,!n)
    done;
    !den

let bind ((r:eval env),i,d) =
  push(Array.create 1 i,namestack);
  push(Array.create 1 d,evalstack);
  push(r,slinkstack);
  push(Standard,tagstack);
  (lungh(namestack):  eval env)

let bindlist(r, il, el) =
  let n = List.length il in
  let ii = Array.create n ``dummy'' in
  let dd = Array.create n Unbound in
  let ri = ref(il) in
  let rd = ref(el) in
```

```
   let index = ref(0) in
    while !index < n do
      let i = List.hd !ri in
      let d = List.hd !rd in
        ( ri := List.tl !ri;
          rd := List.tl !rd;
          Array.set ii !index i;
          Array.set dd !index d;
          index := !index + 1)
    done;
    push(ii, namestack);
    push(dd,evalstack);
    push(r,slinkstack);
    push(Standard,tagstack);
    (lungh(namestack):  eval env)

let emptyenv(x) = currentenv := -1;
   svuota(namestack); svuota(evalstack);
   svuota(slinkstack); svuota(tagstack);
   !currentenv

let svuotaenv() =
   svuota(namestack); svuota(tagstack);
   svuota(evalstack); svuota(slinkstack)

let topenv() = !currentenv

let popenv () = pop(namestack);
   pop(evalstack);
   pop(slinkstack);
   pop(tagstack);
   currentenv := !currentenv - 1

let pushenv(r) =
   if r = !currentenv then
    (push([||],namestack);
    push([||],evalstack);
    push(Standard,tagstack);
    push(r,slinkstack);
    currentenv := lungh(namestack) )
   else currentenv := r

let pushargs ((b:  exp list),continuation) = let br = ref(b) in
   while not(!br = []) do
    push(Expr1(List.hd !br),continuation);
    br := List.tl !br
   done

let getargs ((b:  exp list),(tempstack:  eval stack)) = let br = ref(b)
```

```
in
let er = ref([]) in
   while not(!br = []) do
    let arg=top(tempstack) in
      pop(tempstack); er := !er @ [arg];
      br := List.tl !br
   done;
   !er

let retained (n:eval env) = access(tagstack,n) = Retained

let retain () =
   setta(tagstack, !currentenv, Retained);
   let cont = ref(lungh(namestack)) in
    while !cont > -1 & retained(!cont) do cont := !cont - 1 done;
    currentenv := !cont

let to_be_retained (v:eval) = match v with
   | Funval (e, r1) -> not(r1 < !currentenv)
   | _ -> false

let retainorpopenv (r, valore) =
   if (topenv() = r) then () else
    if !currentenv < lungh(namestack) then retain() else
      if to_be_retained(valore) then retain() else
        (popenv();
        let index = ref(!currentenv) in
          while !index > r do
            if retained(!index)
            then (currentenv := !currentenv - 1;
                 index := !index -1)
            else (index := r) done )

let collectretained (r) = let index =
   ref(lungh(namestack)) in
   while !index > r do
    pop(namestack); pop(evalstack);
    pop(slinkstack); pop(tagstack);
    index := !index -1 done;
   currentenv := lungh(namestack)

let newframes(e,rho) =
   let cframe = emptystack(cframesize(e),Expr1(e)) in
   let tframe = emptystack(tframesize(e),Unbound) in
    push(Expr1(e),cframe);
    push(cframe,cstack);
    push(tframe,tempvalstack);
    pushenv(rho)
```

```
let makefun ((a:exp),(x:eval env)) =
   (match a with
    | Fun(ii,aa) ->
             Funval(a,x)
    | _ -> failwith (``Non-functional object''))

let applyfun ((ev1:eval),(ev2:eval list)) =
   ( match ev1 with
      | Funval(Fun(ii,aa),r) -> newframes(aa,bindlist(r, ii, ev2))
      | _ -> failwith (``attempt to apply a non-functional object''))

let makefunrec (f, (a:exp),(x:eval env)) =
   makefun(a, ((lungh(namestack) + 1):  eval env))

let sem ((e:exp), (r:eval env)) =
   push(emptystack(1,Unbound),tempvalstack);
   newframes(e,r);
   while not(empty(cstack)) do
    while not(empty(top(cstack))) do
     let continuation = top(cstack) in
     let tempstack = top(tempvalstack) in
     let rho = topenv() in
      (match top(continuation) with
       |Expr1(x) ->
          (pop(continuation);
          push(Expr2(x),continuation);
          (match x with
             | Iszero(a) -> push(Expr1(a),continuation)
             | Eq(a,b) -> push(Expr1(a),continuation);
                push(Expr1(b),continuation)
             | Prod(a,b) -> push(Expr1(a),continuation);
                push(Expr1(b),continuation)
             | Sum(a,b) -> push(Expr1(a),continuation);
                push(Expr1(b),continuation)
             | Diff(a,b) ->
                push(Expr1(a),continuation);
                push(Expr1(b),continuation)
             | Minus(a) -> push(Expr1(a),continuation)
             | And(a,b) -> push(Expr1(a),continuation);
                push(Expr1(b),continuation)
             | Or(a,b) -> push(Expr1(a),continuation);
                push(Expr1(b),continuation)
             | Not(a) -> push(Expr1(a),continuation)
             | Ifthenelse(a,b,c) ->
                push(Expr1(a),continuation)
             | Let(i,e1,e2) ->
                push(Expr1(e1),continuation)
             | Appl(a,b) -> push(Expr1(a),continuation);
                pushargs(b,continuation)
```

```
      | _ -> ()))
|Expr2(x) ->
   (pop(continuation);
   (match x with
      | Eint(n) -> push(Int(n),tempstack)
      | Ebool(b) -> push(Bool(b),tempstack)
      | Den(i) ->
         push(applyenv(rho,i),tempstack)
      | Iszero(a) ->
         let arg=top(tempstack) in
          pop(tempstack);
          push(iszero(arg),tempstack)
      | Eq(a,b) ->
         let firstarg=top(tempstack) in
          pop(tempstack);
          let sndarg=top(tempstack) in
            pop(tempstack);
            push(equ(firstarg,sndarg),tempstack)
      | Prod(a,b) ->
         let firstarg=top(tempstack) in
          pop(tempstack);
          let sndarg=top(tempstack) in
            pop(tempstack);
            push(mult(firstarg,sndarg),tempstack)
      | Sum(a,b) ->
         let firstarg=top(tempstack) in
          pop(tempstack);
          let sndarg=top(tempstack) in
            pop(tempstack);
            push(plus(firstarg,sndarg),tempstack)
      | Diff(a,b) ->
         let firstarg=top(tempstack) in
          pop(tempstack);
          let sndarg=top(tempstack) in
            pop(tempstack);
            push(diff(firstarg,sndarg),tempstack)
      | Minus(a) ->
         let arg=top(tempstack) in
          pop(tempstack);
          push(minus(arg),tempstack)
      | And(a,b) ->
         let firstarg=top(tempstack) in
          pop(tempstack);
          let sndarg=top(tempstack) in
            pop(tempstack);
            push(et(firstarg,sndarg),tempstack)
      | Or(a,b) ->
         let firstarg=top(tempstack) in
          pop(tempstack);
```

```
                    let sndarg=top(tempstack) in
                      pop(tempstack);
                      push(vel(firstarg,sndarg),tempstack)
                | Not(a) ->
                   let arg=top(tempstack) in
                    pop(tempstack);
                    push(non(arg),tempstack)
                | Ifthenelse(a,b,c) ->
                   let arg=top(tempstack) in
                    pop(tempstack);
                    if typecheck(``bool'',arg) then
                      (if arg = Bool(true)
                      then push(Expr1(b),continuation)
                      else push(Expr1(c),continuation))
                    else failwith (``type error'')
                | Fun(i,a) ->
                   push(makefun(Fun(i,a),rho),tempstack)
                | Rec(f,e) ->
                   push(makefunrec(f,e,rho),tempstack)
                | Let(i,e1,e2) -> let arg=top(tempstack) in
                   pop(tempstack);
                   newframes(e2,bind(rho, i, arg))
                | Appl(a,b) -> let firstarg=top(tempstack) in
                   pop(tempstack);
                   let sndarg=getargs(b,tempstack) in
                    applyfun(firstarg,sndarg))))
    done;
    let valore= top(top(tempvalstack)) in
     pop(top(tempvalstack));
     pop(cstack);
     pop(tempvalstack);
     push(valore,top(tempvalstack));
     retainorpopenv (r, valore)
  done;
  collectretained(r);
  let valore = top(top(tempvalstack)) in
   pop(tempvalstack); valore
```

13.7 Un esempio che non funziona senza retention

Consideriamo il seguente esempio:

```
# sem(
   Appl(
    Appl(
     Fun([``x''],
      Fun([``y''], Sum(Den(``x''),Den(``y'')))),
     [Eint 3]),
```

```
    [Eint 5])),
  emptyenv Unbound);;
```

La seconda applicazione di funzione pusha sulla pila dei temporanei il valore

```
Funval(Fun([''y''], Sum(Den(''x''),Den(''y'')))), r)
```

ed `r` punta ad un ambiente che contiene l'associazione fra `x` e `Dint 3`.
Senza retention l'ambiente dell'applicazione considerata viene poppato e rimpiazzato da quello dell'applicazione precedente, in cui il link statico punta (per errore) a se stesso. L'applicazione di tale ambiente porta ad un *ciclo infinito*, visto che `applyenv` cerca nell'ambiente puntato dal link statico se non trova il nome.

13.8 Analisi statiche ed ottimizzazioni

Se lo scping è statico è possibile per il compilatore controllare che ogni riferimento ad un nome abbia effettivamente un'associazione, inferire o controllare il tipo per ogni errore ed, eventualmente, segnalare gli errori.
Sono inoltre possibili ottimizzazioni legate alla seguente proprietà:

> Dato che ogni attivazione di funzione avrà come puntatore di catena statica il puntatore all'ambiente locale in cui la funzione è stata definita (sempre lo stesso per tutte le attivazioni, anche ricorsive, relative alla stessa definizione), il numero di passi necessari a tempo di esecuzione lungo la catena statica per trovare un'associazione non locale per l'identificatore `x`, è *costante*.

Il numero di passi necessari non dipende infatti dalla catena di attivazioni a tempo di esecuzione, ma è invece esattamente la differenza fra le profondità di annidamento del blocco in cui `x` è dichiarato e quello in cui è usato.
La proprietà appena descritta permette la **traduzione** statica di ogni riferimento `Den ide` in una coppia (m, n), dove m è la differenza fra le profondità di nesting dei blocchi (0 se `ide` si trova nell'ambiente locale), ed n è la posizione relativa della dichiarazione di *ide* fra quelle contenute nel blocco.
Interprete o supporto a tempo di esecuzione (la nuova `applyenv`) interpreteranno la coppia (m, n) come segue: "effettua m passi lungo la catena statica, partendo dall'ambiente locale attualmente sulla testa della pila, e resituisci il contenuto dell'elemento iin posizione n nell'array di valori denotati così ottenuto".
In questo modo l'accesso diventa più efficiente poichè non viene più eseguita una ricerca per nome. E' possibile inoltre economizzare nella rappresentazione degli ambienti locali, che non necessitano più di memorizzare i nomi, eliminando la pila `namestack`.
In seguito vedremo come queste ottimizzazioni siano ottenibili gratuitamente attraverso la specializzazione dell'interprete via valutazione parziale.

13.9 L'esercizio di traduzione dei nomi

Dato un programma, per tradurre una specifica occorrenza di `Den ide` bisogna: identificare con precisione la struttura di annidamento ed identificare il blocco o la funzione dove occorre l'associazione per `ide` (o scoprire che non c'è) e vedere la posizione che `ide` occupa in tale ambiente.

Un modo conveniente per ottenere questo risultato consite nel costruire una *catena statica*, ovvero eseguire il programma con l'interprete appena definito limitando l'esecuzione ai soli costrutti che hanno a che fare con l'ambiente, costruendo un nuovo ambiente locale seguendo la struttura statica (`Let`, `Fun`, `Rec`) e non quella dinamica (`Let` e `Apply`), e facendo attezione ad associare ad ogni espressione l'ambiente in cui deve essere valutata.
Il risultato sarà chiaramente diverso dalla costruzione dell'ambiente a tempo di esecuzione, basata sulle applicazioni e non sulle definizioni di funzione, ma sappiamo che per la traduzione dei nomi è la struttura statica quella che conta.

13.10 Scoping dinamico

Con l'implementazione vista abbiamo una pila di record di attivazione, che per l'ambiente è realizzata attraverso `namestack` e `evalstack`, con i record (ambienti locali) creati e distrutti all'ingresso ed all'uscita da blocchi ed attivazioni di funzioni.
L'associazione per il riferimento non locale `x` è, con lo scoping dinamico, la prima associazione per `x` che si trova scorrendo la pila all'indietro.
In questo caso è quindi necessaria una ricerca per nome, e vanno mantenuti i nomi.

Quella appena descritta è l'implementazione più comune in LISP (deep binding), dove però la pila di ambienti locali (*A-list*) è tenuta separata dalla pila dei record di attivazione ed è rappresentata con una S-espressione memorizzata nella heap e modificabile come tutte la altre S-espressioni.
L'implementazione semplice dell'ambiente, unico vantaggio dello scoping dinamico, si complica se introdiciamo le chiusure per trattare gli argomenti funzionali e, soprattutto, i ritorno funzionali à la LISP.

13.11 Shallow binding

E' possibile ridurre il costo di un riferimento non locale, effettuando un accesso diretto senza ricerca, pagando il prezzo della complicazione nella gestione della creazione e distruzione delle attivazioni.
L'ambiente viene realizzato con un'unica tabella centrale che contiene *tutte* le associazioni attive (locali e non locali). La tabella ha una entry per ogni *nome* nel programma, ed in corrispondenza di ogni nome mantiene un *flag* che indica se l'associazione è attiva o meno.
I riferimenti, locali e non, sono compilati in accessi diretti alla tabella a tempo di esecuzione di modo che la ricerca viene sostituita da un semplice controllo del bit di attivazione.
Anche in questo caso i nomi possono sparire , rimpiazzati dalla corrispondente posizione nella tabella.
Con questo tipo di gestione dell'ambiente diventa però molto più comlicata la creazione e la distruzione delle attivazioni, che implica l'utilizzo di un'altra pila (detta *pila nascosta*) in cui salvare l'attivazione corrente nel momento in cui si crea una nuova attivazione, che deve essere riprstinata al ritorno dalla nuova attivazione.
La convenienza dello shallow binding rispetto al deep binding dipende soprattutto dallo "stile di programmazione", ovvero conviene se il programma usa molti riferimenti non locali e pochi `Let` e `Apply`.

13.12 Scoping statico e scoping dinamico

Mentre con lo **scoping dinamico** non è possibile effettuare nessuna analisi ed ottimizzazione a tempo di compilazione, lo **scoping statico** consente diavere programmi: più *sicuri*, grazie al rilevamento statico di errori, alla verifica e l'inferenza dei tipi; più *efficienti* grazie all'eliminazione dei nomi e del meccanismo di ricerca. Il problema della non *compilabilità separata* (ALGOL, PASCAL) si risolve (C) combinando la struttura a blocchi con scoping statico, con un meccanismo di moduli separati con regole di visibilità.

14 Implementazione di ambiente e memoria nel linguaggio imperativo

Contenuti del capitolo:

- Ambiente e memoria locale nel linguaggio imperativo:
  - Cosa serve in ogni attivazione.
  - Perchè la restrizione a locazioni denotabili.
  - Implementazione: strutture dati ed operazioni.
  - Cosa cambia nell'interprete iterativo

14.1 Ambiente locale dinamico

Per ogni attivazione (entrata nel blocco o chiamata di procedura) abbiamo attualmente nel record di attivazione ambiente e memoria, nel loro complesso, implementati come funzioni.
In accordo con la semantica dell'ambiente locale dinamico possiamo inserire nel record di attivazione: una tabella che implementa il solo ambiente locale (*catena statica*) ed una tabella che implementa la memoria locale.
Quando un'attivazione termina (uscita dal blocco o ritorno dalla chiamata di procedura) possiamo eliminare ambiente e memoria locali insieme a tutte le altre informazionicontanute nel record di attivazione, imponendo però determinate *restrizioni*.
In sostanza adottiamo la stessa soluzione introdotta per il linguaggio funzionale, con la creazione di un nuovo ambiente locale in corrispondenza dei costrutti `Let`, `Apply`, `Block` e `Call`.
Utilizzeremo quindi le seguenti pile per l'ambiente:

- `namestack`: pila di array di identificatori.
- `dvalstack`: pila di array di valori denotati.
- `slinkstack`: pila di puntatori ad ambienti.
- `tagstack`: pila di etichette per la retention, usata solo per il frammento funzionale, visto che le procedure non sono esprimibili.

14.2 Memoria locale

Per la memoria utilizziamo lo stack

- `storestack`: pila di array di mval.

La creazione di una nuova memoria locale avviene chiaramente quando si entra in un blocco (`Block`) o si chiama una procedura (`Call`), e si crea una nuova associazione tra un nome ed una espressione di tipo `Newloc` per ogni dichiarazione di variabile.
Uno **store** è quindi un puntatore (intero) nella pila, e lo *store* corrente è il valore della variabile `currentstore`.
Una **locazione** è una coppia di interi: il primo identifica lo *store*, il secondo la *posizione* relativa. Tutte le locazioni raggiungibili attraverso l'ambiente non locale sono accessibili: possono essere lette, modificate con l'assegnamento e passate come parametro.

14.3 Realizzazione dell'ambiente

Vediamo strutture dati ed operazioni che usiamo per implementare l'ambiente:

```
type 't store = int

let (currentstore:  mval store ref) = ref(0)

let storestack = emptystack(stacksize,[|Undefined|])

let (newloc,initloc) =
  let count = ref(-1) in
   (fun () -> count := !count +1;
     (!currentstore, !count)),
  (fun () -> count := -1)

let applystore ((x:  mval store), ((n1, n2):  loc)) =
  let a = access(storestack, n1) in
   Array.get a n2

let emptystore(x) = initloc();
  svuota(storestack); currentstore := -1; !currentstore

let allocate ((s:mval store), (m:mval)) = let (n1, n2) = (*2*)
  newloc() in let a =
   access(storestack, n1) in
   Array.set a n2 m; ((n1, n2), s)

let update((s:mval store), (n1,n2), (m:mval)) =
  if applystore(s, (n1,n2)) = Undefined
  then failwith (``wrong assignment'')
  else let a = access(storestack, n1) in
   Array.set a n2 m; s

let pushlocalstore (dl) = let rdl = ref(dl) in (*1*)
let rn = ref(0) in
  while not(!rdl = []) do
   let (i, d) = List.hd !rdl in
     (match d with
       | Newloc(_) -> rn := !rn + 1
       | _ -> ());
     rdl := List.tl !rdl
  done;
  let a = Array.create !rn Undefined in
   pop(storestack); push(a, storestack);
   initloc(); !currentstore
```

Commenti al codice:

1. La `pushlocalstore` ha come argomento una lista di dichiarazioni (costanti e

variabili) e crea l'array locale della dimensione necessaria, sostituendolo con quello vuoto correntemente sulla testa di `storestack`.

2. `allocate` setta l'array creato con la `pushlocalstore`.

14.4 Gestione a pila della memoria locale

Per poter correttamente poppare anche la memoria locale insieme al resto del record di attivazione è necessario essere sicuri che non esistano cammini d'accesso "esterni" alle locazioni interne alla memoria locale.
Un cammino d'accesso esterno può essere:

- Un altro nome, diverso da quello locale, per la locazione (*aliasing*). Questa situazione può verificarsi unicamente attraverso il passaggio della locazione come parametro (per riferimento) ad un'altra procedura, ma al momento del ritorno della procedura che conteneva la dichiarazione di variabile originale qualunque procedura chiamata è già necessariamente ritornata. L'aliasing non deve quindi preoccuparci.
- Una locazione, appartenente ad una diversa memoria locale, che contiene la locazione come valore. Anche questa situazione non si può verificare perchè le locazioni non sono memorizzabili.
- Il valore temporaneo della attuale "applicazione di funzione". Anche questa situaizone non si può verificare perchè le locazioni non sono valori esprimibili.

La memoria "dinamica" può dunque essere gestita a pila solo se le locazioni non sono *nè esprimibili nè memorizzabili*.

La gestione dinamica a pila della memoria è stata introdotta da ALGOL60 insieme ad ambiente locale dinamico ed alla struttura a blocchi. ALGOL non prevedeva l'uso di puntatori, nè vere strutture dati dinamiche.
Quando il linguaggio prevede l'uso di puntatori (PASCAL, C) è necessaria una gestione della memoria ad heap (simile a quella che vedremo per gli oggetti), che può coesistere con una gestione a pila, se si mantiene la distinzione tra locazioni e puntatori (PASCAL).
Una gestione dinamica della memoria può essere necessaria anche in linguaggi che non hanno nella semantica nè *store* nè *heap*, come i linguaggi funzionali o linguaggi logici, semplicemente per poter gestire strutture dati dinamiche, implementate necessariamente con heap e puntatori.
Tutti i casi che richiedono la gestione a heap (puntatori, strutture dati dinamiche, oggetti) permettono che operazioni che richiedono l'allocazione di nuova memoria possano figurare in punti arbitrari del programma, invece che soltanto nelle dichiarazioni all'ingresso dei blocchi.

14.5 Novità nell'interprete iterativo

Vediamo le principali modifice necessarie nell'interprete iterativo con l'implementazione della memoria.
Innanzi tutto cambia l'applicazione delle procedure, in cui un'unico frame contiene l'ambiente locale con i parametri e lo "spazio" necessario per tutte le dichiarazioni, e la memoria locale con lo spazio per tuute le variabili locali:

```
let applyproc ((ev1:dval),(ev2:dval list), s) =
   ( match ev1 with
      | Dprocval(Proc(ii,(l1, l2, l3)), x) ->
         let r = bindlist(x, ii, ev2) in
          newframes(labelcom(l3), r, s);
          push(Rdecl(l2), top(cstack));
          push(labeldec(l1),top(cstack));
          pushlocalenv(l1,l2,r);
          pushlocalstore(l1);
          ()
      | _ -> failwith (``attempt to apply a ...''))
```

Eliminiamo poi il calcolo del punto fisso, poichè l'ambiente delle chiusure (`rho`) è lo stesso in cui vengono inserite le associazioni.

```
let itsemrdecl() =
   let tempstack = top(tempvalstack) in
   let continuation = top(cstack) in
   let tempdstack = top(tempdvalstack) in
   let rho = topenv() in
   let sigma = topstore() in
   let rl =
     (match top(continuation) with
      | Rdecl(rl1) -> rl1
      | _ -> failwith(``impossible in semrdecl'')) in
    pop(continuation);
    let prl = ref(rl) in
     while not(!prl = []) do
       let currd = List.hd !prl in
        prl := List.tl !prl;
        let (i, den) =
          (match currd with
           |(j, Proc(il,b)) -> (j, makeproc(Proc(il,b),rho))
           |(j, Fun(il,b)) -> (j, makefun(Fun(il,b),rho))
           | _ -> failwith(``no more sensible cases in ...''))  in
          currentenv := bind(rho, i, den)
     done
```

Infine cambia la semantica del blocco:

```
let semb ((l1, l2, l3), r, s) = initstate(r,s);
   newframes(labelcom(l3), r, s);
   push(Rdecl(l2), top(cstack));
   push(labeldec(l1),top(cstack));
   pushlocalenv(l1,l2,!currentenv);
   pushlocalstore(l1);
   loop();
   currentenv := !currentenv + 1;
   currentstore := !currentstore + 1;
```

```
  topstore()
```

14.6 Interprete iterativo

Ecco il codice completo dell'interprete iterativo per il linguaggio imperativo, con l'implementazione di ambiente e memoria:

```
type 't env = int
type loc = int * int
type 't store = int

exception Nonstorable
exception Nonexpressible

type eval =
   | Int of int
   | Bool of bool
   | Novalue
   | Funval of efun
and efun = exp * (dval env)
and proc = exp * (dval env)

and dval =
   | Dint of int
   | Dbool of bool
   | Unbound
   | Dloc of loc
   | Dfunval of efun
   | Dprocval of proc

and mval =
   | Mint of int
   | Mbool of bool
   | Undefined

let evaltomval e = match e with
   | Int n -> Mint n
   | Bool n -> Mbool n
   | _ -> raise Nonstorable

let mvaltoeval m = match m with
   | Mint n -> Int n
   | Mbool n -> Bool n
   | _ -> Novalue

let evaltodval e = match e with
   | Int n -> Dint n
   | Bool n -> Dbool n
```

```
   | Novalue -> Unbound
   | Funval f -> Dfunval f

let dvaltoeval e = match e with
   | Dint n -> Int n
   | Dbool n -> Bool n
   | Dloc n -> raise Nonexpressible
   | Dprocval n -> raise Nonexpressible
   | Unbound -> Novalue
   | Dfunval f -> Funval f

type labeledconstruct =
   | Expr1 of exp
   | Expr2 of exp
   | Exprd1 of exp
   | Exprd2 of exp
   | Com1 of com
   | Com2 of com
   | Coml of labeledconstruct list
   | Decl of labeledconstruct list
   | Rdecl of (ide * exp) list
   | Dec1 of (ide * exp)
   | Dec2 of (ide * exp)

let (cstack:  labeledconstruct stack stack) =
   emptystack(stacksize,emptystack(1,Expr1(Eint(0))))

let (tempvalstack:  eval stack stack) =
   emptystack(stacksize,emptystack(1,Novalue))

let (tempdvalstack:  dval stack stack) =
   emptystack(stacksize,emptystack(1,Unbound))

let labelcom (dl:  com list) = let dlr =
   ref(dl) in let ldlr =
     ref([]) in
     while not (!dlr = []) do
       let i = List.hd !dlr in
         ldlr := !ldlr @ [Com1(i)];
         dlr := List.tl !dlr
     done;
     Coml(!ldlr)

let labeldec (dl:  (ide * exp) list) =
   let dlr = ref(dl) in let ldlr =
      ref([]) in
     while not (!dlr = []) do
       let i = List.hd !dlr in
         ldlr := !ldlr @ [Dec1(i)];
```

```
       dlr := List.tl !dlr
    done;
    Decl(!ldlr)

let (labelstack:  labeledconstruct stack) =
  emptystack(stacksize,Expr1(Eint(0)))

let pushargs ((b:  exp list),(continuation:  labeledconstruct stack))
=
  let br = ref(b) in
    while not(!br = []) do
      push(Exprd1(List.hd !br),continuation);
      br := List.tl !br
    done

let getargs ((b:  exp list),(tempstack:  dval stack)) =
  let br = ref(b) in
  let er = ref([]) in
    while not(!br = []) do
      let arg=top(tempstack) in
        pop(tempstack); er := !er @ [arg];
        br := List.tl !br
    done;
    !er

let (currentenv:  dval env ref) = ref(0)

let (initenv:  dval env ref) = ref(0)

let namestack = emptystack(stacksize,[| ``dummy'' |])

let dvalstack = emptystack(stacksize,[| Unbound |])

let slinkstack = emptystack(stacksize, !currentenv)

type tag = Retained| Standard

let tagstack = emptystack(stacksize, Standard)

let applyenv ((x:  dval env), (y:  ide)) =
  let n = ref(x) in
  let den = ref(Unbound) in
    while !n > -1 do
      let lenv = access(namestack,!n) in
      let nl = Array.length lenv in
      let index = ref(0) in
        while !index < nl do
          if Array.get lenv !index = y then
            (den := Array.get(access(dvalstack,!n)) !index;
```

```
          index := nl)
        else index := !index + 1
      done;
      if not(!den = Unbound) then n := -1
      else n := access(slinkstack,!n)
    done;
    !den

let bind ((r:dval env),i,d) =
  let arrnomi = access(namestack, r) in
  let lun = Array.length arrnomi in
  let arrden = access(dvalstack, r) in
  let n = ref(0) in
  let index = ref(-1) in
    while !n < lun do
      let nome = Array.get arrnomi !n in
        if nome = i then
          (Array.set arrden !n d;
          n := lun; index := -2) else
            if nome = ''dummy''
            then (index := !n; n := lun)
            else n := !n + 1
    done;
    if !index = -2 then r else
      if !index > -1
      then
        (Array.set arrnomi !index i;
        Array.set arrden !index d; r)
      else
        (push(Array.create 1 i,namestack);
        push(Array.create 1 d,dvalstack);
        push(r,slinkstack);
        push(Standard,tagstack);
        (lungh(namestack):  dval env))

let bindlist(r, il, el) =
  let n = List.length il in
  let ii = Array.create n ``dummy'' in
  let dd = Array.create n Unbound in
  let ri = ref(il) in
  let rd = ref(el) in
  let index = ref(0) in
    while !index < n do
      let i = List.hd !ri in
      let d = List.hd !rd in
        ( ri := List.tl !ri;
          rd := List.tl !rd;
          Array.set ii !index i;
          Array.set dd !index d;
```

```
        index := !index + 1)
    done;
    push(ii, namestack);
    push(dd,dvalstack);
    push(r,slinkstack);
    push(Standard,tagstack);
    (lungh(namestack):  dval env)

let emptyenv(x) = currentenv := -1; svuota(namestack);
  svuota(dvalstack); svuota(slinkstack); svuota(tagstack);
  !currentenv

let svuotaenv() = svuota(namestack); svuota(tagstack);
  svuota(dvalstack); svuota(slinkstack)

let pushlocalenv (dl,dlr,r) =
  let anomi = access(namestack, r) in
  let aden = access(dvalstack, r) in
  let nn = ref(Array.length anomi) in
  let rn = (List.length (dl @ dlr) + !nn) in
  let ii = Array.create rn ``dummy'' in
  let dd = Array.create rn Unbound in
    while not(!nn = 0) do
      Array.set ii (!nn - 1) (Array.get anomi (!nn - 1));
      Array.set dd (!nn - 1) (Array.get aden (!nn - 1));
      nn := !nn - 1
    done;
    pop(namestack); pop(dvalstack);
    push(ii, namestack);
    push(dd, dvalstack);
    r

let topenv() = !currentenv

let popenv () = pop(namestack);
  pop(dvalstack);
  pop(slinkstack);
  pop(tagstack);
  currentenv := !currentenv - 1

let pushemptylocalenv(r) = push([||],namestack);
  push([||],dvalstack);
  push(Standard,tagstack);
  push(r,slinkstack);
  currentenv := lungh(namestack)

let pushenv(r) = if r = !currentenv then
  pushemptylocalenv(r)
else currentenv := r
```

```
let retained (n:dval env) = access(tagstack,n) = Retained

let retain () =
  setta(tagstack, !currentenv, Retained);
  let cont = ref(lungh(namestack)) in
   while !cont > -1 & retained(!cont) do cont := !cont - 1 done;
   currentenv := !cont

let to_be_retained (v:eval) = match v with
  | Funval (e, r1) -> not(r1 < !currentenv)
  | _ -> false

let retainorpopenv (r, valore) =
  if (topenv() = r) then () else
   if !currentenv < lungh(namestack) then retain() else
    if to_be_retained(valore) then retain() else
     (popenv();
     let index = ref(!currentenv) in
      while !index > r do
       if retained(!index)
       then (currentenv := !currentenv - 1;
            index := !index -1) else (index := r) done )

let (currentstore:  mval store ref) = ref(0)

let storestack = emptystack(stacksize,[|Undefined|])

let (newloc,initloc) =
  let count = ref(-1) in
   (fun () -> count := !count +1;
    (!currentstore, !count)),
  (fun () -> count := -1)

let applystore ((x:  mval store), ((n1, n2):  loc)) =
  let a = access(storestack, n1) in
   Array.get a n2

let emptystore(x) = initloc();
  svuota(storestack); currentstore := -1; !currentstore

let allocate ((s:mval store), (m:mval)) =
  let (n1, n2) = newloc() in
  let a = access(storestack, n1) in
   Array.set a n2 m; ((n1, n2), s)

let update((s:mval store), (n1,n2), (m:mval)) =
  if applystore(s, (n1,n2)) = Undefined
  then failwith (``wrong assignment'')
```

```
   else let a = access(storestack, n1) in
    Array.set a n2 m; s

let pushlocalstore (dl) = let rdl = ref(dl) in
let rn = ref(0) in
   while not(!rdl = []) do
    let (i, d) = List.hd !rdl in
      (match d with
        | Newloc(_) -> rn := !rn + 1
        | _ -> ());
      rdl := List.tl !rdl
   done;
   let a = Array.create !rn Undefined in
    pop(storestack); push(a, storestack);
    initloc(); !currentstore


let pushemptylocalstore () = push([||],storestack);
   currentstore := !currentstore + 1

let pushstore(n) =
   if n = !currentstore
   then pushemptylocalstore()
   else currentstore := n

let popstore () = pop(storestack); currentstore := !currentstore -1

let svuotastore () = svuota(storestack)

let topstore() = !currentstore

let newframes(ss,rho,sigma) =
   pushstore(sigma);
   pushenv(rho);
   let cframe = emptystack(cframesize(ss),Expr1(Eint 0)) in
   let tframe = emptystack(tframesize(ss),Novalue) in
   let dframe = emptystack(tdframesize(ss),Unbound) in
    push(tframe,tempvalstack);
    push(dframe,tempdvalstack);
    push(ss, labelstack);
    push(ss, cframe);
    push(cframe, cstack)

let makefun ((a:exp),(x:dval env)) =
   (match a with
    | Fun(ii,aa) ->
        Dfunval(a,x)
    | _ -> failwith (``Non-functional object''))
```

```
let makeproc ((a:exp),(x:dval env)) =
  (match a with
    | Proc(ii,b) ->
        Dprocval(a,x)
    | _ -> failwith (``Non-procedural object''))

let makefunrec (f, (a:exp),(x:dval env)) =
  dvaltoeval(makefun(a, lungh(namestack) + 1))

let applyfun ((ev1:dval),(ev2:dval list), s) =
  ( match ev1 with
     | Dfunval(Fun(ii,aa),r) ->
        newframes(Expr1(aa),bindlist(r, ii, ev2), s)
     | _ -> failwith (``attempt to apply a non-functional object''))

let applyproc ((ev1:dval),(ev2:dval list), s) =
  ( match ev1 with
     | Dprocval(Proc(ii,(l1, l2, l3)), x) ->
        let r = bindlist(x, ii, ev2) in
         newframes(labelcom(l3), r, s);
         push(Rdecl(l2), top(cstack));
         push(labeldec(l1),top(cstack));
         pushlocalenv(l1,l2,r);
         pushlocalstore(l1);
         ()
     | _ -> failwith (``attempt to apply a non-functional object''))

let itsem() =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
    (match top(continuation) with
     |Expr1(x) ->
        (pop(continuation);
        push(Expr2(x),continuation);
        (match x with
           | Iszero(a) -> push(Expr1(a),continuation)
           | Eq(a,b) -> push(Expr1(a),continuation);
              push(Expr1(b),continuation)
           | Prod(a,b) -> push(Expr1(a),continuation);
              push(Expr1(b),continuation)
           | Sum(a,b) -> push(Expr1(a),continuation);
              push(Expr1(b),continuation)
           | Diff(a,b) -> push(Expr1(a),continuation);
              push(Expr1(b),continuation)
           | Minus(a) -> push(Expr1(a),continuation)
           | And(a,b) -> push(Expr1(a),continuation);
```

```
            push(Expr1(b),continuation)
         | Or(a,b) -> push(Expr1(a),continuation);
            push(Expr1(b),continuation)
         | Not(a) -> push(Expr1(a),continuation)
         | Ifthenelse(a,b,c) -> push(Expr1(a),continuation)
         | Val(e) -> push(Exprd1(e),continuation)
         | Newloc(e) -> failwith (``nonlegal expression for sem'')
         | Let(i,e1,e2) -> push(Exprd1(e1),continuation)
         | Appl(a,b) -> push(Expr1(a),continuation);
            pushargs(b,continuation)
         | Proc(i,b) -> failwith (``nonlegal expression for sem'')
         | _ -> ()))
   |Expr2(x) ->
      (pop(continuation);
      (match x with
         | Eint(n) -> push(Int(n),tempstack)
         | Ebool(b) -> push(Bool(b),tempstack)
         | Den(i) ->
            push(dvaltoeval(applyenv(rho,i)),tempstack)
         | Iszero(a) ->
            let arg=top(tempstack) in
              pop(tempstack);
              push(iszero(arg),tempstack)
         | Eq(a,b) ->
            let firstarg=top(tempstack) in
              pop(tempstack);
              let sndarg=top(tempstack) in
                pop(tempstack);
                push(equ(firstarg,sndarg),tempstack)
         | Prod(a,b) ->
            let firstarg=top(tempstack) in
              pop(tempstack);
              let sndarg=top(tempstack) in
                pop(tempstack);
                push(mult(firstarg,sndarg),tempstack)
         | Sum(a,b) ->
            let firstarg=top(tempstack) in
              pop(tempstack);
              let sndarg=top(tempstack) in
                pop(tempstack);
                push(plus(firstarg,sndarg),tempstack)
         | Diff(a,b) ->
            let firstarg=top(tempstack) in
              pop(tempstack);
              let sndarg=top(tempstack) in
                pop(tempstack);
                push(diff(firstarg,sndarg),tempstack)
         | Minus(a) ->
            let arg=top(tempstack) in
```

```
              pop(tempstack);
              push(minus(arg),tempstack)
          | And(a,b) ->
             let firstarg=top(tempstack) in
              pop(tempstack);
              let sndarg=top(tempstack) in
               pop(tempstack);
               push(et(firstarg,sndarg),tempstack)
          | Or(a,b) ->
             let firstarg=top(tempstack) in
              pop(tempstack);
              let sndarg=top(tempstack) in
               pop(tempstack);
               push(vel(firstarg,sndarg),tempstack)
          | Not(a) ->
             let arg=top(tempstack) in
              pop(tempstack);
              push(non(arg),tempstack)
          | Ifthenelse(a,b,c) ->
             let arg=top(tempstack) in
              pop(tempstack);
              if typecheck(``bool'',arg) then
               (if arg = Bool(true)
               then push(Expr1(b),continuation)
               else push(Expr1(c),continuation))
              else failwith (``type error'')
          | Val(e) -> let v = top(tempdstack) in
             pop(tempdstack);
             (match v with
               | Dloc n ->
                  push(mvaltoeval(applystore(sigma, n)), tempstack)
               | _ -> failwith(``not a variable''))
          | Fun(i,a) -> push(dvaltoeval(makefun(Fun(i,a),rho)),tempstack)
          | Rec(f,e) -> push(makefunrec(f,e,rho),tempstack)
          | Let(i,e1,e2) -> let arg= top(tempdstack) in
             pop(tempdstack);
             newframes(Expr1(e2), bind(rho, i, arg), sigma)
          | Appl(a,b) -> let firstarg=evaltodval(top(tempstack)) in
             pop(tempstack);
             let sndarg=getargs(b,tempdstack) in
              applyfun(firstarg, sndarg, sigma)
          | _ -> failwith(``no more cases for semexpr'')))
     | _ -> failwith(``no more cases for semexpr''))

let itsemden() =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
```

```
  let sigma = topstore() in
    (match top(continuation) with
      |Exprd1(x) ->
         (pop(continuation); push(Exprd2(x),continuation);
         match x with
           | Den i -> ()
           | Fun(i,e) -> ()
           | Proc(il,b) -> ()
           | Newloc(e) -> push(Expr1( e), continuation)
           | _ -> push(Expr1(x), continuation))
      |Exprd2(x) -> (pop(continuation); match x with
                     | Den i ->
                        push(applyenv(rho,i), tempdstack)
                     | Fun(i,e) ->
                        push(makefun(x,rho),tempdstack)
                     | Proc(il,b) ->
                        push(makeproc(x,rho),tempdstack)
                     | Newloc(e) ->
                        let m=evaltomval(top(tempstack)) in
                          pop(tempstack);
                          let (l, s1)=
                            allocate(sigma, m) in
                            push(Dloc l, tempdstack);
                            currentstore := s1
                     | _ -> let arg = top(tempstack)in
                        pop(tempstack);
                        push(evaltodval(arg),tempdstack))
      | _ -> failwith(``No more cases for demden''))

let itsemdecl () =
  let tempstack = top(tempvalstack) in
  let continuation = top(cstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  let dl =
    (match top(continuation) with
      | Decl(dl1) -> dl1
      | _ -> failwith(``impossible in semdecl'')) in
    if dl = [] then pop(continuation) else
      (let currd = List.hd dl in
      let newdl = List.tl dl in
        pop(continuation); push(Decl(newdl),continuation);
        (match currd with
           | Dec1( (i,e)) ->
              pop(continuation);
              push(Decl(Dec2((i, e))::newdl),continuation);
              push(Exprd1(e), continuation)
           | Dec2((i,e)) ->
```

```
            let arg = top(tempdstack) in
              pop(tempdstack);
              currentenv := bind(rho, i, arg)
          | _ -> failwith(``no more sensible cases for semdecl''))))

let itsemrdecl() =
  let tempstack = top(tempvalstack) in
  let continuation = top(cstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  let rl =
    (match top(continuation) with
      | Rdecl(rl1) -> rl1
      | _ -> failwith(``impossible in semrdecl'')) in
    pop(continuation);
    let prl = ref(rl) in
      while not(!prl = []) do
        let currd = List.hd !prl in
          prl := List.tl !prl;
          let (i, den) =
            (match currd with
              |(j, Proc(il,b)) -> (j, makeproc(Proc(il,b),rho))
              |(j, Fun(il,b)) -> (j, makefun(Fun(il,b),rho))
              | _ -> failwith(``no more sensible cases in ...''))  in
            currentenv := bind(rho, i, den)
      done

let itsemcl() =
  let tempstack = top(tempvalstack) in
  let continuation = top(cstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
  let cl =
    (match top(continuation) with
      | Coml(dl1) -> dl1
      | _ -> failwith(``impossible in semcl'')) in
    if cl = [] then pop(continuation) else
      (let currc = List.hd cl in
      let newcl = List.tl cl in
        pop(continuation); push(Coml(newcl),continuation);
        (match currc with
           | Com1(Assign(e1, e2)) -> pop(continuation);
             push(Coml(Com2(Assign(e1,e2))::newcl),continuation);
             push(Exprd1(e1), continuation);
             push(Expr1(e2), continuation)
           | Com2(Assign(e1, e2)) ->
             let arg2 = evaltomval(top(tempstack)) in
```

```
    pop(tempstack);
    let arg1 = top(tempdstack) in
     pop(tempdstack);
     (match arg1 with
       | Dloc(n) ->
          currentstore := update(sigma,n,arg2)
       | _ -> failwith (``wrong location in assignment''))
| Com1(While(e, cl)) ->
   pop(continuation);
   push(Coml(Com2(While(e, cl))::newcl),continuation);
   push(Expr1(e), continuation)
| Com2(While(e, cl)) ->
   let g = top(tempstack) in
    pop(tempstack);
    if typecheck(``bool'',g) then
     (if g = Bool(true) then
      (let old = newcl in
       let newl =
         (match labelcom cl with
          | Coml newl1 -> newl1
          | _ -> failwith(``impossible in while'')) in
       let nuovo =
         Coml(newl@[Com1(While(e, cl))] @ old) in
         pop(continuation); push(nuovo,continuation))
     else ())
    else failwith (``nonboolean guard'')
| Com1(Cifthenelse(e, cl1, cl2)) ->
   pop(continuation);
   push(Coml(
       Com2(Cifthenelse(e,cl1,cl2))::newcl),
      continuation);
   push(Expr1(e), continuation)
| Com2(Cifthenelse(e, cl1, cl2)) ->
   let g = top(tempstack) in
    pop(tempstack);
    if typecheck(``bool'',g) then
     (let temp = if g = Bool(true) then
      labelcom (cl1) else labelcom (cl2) in
     let newl =
       (match temp with
          | Coml newl1 -> newl1
          | _ -> failwith(``impossible in cifthenelse'')) in
     let nuovo = Coml(newl @ newcl) in
         pop(continuation); push(nuovo,continuation))
   else failwith (``nonboolean guard'')
| Com1(Call(e, el)) ->
   pop(continuation);
   push(Coml(Com2(Call(e, el))::newcl),continuation);
   push(Exprd1( e), continuation);
```

```
            pushargs(el, continuation)
          | Com2(Call(e, el)) ->
            let p = top(tempdstack) in
             pop(tempdstack);
             let args = getargs(el,tempdstack) in
               applyproc(p, args, sigma)
          | Com1(Block((l1, l2, l3))) ->
            newframes(labelcom(l3), rho, sigma);
            push(Rdecl(l2), top(cstack));
            push(labeldec(l1),top(cstack));
            pushlocalenv(l1,l2,rho);
            pushlocalstore(l1);
            ()
          | _ -> failwith(``no more sensible cases in commands'') ))

let initstate (r, s) =
  svuota(cstack); svuota(tempvalstack); svuota(tempdvalstack);
  svuota(labelstack); currentenv := r; currentstore := s

let loop () =
  while not(empty(cstack)) do
   while not(empty(top(cstack))) do
    let currconstr = top(top(cstack)) in
     (match currconstr with
       | Expr1(e) -> itsem()
       | Expr2(e) -> itsem()
       | Exprd1(e) -> itsemden()
       | Exprd2(e) -> itsemden()
       | Coml(cl) -> itsemcl()
       | Decl(l) -> itsemdecl()
       | Rdecl(l) -> itsemrdecl()
       | _ -> failwith(``non legal construct in loop''))
   done;
   (match top(labelstack) with
     | Expr1(_) -> let valore = top(top(tempvalstack)) in
        pop(top(tempvalstack));
        pop(tempvalstack); push(valore,top(tempvalstack));
        retainorpopenv(!initenv, valore);
        popstore();
        pop(tempdvalstack)
     | Exprd1(_) -> let valore = top(top(tempdvalstack)) in
        pop(top(tempdvalstack));
        pop(tempdvalstack); push(valore,top(tempdvalstack));
        retainorpopenv(!initenv, dvaltoeval(valore));
        popstore();
        pop(tempvalstack)
     | Decl(_) ->
        pop(tempvalstack); pop(tempdvalstack)
     | Rdecl(_) ->
```

```
        pop(tempvalstack); pop(tempdvalstack)
      | Coml(_) -> pop(tempvalstack); pop(tempdvalstack);
        popenv(); popstore()
      | _ -> failwith(``non legal label in loop''));
    pop(cstack);
    pop(labelstack)
  done


let sem (e,(r:  dval env), (s:  mval store)) =
  initstate(r,s);
  initenv := r;
  push(emptystack(tframesize(e),Novalue),tempvalstack);
  newframes(Expr1(e), r, s);
  loop();
  let valore= top(top(tempvalstack)) in
   pop(tempvalstack);
   valore

let semden (e,(r:  dval env), (s:  mval store)) =
  initstate(r,s);
  initenv := r;
  push(emptystack(tdframesize(e),Unbound),tempdvalstack);
  newframes(Exprd1(e), r, s);
  loop();
  let valore= top(top(tempdvalstack)) in
   pop(tempdvalstack);
   valore

let semcl (cl,(r:  dval env), (s:  mval store)) =
  initstate(r,s);
  newframes(labelcom(cl), r, s);
  loop();
  let st = topstore() in st

let semc((c:  com), (r:dval env), (s:  mval store)) =
  semcl ([c],r, s)

let semdv(dl, r, s) =
  initstate(r,s);
  newframes(labeldec(dl), r, s);
  pushlocalenv(dl,[],!currentenv);
  pushlocalstore(dl);
  loop();
  let st = topstore() in
  let rt = topenv() in
   (rt, st)

let semdr(dl, r, s) =
```

```
   initstate(r,s);
   newframes(Rdecl(dl), r, s);
   pushlocalenv([],dl,!currentenv);
   loop();
   let rt = topenv() in
     (rt, s)

let semdl((dl,rl), r, s) =
   initstate(r,s);
   newframes(Rdecl(rl), r, s);
   push(labeldec(dl),top(cstack));
   pushlocalenv(dl,rl,!currentenv);
   pushlocalstore(dl);
   loop();
   let st = topstore() in
   let rt = topenv() in
     (rt, st)

let semb ((l1, l2, l3), r, s) = initstate(r,s);
   newframes(labelcom(l3), r, s);
   push(Rdecl(l2), top(cstack));
   push(labeldec(l1),top(cstack));
   pushlocalenv(l1,l2,!currentenv);
   pushlocalstore(l1);
   loop();
   currentenv := !currentenv + 1;
   currentstore := !currentstore + 1;
   topstore()
```

15 Implementazione degli oggetti

Contenuti del capitolo:

- Effetti dell'implementazione di amnìbiente e memoria sugi oggetti: oggetti con ambiente e memoria permanenti.
- Effetto degli oggetti sull'implementazione di ambiente e memoria: ambienti e memoria esistono sia sulla pila che sulla heap.
- Modifiche dell'interprete iterativo.

15.1 Oggetti ed implementazione dello stato

Nella semantica attuale un oggetto è un ambiente, rappresentato da una funzione, ottenuto a partire dall'ambiente complessivo esistente dopo l'esecuzione del blocco, limitandosi (`localenv`) alle associazioni per `this`, campi e metodi (compresi quelli ereditati).
Tale ambiente è utilizzato unicamente per selezionare campi e metodi dell'ogggetto (`Field`).
L'ambiente *non locale* dell'oggetto è soltanto l'ambiente globale delle classi (ma potrebbe essere un qualunque ambiente se permettessimo l'annidamento tra classi), ed è rappresentato completamente nelle chiusure dei suoi metodi.
La memoria esportata nella posizione della pila corrispondente all'attivazione precedente contiene tutto, comprese: locazioni locali ad attivazioni ritornate non accessibili perchè non denotate nell'ambiente, e locazioni locali dell'oggetto che restano accessibili (permanenti) in quanto denotate nell' ambiente-oggetto (che rimane nella heap).
Quando ambiente e memoria vengono implementati come pile di ambienti e memorie locali, bisogna adattare anche l'implementazione dell'oggetto.

15.2 Cosa cambia, come cambia

La nuova struttura degli **oggetti** diventa:

```
type obj = ide array * dval array * dval env * mval array
```

Come negli ambienti e memorie locali:

- `ide array`, array di identificatori, come gli elementi di `namestack`.
- `dval array`, array di valori denotati, come gli elementi di `dvalstack`.
- `dval env`, (puntatore ad) ambiente, come gli elementi di `slinkstack`.
- `mval array`, (puntatore a) memoria, come gli elementi di `storestack`.

In questo caso non serve un tag per la retention, perchè l'ambiente locale così costruito è permanente.
Anche la **heap**, come nel caso di memoria ed ambiente, non può più essere una funzione. Passiamo quindi

da `type heap = pointer -> obj`
a `type heap = obj array`

Manteniamo per il momento l'implementazione "banale" (in cui non si elimina mai niente) di `pointer` e `newpoint`:

```
type pointer = int;
let newpoint = let count = ref(-1) in
    function () -> count := count+1; !count
```

Cambiano però le varie operazioni sulla heap:

```
let emptyheap () = initpoint();
  (Array.create 100 ((Array.create 1 ''dummy''),
       (Array.create 1 Unbound),
       Denv(-1), (Array.create 1 Undefined)):  heap)

let currentheap = ref(emptyheap())

let applyheap ((x:  heap), (y:pointer)) = Array.get x y

let allocateheap ((x:heap), (i:pointer), (r:obj)) =
  Array.set x i r; x

let getnomi(x:obj) = match x with
  | (nomi, den, slink, st) -> nomi

let getden(x:obj) = match x with
  | (nomi, den, slink, st) -> den

let getslink(x:obj) = match x with
  | (nomi, den, slink, st) -> slink

let getst(x:obj) = match x with
  | (nomi, den, slink, st) -> st
```

Anche **ambiente** e **memoria** devono cambiare. Nel linguaggio imperativo erano semplicemente interi interpretati come puntatori alle tabelle locali nelle varie pile che realizzano concretamente di ambiente e memoria. In presenza di oggetti ambienti e memorie locali possono essere permanenti essendo allocate sulla heap. Pssiamo astrarre dalle differenze e definire un unico tipo di ambiente e memoria:

```
type 't env = Denv of int | Penv of pointer
type 't store = Dstore of int | Pstore of pointer
```

Le versioni che iniziano con `D` (dinamiche) sono puntatori nelle pile, mentre le versioni che iniziano per `P` (permanenti) sono puntatori nella heap.
Le implementazioni delle operazioni si preoccuperanno di trattare i due casi in modo appropriato.
Ecco alcune delle operazioni che devono essere ridefinite:

```
let applyenv ((r:  dval env), (y:  ide)) =
  let den = ref(Unbound) in
  let (x, caso) =
    (match r with
     | Denv(x1) -> (x1, ref(``stack''))
     | Penv(x1) -> (x1, ref(``heap''))) in
  let n = ref(x) in
   while !n > -1 do
    let lenv =
     if !caso = ``stack''
     then access(namestack,!n)
     else getnomi(applyheap(!currentheap,!n)) in
    let nl = Array.length lenv in
    let index = ref(0) in
     while !index < nl do
      if Array.get lenv !index = y
      then
        (den :=
         (if !caso = ``stack''
          then
            Array.get(access(dvalstack,!n)) !index
          else
            Array.get(getden(applyheap(!currentheap,!n))) !index);
        index := nl)
      else index := !index + 1
     done;
     if not(!den = Unbound) then n := -1
     else let next =
      (if !caso = ''stack'' then access(slinkstack,!n)
      else getslink(applyheap(!currentheap,!n))) in
      caso := (match next with
               |Denv(_) -> ``stack''
               |Penv(_) -> ``heap'');
      n := (match next with
            |Denv(n1) -> n1
            |Penv(n1) -> n1)
   done;
   !den;;

let applystore ((x:  mval store), (d:  loc)) =
  match d with
   | (s2,n2)->
      (match s2 with
       | Dstore(n1) -> let a =
          access(storestack, n1) in
          Array.get a n2
       | Pstore(n1) -> let a =
          getst(applyheap(!currentheap,n1)) in
          Array.get a n2)
```

```
    | _ -> failwith(``not a location in applystore'')

let allocate ((s:mval store), (m:mval)) =
  let (s2, n2) = newloc() in
    (match s2 with
      | Pstore(n1) -> let a = access(storestack, n1) in
         Array.set a n2 m; ((s2, n2),s)
      | Dstore(n1) -> let a = access(storestack, n1) in
         Array.set a n2 m; ((s2, n2),s))

let update((s:mval store), (d:  loc), (m:mval)) =
  if applystore(s, d) = Undefined
  then failwith (``wrong assignment'')
  else match d with
    | (s2,n2) ->
       (match s2 with
         | Dstore(n1) -> let a =
            access(storestack, n1) in
            Array.set a n2 m; s
         | Pstore(n1) -> let a =
            getst(applyheap(!currentheap,n1)) in
            Array.set a n2 m; s )

let eredita ((rho:  dval env), ogg, (h:  heap)) =
  let currn = (match rho with | Denv(n) -> n) in
  let (point, arrnomisotto, arrdensotto, arrstore) =
    (match ogg with
      | Object(n) -> let oo = applyheap(!currentheap,n) in
         (n, getnomi(oo), getden(oo), getst(oo))
      | _ -> failwith(``not an object in eredita'')) in
  let stl = Array.length arrstore in
  let newstore = Array.create stl Undefined in
  let index = ref(0) in
    while !index < stl do
      Array.set newstore !index (Array.get arrstore !index);
      index := !index + 1
    done;
    pop(storestack);
    push(newstore, storestack);
    let currarrnomi = access(namestack, currn) in
    let currarrden = access(dvalstack, currn) in
    let r = access(slinkstack, currn) in
    let currl = Array.length currarrnomi in
    let oldlen = Array.length arrnomisotto in
      index := 0;
      while not(Array.get arrnomisotto !index = ``this'') do
        index := !index + 1 done;
      index := !index + 1;
      let newlen = (currl + oldlen - !index ) in
```

```
      let newarrnomi = Array.create newlen ``dummy'' in
      let newarrden = Array.create newlen Unbound in
      let newindex = ref(0) in
       while !newindex < currl do
        Array.set newarrnomi !newindex (Array.get currarrnomi !newindex);
        Array.set newarrden !newindex (Array.get currarrden !newindex);
        newindex := !newindex + 1
       done;
       while !newindex < newlen do
        Array.set newarrnomi !newindex (Array.get arrnomisotto !index);
        Array.set newarrden !newindex (Array.get arrdensotto !index);
        newindex := !newindex + 1;
        index := !index + 1
       done;
       pop(namestack);pop(dvalstack);pop(slinkstack);
       push(newarrnomi, namestack);
       push(newarrden,dvalstack);
       push(r,slinkstack);;

let localenv((r:dval env),(s:  mval store),
          Dobject(ob),(li:ide list), (r1:  dval env)) =
  let (rint, sint) =
    (match (r,s) with
      | (Denv nr, Dstore ns) -> (nr, ns)
      | _ -> failwith(``heap structures in localenv'')) in
  let oldst = access(storestack, sint) in
  let oldnomi = access(namestack, rint) in
  let oldden = access(dvalstack, rint) in
  let storesize = Array.length oldst in
  let newst = Array.create storesize Undefined in
  let index = ref(0) in
   while not(!index = storesize) do
    Array.set newst !index (Array.get oldst !index);
    index := !index + 1
   done;
   let oldenvlength = Array.length oldnomi in
   let newenvlength = oldenvlength - (List.length li) in
   let newnomi = Array.create newenvlength ``dummy'' in
   let newden = Array.create newenvlength Unbound in
   let index = ref(0) in
   let newindex = ref(0) in
    while not(!index = oldenvlength) do
     let lname = Array.get oldnomi !index in
     let lden = Array.get oldden !index in
      if notoccur(lname, li) then (
       Array.set newnomi !newindex lname;
       let ldennuova =
        (match lden with
         | Dfunval(e,rho) ->
```

```
              if rho >= r
              then Dfunval(e,Penv(ob))
              else lden
           | Dprocval(e,rho) ->
              if rho >= r
              then Dprocval(e,Penv(ob))
              else lden
           | Dloc(sigma, n) ->
              if sigma >= s
              then Dloc(Pstore(ob),n)
              else lden
           | _ -> lden) in
         Array.set newden !newindex ldennuova;
         newindex := !newindex + 1)
      else ();
      index := !index + 1
   done;
   (newnomi, newden, r1, newst)
```

15.3 Novità nell'interprete iterativo

L'introduzione dei due tipi ambiente e memoria comporta una lunga serie di adattamenti nell'interprete iterativo.
Vediamo le modifiche più importanti, relative al trattamento degli oggetti:

```
let itsemobj() =
   ...
   | Ogg1(Class(name,fpars,extends,(b1,b2,b3))) ->
      pop(continuation);
      (match extends with
         | (``Object'',_) ->
            push(
              Ogg3(
                Class(name,fpars,extends,(b1,b2,b3))),
              continuation);
            push(labelcom(b3), top(cstack));
            push(Rdecl(b2), top(cstack));
            push(labeldec(b1),top(cstack));
            pushlocalenv(b1,b2,rho);
            pushlocalstore(b1);
            ()
         | (super,superpars) ->
            let lobj = applyenv(rho, ``this'') in
            let superargs =
              findsuperargs(fpars,
                            dlist(fpars,rho),
                            superpars) in
              push(Ogg2(Class(name,
```

```
                        fpars,
                        extends,
                        (b1,b2,b3) )),
                  continuation);
              (match applyenv(rho, super) with
                | Classval(Class(snome,
                             superfpars,
                             sextends, sb),
                           r) ->
                  newframes(Ogg1(Class(snome,
                                       superfpars,
                                       sextends,
                                       sb)),
                            bindlist(r,
                                     superfpars @ [``this''],
                                     superargs @ [lobj]),
                            sigma)
                | _ -> failwith(``not a superclass name'')))
  | Ogg2(Class(name, fpars, extends, (b1,b2,b3) )) ->
     pop(continuation);
     let v = top(tempstack) in
      pop(tempstack);
      eredita(rho, v, !currentheap);
      push(Ogg3(Class(name,
                  fpars,
                  extends,
                  (b1,b2,b3) )),
           continuation);
      push(labelcom(b3), top(cstack));
      push(Rdecl(b2), top(cstack));
      push(labeldec(b1),top(cstack));
      pushlocalenv(b1,b2,rho);
      pushlocalstore(b1);
      ()
  | Ogg3(Class(name, fpars, extends, (b1,b2,b3) )) ->
     pop(continuation);
     let r = (match applyenv(rho,name) with
               | Classval(_, r1) -> r1
               | _ -> failwith(``not a class name'')) in
     let lobj =
       (match applyenv(rho, ``this'') with
          | Dobject n -> n) in
     let newobj =
       localenv(rho, sigma, Dobject(lobj), fpars, r) in
       currentheap := allocateheap (!currentheap,
                                    lobj, newobj);
       push(Object lobj, tempstack)
  | _ -> failwith(``impossible in semobj''))
```

15.4 Interprete iterativo

Interprete iterativo completo per il linguaggio object-oriented, con ambiente e memoria:

```
exception Nonstorable
exception Nonexpressible
type pointer = int
type 't env = Denv of int | Penv of pointer
and loc = mval store * int
and 't store = Dstore of int | Pstore of pointer
and eval =
   | Int of int
   | Bool of bool
   | Novalue
   | Funval of efun
   | Object of pointer

and efun = exp * (dval env)
and proc = exp * (dval env)
and eclass = cdecl * (dval env)

and obj = ( ide array * dval array *
            dval env * mval array )
and heap = obj array

and dval =
   | Dint of int
   | Dbool of bool
   | Unbound
   | Dloc of loc
   | Dfunval of efun
   | Dprocval of proc
   | Dobject of pointer
   | Classval of eclass

and mval =
   | Mint of int
   | Mbool of bool
   | Undefined
   | Mobject of pointer

let evaltomval e = match e with
   | Int n -> Mint n
   | Bool n -> Mbool n
   | Object n -> Mobject n
   | _ -> raise Nonstorable

let mvaltoeval m = match m with
   | Mint n -> Int n
```

```
   | Mbool n -> Bool n
   | Mobject n -> Object n
   | _ -> Novalue

let evaltodval e = match e with
   | Int n -> Dint n
   | Bool n -> Dbool n
   | Novalue -> Unbound
   | Funval f -> Dfunval f
   | Object n -> Dobject n

let dvaltoeval e = match e with
   | Dint n -> Int n
   | Dbool n -> Bool n
   | Dloc n -> raise Nonexpressible
   | Dprocval n -> raise Nonexpressible
   | Unbound -> Novalue
   | Dfunval f -> Funval f
   | Dobject n -> Object n
   | Classval n -> raise Nonexpressible

type labeledconstruct =
   | Expr1 of exp
   | Expr2 of exp
   | Exprd1 of exp
   | Exprd2 of exp
   | Com1 of com
   | Com2 of com
   | Coml of labeledconstruct list
   | Decl of labeledconstruct list
   | Rdecl of (ide * exp) list
   | Dec1 of (ide * exp)
   | Dec2 of (ide * exp)
   | Ogg1 of cdecl
   | Ogg2 of cdecl
   | Ogg3 of cdecl

let (cstack:  labeledconstruct stack stack) =
   emptystack(stacksize,emptystack(1,Expr1(Eint(0))))

let (tempvalstack:  eval stack stack) =
   emptystack(stacksize,emptystack(1,Novalue))

let (tempdvalstack:  dval stack stack) =
   emptystack(stacksize,emptystack(1,Unbound))

let labelcom (dl:  com list) = let dlr = ref(dl) in
let ldlr = ref([]) in
   while not (!dlr = []) do
```

```
    let i = List.hd !dlr in
      ldlr := !ldlr @ [Com1(i)];
      dlr := List.tl !dlr
  done;
  Coml(!ldlr)

let labeldec (dl:  (ide * exp) list) = let dlr = ref(dl) in
let ldlr = ref([]) in
  while not (!dlr = []) do
    let i = List.hd !dlr in
      ldlr := !ldlr @ [Dec1(i)];
      dlr := List.tl !dlr
  done;
  Decl(!ldlr)

let (labelstack:  labeledconstruct stack) =
  emptystack(stacksize,Expr1(Eint(0)))

let pushargs ((b:  exp list),
           (continuation:  labeledconstruct stack)) =
  let br = ref(b) in
    while not(!br = []) do
      push(Exprd1(List.hd !br),continuation);
      br := List.tl !br
    done

let getargs ((b:  exp list),(tempstack:  dval stack)) =
  let br = ref(b) in
     let er = ref([]) in
       while not(!br = []) do
         let arg=top(tempstack) in
           pop(tempstack); er := !er @ [arg];
           br := List.tl !br
       done;
       !er

let (newpoint,initpoint) =
  let count = ref(-1) in
  (fun () -> count := !count +1;
     !count),
  (fun () -> count := -1)

let emptyheap () = initpoint();
  (Array.create 100 ((Array.create 1 ``dummy''),
                (Array.create 1 Unbound),
                Denv(-1),
                (Array.create 1 Undefined))
    :  heap)
```

```
let currentheap = ref(emptyheap())

let applyheap ((x:  heap), (y:pointer)) = Array.get x y

let allocateheap ((x:heap), (i:pointer), (r:obj)) =
  Array.set x i r; x

let getnomi(x:obj) = match x with
  | (nomi, den, slink, st) -> nomi

let getden(x:obj) = match x with
  | (nomi, den, slink, st) -> den

let getslink(x:obj) = match x with
  | (nomi, den, slink, st) -> slink

let getst(x:obj) = match x with
  | (nomi, den, slink, st) -> st

let currentenv = ref(0)

let namestack = emptystack(stacksize,[| ``dummy'' |])

let dvalstack = emptystack(stacksize,[| Unbound |])

let slinkstack = emptystack(stacksize, Denv(0))

let overriden ((a:  ide array), (i:  ide)) =
  let len = Array.length a in
  let ind = ref(0) in
  let res = ref(false) in
   while !ind < len do
     if Array.get a !ind = i
     then (res := true; ind := len)
     else ind := !ind + 1
   done;
   !res

let bindlist(r, il, el) =
  let n = List.length il in
  let ii = Array.create n ''dummy'' in
  let dd = Array.create n Unbound in
  let ri = ref(il) in
  let rd = ref(el) in
  let index = ref(0) in
   while !index < n do
     let i = List.hd !ri in
     let d = List.hd !rd in
       ( ri := List.tl !ri;
```

```
        rd := List.tl !rd;
        Array.set ii !index i;
        Array.set dd !index d;
        index := !index + 1)
    done;
    push(ii, namestack);
    push(dd,dvalstack);
    push(r,slinkstack);
    (Denv(lungh(namestack)):  dval env)

let bind ((x:dval env),i,d) =
  match x with
    | Penv(x1) -> failwith(``bind for objects'')
    | Denv(r) ->
       let arrnomi = access(namestack, r) in
       let lun = Array.length arrnomi in
       let arrden = access(dvalstack, r) in
       let n = ref(0) in
       let index = ref(-1) in
        while !n < lun do
          let nome = Array.get arrnomi !n in
            if nome = i then
              (Array.set arrden !n d;
              n := lun; index := -2) else
                if nome = ``dummy''
                then (index := !n; n := lun)
                else n := !n + 1
        done;
        if !index = -2 then x else
          if !index > -1
          then
            (Array.set arrnomi !index i;
            Array.set arrden !index d; x)
          else
            (push(Array.create 1 i,namestack);
            push(Array.create 1 d,dvalstack);
            push(x,slinkstack);
            (Denv(lungh(namestack)):  dval env))

let emptyenv(x) = currentenv := -1; svuota(namestack);
  svuota(dvalstack); svuota(slinkstack);
  Denv(!currentenv)

let svuotaenv() = svuota(namestack);
  svuota(dvalstack); svuota(slinkstack)

let applyenv ((r:  dval env), (y:  ide)) =
  let den = ref(Unbound) in
  let (x, caso) =
```

```
    (match r with
      | Denv(x1) -> (x1, ref(``stack''))
      | Penv(x1) -> (x1, ref(``heap''))) in
  let n = ref(x) in
    while !n > -1 do
      let lenv =
        if !caso = ``stack''
        then access(namestack,!n)
        else getnomi(applyheap(!currentheap,!n)) in
      let nl = Array.length lenv in
      let index = ref(0) in
        while !index < nl do
          if Array.get lenv !index = y then
            (den := (if !caso = ''stack''
                  then
                    Array.get (access(dvalstack,!n)) !index
                  else
                    Array.get
                      (getden(applyheap(!currentheap,!n)))
                      !index);
           index := nl)
          else index := !index + 1
        done;
        if not(!den = Unbound) then n := -1
        else let next =
          (if !caso = ``stack''
          then access(slinkstack,!n)
          else getslink(applyheap(!currentheap,!n))) in
          caso := (match next with
                  |Denv(_) -> ``stack''
                  |Penv(_) -> ``heap'');
          n := (match next with
                |Denv(n1) -> n1
                |Penv(n1) -> n1)
    done;
    !den;;

let pushlocalenv (dl,dlr,x) =
  match x with
    | Penv(r) -> failwith(``pushlocalenv for objects'')
    | Denv(r) ->
       let anomi = access(namestack, r) in
       let aden = access(dvalstack, r) in
       let nn = ref(Array.length anomi) in
       let lun1 = ref(0) in
       let dlp = ref(dl) in
        while not(!dlp = []) do
          let i = (match List.hd !dlp with
                  | (i1, _) -> i1) in
```

```
            dlp := List.tl !dlp;
            if overriden(anomi, i)
            then failwith(``field overriding is not allowed'')
            else lun1:= !lun1 + 1
          done;
          let lun2 = ref(0) in
          let dlp = ref(dlr) in
           while not(!dlp = []) do
            let i = (match List.hd !dlp with
                    | (i1, _) -> i1) in
              dlp := List.tl !dlp;
              if overriden(anomi, i) then ()
              else lun1:= !lun1 + 1
           done;
           let rn = !lun1 + !lun2 + !nn in
           let ii = Array.create rn ''dummy'' in
           let dd = Array.create rn Unbound in
            while not(!nn = 0) do
              Array.set ii
                (!nn - 1)
                (Array.get anomi (!nn - 1));
              Array.set dd
                (!nn - 1)
                (Array.get aden (!nn - 1));
              nn := !nn - 1
            done;
            pop(namestack); pop(dvalstack);
            push(ii, namestack);
            push(dd, dvalstack);
            x

let topenv() = Denv(!currentenv)

let popenv () = pop(namestack);
   pop(dvalstack);
   pop(slinkstack);
   currentenv := !currentenv - 1

let pushemptylocalenv(r) = push([||],namestack);
   push([||],dvalstack);
   push(r,slinkstack);
   currentenv := lungh(namestack)

let pushenv(r1) = match r1 with Denv(r) ->
   if r = !currentenv then
     pushemptylocalenv(r1)
   else currentenv := r

let currentstore = ref(0)
```

```
let storestack = emptystack(stacksize,[|Undefined|])

let (newloc,initloc) =
  let count = ref(-1) in
   (fun () -> let arrst = access(storestack, !currentstore) in
   let len = Array.length arrst in
   let index = ref(0) in
     while !index < len do
       if Array.get arrst !index = Undefined then
         (count := !index; index := len)
       else index := !index + 1
     done;
     (Dstore(!currentstore), !count)),
  (fun () -> count := -1)

let applystore ((x:  mval store), (d:  loc)) =
  match d with
    | (s2,n2)->
       (match s2 with
         | Dstore(n1) -> let a =
            access(storestack, n1) in
            Array.get a n2
         | Pstore(n1) -> let a =
            getst(applyheap(!currentheap,n1)) in
            Array.get a n2)
    | _ -> failwith(``not a location in applystore'')

let emptystore(x) = initloc();
  svuota(storestack); currentstore := -1;
  Dstore(!currentstore)

let allocate ((s:mval store), (m:mval)) =
  let (s2, n2) = newloc() in
    (match s2 with
      | Pstore(n1) -> let a = access(storestack, n1) in
         Array.set a n2 m; ((s2, n2),s)
      | Dstore(n1) -> let a = access(storestack, n1) in
         Array.set a n2 m; ((s2, n2),s))

let update((s:mval store), (d:  loc), (m:mval)) =
  if applystore(s, d) = Undefined
  then failwith (``wrong assignment'')
  else match d with
    | (s2,n2) ->
       (match s2 with
         | Dstore(n1) ->
            let a = access(storestack, n1) in
              Array.set a n2 m; s
```

```
        | Pstore(n1) ->
           let a = getst(applyheap(!currentheap,n1)) in
            Array.set a n2 m; s )

let pushlocalstore (dl) =
  let currarr = access(storestack, !currentstore) in
  let currl = Array.length currarr in
  let rdl = ref(dl) in
  let rn = ref(0) in
   while not(!rdl = []) do
    let (i, d) = List.hd !rdl in
     (match d with
      | Newloc(_) -> rn := !rn + 1
      | _ -> ());
     rdl := List.tl !rdl
   done;
   let a = Array.create (!rn + currl) Undefined in
   let index1 = ref(0) in
    while !index1 < currl do
     let y = Array.get currarr !index1 in
      Array.set a !index1 y;
      index1 := !index1 + 1
    done;
    pop(storestack); push(a, storestack);
    Dstore(!currentstore)

let pushemptylocalstore () = push([||],storestack);
  currentstore := !currentstore + 1

let pushstore(n1) = match n1 with Dstore(n) ->
  if n = !currentstore
  then pushemptylocalstore()
  else currentstore := n

let popstore () = pop(storestack); currentstore := !currentstore -1

let svuotastore () = svuota(storestack)

let topstore() = Dstore(!currentstore)

let dlist ((b:  ide list),(r:  dval env)) =
  let br = ref(b) in
  let er = ref([]) in
   while not(!br = []) do
    let arg= applyenv(r, List.hd !br) in
     er := !er @ [arg];
     br := List.tl !br
   done;
   !er
```

```
let notoccur ((i:ide), l) = let vl = ref(l) in
let res = ref(true) in
   while not (!vl = []) do
    if i = List.hd(!vl) then (res := false; vl := [])
    else vl := List.tl(!vl)
   done;
   !res

let findone ((args:  ide list), (el:  dval list), (s:  ide )) =
   let vargs = ref(args) in
   let dargs = ref(el) in
   let valore = ref(Unbound) in
    while not(!vargs = []) do
     if s = List.hd !vargs then
       (vargs := []; valore := List.hd !dargs)
     else vargs := List.tl !vargs; dargs := List.tl !dargs
    done;
    !valore

let findsuperargs ((args:  ide list),
               (el:  dval list),(sargs:  ide list)) =
   let vsargs = ref(sargs) in
   let res = ref([]) in
    while not(!vsargs = []) do
     let i = findone(args, el, List.hd(!vsargs)) in
       if i = Unbound then failwith(``problem with super parameters'')
       else res := !res @ [i]; vsargs := List.tl !vsargs
    done;
    !res

let localenv ((r:dval env) , (s:  mval store),
           Dobject(ob), (li:ide list), (r1:  dval env)) =
   let (rint, sint) =
    (match (r,s) with
     | (Denv nr, Dstore ns) -> (nr, ns)
     | _ -> failwith(``heap structures in localenv'')) in
   let oldst = access(storestack, sint) in
   let oldnomi = access(namestack, rint) in
   let oldden = access(dvalstack, rint) in
   let storesize = Array.length oldst in
   let newst = Array.create storesize Undefined in
   let index = ref(0) in
    while not(!index = storesize) do
     Array.set newst !index (Array.get oldst !index);
     index := !index + 1
    done;
    let oldenvlength = Array.length oldnomi in
    let newenvlength = oldenvlength - (List.length li) in
```

```
    let newnomi = Array.create newenvlength ''dummy'' in
    let newden = Array.create newenvlength Unbound in
    let index = ref(0) in
    let newindex = ref(0) in
     while not(!index = oldenvlength) do
      let lname = Array.get oldnomi !index in
      let lden = Array.get oldden !index in
       if notoccur(lname, li) then (
        Array.set newnomi !newindex lname;
        let ldennuova =
          (match lden with
            | Dfunval(e,rho) ->
               if rho >= r
               then Dfunval(e,Penv(ob))
               else lden
            | Dprocval(e,rho) ->
               if rho >= r
               then Dprocval(e,Penv(ob))
               else lden
            | Dloc(sigma, n) ->
               if sigma >= s
               then Dloc(Pstore(ob),n)
               else lden
            | _ -> lden) in
          Array.set newden !newindex ldennuova;
          newindex := !newindex + 1)
       else ();
       index := !index + 1
     done;
     (newnomi, newden, r1, newst)

let eredita ((rho:  dval env), ogg, (h:  heap)) =
  let currn = (match rho with | Denv(n) -> n) in
  let (point, arrnomisotto, arrdensotto, arrstore) =
    (match ogg with
      | Object(n) -> let oo =
         applyheap(!currentheap,n) in
         (n, getnomi(oo), getden(oo), getst(oo))
      | _ -> failwith(``not an object in eredita'')) in
  let stl = Array.length arrstore in
  let newstore = Array.create stl Undefined in
  let index = ref(0) in
   while !index < stl do
    Array.set newstore !index (Array.get arrstore !index);
    index := !index + 1
   done;
   pop(storestack);
   push(newstore, storestack);
   let currarrnomi = access(namestack, currn) in
```

```
    let currarrden = access(dvalstack, currn) in
    let r = access(slinkstack, currn) in
    let currl = Array.length currarrnomi in
    let oldlen = Array.length arrnomisotto in
     index := 0;
     while not(Array.get arrnomisotto !index = ‘‘this’’) do
      index := !index + 1 done;
     index := !index + 1;
     let newlen = (currl + oldlen - !index ) in
     let newarrnomi = Array.create newlen ’’dummy’’ in
     let newarrden = Array.create newlen Unbound in
     let newindex = ref(0) in
      while !newindex < currl do
       Array.set newarrnomi !newindex
        (Array.get currarrnomi !newindex);
       Array.set newarrden !newindex
        (Array.get currarrden !newindex);
       newindex := !newindex + 1
      done;
      while !newindex < newlen do
       Array.set newarrnomi
        !newindex (Array.get arrnomisotto !index);
       Array.set newarrden
        !newindex (Array.get arrdensotto !index);
       newindex := !newindex + 1;
       index := !index + 1
      done;
      pop(namestack);pop(dvalstack);
      pop(slinkstack);
      push(newarrnomi, namestack);
      push(newarrden,dvalstack);
      push(r,slinkstack);;

let newframes(ss,rho,sigma) =
  pushstore(sigma);
  pushenv(rho);
  let cframe =
   emptystack(cframesize(ss),Expr1(Eint 0)) in
  let tframe =
   emptystack(tframesize(ss),Novalue) in
  let dframe =
   emptystack(tdframesize(ss),Unbound) in
   push(tframe,tempvalstack);
   push(dframe,tempdvalstack);
   push(ss, labelstack);
   push(ss, cframe);
   push(cframe, cstack)

let makefun ((a:exp),(x:dval env)) =
```

```
   (match a with
    | Fun(ii,aa) ->
        Dfunval(a,x)
    | _ -> failwith (``Non-functional object''))

let makeproc ((a:exp),(x:dval env)) =
   (match a with
    | Proc(ii,b) ->
        Dprocval(a,x)
    | _ -> failwith (``Non-procedural object''))

let makefunrec (f, (a:exp),(x:dval env)) =
   dvaltoeval(makefun(a, Denv(lungh(namestack) + 1)) )

let applyfun ((ev1:dval),(ev2:dval list), s) =
   ( match ev1 with
      | Dfunval(Fun(ii,aa),r) ->
         newframes(Expr1(aa),bindlist(r, ii, ev2), s)
      | _ -> failwith (``attempt to apply a non-functional object''))

let applyproc ((ev1:dval),(ev2:dval list), s) =
   ( match ev1 with
      | Dprocval(Proc(ii,(l1, l2, l3)), x) ->
         let r = bindlist(x, ii, ev2) in
          newframes(labelcom(l3), r, s);
          push(Rdecl(l2), top(cstack));
          push(labeldec(l1),top(cstack));
          pushlocalenv(l1,l2,r);
          pushlocalstore(l1);
          ()
      | _ -> failwith (``attempt to apply a non-functional object''))

let makeclass((c:  cdecl), r) = Classval(c, r)

let applyclass ((ev1:dval),(apars:dval list), s, h) =
   ( match ev1 with
      | Classval(Class(name,fpars,extends,(b1,b2,b3)),r) ->
         let j = newpoint() in
          newframes(Ogg1(Class(name,
                          fpars,
                          extends,
                          (b1,b2,b3) )),
                  bindlist(r,
                        fpars @ [``this''],
                        apars @ [Dobject(j)]), s)
      | _ -> failwith(``not a class''))

let itsem() =
   let continuation = top(cstack) in
```

```
let tempstack = top(tempvalstack) in
let tempdstack = top(tempdvalstack) in
let rho = topenv() in
let sigma = topstore() in
  (match top(continuation) with
    |Expr1(x) ->
       (pop(continuation);
       push(Expr2(x),continuation);
       (match x with
          | Iszero(a) ->
             push(Expr1(a),continuation)
          | Eq(a,b) ->
             push(Expr1(a),continuation);
             push(Expr1(b),continuation)
          | Prod(a,b) ->
             push(Expr1(a),continuation);
             push(Expr1(b),continuation)
          | Sum(a,b) ->
             push(Expr1(a),continuation);
             push(Expr1(b),continuation)
          | Diff(a,b) ->
             push(Expr1(a),continuation);
             push(Expr1(b),continuation)
          | Minus(a) ->
             push(Expr1(a),continuation)
          | And(a,b) ->
             push(Expr1(a),continuation);
             push(Expr1(b),continuation)
          | Or(a,b) ->
             push(Expr1(a),continuation);
             push(Expr1(b),continuation)
          | Not(a) ->
             push(Expr1(a),continuation)
          | Ifthenelse(a,b,c) ->
             push(Expr1(a),continuation)
          | Val(e) -> push(Exprd1(e),continuation)
          | Newloc(e) ->
             failwith (``nonlegal expression for sem'')
          | Let(i,e1,e2) ->
             push(Exprd1(e1),continuation)
       | Appl(a,b) ->
          push(Exprd1(a),continuation);
          pushargs(b,continuation)
       | Proc(i,b) ->
          failwith (``nonlegal expression for sem'')
       | New(i,ge) ->
          pushargs(ge, continuation)
       | _ -> ()))
    |Expr2(x) ->
```

```
(pop(continuation);
(match x with
   | Eint(n) -> push(Int(n),tempstack)
   | Ebool(b) -> push(Bool(b),tempstack)
   | Den(i) ->
      push(dvaltoeval(applyenv(rho,i)),tempstack)
   | Iszero(a) ->
      let arg=top(tempstack) in
       pop(tempstack);
       push(iszero(arg),tempstack)
   | Eq(a,b) ->
      let firstarg=top(tempstack) in
       pop(tempstack);
       let sndarg=top(tempstack) in
         pop(tempstack);
         push(equ(firstarg,sndarg),tempstack)
   | Prod(a,b) ->
      let firstarg=top(tempstack) in
       pop(tempstack);
       let sndarg=top(tempstack) in
         pop(tempstack);
         push(mult(firstarg,sndarg),tempstack)
   | Sum(a,b) ->
      let firstarg=top(tempstack) in
       pop(tempstack);
       let sndarg=top(tempstack) in
         pop(tempstack);
         push(plus(firstarg,sndarg),tempstack)
   | Diff(a,b) ->
      let firstarg=top(tempstack) in
       pop(tempstack);
       let sndarg=top(tempstack) in
         pop(tempstack);
         push(diff(firstarg,sndarg),tempstack)
   | Minus(a) ->
      let arg=top(tempstack) in
       pop(tempstack);
       push(minus(arg),tempstack)
   | And(a,b) ->
      let firstarg=top(tempstack) in
       pop(tempstack);
       let sndarg=top(tempstack) in
         pop(tempstack);
         push(et(firstarg,sndarg),tempstack)
   | Or(a,b) ->
      let firstarg=top(tempstack) in
       pop(tempstack);
       let sndarg=top(tempstack) in
         pop(tempstack);
```

```
                push(vel(firstarg,sndarg),tempstack)
            | Not(a) ->
               let arg=top(tempstack) in
                pop(tempstack);
                push(non(arg),tempstack)
            | Ifthenelse(a,b,c) ->
               let arg=top(tempstack) in
                pop(tempstack);
                if typecheck(``bool'',arg) then
                  (if arg = Bool(true)
                  then push(Expr1(b),continuation)
                  else push(Expr1(c),continuation))
                else failwith (``type error'')
            | Val(e) -> let v = top(tempdstack) in
               pop(tempdstack);
               (match v with
                | Dloc n ->
                   push(
                     mvaltoeval(applystore(sigma, n)),
                     tempstack)
                | _ -> failwith(``not a variable''))
            | Fun(i,a) ->
               push(dvaltoeval(makefun(Fun(i,a),rho)),tempstack)
            | Rec(f,e) ->
               push(makefunrec(f,e,rho),tempstack)
            | Let(i,e1,e2) -> let arg= top(tempdstack) in
               pop(tempdstack);
               newframes(Expr1(e2), bind(rho, i, arg), sigma)
            | Appl(a,b) -> let firstarg=top(tempdstack) in
               pop(tempdstack);
               let sndarg=getargs(b,tempdstack) in
                applyfun(firstarg, sndarg, sigma)
            | New(i,ge) ->
               let arg=getargs(ge,tempdstack) in
                applyclass(applyenv(rho,i), arg, sigma, !currentheap)
            | This ->
               push(dvaltoeval(applyenv(rho,''this'')), tempstack)
            | _ -> failwith(``no more cases for semexpr'')))
      | _ -> failwith(``no more cases for semexpr''))

let itsemden() =
  let continuation = top(cstack) in
  let tempstack = top(tempvalstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
    (match top(continuation) with
     |Exprd1(x) ->
        (pop(continuation); push(Exprd2(x),continuation);
```

```
        match x with
          | Den i -> ()
          | Fun(i,e) -> ()
          | Proc(il,b) -> ()
          | Field(e,i) ->
             push(Expr1(e), continuation)
          | Newloc(e) -> push(Expr1( e), continuation)
          | _ -> push(Expr1(x), continuation))
      |Exprd2(x) ->
         (pop(continuation); match x with
          | Den i -> push(applyenv(rho,i), tempdstack)
          | Fun(i,e) -> push(makefun(x,rho),tempdstack)
          | Proc(il,b) -> push(makeproc(x,rho),tempdstack)
          | Field(e,i) -> let ogg = top(tempstack) in
             pop(tempstack);
             (match ogg with
                | Object i1 -> let r1 = Penv(i1) in
                 let field = applyenv(r1,i) in
                   push(field, tempdstack)
                | _ -> failwith(``notanobject''))
          | Newloc(e) ->let m=evaltomval(top(tempstack)) in
             pop(tempstack);
             let (l, ss) = allocate(sigma, m) in
              push(Dloc l, tempdstack);
              currentstore :=
                (match ss with
                   | Dstore(s1) -> s1
                   | _ -> failwith(``object store in newloc''))
          | _ -> let arg = top(tempstack) in pop(tempstack);
             push(evaltodval(arg), tempdstack))
      | _ -> failwith(``No more cases for demden'') )

let itsemdecl () =
   let tempstack = top(tempvalstack) in
   let continuation = top(cstack) in
   let tempdstack = top(tempdvalstack) in
   let rho = topenv() in
   let sigma = topstore() in
   let dl =
     (match top(continuation) with
      | Decl(dl1) -> dl1
      | _ -> failwith(``impossible in semdecl'')) in
     if dl = [] then pop(continuation) else
      (let currd = List.hd dl in
      let newdl = List.tl dl in
       pop(continuation); push(Decl(newdl),continuation);
       (match currd with
          | Dec1( (i,e)) ->
             pop(continuation);
```

```
              push(Decl(Dec2((i, e))::newdl),continuation);
              push(Exprd1(e), continuation)
            | Dec2((i,e)) ->
              let arg = top(tempdstack) in
                 pop(tempdstack);
                let rr = bind(rho, i, arg) in
                 currentenv :=
                   (match rr with
                     | Denv(rr1) -> rr1
                     | Penv(_) ->
                        failwith(``object env in a declaration''))
          | _ -> failwith(``no more sensible cases for semdecl''))))

let itsemrdecl() =
   let tempstack = top(tempvalstack) in
   let continuation = top(cstack) in
   let tempdstack = top(tempdvalstack) in
   let rho = topenv() in
   let sigma = topstore() in
   let rl =
     (match top(continuation) with
      | Rdecl(rl1) -> rl1
      | _ -> failwith(``impossible in semrdecl'')) in
    pop(continuation);
    let prl = ref(rl) in
     while not(!prl = []) do
      let currd = List.hd !prl in
       prl := List.tl !prl;
       let (i, den) =
         (match currd with
          |(j, Proc(il,b)) ->
             (j, makeproc(Proc(il,b),rho))
          |(j, Fun(il,b)) ->
             (j, makefun(Fun(il,b),rho))
          | _ ->
             failwith(``no more sensible cases in ...''))  in
        let rr = bind(rho, i, den) in
         currentenv :=
           (match rr with
            | Denv(rr1) -> rr1
            | Penv(_) ->
               failwith(``object env in a declaration''))
     done

let itsemcl() =
   let tempstack = top(tempvalstack) in
   let continuation = top(cstack) in
   let tempdstack = top(tempdvalstack) in
   let rho = topenv() in
```

```
let sigma = topstore() in
let cl =
  (match top(continuation) with
    | Coml(dl1) -> dl1
    | _ -> failwith(``impossible in semcl'')) in
  if cl = [] then pop(continuation) else
    (let currc = List.hd cl in
    let newcl = List.tl cl in
     pop(continuation);
     push(Coml(newcl),continuation);
       (match currc with
          | Com1(Assign(e1, e2)) ->
           pop(continuation);
           push(Coml(Com2(Assign(e1, e2))::newcl),continuation);
           push(Exprd1(e1), continuation);
           push(Expr1(e2), continuation)
        | Com2(Assign(e1, e2)) ->
           let arg2 = evaltomval(top(tempstack)) in
            pop(tempstack);
            let arg1 = top(tempdstack) in
             pop(tempdstack);
             (match arg1 with
                | Dloc(ss, n) ->
                 update(sigma, (ss,n), arg2); ()
                | _ -> failwith (``wrong location in assignment''))
        | Com1(While(e, cl)) ->
           pop(continuation);
           push(Coml(Com2(While(e, cl))::newcl),continuation);
           push(Expr1(e), continuation)
        | Com2(While(e, cl)) ->
           let g = top(tempstack) in
            pop(tempstack);
            if typecheck(``bool'',g) then
              (if g = Bool(true) then
                 (let old = newcl in
                 let newl =
                   (match labelcom cl with
                     | Coml newl1 -> newl1
                     | _ ->
                        failwith(``impossible in while'')) in
                 let nuovo =
                 Coml(newl @ [Com1(While(e, cl))] @ old) in
                  pop(continuation); push(nuovo,continuation))
               else ())
             else failwith (``nonboolean guard'')
        | Com1(Cifthenelse(e, cl1, cl2)) ->
           pop(continuation);
           push(Coml(
                Com2(Cifthenelse(e, cl1, cl2))::newcl),
```

```
              continuation);
            push(Expr1(e), continuation)
        | Com2(Cifthenelse(e, cl1, cl2)) ->
            let g = top(tempstack) in
             pop(tempstack);
             if typecheck(``bool'',g) then
              (let temp = if g = Bool(true) then
               labelcom (cl1) else labelcom (cl2) in
              let newl =
               (match temp with
                  | Coml newl1 -> newl1
                  | _ -> failwith(``impossible in cifthenelse'')) in
              let nuovo = Coml(newl @ newcl) in
                 pop(continuation); push(nuovo,continuation))
             else failwith (``nonboolean guard'')
        | Com1(Call(e, el)) ->
            pop(continuation);
            push(Coml(Com2(Call(e, el))::newcl),continuation);
            push(Exprd1( e), continuation);
            pushargs(el, continuation)
        | Com2(Call(e, el)) ->
            let p = top(tempdstack) in
             pop(tempdstack);
             let args = getargs(el,tempdstack) in
               applyproc(p, args, sigma)
         | Com1(Block((l1, l2, l3))) ->
            newframes(labelcom(l3), rho, sigma);
            push(Rdecl(l2), top(cstack));
            push(labeldec(l1),top(cstack));
            pushlocalenv(l1,l2,rho);
            pushlocalstore(l1);
            ()
        | _ -> failwith(``no more sensible cases in commands'') ))

let itsemobj() =
  let tempstack = top(tempvalstack) in
  let continuation = top(cstack) in
  let tempdstack = top(tempdvalstack) in
  let rho = topenv() in
  let sigma = topstore() in
    (match top(continuation) with
     | Ogg1(Class(name,fpars,extends,(b1,b2,b3))) ->
        pop(continuation);
        (match extends with
           | (``Object'',_) ->
              push(
               Ogg3(
                Class(name,fpars,extends,(b1,b2,b3))),
               continuation);
```

```
          push(labelcom(b3), top(cstack));
          push(Rdecl(b2), top(cstack));
          push(labeldec(b1),top(cstack));
          pushlocalenv(b1,b2,rho);
          pushlocalstore(b1);
          ()
        | (super,superpars) ->
          let lobj = applyenv(rho, ''this'') in
          let superargs =
           findsuperargs(fpars,
                      dlist(fpars,rho),
                      superpars) in
           push(Ogg2(Class(name,
                       fpars,
                       extends,
                       (b1,b2,b3) )),
              continuation);
           (match applyenv(rho, super) with
            | Classval(Class(snome,
                          superfpars,
                          sextends, sb),
                      r) ->
              newframes(Ogg1(Class(snome,
                                   superfpars,
                                   sextends,
                                   sb)),
                        bindlist(r,
                                 superfpars @ [``this''],
                                 superargs @ [lobj]),
                        sigma)
            | _ -> failwith(``not a superclass name'')))
| Ogg2(Class(name, fpars, extends, (b1,b2,b3) )) ->
   pop(continuation);
   let v = top(tempstack) in
    pop(tempstack);
    eredita(rho, v, !currentheap);
    push(Ogg3(Class(name,
                fpars,
                extends,
                (b1,b2,b3) )),
         continuation);
    push(labelcom(b3), top(cstack));
    push(Rdecl(b2), top(cstack));
    push(labeldec(b1),top(cstack));
    pushlocalenv(b1,b2,rho);
    pushlocalstore(b1);
    ()
| Ogg3(Class(name, fpars, extends, (b1,b2,b3) )) ->
   pop(continuation);
```

```
        let r = (match applyenv(rho,name) with
                 | Classval(_, r1) -> r1
                 | _ -> failwith(``not a class name'')) in
        let lobj =
          (match applyenv(rho, ''this'') with
             | Dobject n -> n) in
        let newobj =
          localenv(rho, sigma, Dobject(lobj), fpars, r) in
          currentheap := allocateheap (!currentheap,
                                lobj, newobj);
          push(Object lobj, tempstack)
      | _ -> failwith(``impossible in semobj''))

let initstate(r, s) =
   svuota(cstack); svuota(tempvalstack); svuota(tempdvalstack);
   svuota(labelstack);
   match (r,s) with (Denv r1, Dstore s1) ->
    currentenv := r1; currentstore := s1

let loop () =
   while not(empty(cstack)) do
    while not(empty(top(cstack))) do
     let currconstr = top(top(cstack)) in
      (match currconstr with
        | Expr1(e) -> itsem()
        | Expr2(e) -> itsem()
        | Exprd1(e) -> itsemden()
        | Exprd2(e) -> itsemden()
        | Coml(cl) -> itsemcl()
        | Decl(l) -> itsemdecl()
        | Rdecl(l) -> itsemrdecl()
        | Ogg1(e) -> itsemobj()
        | Ogg2(e) -> itsemobj()
        | Ogg3(e) -> itsemobj()
        | _ -> failwith(``non legal construct in loop''))
    done;
    (match top(labelstack) with
     | Expr1(_) -> let valore = top(top(tempvalstack)) in
        pop(top(tempvalstack));
        pop(tempvalstack); push(valore,top(tempvalstack));
        popenv();popstore(); pop(tempdvalstack)
     | Exprd1(_) -> let valore = top(top(tempdvalstack)) in
        pop(top(tempdvalstack));
        pop(tempdvalstack); push(valore,top(tempdvalstack));
        popenv();popstore(); pop(tempvalstack)
     | Decl(_) -> pop(tempvalstack); pop(tempdvalstack)
     | Rdecl(_) -> pop(tempvalstack); pop(tempdvalstack)
     | Ogg1(_) -> let valore = top(top(tempvalstack)) in
        pop(top(tempvalstack));
```

```
        pop(tempvalstack); push(valore,top(tempvalstack));
        popenv();popstore(); pop(tempdvalstack)
      | Coml(_) -> popenv();popstore();
        pop(tempvalstack); pop(tempdvalstack)
      | _ -> failwith(``non legal label in loop''));
    pop(cstack);
    pop(labelstack)
  done

let sem (e,(r:  dval env), (s:  mval store), h) =
  initstate(r,s);
  currentheap := h;
  push(emptystack(tframesize(e),Novalue),tempvalstack);
  newframes(Expr1(e), r, s);
  loop();
  let valore= top(top(tempvalstack)) in
   pop(tempvalstack);
   let st = topstore() in
     (valore, st, !currentheap)

let semden (e,(r:  dval env), (s:  mval store), h) =
  initstate(r,s);
  currentheap := h;
  push(emptystack(tdframesize(e),Unbound),tempdvalstack);
  newframes(Exprd1(e), r, s);
  loop();
  let valore= top(top(tempdvalstack)) in
   pop(tempdvalstack);
   let st = topstore() in
     (valore, st, !currentheap)

let semcl (cl,(r:  dval env), (s:  mval store), h) =
  initstate(r,s);
  currentheap := h;
  newframes(labelcom(cl), r, s);
  loop();
  let st = topstore() in
    (st, !currentheap)

let semdv(dl, r, s, h) =
  initstate(r,s);
  currentheap := h;
  newframes(labeldec(dl), r, s);
  pushlocalenv(dl,[],Denv(!currentenv));
  pushlocalstore(dl);
  loop();
  let st = topstore() in
  let rt = topenv() in
    (rt, st, !currentheap)
```

```
let semdr(dl, r, s, h) =
   initstate(r,s);
   currentheap := h;
   newframes(Rdecl(dl), r, s);
   pushlocalenv([],dl,Denv(!currentenv));
   loop();
   let rt = topenv() in
     (rt, s, !currentheap)

let semdl((dl,rl), r, s, h) =
   initstate(r,s);
   currentheap := h;
   newframes(Rdecl(rl), r, s);
   push(labeldec(dl),top(cstack));
   pushlocalenv(dl,rl,Denv(!currentenv));
   pushlocalstore(dl);
   loop();
   let st = topstore() in
   let rt = topenv() in
     (rt, st, !currentheap)

let semc((c:  com), (r:dval env), (s:  mval store), h) =
   semcl ([c],r, s, h)

let semb ((l1, l2, l3), r, s, h) =
   initstate(r,s);
   currentheap := h;
   newframes(labelcom(l3), r, s);
   push(Rdecl(l2), top(cstack));
   push(labeldec(l1), top(cstack));
   pushlocalenv(l1,l2,Denv(!currentenv));
   pushlocalstore(l1);
   loop();
   currentenv := !currentenv + 1;
   currentstore := !currentstore + 1;
   (topstore(), !currentheap)

let semclasslist (cl, Denv(r)) =
   let rn = List.length cl in
   let ii = Array.create rn ‘‘dummy’’ in
   let dd = Array.create rn Unbound in
     push(ii, namestack);
     push(dd, dvalstack);
     push(Denv(r), slinkstack);
     currentenv:= lungh(namestack);
     let rr = Denv(!currentenv) in
     let rcl = ref(cl) in
       while not(!rcl = []) do
```

```
       let thisclass = List.hd !rcl in
        rcl := List.tl !rcl;
        match thisclass with
          | Class(nome,_,_,_) ->
             bind(rr,
                nome,
                makeclass(thisclass, rr))
     done;
     rr

let semprog ((cdl,b),r,s,h) = semb(b,semclasslist(cdl,r),s,h)
```

16 Gestione dinamica della memoria a heap (nel Linguaggio OO)

Contenuti del capitolo:

- Heap di oggetti, allocazione di oggetti con lista libera, restituzione di oggetti.
- Gestione della heap da parte del sistema: garbage collector.
- Altri meccanismi per gestire la heap.

16.1 Heap e gestione dinamica della memoria

```
type heap = obj array
type poiner = int
let newpoint = let count = ref(-1) in
   function () -> count := !count + 1; !count
```

Nell'implementazione corrente gli oggetti allocati sulla heap sono realmente permanenti perchè non esiste un modo per disallocarli.
Nella tradizionale gestione della memoria ad heap (come abbiamo visto per le liste) la heap non è gestita come un banale array sequenziale, ma attraverso una lista libera: le allocazioni sono fatte prendendo un puntatore dalla lista libera, ed esiste una operazione che permette di restituire un elemento alla lista libera.

16.2 La nuova heap

Adattiamo l'implementazione della heap già vista al caso in cui gli elemento da allocare siano oggetti.

```
let objects = (*1*)
   (Array.create heapsize ((Array.create 1 ‘‘dummy’’),
                     (Array.create 1 Unbound),
                     Denv(-1),
                     (Array.create 1 Undefined)) :  heap)

let nexts = Array.create heapsize (-1:  pointer) (*2*)

let next = ref((0:  pointer)) (*3*)

let emptyheap() = (*4*)
   let index = ref(0) in
    while !index < heapsize do
      Array.set nexts !index (!index + 1);
      Array.set marks !index false;
      index := !index + 1
    done;
    Array.set nexts (heapsize - 1) (-1);
    next := 0;
    svuota (markstack);
    objects
```

Commenti:

1. Definizione della heap.
2. Array utilizzato per gestire la lista libera.
3. Puntatore alla testa della lista libera.
4. Heap iniziale, in cui c'è solo lista libera.

Vediamo ora le operazioni sulla heap:

```
let applyheap ((x:  heap), (y:pointer)) = Array.get x y

let allocateheap ((x:heap), (i:pointer), (r:obj)) =
   Array.set x i r;
   next := Array.get nexts i;
   x

let disallocate (i:pointer) =
   let pre = !next in
   next := i;
   Array.set nexts i pre
```

Per il momento abbiamo definito un'operazione di disallocazione (`disallocate`), che porta con sè una serie di considerazioni riguardo alla rimozione degli oggetti e, quindi, al *garbage collector*.

16.3 Disallocazione e marcatura: garbage collector

Nel linguaggio didattico, come in Java ed OCAML, la disallocazione non è prevista come un'operazione a disposizione del programmatore, ma piuttosto un'operazione invocata dal sistema (implementazione) quando la lista libera diventa vuota e non permette di allocare un nuovo oggetto.
Quando questo avviene: gli oggetti che non sono più utilizzati vengono disallocati e lo spazio da loro occupato nella heap viene utilizzato per allocare nuovi oggetti.
Con questo procedimento gli oggetti continuano ad essere comunque *logicamente persistenti*, poichè vengono eventualmente distrutti solo quando non servono più.
Gli oggetti che "non servono più" vengono determinati attraverso un procedimento complesso (**marcatura**) il cui effetto è quello di "marcare" tutti gli oggetti che servono ancora.
Per effettuare la marcatura utilizziamo un terzo array parallelo a quello degli oggetti, che verrà settato a true in corrispondenza degli oggetti ancora utili:

```
let marks = Array.create heapsize false
```

Una volta effettuata la marcatura, tutti gli oggetti non marcati vengono disallocati, restituendoli alla lista libera:

```
let collect = function () ->
   let i = ref(0) in
```

```
    while !i < heapsize do
      (if Array.get marks !i
      then (Array.set marks !i false)
      else disallocate(!i));
      i := !i +1
    done
```

Adesso pendiamo in considerazione il problema di dover individuare e marcare tutti gli *oggetti attivi*, ovvero quegli oggetti raggiungibili a partire dalle strutture che realizzano la pila dei record di attivazione (eventualmente passando attraverso altri oggetti attivi).
Per poter applicare la marcatura è necessario visitare le strutture a grafo costituite da puntatori e radicate in strutture dati esterne alla heap stessa (ambiente, memoria, temporanei). Per visitare tale struttura è necessario disporre di una "pila per la marcatura", che chiamiamo `markstack`:

```
let markstacksize = 100;
let markstack = emptystack(markstacksize, (0:pointer))

let pushmarkstack (i:  pointer) =
   if lungh(markstack) = markstacksize
   then failwith(``markstack length has to be increased'')
   else push(i, markstack)
```

Introduciamo ora la procedura `markobject` che gestisce la visita della struttura di puntatori:

```
let markobject (i:  pointer) =
   if Array.get marks i then ()
   else
     (Array.set marks i true;
     let ob = Array.get objects i in
     let den = getden(ob) in
     let st = getst(ob) in
     let index = ref(0) in
       while !index < Array.length den do
         (match Array.get den !index with
            | Dobject j -> pushmarkstack(j)
            | _ -> ());
         index := !index + 1
       done;
       index := 0;
       while !index < Array.length st do
         (match Array.get st !index with
            | Mobject j -> pushmarkstack(j)
            | _ -> ());
         index := !index + 1
       done)
```

Non abbiamo ancora definito il modo in cui i vari puntatori (*punti di partenza* del garbage collector), provenienti da strutture dati diverse dalla heap, vengono inseriti all'interno di `markstack`. Prima di definire la procedura appropriata, consideraiamo quali sono le strutture dello stato che possono contenere puntatori alla heap: `tempvalstack`, `tempdvalstack`, `dvalstack` e `storestack`.
E' nelle pile appena individuate che andiamo a ricercare i puntatori (con prefisso `Object`, `Dobject` e `Mobject`), punti di partenza del garbage collector, attraverso l'operazione `startingpoints`:

```
let startingpoints() =
   let index1 = ref(0) in
   let index2 = ref(0) in
      (* dvalstack *)
     while !index1 <= lungh(dvalstack) do
      let adval = access(dvalstack, !index1) in
        index2 := 0;
        while !index2 < Array.length adval do
          (match Array.get adval !index2 with
            | Dobject j -> pushmarkstack(j)
            | _ -> ());
          index2 := !index2 + 1
        done;
        index1 := !index1 + 1
    done;
    index1 := 0;
     (* storestack *)
    while !index1 <= lungh(storestack) do
      let adval = access(storestack, !index1) in
        index2 := 0;
        while !index2 < Array.length adval do
          (match Array.get adval !index2 with
            | Mobject j -> pushmarkstack(j)
            | _ -> ());
          index2 := !index2 + 1
        done;
        index1 := !index1 + 1
    done;
    (* tempvalstack *)
    index1 := 0;
    while !index1 <= lungh(tempvalstack) do
      let tempstack = access(tempvalstack, !index1) in
        index2 := 0;
        while !index2 <= lungh(tempstack) do
          (match access(tempstack,!index2) with
            | Object j -> pushmarkstack(j)
            | _ -> ());
          index2 := !index2 + 1
        done;
        index1 := !index1 + 1
```

```
    done;
     (* tempdvalstack *)
    index1 := 0;
    while !index1 <= lungh(tempdvalstack) do
     let tempstack = access(tempdvalstack, !index1) in
      index2 := 0;
      while !index2 <= lungh(tempstack) do
       (match access(tempstack,!index2) with
         | Dobject j -> pushmarkstack(j)
         | _ -> ());
       index2 := !index2 + 1
      done;
      index1 := !index1 + 1
    done
```

Vediamo infine le procedure usate per allocare nuovi oggetti, eventualmente marcando gli oggetti inutili:

```
let mark() =
  startingpoints();
  while not(empty(markstack)) do
   let current = top(markstack) in
    pop(markstack);
    markobject(current)
  done

let newpoint() =
  if not(!next = -1)
  then (!next)
  else
   (mark(); collect();
   if !next = -1
   then
    failwith(``the heap size is not sufficient'')
   else !next)
```

16.4 Condizioni per poter realizzare un garbage collector

Per poter realizzare un garbage collector è necessario sapere:

- Per ogni *struttura dello stato* (pila dei record di attivazione): dove possono esserci puntatori alla heap, per poter realizzare `startingpoints`.
- Per ogni *struttura della heap*: dove possono esserci puntatori ad altri elementi della heap, per poter realizzare `markobject`.

16.5 Altri costrutti ed altre tecniche

La realizzazione di una gestione automatica della heap attraverso un garbage collector è stata, prima di Java, limitata ad alcuni linguaggi funzionali e logici sfruttando

la semplice struttura dei record di attivazione e l'uniformità delle strutture allocate sulla heap (s-espressioni, termini).
Linguaggi come PASCAL, C o C++ hanno scelto di affidare al programmatore la restituzione di strutture ed oggetti alla memoria libera, fornendo costrutti del tipo `free` o `dispose`. In questo modo però le strutture non sono più permanenti e si rischia di creare *garbage* o, ben più grave, di creare *dangling references*.
Si hanno **dangling references** quando il programmatore restituisce alla lista libera una struttura che ha ancora dei cammini di accesso, con il risultato di poter incappare in stati di errore, se si cerca di seguire un "puntatore a nulla", o di inconsistenza, se la cella della heap restituita viene riutilizzata per allocare nuove strutture. Nei casi in cui una parte del contenuto della struttura (ad esempio liste ed s-espressioni in LISP) è utilizzata per rappresentare la lista libera, un accesso ad una dangling reference potrebbe portare addirittura a distruggere senza rimedio parte della lista libera.
Quelli dovuti ad una scorretta disallocazione da parte del programmatore sono tutti errori molto difficili da individuare, anche perchè non necessariamente ripetibili (l'effetto dipende dalle dimensioni ella heap e persino da possibili esecuzioni pregresse). E' proprio per questo che Java è tanto migliore di C++.

16.6 Altre gestioni da parte del sistema

Esistono altri algoritmi di garbage collection che apportano dei miglioramenti all'algoritmo di marcatura "naif" descritto finora.
Segnaliamo soltatnto due problemi "storici" dell'algoritmo descritto in precedenza:

- La marcatura richiede una pila tanto più grande quanto maggiore è il numero di strutture marcate, e quindi quanto meno è utile il garbage collector.
  Questo problema è stato affrontato con l'algoritmo classico di Schorr & Waite che utilizza la struttura a grafo stessa per "ricordare quello che ancora c'è da visitare".
- La marcatura parte quando si esaurisce la lista libera e si cerca di allocare una nuova struttura, sospendendo la computazione in maniera visibile e fastidiosa (basti pensare alle applicazioni interattive).
  Questo problema viene affrontato dai garbage collectors incrementali.

Il secondo problema descritto può essere risolto con una tecnica alternativa al garbage collector, conosciuta come **contatori di riferimenti**.
Con i contatori di riferimenti, ogni struttura allocata nella heap ha associato un contatore che conta il numero di cammini di accesso alla sruttura. In questo modo, pur appesantendo tutte le operazioni che manipolano i puntatori (bisogna incrementare/decrementare i contatori), è possibile restituire una struttura alla lista libera quando il contatore associato diventa 0.
Con questa tecnica il costo della gestione viene distribuito nel tempo, al prezzo di una maggiore occupazione di memoria. La tecnica dei contatori di riferimenti, inoltre ha il limite di non poter funzionare con strutture circolari.

17 Interpretazione astratta

Il capitolo introduce e descrive i concetti fondamentali relativi alla tecnica dell'*interpretazione astratta*, che permette di gestire in modo sistematico astrazione ed approssimazione.

17.1 Interpretazione astratta: astrazione ed approssimazione

Astrazione ed approssimazione sono due concetti rilevanti in numerose aree dell'informatica: per analizzare il comportamento di sistemi complessi, e per rendere l'analisi effettiva dal punto di vista computazionale.
Uno strumento utile per gestire astrazione ed approssimazione è rappresentato dalla tecnica dell'**interpretazione astratta**, nata 30 anni fa per descrivere (e dimostrare corrette) le analisi statiche.
Applicata con successo a molti paradigmi e ad altri compiti "basati sulla semantica" (ad esempio la verifica), oggi l'interpretazione astratta è vista come una tecnica generale per ragionare sulle semantiche a differenti livelli di astrazione.
Spesso l'interpretazione astratta viene usata nel seguente modo:

> Data una *semantica* ed un *algoritmo di analisi* sviluppato con tecniche ad-hoc (quelli dei compilatori con tecniche data flow, quelli dei teorici dei tipi con sistemi di tipi, ecc.), la teoria è usata per dimostrare che l'algoritmo è *corretto*, cioè che i suoi risultati sono un'approssimazione corretta della proprietà da analizzare (osservata sulla semantica).

Da un punto di vista sistematico qusto si traduce in:

- Prendere una semantica, senza alcun tipo di limitazione sullo stile di definizione, che assegna un significato ai programmi su un opportuno dominio concreto (**dominio delle computazioni concrete**).
- Prendere un dominio astratto (**dominio delle computazioni astratte**) che modella solo alcune delle proprietà delle computazioni concrete, tralasciando (astraendo da) le altre proprietà.
- Usare la teoria per derivare una semantica astratta che ci permette di "eseguire" il programma sul dominio astratto per calcolare il suo significato astratto, cioè la proprietà modellata.

17.2 I domini concreto ed astratto

I domini *completo* ed *astratto* sono due reticoli completi, in cui gli ordinamenti parziali riflettono la "precisione" (il più piccolo è anche il migliore, in quanto più preciso).

- **Dominio concreto**: $(P(C), \subseteq, \varnothing, C, \cup, \cap)$
  ha la struttura di insieme delle parti (vedremo in seguito perchè)
- **Dominio atratto**: $(A, \leq, bottom, top, lub, glb)$
  ogni valore astratto è una descrizione di *un insieme di* valori concreti.

Esempio
Prendiamo in considerazione un esempio in cui vogliamo modellare il segno, astraendo da un dominio concreto che contiene tutti gli interi.

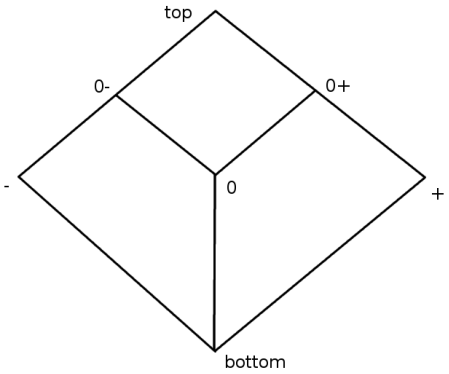

Figura 4: Il dominio astratto *Sign*

I domini:

- Concreto (insiseme di interi): $(P(Z), \subseteq, \varnothing, C, \cup, \cap)$
- Astratto: $(Sign, \leq, bottom, top, lub, glb)$

17.3 Concretizzazione ed astrazione

Dominio concreto: $(P(C), \subseteq, \varnothing, C, \cup, \cap)$
Dominio atratto: $(A, \leq, bottom, top, lub, glb)$

Il significato dei valori astratti è definito da una **funzione di concretizzazione**:

$$\gamma : A \to P(C)$$

dove, $\forall a \in A$, $\gamma(a)$ è l'insieme dei valori concreti descritti da a (per questo il domino concreto deve avere la struttura di insieme di parti).
La funzione di concretizzazione deve essere *monotona*:

$$\forall a_1, a_2 \in A, a_1 \leq a_2 \text{ implica } \gamma(a_1) \subseteq \gamma(a_2)$$

La concretizzazione *preserva la precisione* relativa.
Quando gni elemento di $P(C)$ ha un'unica "migliore" (più precisa) descrizione in A[24], è possibile definire una **funzione di astrazione**:

$$\alpha : P(C) \to A$$

dove, $\forall c \in P(C)$, $\alpha(c)$ è la migliore descrizione astratta di c.
La funzione di astrazione deve essere *monotona*:

$$\forall c_1, c_2 \in P(C), c_1 \subseteq c_2 \text{ implica } \alpha(c_1) \leq \alpha(c_2)$$

L'astrazione *preserva la precisione* relativa.

[24]Questo è possibile solo se A è una famiglia di Moore, chiuso sotto *glb*.

17.4 Connessioni di Galois

Date:

$$\gamma : A \to P(C) \text{ (cocretizzazione) e } \alpha : P(C) \to A \text{ (astrazione)}$$

Secondo la **connessione di Galois**:

$$\forall c \in P(C), c \subseteq \gamma(\alpha(c))$$

Ci può essere quindi perdita di informazioni passando da $P(C)$ ad A, e

$$\forall a \in A, \alpha(\gamma(a)) \leq a$$

Quando in questa formula vale l'uguaglianza parliamo di **intersezione di Galois**.

Con riferimento all'esempio introdotto in precedenza abbiamo che:

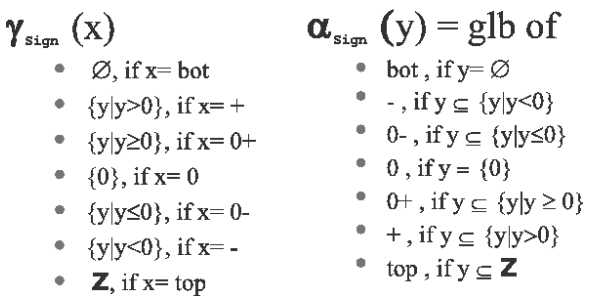

Figura 5: Concretizzazione ed astrazione su *Sign*

17.5 Semantica concreta e semantica collecting

La **semantica concreta** è definita come minimo (o massimo) punto fisso[25] di una funzione di valutazione semantica concreta F definita sul dominio C.
F è definita in termini di operatori semantici primitivi f_i su C. La funzione di valutazione semantica astratta viene ottenuta rimpiazzando in F ogni operatore semantico concreto con un corrispondente operatore astratto.
Dato che l'effettivo dominio concreto è $P(C)$ dobbiamo prima estendere la semantica concreta $lfp\ F$ alla **semantica collecting** definita su $P(C)$.
L'esetensione di $lfp\ F$ all'insieme delle parti, in modo da ottenere la semantica collecting, è soltanto un'operazione concettuale

$$semantica\ collecting\ =\ \{lfp\ F\}$$

Non abbiamo infatti bisogno di definire una nuova funzione di valutazione semantica collecting su $P(C)$, ma ci basta ragionare in termini di estensioni alle parti di tutti gli operatori primitivi, nella progettazione degli operatori astratti e nella dimostrazione delle loro proprietà.
E' esattamente lo stesso anche per gli operatori concreti e le loro versioni collecting.

[25] Questo non vuol dire che lo stile di definizione debba essere necessariamente denotazionale.

17.6 Correttezza degli operatori astratti

Per ciascun f_i (concreto) dobbiamo fornire un corrispondente f_i^α (astratto) definito su A.
f_i^α deve essere localmente corretto, cioè

$$\forall x_1, ..., x_n \in P(C).\ f_i(x_1, ..., x_n) \subseteq \gamma(f_i^\alpha(\alpha(x_1), ..., \alpha(x_n)))$$

che indica come un passo di calcolo concreto sia più preciso della concretizzazione del "corrispondente" passo di calcolo astratto.
Notiamo che, comunque, quello appena descritto è un requisito molto debole, soddisfatto, per esempio, da un operatore che calcola sempre il peggior valore astratto (il top del reticolo).
La precisione è la cosa importante nella progettazione di operazioni astratte.

17.7 Operazioni astratte: ottimalità e completezza

Correttezza:

$$\forall x_1, ..., x_n \in P(C).\ f_i(x_1, ..., x_n) \subseteq \gamma(f_i^\alpha(\alpha(x_1), ..., \alpha(x_n)))$$

Ottimalità:

$$\forall y_1, ..., y_n \in A.\ f_i^\alpha(y_1, ..., y_n) \subseteq \alpha(f_i(\gamma(y_1), ..., \gamma(y_n)))$$

l'ottimalità individua il più preciso operatore astratto f_i^α corretto rispetto ad f_i, che rappresenta un limite teorico ed una base per la progettazione, piuttosto che una definizione da implementare.

Completezza (precisione assoluta):

$$\forall x_1, ..., x_n \in P(C).\ f_i(x_1, ..., x_n) = \gamma(f_i^\alpha(\alpha(x_1), ..., \alpha(x_n)))$$

non c'è perdita di informazione, l'astrazione del passo di calcolo concreto è esattamente il risultato del corrispondente passo di calcolo astratto.

Esempio:
Riprendiamo l'esempio introdotto in precedenza, definendo delle operazioni astratte su *Sign*:

Times$^{\mathbf{Sign}}$

	bot	**-**	**0-**	**0**	**0+**	**+**	**top**
bot	bot	bot	bot	bot	bot	bot	bot
-	bot	+	0+	0	0-	-	top
0-	bot	0+	0+	0	0-	0-	top
0	bot	0	0	0	0	0	0
0+	bot	0-	0-	0	0+	0+	top
+	bot	-	0-	0	0+	+	top
top	bot	top	top	0	top	top	top

Plus$^{\mathbf{Sign}}$

	bot	**-**	**0-**	**0**	**0+**	**+**	**top**
bot	bot	bot	bot	bot	bot	bot	bot
-	bot	-	-	-	top	top	top
0-	bot	-	0-	0-	top	top	top
0	bot	-	0-	0	0+	+	top
0+	bot	top	top	0+	0+	+	top
+	bot	top	top	+	+	+	top
top	bot	top	top	top	top	top	top

`Times` e `Plus` sono gli operatori classici estesi a $P(Z)$.
Entrambi sono *ottimi* (e quindi corretti), `Times` è anche *completo* (non c'è approssimazione), mentre `Plus` è necessariamente incompleto.

17.8 Correttezza globale

La composizione di operatori astratti localmente corretti è localemente corretta rispetto alla composizione degli operatori concreti.
La composizione non conserva però l'ottimalità: la composizione di operatori ottimi può essere meno precisa della versione astratta ottimale della composizione.
Se F^α (funzione di valutazione semantica astratta) è ottenuta rimpiazzando in F ogni oeratore semantico concreto con un corrispondente operatore astratto localmente corretto, allora anche F^α è localemnte corretta:

$$\forall x \in P(C).\ F(x) \subseteq \gamma(F^\alpha(\alpha(x)))$$

la correttezza locale implica la **correttezza globale**, cioè viene preservata anche dal calcolo del punto fisso:

$$lfp\ F \subseteq \gamma(lfp\ F^\alpha) \quad \text{ed} \quad \alpha(lfp\ F) \leq lfp\ F^\alpha$$

la semantica astratta è quindi meno precisa dell'astrazione di quella concreta.
L'approssimazione dipende dal fatto che gli operatori astratti sono incompleti e che, in generale, ci sono più cammini di esecuzione nella semantica astratta poichè lo stato astratto non ha abbastanza informazioni per permettere scelte deterministiche (condizionali, pattern matching, ecc.). In questi casi l'insieme degli stati astratti risultanti viene trasformato in un unico stato astratto, effettuando un'operazione di *lub astratto*.

17.9 Perchè calcolare $lfp\ F^\alpha$

Come già detto:

$$\alpha(lfp\ F) \leq lfp\ F^\alpha$$

quindi perchè calcolare $lfp\ F^\alpha$ e non piuttosto $\alpha(lfp\ F)$?
Il motivo risiede nel fatto che $\alpha(lfp\ F)$ non può essere calcolato in un numero finito di passi, mentre $lfp\ F^\alpha$ può essere calcolato in un numero finito di passi se il dominio astratto è finito o almeno noetheriano.
Sfruttando il fatto che non ci sono catene infinite crescenti, riusciamo a sfruttare il calcolo del punto fisso per l'analisi statica dei programmi, dove il calcolo del punto fisso deve terminare. La maggior parte delle proprietà considerate nell'analisi statica sono indecidibili, quindi accettiamo una perdita di precisione (approsimazione corretta) per rendere l'analisi fattibile.

17.10 Applicazioni

Analisi statica = Calcolo effettivo della semantica astratta

Se il dominio astratto è noetheriano e gli operatori hanno una complessità accettabile.
Quando il dominio è non noetheriano oppure se il calcolo del punto fisso è troppo complesso si usano gli *operatori di widening*, che calcolano in modo effettivo un'approssimazione (superiore) di $lfp\ F^{\alpha}$.

17.11 Riepilogo

Breve riepilogo schematico riguardo a quanto detto sull'interpretazione astratta:

- Dominio concreto: $(P(C), \subseteq, \varnothing, C, \cup, \cap)$
- Dominio astratto: $(A, \leq, bottom, top, lub, glb)$
- Funzione di concretizzazione: $\gamma : A \to P(C)$ (monotona)
- Funzione di astrazione: $\alpha : P(C) \to A$ (monotona)
- Connessione di Galois: $\begin{array}{c} \forall c \in P(C), c \subseteq \gamma(\alpha(c)) \\ \forall a \in A, \alpha(\gamma(a)) \leq a \end{array}$
- Correttezza locale:

$$\forall f_i \exists f_i^{\alpha} | \forall x_1, ..., x_n \in P(C).\ f_i(x1, ..., x_n) \subseteq \gamma(f_i^{\alpha}(\alpha(x_1), ..., \alpha(x_n)))$$

Le scelte critiche sono: il *dominio astratto* per modellare la proprietà, e gli *operatori astratti* corretti (possibilmente ottimali).

18 Valutazione parziale

Contenuti del capitolo:

- Valutazione parziale.
- Specializzazione di interpreti: prima e seconda proiezione di Futamura.
- Caratteristiche dell'interprete da specializzare.
- Cosa fa lo specializzatore di interpreti.

18.1 Trasormazione sistematica di un interprete in un compilatore

La **valutazione parziale** è una tecnica che permette di generare automaticamente un compilatore a partire da un interprete.
In effetti l'approccio è l'equivalente "semantico" di quello usato per i generatori di analizzatori sintattici a partire dalla specifica della grammatica del linguaggio, usando però la semantica, denotazionale o operazionale, come vera specifica.
Oltre alla sua utilità pratica, lo studio della valutaizone parziale consente di comprendere meglio la differenza tra interpretazione e compilazione.

18.2 Valutazione parziale: il problema

Fissati un programma P ed una n-upla di valori $v_1, ..., v_n$ per i suoi "primi" n dati in ingresso, vogliamo determinare un programma P', specializzazione di P per $v_1, ..., v_n$, che si comporta *esattamente* come P per ogni valore degli altri dati, quando i suoi "primi" n dati sono $v_1, ..., v_n$, ed è *più efficiente* di P.

Esempio
Supponiamo di avere

```
f(x,y) = if x=0 then g(x) else h(y)
```

vogliamo specializzare `f` per il valore `x=2`

```
f2(y) = f(2,y) = h(y)
```

Otteniamo `f2` "valutando" il corpo di `f` quanto possibile a tempo di specializzazione, riuscendo a rimpiazzare il condizionale con il solo ramo `else`, visto che la guardia risulta essere sempre falsa.
`f2` è più efficiente di `f` per qualunque valore di `y`.

18.3 Teorema *s-m-n* di Kleene

Il teorema *s-m-n* di Kleene garantisce che, dati una funzione

$$f = \lambda x_1, ..., x_n.e$$

ed una k-upla di valori

$$a_1, ..., a_k$$

è possibile calcolare la funzione

$$f' = \lambda x_{k+1}, ..., x_n.e'$$

tale che

$$\forall x_{k+1}, ..., x_n.f(a_1, ..., a_k, x_{k+1}, ..., x_n) = f'(x_{k+1}, ..., x_n)$$

18.4 Come si calcola effettivamente la soluzione

Per calcolare la soluzione il valutatore parziale esegue simbolicamente il programma valutando una volta per tutte ed eliminando dal codice le istruzioni per le quali si hanno sufficienti informazioni perchè siano eseguite, e lasciando le altre nel "codice residuo" (eventalmente semplificandolo usando le regole di una *semantica algebrica*).
Valutare un volta per tutte, eliminando o lasciando nel codice, si applica in maniera particolare alle strutture dati, ottenendo moglioramenti notevoli *eliminando i condizionali*, *sfogliando i cicli* e *rimpiazzando le procedure* (non ricorsive) con le loro definizioni.

18.5 Il valutatore parziale del linguaggio M

Consideriamo `peval` come un valutatore parziale in grado di specializzare programmi scritti nel linguaggio `M`. `peval` è molto simile ad in interprete di `M`.
Un generico programma `P` di `M` ha i propri dati raggruppati in due tuple: la prima (`D`) è quella dei dati forniti nella specializzazione, la seconda (`X`) è la tupla dei dati non conosciuti.
Abbiamo:

$\texttt{peval} : \texttt{Prog}_{\texttt{M}} * \texttt{dati} -> \texttt{Prog}_{\texttt{M}}$

$\texttt{peval}(\texttt{P}, \texttt{D}) = \texttt{P}'$ tale che $\forall \texttt{X}.\ \texttt{P}'(\texttt{X}) = \texttt{P}(\texttt{D}, \texttt{X})$

Quelle appena descritte sono le equazioni che danno le proprietà del valutatore parziale.

18.6 Specializziamo un interprete

Con riferimanto a quanto detto nel paragrafo precedente, specializziamo con `peval` il programma `int`, ovvero un interprete[26] del linguaggio `L` scritto nel linguaggio `M`, che ha come dati: un programma (`prog`) scritto in `L`, ed uno stato iniziale `s`.
Applicando le equazioni della valutazione parziale abbiamo:

$\texttt{peval}(\texttt{int}, \texttt{prog}) = \texttt{int}'$ tale che $\forall \texttt{s}.\ \texttt{int}'(\texttt{s}) = \texttt{int}(\texttt{prog}, \texttt{s})$

Cosa abbiamo ottenuto con `int'`?

- Dato che `int` è un interprete, `int(prog,s)` fornisce lo stato finale per ogni stato iniziale `s` ottenuto dalla semantica di `prog`.

[26] Possiamo pensare di trattare in questo modo l'interprete del nostro linguaggio didattico (`L`), scritto in ML (`M`=ML).

- `int'` fa la stessa cosa ed è un programma di `M` (come `int`).
- `int'` è la traduzione di `prog` da `L` a `M`.

Quello che abbiamo fatto corrisponde quindi (più o meno)ad un passo di compilazione, in cui `int'` rappresenta il codice compilato.

18.7 Prima proiezione di Futamura: il codice compilato

Avendo:

- `peval`: valutatore parziale di `M`.
- `int`: interprete del linguaggio `L` scritto nel linguaggio `M`.
- `prog`: programma di `L`.

ed applicando:

$$\texttt{peval}(\texttt{int}, \texttt{prog}) = \texttt{int}' \text{ tale che } \forall \texttt{s}.\ \texttt{int}'(\texttt{s}) = \texttt{int}(\texttt{prog}, \texttt{s})$$

`int'`, risultato della *compilazione* di `prog` sulla macchina astratta `M`:

- Può essere *eseguito direttamente* su `M` senza avere bisogno dell'interprete.
- E' funzionalmente *equivalente* a `prog`.
- L'esecuzione diretta di `int'` dovrebbe essere *più efficiente* dell'esecuzione interpretativa di `prog`.

18.8 Seconda proiezione di Futamura: il compilatore

Avendo:

- `peval`: valutatore parziale di `M`, scritto in `M` (*autoapplicabile*).
- `int`: interprete del linguaggio `L` scritto nel linguaggio `M`.
- `prog`: programma di `L`.
- `int'`: risultato della *compilazione* di `prog` sulla macchina astratta `M`, come descritto nel paragrafo precedente.

Applicando:

$\texttt{peval}(\texttt{peval}, \texttt{int}) = \texttt{peval}'$
$\forall \texttt{prog}.\ \texttt{peval}'(\texttt{prog}) = \texttt{peval}(\texttt{int}, \texttt{prog}) = \texttt{int}'$

Abbiamo ottenuto `peval'`, *compilatore* da `L` a `M`.

18.9 Generazione del codice ed aggiustamenti sull'interprete

Per capire come l'interprete (`int`) si semplifica nella specializzazione, bisogna guardare la sua definizione.
In generale, può essere corstuito, valutato ed eliminato tutto ciò che dipende solo da `prog` (che è statico) e quindi si può agire su

- Strutture dati destinate a contenere i costrutti sintattici (`cstack`).

- Il ciclo di decodifica, determinato appunto dai costrutti sintattici.
- Eventuali analizzatori statici (inferenza dei tipi ecc.)
- In caso di scoping statico, le pile di array di nomi ed i link statici.

Le strutture dati che restano finiscono nel supporto a tempo di esecuzione.
Quanto detto implica anche la possibilità di effettuare modifiche all'interprete in modo da favorire i vantaggi in compilazione.
In particolare:

- Inserire tutti gli eventuali analizzatori (nomi, tipi, dimensionamenti) in modo da rilevare gli errori una volta per tutte a tempo di compilazione, dove possibile rimuovere parte del codice (tipi) e precalcolare parte delle informazioni.
- Reintrodurre uno stile "denotazionale" nel trattamento delle funzioni, in modo da specializzare il loro codice una sola volta, al momento della definizione.
- Volendo una "compilazione separata" può addirittura convenire reintrodurre la ricorsione (al posto delle chiamate di `newframes`).

18.10 Le scelte sulla specializzazione: strutture dati e funzioni

Vediamo, nel caso del frammento funzionale, quali sono le cose maggiormente interessate dal processo di specializzazione.
La scelta principale per la specializzazione riguarda, come già detto, le **strutture dati** ed in particolare le strutture dati che compongono la pila dei record di attivazione. In concreto:

- Strutture che possono essere *costruite* ed *eliminate*: `cstack` (pila di pile di espressioni etichettate) e `namestack` (pila di array di identificatori).
- Strutture che possono essere costruite (in versione statica) ma che devono rimanere nel rts: `slinkstack` (pila di puntatori ad ambienti).
- Strutture che devono essere lasciate nel rts: `tempvalstack`, pila di pile di valori esprimibli, `evalstack` (pila di array di valori denotati) e `tagstack` (pila di etichette per la retention).

Se lo scoping fosse dinamico non potremmo eliminare `namestack`.
Nel linguaggio *imperativo* è tutto uguale e non si può eliminare `storestack`, mentre nel linguaggio *ad oggetti* si può costruire ed eliminare anche la parte dei nomi degli ambienti permanenti degli oggetti.
Per quanto rigurda le **funzioni**, invece, è possibile usare l'*unfolding*, ovvero il rimpiazzamento di una chiamata con il corpo e la valutazione parziale del corpo della chiamata stessa.
Possiamo applicare l'unfolding in corrispondenza delle `newframes` e delle operazioni relative all'ambiente, visto che sono noti `namestack` e `slinkstack`.
Il ciclo di interpretazione (i due `while` annidati) può infine essere sfogliato completamente ed eeliminato, perchè controllato da informazione unicamente sintattica (il contenuto di `cstack`).

18.11 Esempio di simulazione: quando si trova un `Den x`

Quando trova un `Den ide` viene chiamata l'operazione `applyenv`:

```
let applyenv ((x:  eval env), (y:  ide)) =
  let n = ref(x) in
  let den = ref(Unbound) in
  while !n > -1 do
    let lenv = access(namestack, !n) in
    let nl = Array.length lenv in
    let index = ref(0) in
    while !index < nl do
      if Array.get lenv !index = y then
      (den := Array.get (access(evalstack, !n)) !index;
      index := nl)
      else index := index + 1
    done;
    if not(!den = Unbound) then n:=-1 else n:=access(slinkstack, !n)
  done;
  !den
```

L'interprete specializzato conosce `namestack` e `slinkstack` (nella versione statica, sufficiente alla traduzione dei nomi) e può quindi sostituire la traduzione di un nome con il corrispondente codice per l'accesso "diretto" ad `evalstack`

Esempio
Un riferimento che verrebbe tradotto in `(2,1)`, diventa, nel codice prodotto dall'interprete specializzato:

```
Array.get (access(evalstack, access(slinkstack, access(slinkstack,
                        !currentenv)))) 1
```

Ovvero, considerando la generica coppia *(m,n)*: m "salti" sulla catena statica (a partire dall'ambiente corrente), un accesso ad `evalstack` (all'indice ottenuto su `slinkstack`) per prendere il giusto array, infine un accesso all'array alla posizone n-esima.

19 Strutture dati nel supporto a runtime

Panoramica delle strutture dati presenti nel supporto a runtime dei principali linguaggi di programmazione.

19.1 Entità presenti quando un programma va in esecuzione

Cosa è, in generale, presente nel support a runtime quando un programma è in esecuzione:

- Programmi utente compilati.
- Routines del supporto: interprete, I/O, librerie, gestione delle altre strutture, garbage collector, ecc.
- Strutture dati per gestire le attivazioni (funzioni, procedure, classi): ambiente, memoria, temporanei, punti di ritorno.

19.2 FORTRAN

Caratteristiche del linguaggio:

- Permette la compilazione separata dei sottoprogrammi.
- Non permette la ricorsione.
- Ambiente e memoria locali statici.
- Non esiste ambiente non locale.

La gestione è *completamente statica*, il compilatore crea una "unità completa" che contiene: codice compilato, punto di ritorno, l'area dati locali (ambiente e memoria), temporanei.
Linker e loader risolvono i riferimenti globali ed allocano memoria per tutte le unità necessarie e per le routines del supporto a tempo di esecuzione.

19.3 ALGOL

Caratteristiche del linguaggio:

- Il programma è un unico blocco, con blocchi e procedure annidati e senza la possibilità di compilare separatamente i sottoprogrammi.
- Permette la ricorsione.
- Ambiente e momoria locali dinamici (anche statici se dichiarati tali).
- Scoping statico.
- Non permette puntatori.

Semplice gestione dinamica basata sulla pila dei record di attivazione.
Il compilatore genera: codice per l'intero programma, compreso quello per la generazione dei record di attivazione a tempo di esecuzione, le costanti e l'area dati locale statica (ambiente e memoria).
Un record di attivazione contiene: punto di ritorno (puntatore di catena dinamica), puntatore di catena statica, ambiente e momoria locali senza nomi, temporanei.

19.4 PASCAL

Caratteristiche del linguaggio:

- Il programma è un unico blocco, con blocchi e procedure annidati e senza la possibilità di compilare separatamente i sottoprogrammi.
- Permette la ricorsione.
- Ambiente e momoria locali dinamici (anche statici se dichiarati tali).
- Scoping statico.
- Permette puntatori

In questo caso si utilizza la pila dei record di attivazione più una heap.
Il compilatore genera: il codice per l'intero programma, compreso quello per la generazione dei record di attivazione, le costanti, l'area dati locali statica (ambiente e memoria).
Un record di attivazione contiene: punto di ritorno (puntatore di catena dinamica), puntatore di catena statica, ambiente e momoria locali senza nomi, temporanei.
La heap non è gestita attraverso un garbage collector.

19.5 C

Caratteristiche del linguaggio:

- Il programma è composto da moduli compilati separatamente.
- Permette la ricorsione.
- Ambiente e momoria locali dinamici (anche statici se dichiarati tali).
- Scoping statico.
- Permette puntatori.

Ogni record di attivazione contiene: punto di ritorno (puntatore di catena dinamica), puntatore di catena statica, ambiente e momoria locali senza nomi, temporanei.
Anche in questo caso la heap non ha garbage collector.
In sostanza C si differenzia da PASCAL solo per la compilazione separata.

19.6 Java

Caratteristiche del linguaggio:

- Ilprogramma consiste di un insieme di classi, compilate separatamente.
- Permette la ricorsione.
- Ambiente e momoria locali dinamici (anche statici se dichiarati tali).
- Scoping statico (per i blocchi).
- Oggetti e puntatori.

Il supporto a runtime di Java utilizza una pila di record di attivaizone più una heap per gli oggetti.
Il compilatore genera per ogni classe: il codice compilato incluso quello per la generazione a tempo di esecuzione di oggetti e record di attivazione dei metodi; l'area dati statica (ambiente e memoria relativi alle dichiarazioni `static`). Ogni record di attivazione contiene: punto di ritorno (puntatore di catena dinamica), puntatore di catena statica, ambiente e momoria locali senza nomi, temporanei.
In questo caso la heap viene gestita attraverso un garbage collector.

19.7 ML

Caratteristiche del linguaggio:

- Il programma è un insieme di definizioni di funzioni compilabili separatamente.
- Permette la ricorsione.
- Ambiente locale dinamico.
- Scoping statico.
- Valori di ordine superiore.

ML usa una pila per i record di attivazione più una heap per termini e liste.
Il compilatore genera: il codice per ogni funzione, compreso quello per la generazione a tempo di esecuzione dei record di attivazione.
Ogni record di attivazione contiene: punto di ritorno (puntatore di catena dinamica), puntatore di catena statica, ambiente e momoria locali senza nomi, temporanei.
Può essere necessario efffettuare la retention di record di attivazione.
La heap viene gestita con un garbage collector.

19.8 LISP

Caratteristiche del linguaggio:

- Il programma è un insieme di definizioni di funzioni compilabili separatamente.
- Permette la ricorsione.
- Ambiente locale dinamico.
- Scoping dinamico.
- Valori di ordine superiore.

Le definizioni di funzione (ed il loro codice compilato) sono associate al nome di funzione in una tabella degli atomi, in una sorta di ambiente globale.
Anche in questo caso si usa una pila di record di attivazione più una heap per le s-espressioni.
Un record di attivazione dovrebbe contenere: punto di ritorno (puntatore di catena dinamica), ambiente locale con nomi, temporanei.
La pila di ambienti locali (a-list) è a sua volta rappresentata come s-espressione e risiede nella heap.
Il record di attivazione contiene un puntatore alla a-list.
La heap è gestita attraverso un garbage collector.