PLP - Modulo II
Semantica Denotazionale e Concetti di Linguaggi di Programmazione
Draft

prof. Marco Bellia

March 6, 2012

Prefazione. Queste note sono integrative alla prima parte del modulo II sui paradigmi. Il loro scopo è richiamare i concetti di base dei linguaggi e diffusamente presenti in essi, con il fine di padroneggiarli quando, nella seconda parte, useremo in modo sistematico linguaggi dei tre paradigmi scelti. Queste note sono destinate a crescere con la quantità di argomenti che ci sembrerà utile trattare e con l'accuratezza con cui li tratteremo, nel corso. Gli esercizi sono quasi sempre presentati all'interno delle varie sezioni e sotto sezioni. Lo scopo non è quello di disturbare la lettura. Al contrario gli esercizi sono parte dalle considerazioni della (sotto) sezione e sono posti come attività che deve condotta dal lettore per completare la presentazione di quanto li precede. Alcuni sono di soluzione immediata, altri possono richiedere più tempo e talora, la conoscenza di argomenti già trattati nella triennale. In tutti i casi, lo svolgimento degli esercizi è obbligatorio prima di procedere con la lettura (o, detto in altro modo, è inutile procedere con la lettura se non si sono risolti gli esercizi che precedono). Marco Bellia

1 Le Strutture di Base

1.1 Domini Sintattici

Introducono i termini (strutture) del linguaggio e possiamo considerarli alberi di sintassi astratta con i loro bravi costruttori che, applicati ad alberi argomento, generano un nuovo albero avente tali alberi come figli.

Table 1

Domini Sintattici

`D`	::=	`Proc I() C;` | `D D` | ...	(*Dichiarazioni*)
`C`	::=	{`D C`} | `I := E` | `Call I ()` | ...	(*Comandi*)
`E`	::=	`I` | `LV` | ...	(*Espressioni*)
`LV`	::=	...	(*Literals*)

Esercizio 1 *(a) Si elenchino i costruttori, indicandone la segnatura, utilizzati nel dominio* `C` *in aggiunta al costruttore binario* {_ _} *con segnatura* $\texttt{D} \times \texttt{C} \rightarrow \texttt{C}$ *(ovvero,*

indicheremo il tutto con la scrittura: $\{\_\ \_\} : \mathtt{D} \times \mathtt{C} \to \mathtt{C}$*).*
(b) Qual'e il costruttore della sequenza di dichiarazione. □

Esercizio 2 *Sotto quali condizioni il linguaggio in Table1 è un linguaggio Turing Completo (abbr. TC)?: Si dica quali caratteristiche devono o non devono avere i costrutti in tabella affichè il linguaggio sia o non sia TC.* □

1.2 Domini Semantici

Introducono i valori che utilizzeremo per definire le funzioni semantiche.

Notazione 1 (Uso dei Simboli) *Useremo i nomi dei domini (e i nomi posti a sinistra di '::='), eventualmente indiciati, come nomi di metavariabili che variano sul dominio. In tal modo, ad esempio, quando incontreremo il simbolo* `D` *sappiamo che esso indica, a seconda del contesto, l'intero dominio delle dichiarazioni oppure una dichiarazione tra quelle possibili. In tabella 2, nell'anticipare le funzioni* `bind`, `find`, `empty` *ricorriamo all'uso di meta termini la cui notazione e significato associato sono presentati in Notazione 3*

Table 2

Domini Semantici

$\mathtt{Env}, \rho, \delta$	$\equiv$	$\mathtt{I} \to \mathtt{Den}$	(*Ambienti*)
		operazioni di `Env` :	
		$\mathtt{bind} : \mathtt{I} \times \mathtt{Den} \times \mathtt{Env} \to \mathtt{Env}$	
		$\mathtt{bind}(\mathtt{I}, \mathtt{d}, \rho) \equiv \lambda \mathtt{x}.\ \mathtt{if}((\mathtt{x}\ eq\ \mathtt{I}), \mathtt{d}, \rho(\mathtt{x}))$	
		$\mathtt{find} : \mathtt{I} \times \mathtt{Env} \to \mathtt{Den}$	
		$\mathtt{find}(\mathtt{I}, \rho) \equiv \rho(\mathtt{I})$	
		$\mathtt{empty} : \mathtt{Env}$	
		$\mathtt{empty} \equiv \lambda \mathtt{x}.\ \mathtt{x}$	
$\mathtt{Store}, \mathtt{s}$	$\equiv$	...	(*Memoria*)
		operazioni di `Store` :	
		$\mathtt{upd} : \mathtt{Loc} \times \mathtt{Mem} \times \mathtt{Store} \to \mathtt{Store}$	
		$\mathtt{look} : \mathtt{Loc} \times \mathtt{Store} \to \mathtt{Mem}$	
`State`	::=	...	(*Stato*)

Domini Semantici Ausiliari

$\mathtt{Val}, \mathtt{v}$	::=	...	(*Valori* : *Esprimibili*)
$\mathtt{Den}, \mathtt{d}$	::=	...	(*Valori* : *Denotabili*)
$\mathtt{Mem}, \mathtt{d}$	::=	...	(*Valori* : *Memorizzabili*)
$\mathtt{Loc}, \mathtt{l}$	::=	...	(*Locazioni*)
`Input`	::=	...	(*I/O* : *Input*)
`Output`	::=	...	(*I/O* : *Output*)

Esercizio 3 *Per alcuni linguaggi il dominio* `State` *coincide con il dominio* `Store`*. Per tali linguaggi si dice che le operazioni di I/O non fanno parte del linguaggio ma il linguaggio ne permette l'uso attraverso la propria interfaccia con codice nativo di sistema (*language native interface*). Quando le operazioni di I/O fanno invece, parte del linguaggio quale potrebbe essere una definizione per* `State`*.* □

Esercizio 4 *Osservando le definizioni in Table 2 e in Table 3 si dica cosa sono i valori* denotabili*, i valori* memorizzabili*, i valori* esprimibili*. Si giustifichi la risposta richiamando le definizioni, nelle due tabelle, a conferma di quanto asserito.* □

Esercizio 5 *Si consideri il seguente frammento di programma C*

```
#define max 100
typedef enum colore {bianco, nero, giallo, rosso} colore;
typedef struct list{colore *head; struct list *next;}list;
void printColore(colore *j){...}
{
list *quad, *temp; colore x;
goto addr;
addr:   x = bianco;
quad = NULL;
temp = (struct list *)malloc(sizeof(struct list));
temp->head = &x;  temp->next = quad; quad = temp;  colore p = *quad->head;
if(quad == NULL) printf("puntatore nullo");
else printColore(&(*quad->head));
}}
```

Per ciascuno dei domini di valori (denotabili, esprimibili, memorizzabili) si indichino almeno 2 strutture, se esistono, che coinvolgono anche separatamente, valori del dominio: specificando quali valori sono coinvolti e perchè. □

Esercizio 6 *Si consideri il seguente frammento di programma Pascal*

```
label addr;
const max = 100;
type colore = (bianco, nero, verde, giallo, rosso); { nessun commento }
        colorePtr = ^colore;
        listPtr = ^list;
       list = record head : colorePtr; next : listPtr;  end;
var  quad, temp : listPtr;   x : colorePtr;
procedure printColore(var j: colorePtr); begin ... end
begin
       goto addr;
       addr : new(x);
       x^ := bianco; quad := nil;
       new(temp); temp^.head := x; temp^.next := quad; quad := temp;
       if (quad = nil) then Writeln("puntatore nullo")
                       else printColore(quad^.head);
     end.
```

Per ciascuno dei domini di valori (denotabili, esprimibili, memorizzabili) si indichino almeno 2 strutture, se esistono, che coinvolgono anche separatamente, valori del dominio: specificando quali valori sono coinvolti e perchè. □

Esercizio 7 *Si elenchino quali sono i valori denotabili di Java fornendo i costruttori (nomi e segnatura) della loro sintassi astratta (si ricordi che la scelta dei nomi è inessenziale per le caratteristiche della sintassi astratta risultante).* □

Esercizio 8 *(a) Confrontando il programma C e il programma Pascal degli esercizi sopra, si dica perchè possiamo asserire che i puntatori sono valori denotabili in C mentre lo stesso non si può affermare per il Pascal: In particolare si indichi il costrutto del programma C che contiene tale valore e si discuta sull'impossibilità in Pascal di esprimere la stessa struttura. (b) Si dica poi perchè, contrariamente a quanto avviene in generale, in C le locazioni di variabile sono puntatori.* □

Notazione 2 (Abbreviazione per l'Ambiente) *Negli esempi e negli esercizi, talora useremo delle abbreviazioni per rendere più leggibili (lunghe) composizioni di operazioni. Alla base di queste è la seguente notazione* $[\mathtt{d}/\mathtt{I}]\rho$ *che sta per* $\mathtt{bind}(\mathtt{I},\mathtt{d},\rho)$*. Quindi:* $[\mathtt{d}_1/\mathtt{I}_1][\mathtt{d}_2/\mathtt{I}_2][\mathtt{d}_3/\mathtt{I}_3][\mathtt{d}_4/\mathtt{I}_4]\rho$ *è un abbreviazione per* $\mathtt{bind}(\mathtt{I}_1,\mathtt{d}_1,\mathtt{bind}(\mathtt{I}_2,\mathtt{d}_2,\mathtt{bind}(\mathtt{I}_3,\mathtt{d}_3,\mathtt{bind}(\mathtt{I}_4,\mathtt{d}_4,\rho))))$.

1.3 Funzioni Semantiche

Introducono le strutture con cui daremo significato alle strutture del linguaggio. In questo caso useremo tre funzioni con arità e segnatura come indicato in Table 3.

Table 3

Funzioni Semantiche

$\mathcal{D} : \mathtt{D} \to \mathtt{Env} \to \mathtt{Env}$ $\quad (Significato\ delle\ Dichiarazioni)$

$\mathcal{M} : \mathtt{C} \to \mathtt{Env} \to \mathtt{State} \to \mathtt{State}$ $\quad (Significato\ dei\ Comandi)$

$\mathcal{E} : \mathtt{E} \to \mathtt{Env} \to \mathtt{State} \to \mathtt{Val}$ $\quad (Significato\ delle\ Espressioni)$

Domini Ausiliari

$\mathtt{VL} ::= \mathtt{Int} + \mathtt{Char}$ $\quad (Valori\ Literals:\ unione\ disgiunta)$
$\mathtt{Int} ::= ...$ $\quad (Valori\ Literals\ per\ interi)$
$\mathtt{Char} ::= ...$ $\quad (Valori\ Literals\ per\ caratteri)$

Funzioni Ausiliarie

$\mathtt{N} : \mathtt{LV} \to \mathtt{Int}$ $\quad (iniettiva, i.e.\ costruttore - -suriettiva??)$
$\mathtt{C} : \mathtt{LV} \to \mathtt{Char}$ $\quad (iniettiva, i.e.\ costruttore)$
$\mathtt{V} : \mathtt{Int} \cup \mathtt{Char} \to \mathtt{VL}$ $\quad (iniettiva, i.e.\ costruttore)$
$\mathtt{NtoVL} : \mathtt{Int} \to \mathtt{VL}$ $\qquad$ $\mathtt{CtoVL} : \mathtt{Char} \to \mathtt{VL}$
$\mathtt{NtoVL} = \lambda \mathtt{x}.\mathtt{V}(\mathtt{x})$ $\qquad$ $\mathtt{CtoVL} = \lambda \mathtt{x}.\mathtt{V}(\mathtt{x})$
$\mathtt{VLtoN} : \mathtt{VL} \to \mathtt{Int}_\perp$ $\qquad$ $\mathtt{VLtoC} : \mathtt{VL} \to \mathtt{Char}$
$\mathtt{VLtoN} = \lambda \mathtt{x}.\mathtt{if}((\mathtt{x} \in \mathtt{Int}), \mathtt{N}(u), \perp_{\mathtt{Int}})$
$\in \mathtt{Int} : \mathtt{VL} \to \mathtt{TruthV}$ $\qquad$ $\in \mathtt{Char} : \mathtt{VL} \to \mathtt{TruthV}$
$\in \mathtt{Int} = \lambda \mathtt{x}.\mathtt{x}\ eq\ \mathtt{V}(\mathtt{N}(u))$
$\mathtt{MV} : \mathtt{Mem} \to \mathtt{Val}$
$\mathtt{IntoVal} : \mathtt{LV} \to \mathtt{Val}$

Esercizio 9 *In Table3 si vede come per lavorare, senza ambiguità, sui valori di domini correlati tra loro, occorrono iniezioni, proiezioni, predicati. Riflettendo su tutto ciò: (a) Si dica quali sono le funzioni di iniezione, quali quelle di proiezione, e quali i predicati definiti, e a cosa servono; (b) Si dica in cosa differiscono i domini* $\mathtt{Int}+\mathtt{Char}$ *e* $\mathtt{Int}\cup\mathtt{Char}$

entrambi usati nelle definizioni in Table3; (c) Si dica in cosa differisca `Int` *da* $\texttt{Int}_\perp$*; (d) Si completino le definizioni di* `VLtoC` *e di* $\in$ `Char`*. (e) Si dia una definizione di* `MV` *per un dominio* `Mem` *e un dominio* `Val` *di propria scelta.* □

Esercizio 10 *(a) Si elenchino le classi di literals del linguaggio C e si dia una definizione del dominio* `LV` *sempre per il linguaggio C. (b) Si faccia la stessa cosa per il linguaggio Java; (c) Si faccia la stessa cosa per ciascun ulteriore linguaggio noto.* □

Esercizio 11 *(a) Si mostri la definizione del dominio* `Val` *del linguaggio C limitatamente ai valori coinvolti nell'esercizio 5; (b) Si mostri una definizione per* `IntoVal` *del linguaggio C.* □

Esercizio 12 *(a) Si faccia la stessa cosa dell'esercizio precedente per* `Val` *di Pascal facendo riferimento al programma dell'esercizio 6; (b) Si mostri una definizione per* `IntoVal` *del linguaggio Pascal.* □

2 Semantica: Linguaggi con Naming ma Senza Blocchi o con solo Blocchi In-Line

Definiamo in Table 4 le funzioni semantiche nel caso di Linguaggi Imperativi con strutture estremamente semplici.

Notazione 3 *Per esprimere le funzioni semantiche useremo la* λ*notazione, pertanto* $\lambda\ \texttt{x}_1...\texttt{x}_n\ .\ \texttt{F}$ *esprime la funzione nelle variabili* $\texttt{x}_1...\texttt{x}_n$ *che, applicata a* $\texttt{F}_1...\texttt{F}_n$*, calcola il valore espresso da* $[\texttt{F}_1...\texttt{F}_n/\texttt{x}_1...\texttt{x}_n]\texttt{F}$*, ovvero il valore espresso da* `F` *dove le occorrenze delle variabili* $\texttt{x}_1...\texttt{x}_n$ *sono simultaneamente sostituite dai corrispondenti argomenti* $\texttt{F}_1...\texttt{F}_n$*. Ovviamente,* $\texttt{F}, \texttt{F}_1, ..., \texttt{F}_n$ *sono tutti termini con cui esprimiamo i significati definiti dalla semantica.*
Usiamo la composizione di funzione $\circ$ *nella seguente forma:* $\texttt{f} \circ \texttt{g}\ (\texttt{F}) \ \equiv\ \texttt{g}(\texttt{f}(\texttt{F}))$

Table4 : Linguaggi Imperativi : Solo Naming e Blocchi In – Line

Funzioni Semantiche

$\mathcal{M}[\![\mathtt{C}]\!]_\rho : \mathtt{State} \to \mathtt{State}$ (*Significato dei Dichiarazioni*)

$\mathcal{M}[\![\mathtt{I} := \mathtt{E}]\!]_\rho = \lambda \mathtt{s}.\ \mathtt{upd}(\mathtt{find}(\mathtt{I}, \rho), \mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s}), \mathtt{s})$
$\mathcal{M}[\![\mathtt{C}_1; \mathtt{C}_2]\!]_\rho = \mathcal{M}[\![\mathtt{C}_1]\!]_\rho \circ \mathcal{M}[\![\mathtt{C}_2]\!]_\rho$

$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{State} \to \mathtt{Val}$ (*Significato delle Espressioni*)

$\mathcal{E}[\![\mathtt{I}]\!]_\rho = \lambda \mathtt{s}.\ \mathtt{MV}(\mathtt{look}(\mathtt{find}(\mathtt{I}, \rho), \mathtt{s}))$
$\mathcal{E}[\![\mathtt{LV}]\!]_\rho = \lambda \mathtt{s}.\ \mathtt{IntoVal}(\mathtt{LV})$

Domini Semantici

$\mathtt{State} ::= \mathtt{Store}$ (*Stato*)

Domini Ausiliari

$\mathtt{Den} ::= \mathtt{Loc}$

Funzioni Ausiliarie

$\mathtt{VM} : \mathtt{Val} \to \mathtt{Mem}$ (*iniettiva, i.e. costruttore*)

Esercizio 13 *Nel termine* $\mathtt{upd}(\mathtt{find}(\mathtt{I}, \rho), \mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s}), \mathtt{s})$ *è usato il termine* $\mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s})$ *come definente un valore memorizzabile. È corretta tale assunzione? Si dica sotto quali ipotesi ciò è corretto e perchè taluni preferiscono ricorrere al termine* $\mathtt{VM}(\mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s}))$*. Si utilizzi il caso come esempio per confrontare il dominio unione con quello di somma disgiunta.* □

Esercizio 14 *Si dia una definizione di* `Store` *e delle funzioni* `upd` *e* `look` *introdotte in Table 2 sul dominio* `Store`*. Allo scopo si scelga come dominio* `Loc` *un intervallo dei naturali, ad esempio [0..K], per arbitrario K≥0. Ci si limiti ad un modello di memoria ad allocazione statica, che in aggiunta a fornire il valore di ogni locazione utilizzata, dia indicazione sulla memoria ancora allocabile.* □

3 Semantica: Linguaggi a Blocchi e/o Blocchi Procedura

Definiamo le funzioni semantiche nel caso di Linguaggi Imperativi con strutture a blocchi includenti blocchi procedura/funzioni, oltre a blocchi in-line. Queste estendono le definizioni di Table 4, mantenendole integralmente. Distinguiamo i due tipi di Scoping.

3.1 Semantica: Linguaggi con Scoping Statico

Definiamo il comportamento della dichiarazione di procedura (e funzione) e dell'invocazione rispetto all'ambiente quando la portata degli identificatori definiti è statica.

Table5 – Linguaggi Imperativi : Scoping Statico

Funzioni Semantiche

$\mathcal{D}[\![\texttt{D}]\!]_\rho : Env$ $\quad$ (*Significato delle Dichiarazioni*)

$\mathcal{D}[\![\texttt{Proc I() C}]\!]_\rho = \texttt{bind}(\texttt{I}, \mathcal{M}[\![\texttt{C}]\!]_\rho, \rho)$

$\mathcal{M}[\![\texttt{C}]\!]_\rho : \texttt{State} \to \texttt{State}$ $\quad$ (*Significato dei Dichiarazioni*)

$\mathcal{M}[\![\texttt{Call I()}]\!]_\rho = \texttt{find}(\texttt{I}, \rho)$

Domini Ausiliari

$\texttt{Den} ::= \texttt{Loc} + \texttt{ProcFun}$ $\quad$ (*Unione disgiunta*)

$\texttt{ProcFun} ::= \texttt{State} \to \texttt{State}$ $\quad$ (*Valori Procedure*)

Funzioni Ausiliarie

$\texttt{Q} : (\texttt{State} \to \texttt{State}) \to \texttt{ProcFun}$ $\quad$ (*iniettiva, i.e. costruttore*)

Esercizio 15 *In Table5 occorre un termine t che andrebbe più correttamente rimpiazzato con il termine* $\texttt{Q}(t)$*: Si dica quale e perchè.* □

Esercizio 16 *(a) Cosa succede se il programma contiene un'invocazione ad una procedura* `I` *non dichiarata, o dichiarata ma non avente tale invocazione nel proprio scope? Si giustifuchi la risposta avvalendosi della semantica per mostrare quanto asserito.*
(b) L'analisi statica è in grado di controllare che tutti gli identificatori usati siano stati dichiarati con tipo adatto all'uso previsto. Tuttavia, se ciò non fosse, come andrebbe modificata la semantica dell'invocazione: allo scopo si utilizzi il dominio $\texttt{Den}_\perp \equiv \texttt{Den} + \perp_D$*, dove* $\perp_D \equiv (\texttt{Y}f.\lambda\texttt{x}.f(\texttt{x}))(d)$ *con* $d \in \texttt{Den}$ □

3.1.1 Implementazione: Chiusure

La semantica delle procedure o funzioni con scoping statico, introduce una prima fondamentale struttura per l'implementazione delle macchine astratte. Questa struttura si chiama chiusura (Landin 1966?) ed è una coppia $(\texttt{C}, \rho)$ contenente un codice `C`(che potrebbe essere anche un'espressione, a seconda della struttura del linguaggio considerati), ed un ambiente. L'ambiente contiene i bindings (legami) per tutti gli identificatori usati (ma non definiti) in `C`. Da un punto di vista strettamente implementativo, La chiusura implementatta con una struttura contenente un puntatore, $p_{\texttt{C}}$ al codice e un puntatore, p_ρ ad un frame, implementante l'ambiente ρ. Ma allora, come è fatto il frame implementante l'ambiente, δ, definito dalla semantica $\texttt{bind}(\texttt{I}, \mathcal{M}[\![\texttt{C}]\!]_\rho, \rho)$ della dichiarazione di procedura in Table5? Ovvero, che cosa contiene δ in corrispondenza del nome `I` della procedura? Nell"implementazione descritta sopra, $\delta(\texttt{I})$ contiene esattamente una coppia di puntatori. Ciò implica che l'implementazione dell'invocazione

non può limitarsi ad estrarre la denotazione di `I`, come espresso dalla semantica denotazionale $\texttt{find}(\texttt{I}, \rho)$, e applicarla allo stato corrente dell'invocazione. In effetti, utilizzando la coppia$(p_{\texttt{C}}, p_\rho$ è creato un nuovo AR, aggiunto in testa allo stack degli Activation Records. Questo AR ha i componenti: (1) *cd* che punta all'AR in cui è avvenuta l'invocazione (ancora presente sullo stack), lo indicheremo con AR invocante, (2) *cs* che punta all'AR avente come frame un istanza del frame indirizzato da p_ρ, (3) *irit* l'indirizzo del primo statement del codice indirizzato da $p_{\texttt{C}}$, (4) *iris* l'indirizzo nella memoria temporanea (stack valori intermedi) dove inserire l'eventuale valore calcolato dall'invocazione (se previsto perchè funzione), (5) fr che contiene il frame della locali, inclusi i parametri (in accordo [numero, posizione, tipo] a quanto definito dalla semantica $\mathcal{M}[\![\texttt{C}]\!]_\rho$ e specificato dall'implementazione $p_{\texttt{C}}$) (6) *sr* che contiene lo stack, vuoto, utilizzato dalle primitive del linguaggio, per memorizzare i risultati intermedi (principalmente durante la valutazione di espressioni contenute negli statements del codice indirizzato da $p_{\texttt{C}}$)

3.2 Semantica: Linguaggi con Scoping Dinamico

Definiamo il comportamento della dichiarazione di procedura (e funzione) e dell'invocazione rispetto all'ambiente quando la portata degli identificatori definiti è dinamica.

Table6 – Linguaggi Imperativi : Scoping Dinamico

Funzioni Semantiche

$\mathcal{D}[\![\texttt{D}]\!]_\rho : Env$ (*Significato delle Dichiarazioni*)

$\mathcal{D}[\![\texttt{Proc I() C}]\!]_\rho = \texttt{bind}(\texttt{I}, \lambda\delta\,.\,\mathcal{M}[\![\texttt{C}]\!]_\delta, \rho)$

$\mathcal{M}[\![\texttt{C}]\!]_\rho : \texttt{State} \to \texttt{State}$ (*Significato dei Dichiarazioni*)

$\mathcal{M}[\![\texttt{Call I()}]\!]_\rho = \texttt{find}(\texttt{I}, \rho)(\rho)$

Ovviamente l'implementazione dello scoping dinamico non richiede l'uso di chiusure da associare come denotazione del nome della procedura: È sufficiente il puntatore $p_{\texttt{C}}$. Nondimeno, l'implementazione dell'invocazione richiede di creare un AR, come lasciato per esercizio.

Esercizio 17 *Si descriva la struttura, in particolare il contenuto di ogni componente, dell'AR creato all'invocazione di una procedura/funzione con scoping dinamico. Allo scopo si rifrasi, opportunamente, quanto descritto per il caso con scoping statico in paragrafo 3.1.1* □

Esercizio 18 *Si consideri il seguente frammento di programma:*

```
{int x = 7;
  Proc A(){x = 2;};
  {int x = 10;
    Call A();
```

Si mostri la struttura degli Activation Record (AR) durante l'esecuzione del frammento nell'ipotesi (1) il linguaggio utilizza scoping statico; (2) il linguaggio utilizza scoping dinamico. Si giustifichino gli AR introdotti facendo riferimento alla semantica data. □

3.3 Blocchi: Dichiarazioni Sequenziali, Parallele (simmetriche, mutuamente ricorsive o di punto fisso), Miste

Definiamo lo scope di una dichiarazione all'interno del blocco in cui è dichiarata. Sia `I`un identificatore dichiarato in un blocco (indifferentemente, In-line o procedura) il suo scope contiene l'intero blocco? Tre possibili risposte:

- Sequenziale: NO - solo quanto nel blocco segue la sua dichiarazione;
- Parallela: SI - anche tutto quello che precede la sua dichiarazione;
- Mista: NO - dipende da cosa stiamo denotando (ad es. no per variabili, si per procedure, si per tipi astratti).

Finora abbiamo omesso di dare significato alla composizione di dichiarazione. È arrivato il momento di farlo. Potremmo farlo attribuendo 3 semantiche differenti per le differenti 3 risposte date dai vari linguaggi. Invece, introduciamo un nuovo costrutto per la dichiarazione che può essere utilizzato tanto per la dichiarazione parallela quanto per quella mista.

Table7 – Linguaggi : Dichiarazione Sequenziale

Domini Sintattici

$\texttt{D} ::= \texttt{Proc I() C;} \mid \texttt{D D} \mid \texttt{Const I} = \texttt{LV}...$ (*Dichiarazioni*)

Funzioni Semantiche

$\mathcal{D}[\![\texttt{D}]\!]_\rho : Env$ (*Significato delle Dichiarazioni*)

$\mathcal{D}[\![\texttt{D}_1\ \texttt{D}_2]\!]_\rho = \mathcal{D}[\![\texttt{D}_2]\!]_{\mathcal{D}[\![\texttt{D}_1]\!]_\rho}$
$\mathcal{D}[\![\texttt{Const I} = \texttt{LV}]\!]_\rho = \texttt{bind}(\texttt{I}, \texttt{IntoVal}(\texttt{LV}), \rho)$

Domini Ausiliari

$\texttt{Den} ::= \texttt{Loc} + \texttt{ProcFun} + \texttt{LV}$ (*unione disgiunta*)

Esercizio 19 *Si mostri come devono essere modificati i domini e come devono essere definite le funzioni semantiche per il costrutto di naming e invocazione di valori funzione come sotto:*

`Function int A (){if (x==0) then return B () else return 5;}`

Si considerino i due casi di scoping ma si tralasci la funzione semantica del costrutto `return`.

Esercizio 20 *(a) Si applichi la definizione alla dichiarazione:*

```
procedure printColore();
    begin
        Const bianco = "bianco";
        Const nero = "nero";
    end;
Const verde = "verde";
```

(b) Si discuta l'ambiente ρ ottenuto dopo la dichiarazione dell'identificatore `verde` *, in particolare si dica quanto vale ρ(bianco), ρ("bianco"), ρ(printColore).* □

Esercizio 21 *(a) Si applichi la definizione alla dichiarazione:*

```
procedure A(); begin ... Call B(); ... end;
procedure B(); begin ... Call A(); ... end;
```

(b) Si discuta (b1) l'ambiente ρ_A ottenuto prima dell'invocazione `Call B()`*, in particolare si dica quanto vale $\rho(A)$, $\rho(B)$; (b2) l'ambiente ρ_B ottenuto prima dell'invocazione* `Call A()`*, in particolare si dica quanto vale $\rho(A)$, $\rho(B)$.* □

Notazione 4 *Indichiamo l'operatore di punto fisso con* `Y`*. Pertanto, dato un funzionale* $\mathtt{H} \equiv \lambda\ \mathtt{f.F}$ *nella variabile* `f`*, ovvero* `f` *è una variabile che assume valori sul dominio delle funzioni calcolabili,* `Y f.H` *esprime la più piccola (i.e. meno definita) funzione calcolabile che rende identica la seguente equazione:* $\mathtt{g} = \mathtt{H}(\mathtt{g})$*. Un'abbreviazione molto diffusa con punti fissi di funzionali è quella di contrarre l'operatore* `Y`*con il qualificatore λ limitatamente alla variabile funzionale, così da scrivere* `Y f.F` *invece di* `Y f`λ `f.F`*. Ovviamente, solo la seconda è una forma con senso la prima la usiamo solo come abbreviazione. Anche noi faremo così e questo può essere notato nella tabella sulla semantica delle* Dichiarazioni Mutuamente Ricorsive

Osservazione 1 (Calcolo del punto fisso) *Come sappiamo tale funzione può essere ottenuta come limite di un processo di approssimazioni finite descritto dalla formula di Tarski introdotta nel suo teorema sul punto fisso di equazioni tra funzioni:* $\mathtt{Y\ f.H} = \bigcup_{i\in\mathbb{N}} \mathtt{H}^i_\perp$*. La formula qui espressa in forma non dissimile da quella utilizzata nel corso di compilatori per punti fissi di equazioni tra linguaggi. In particolare, l'approssimazione* $\mathtt{H}^{i+1}_\perp \equiv \mathtt{H}(\mathtt{H}^i_\perp)$*, mentre* $\mathtt{H}^0_\perp \equiv \perp$ *(dove $\perp$, ovviamente, esprime la funzione ovunque indefinita).*

Esempio 1 *Sia* `if` *l'usuale operatore condizionale a due vie,* `=`*,* `*`*,* `-` *gli usuali, rispettivamente, predicato di uguaglianza, operatore prodotto, operatore sottrazione, tutti su interi e tutti infissi, ed infine 0 ed 1 i neutri di somma e prodotto. Allora:*

$$\mathtt{H} \equiv \lambda\ \mathtt{f}.\lambda\ \mathtt{x.if}(\mathtt{x}=0, 1, \mathtt{x}*\mathtt{f}(\mathtt{x}-1))$$

è un funzionale che esprime funzioni diverse al variare della variabile `f` *sul dominio delle funzioni. Ad esempio provate ad applicarla alla funzione identit,* $\mathtt{I} \equiv \lambda\ \mathtt{y.\ y}$*, oppure alla funzione successore,* $\mathtt{succ} \equiv \lambda\ \mathtt{y.\ y}+1$*: Dite la funzione espressa da* $\mathtt{H}(\mathtt{I})$*; Dite la funzione espressa da* $\mathtt{H}(\mathtt{succ})$*.*
Torniamo al funzionale `H` *e calcoliamo le prime approssimazioni del suo minimo punto fisso, ovvero di quella funzione* $\mathtt{g} \equiv \mathtt{Y\ f.H}$ *che come abbiamo detto è tale che* $\mathtt{g} = \mathtt{H}(\mathtt{g})$*.*

$$\begin{aligned}
\mathtt{H}^0_\perp &\equiv \perp \\
\mathtt{H}^1_\perp &\equiv \lambda\ \mathtt{x.if}(\mathtt{x}=0, 1, \perp) \\
\mathtt{H}^2_\perp &\equiv \lambda\ \mathtt{x.if}(\mathtt{x}=0, 1, \mathtt{x}*\mathtt{H}^1_\perp(\mathtt{x}-1)) \\
&= \lambda\ \mathtt{x.if}(\mathtt{x}=0, 1, \mathtt{x}*\mathtt{if}((\mathtt{x}-1)=0, 1, \perp)) \\
&= \lambda\ \mathtt{x.if}(\mathtt{x}=0, 1, \mathtt{if}((\mathtt{x}-1)=0, \mathtt{x}*1, \mathtt{x}*\perp)) \\
&= \lambda\ \mathtt{x.if}(\mathtt{x}=0, 1, \mathtt{if}(\mathtt{x}=1, 1*1, \perp)) \\
&= \lambda\ \mathtt{x.if}(\mathtt{x}=0, 1, \mathtt{if}(\mathtt{x}=1, 1, \perp)) \\
\mathtt{H}^3_\perp &\equiv \lambda\ \mathtt{x.if}(\mathtt{x}=0, 1, \mathtt{if}(\mathtt{x}=1, 1, \mathtt{if}(\mathtt{x}=2, 2, \perp)))
\end{aligned}$$

passo induttivo:
$\mathtt{H}^{n+1}_\perp \equiv \lambda\ \mathtt{x.if}(\mathtt{x}=0, 0!, \mathtt{if}(\mathtt{x}=1, 1!, \mathtt{if}(\mathtt{x}=2, 2!, ...\mathtt{if}(\mathtt{x}=n, n!, \perp)))...)$ □

Osservazione 2 (Trasparenza referenziale) *Nel calcolo di* $\mathtt{H}^2_\perp$ *(e, in generale in questo tipo di calcolo algebrico) abbiamo fatto ricorso a varie forme di semplificazioni quali:*

- *Rimpiazzamento di* $(\mathtt{x}-1)=0$ *con* $\mathtt{x}=1$.
- *Rimpiazzamento di* $\mathtt{x}*\mathtt{if}((\mathtt{x}-1)=0,1,\perp)$ *con* $\mathtt{if}((\mathtt{x}-1)=0,\mathtt{x}*1,\mathtt{x}*\perp)$

Queste semplificazioni sono lecite perchè nel calcolo con cui stiamo esprimendo la semantica denotazionale vale la proprietà nota come Trasparenza Referenziale *che afferma che il significato di un'espressione dipende unicamente dall'espressione stessa e non dal contesto in cui essa può occorrere: Pertanto possiamo sempre rimpiazzare un'espressione con un'altra espressione di stesso valore senza che alterare il significato della forma in cui tale espressione compare. Questa proprietà è presente anche in alcuni linguaggi di programmazione quali i* Linguaggi Funzionali Puri *(i.e. Haskell, FOL).* □

Esercizio 22 *In che senso il minimo punto fisso,* g, *del funzionale* H *in Esempio 1 è l'unione delle sue approssimazioni* $\bigcup_{i\in\mathbb{N}}\mathtt{H}^i_\perp$ *piuttosto che limite di esse? Ovvero perchè volendo il valore* $\mathtt{g}(n)$ *invece di usare tale limite consideriamo un'opportuna approssimazione quale* $\mathtt{H}^{n+1}_\perp$*, nel caso dell'esempio, e diciamo* $\mathtt{g}(n)=\mathtt{H}^{n+1}_\perp(n)$ □

A questo punto siamo in grado di estendere il linguaggio con un costrutto per la dichiarazione parallela o di punto fisso, e di esprimere precisamente il significato relativo. Questo è mostrato nella tabella sotto.

Table8 – Linguaggi : Dichiarazioni Mutuamente Ricorsive

Domini Sintattici

$\mathtt{D} ::= \ldots \mid \mathtt{Mut\ D_1\ D_2\ Ally} \mid \ldots$ $\quad$ (*Dichiarazioni*)

Funzioni Semantiche

$\mathcal{D}[\![\mathtt{D}]\!]_\rho : Env$ $\quad$ (*Significato delle Dichiarazioni*)

$\mathcal{D}[\![\mathtt{Mut\ D_1\ D_2\ Ally}]\!]_\rho = \mathtt{Y}\ \delta.(\mathcal{D}[\![\mathtt{D_1}]\!]_\delta \circ \mathcal{D}[\![\mathtt{D_2}]\!]_\delta \circ \rho)$

Esempio 2 *Siano* A *e* B *due identificatori. Consideriamo il seguente frammento di programma:*

```
...
{...
 Mut
   Proc A() Call B();
   Proc B() Call A();
 Ally
...
```

Applichiamo alla sua dichiarazione la semantica definita per il costrutto Mut_Ally*. Otteniamo la seguente funzione:*

$\mathtt{g} \equiv \mathtt{Y}\ \delta.\ \mathtt{bind}(\mathtt{A}, \mathcal{M}[\![\mathtt{Call\ B()}]\!]_\delta, \delta) \circ \mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\mathtt{Call\ A()}]\!]_\delta, \delta) \circ \rho$

Calcoliamo le prime tre approssimazioni. Quindi, consideriamo il funzionale:

$\mathtt{H} \equiv \lambda\ \delta.\mathtt{bind}(\mathtt{A}, \mathcal{M}[\![\mathtt{Call\ B()}]\!]_\delta, \delta) \circ \mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\mathtt{Call\ A()}]\!]_\delta, \delta) \circ \rho$

otteniamo:

$\mathtt{H}^0_\perp \equiv \perp$

$\mathtt{H}^1_\perp \equiv \mathtt{bind}(\mathtt{A}, \mathcal{M}[\![\mathtt{Call\ B()}]\!]_\perp, \perp) \circ \mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\mathtt{Call\ A()}]\!]_\perp, \perp) \circ \rho$

$\mathtt{H}^2_\perp \equiv \mathtt{bind}(\mathtt{A}, \mathcal{M}[\![\mathtt{Call\ B()}]\!]_{\mathtt{H}^1_\perp}, \perp^1) \circ \mathtt{bind}(\mathtt{B}, \mathcal{M}[\![\mathtt{Call\ A()}]\!]_{\mathtt{H}^1_\perp}, \perp^1) \circ \rho$

Sostituiamo e vediamo cosa otteniamo. □

Esercizio 23 *Si spieghi perchnell'esempio dato sopra invece, il punto fisso è* $\mathtt{H}^\omega_\perp$ *e non coincide con nessuna sua approssimazione finita.*

Esercizio 24 *Si consideri il seguente frammento di programma, P, di un linguaggio con scope statico:*

```
...
{...
  Mut
    Function in A (){if (x==0) then return B () else return 5;}
    Function in B (){if (x!=0) then return A () else return 7;}
  Ally
...
```

(a) Si applichi a P, quanto visto sopra. Allo scopo, si elenchino prima, le assunzioni, che si ritengono pi adatte, sulla semantica di `Function` *e il significato degli operatori usati in P. (b) Si dica sotto quali ipotesi il punto fisso del costrutto* `Mut_Ally` *di P può essere ottenuto con un numero di approssimazioni finito. Si giustifichi ciò anche in considerazione del fatto che nell'esempio dato sopra invece, il punto fisso non coincide con nessuna sua approssimazione finita.*

Esercizio 25 *Si utilizzi il costrutto* `Mut_Ally` *per scrivere, nel linguaggio finora definito, una procedura che, calcolato il fattoriale del valore associato ad una variable non locale* `x`*, assegna tale valore ad una variabile non locale* `y`*.* □

Esercizio 26 *Si applichi la semantica della dichiarazione per mostrare che la procedura scritta per l'esercizio precedente calcola effettivamente la funzione fattoriale* □

Esercizio 27 *Si consideri la seguente definizione:*
$\mathcal{D}[\![\mathtt{Mut\ D_1\ D_2\ Ally}]\!]_\rho = \mathtt{Y}\ \delta.\ (\rho\ \circ\ \mathcal{D}[\![\mathtt{D_1}]\!]_\delta\ \circ\ \mathcal{D}[\![\mathtt{D_2}]\!]_\delta)$ □

Esercizio 28 *Si consideri la seguente definizione:*
$\mathcal{D}[\![\mathtt{Mut\ D_1\ D_2\ Ally}]\!]_\rho = \mathtt{Y}\ \delta.\ (\mathcal{D}[\![\mathtt{D_1}]\!]_{\delta\ \circ\ \rho}\ \circ\ \mathcal{D}[\![\mathtt{D_2}]\!]_{\delta\ \circ\ \rho})$ □

Combinando opportunamente il costrutto per la dichiarazione sequenziale con quello per la dichiarazione parallela possiamo riscrivere i programmi di quei linguaggi che utilizzano la dichiarazione mista.

4 Le Espressioni

Le espressioni negli LP sono presenti con un unico scopo: esprimere valori calcolabili. Per questa ragione hanno un ruolo fondamentale e permeano tutte le strutture del linguaggio, in particolare le dichiarazioni e i comandi. E proprio per questa ragione la loro struttura sintattica e la semantica associata sono più complicate di quelle viste fin'ora. Dividiamo le espressioni dei linguaggi in due gruppi: Linguaggi con Trasparenza

Referenziale (referential transparency) e Linguaggi con Effetti Laterali (side effects). In entrambi i casi, il run-time support richiesto per la valutazione dell'espressioni consiste di una pila, per i valori intermedi calcolati per gli operandi (o una struttura di registri opportuna), che risiede nell'Activation Record del blocco o della procedura/funzione in cui l'espressione occorre. La gestione degli operandi/operatori sulla pila avviene conformemente alla visita *postorder* depth-first (la più usata dagli interpreti) oppure a quella *preorder* depth-first . La simmetrica o *in-order* è talora utilizzata dai compilatori ma solo quando si usano registri.

Esercizio 29 *Si consideri la seguente espressione in C:* $-(3*x+y/5)*z$. *(a) Si mostri la sintassi astratta; (b) La polacca postfissa, (c) la polacca prefissa; (d) la simmetrica. (e) Si dica perchè la simmetrica è ambigua e quali ipotesi dobbiamo fare su associatività e precedenza degli operatori per farle mantenere il significato originario. (f) Infine, si mostri l'allocazione sullo stack della postfissa e i passi della sua valutazione allorchè: x sia 2, y sia 20, z sia 7.* □

4.1 Le Espressioni con Trasparenza Referenziale

Si parla di Trasparenza Referenziale quando il significato dell'espressione non dipende dal contesto in cui è usata, pertanto la sua occorrenza può essere rimpiazzata dal suo valore (o da ogni espressione con stesso valore) senza che il significato del programma cambi. La funzione semantica di queste espressioni è quella data in Table 3:

$$\mathcal{E} : \texttt{E} \rightarrow \texttt{Env} \rightarrow \texttt{State} \rightarrow \texttt{Val}$$

Questa classe di espressioni è la tipica classe di espressioni dell'algebra e si prestano bene ad essere trattate con tecniche algebriche, anche per la dimostrazione di proprietà. Linguaggi di Programmazione con questa classe di espressioni sono i Linguaggi Funzionali Puri, quali il linguaggio Haskell, i Linguaggi Logici e Algebrici Puri, quali Pure Prolog e Obj, e comunque Linguaggi che non hanno valori modificabili, nè assegnamento nè operazioni che modificano lo stato.

4.2 Le Espressioni con Effetti Laterali

Si parla di Effetti Laterali quando il significato dell'espressione può dipendere dal contesto in cui è usata, pertanto il valore della sua occorrenza è determinato dallo stato della valutazione al momento del calcolo di tale valore (in questo caso: l'espressione non ha trasparenza referenziale). Il calcolo del valore, inoltre può modificare lo stato della valutazione (in questo caso: l'espressione crea un effetto laterale). In tutti i casi, il rimpiazzamento di una tale espressione con il suo valore può condurre a programmi che calcolano in modo difforme dal programma originale. La funzione semantica di queste espressioni è diversa da quella data in Table 3 ed è riportata sotto:

$$\mathcal{E} : \texttt{E} \rightarrow \texttt{Env} \rightarrow \texttt{State} \rightarrow (\texttt{Val} \times \texttt{State})$$

Linguaggi di Programmazione con questa classe di espressioni sono la quasi totalità dei linguaggi, includono certamente i Linguaggi Imperativi, ed anche molti Linguaggi Funzionali, quali Caml, ML, ed Object Oriented, quali, Java, C♯, Eiffel, Scala, F♯, OCaml, Python, etc...

Esercizio 30 *Si considerino la seguenti espressioni in C:* $e_1 \equiv x - (x = y)$, $e_2 \equiv y - (x = y)$, $e_3 \equiv (x = y) - x$. *Nell'ipotesi che le variabili usate siano ben definite (i.e. dichiarate e con valore memorizzabile definito), si dica (a) quale delle tre calcola sempre*

0*; (b) perchè nessuna delle tre espressioni può essere rimpiazzata dal valore calcolato; (c) perchè la variabile è un'espressione che non ha trasparenza referenziale (in C).* □

4.3 Le Espressioni in un Linguaggio di Programmazione

I linguaggi di Programmazione sono Turing Completi o equivalentemente, devono poter esprimere tutte le funzioni calcolabili che includono le *funzioni parziali*. La presenza di strutture, nel linguaggio, in grado di esprimere funzioni parziali conduce inevitabilmente ad espressioni che includono una particolare forma di valore che chiameremo *valore indefinito*, lo indicheremo con $\perp_{\texttt{Val}}$, ed estende il dominio (di valori) `Val` con il seguente valore: $(\texttt{Y}f.\lambda\texttt{x}.f(\texttt{x}))(v)$, con $v \in \texttt{Val}$. Il nuovo dominio risultante, $\texttt{Val}_\perp \equiv \texttt{Val}+\{\perp_{\texttt{Val}}\}$, pone nuovi problemi sul significato di un'espressione, che si aggiungono in modo ortogonale ai problemi osservati prima sulla trasparenza referenziale e sugli effetti laterali. Ed estende anche la terminologia, introducendo il termine *diverge* per indicare (un'espressione la cui valutazione richiede) un calcolo che non termina (ed anche, calcolo/espressione *divergente*)

Esercizio 31 *Si dica perchè il significato dell'indefinito $\perp_{\texttt{Val}}$ è da considerare una buona astrazione per il valore calcolato da tutte le espressioni la cui esecuzione non termina.* □

Esercizio 32 *Si estenda il seguente frammento di programma C:*

```
{... x+f(5)...
```

in modo tale che l'espressione diverga sempre. □

4.3.1 Ordine di Valutazione

La terminologia *ordine di valutazione* si è andata diffondendo per spiegare il significato delle espressioni su domini estesi con indefiniti. Il problema si pone sull'applicazione di un operatore di arità $k > 0$ alle k (sotto-)espressioni che formano gli argomenti dell'applicazione. Possiamo parlare di due ordini di valutazione diversi. Questi sono l'ordine di valutazione:

Interna [Operatori Stretti e Eager Evaluation]. In questo caso, gli argomenti sono valutati e solo dopo si applica l'operatore ai valori così ottenuti. Cosa accade quando uno o più degli argomenti diverge? Ovviamente, l'applicazione diverge, indipendentemente dalla rilevanza degli argomenti divergenti rispetto a quelli non divergenti (per i quali un valore stato calcolato) e al *significato inteso* dell'operatore. Sia `op` il nome di tale operatore di arità k, e sia `op` l'operazione da esso calcolata (il contesto di uso eliminerà ogni ambiguità sull'uso di uno stesso simbolo per la sintassi e la semantica), ovvero $\texttt{op}{:}\texttt{Val}^k \rightarrow \texttt{Val}$, sul dominio `Val`. In accordo alla valutazione interna, l'operazione `op` è estesa sul dominio $\texttt{Val}_\perp$, nella *funzione stretta* $\texttt{op}_{\perp_S} : \texttt{Val}_\perp^k \rightarrow \texttt{Val}_\perp$, così definita:

$$\texttt{op}_{\perp_S}(e_1, ..., e_k) = \begin{cases} \texttt{op}(e_1, ..., e_k) & \text{se e solo se } (\forall i \in [1..k])e_i \in \texttt{Val} \\ \perp_{\texttt{Val}} & \text{otherwise} \end{cases}$$

Esterna [Operatori Non Stretti e Lazy Evaluation]. In questo caso, si applica l'operatore agli argomenti non valutati. Sarà l'operatore stesso a determinare quali argomenti dovranno essere necessariamente valutati. Cosa accade quando uno o più degli argomenti diverge? Dipende dalla definizione dell'operatore e da quali argomenti divergono. Sia `op` il nome di tale operatore di arità k, e sia `op` l'operazione da esso calcolata (il contesto di uso eliminerà ogni ambiguità sull'uso di uno stesso simbolo per la sintassi e la semantica), ovvero `op`:$\texttt{Val}^k \to \texttt{Val}$, sul dominio `Val`. In accordo alla valutazione esterna, l'operazione `op` è estesa sul dominio $\texttt{Val}_\perp$, nella *funzione non stretta* $\texttt{op}_{\perp_S} : \texttt{Val}_\perp^k \to \texttt{Val}_\perp$, tale che:

$$e_i \in \texttt{Val}\ (\forall i \in [1..k]), \text{ implica } \texttt{op}_{\perp_{NS}}(e_1, ..., e_k) = \texttt{op}(e_1, ..., e_k)$$

In effetti, uno stesso operatore può essere esteso da più funzioni non strette e in alcuni casi, queste sono una più definita dell'altra (quando questo è il caso, il linguaggio usa la più definita).

Questi due ordini di valutazione hanno dato origine a delle vere e proprie strategie di valutazione che vedremo quando si parlerà di trasmissione dei parametri (*by Value*, *by Result*, *by ValueResult*, *by Name*, *by Need*, *by Reference*) di valori Finitari Infiniti (approssimabili finitamente) (*Demand Driven*) e di valori Infini rappresentabili finitamente. Inoltre, è opportuno distinguere tra linguaggi in cui sono presenti operatori non stretti e trasmissione by need, e linguaggi che adottano di una vera e propria valutazione lazy. Questi ultimi, infatti hanno valori strutturati i cui costruttori sono Lazy e non valutano gli argomenti quando sono applicati e costruiscono il valore.

Esempio 3 *Valori Infiniti (approssimabili finitamente):*

```
nat n = n:nat(n+1)
naturali = nat 0
v = fst 3 naturali
```

In Haskell le tre espressioni definiscono una funzione `nat`*. Questa funzione applicata ad un intero* `n`*, applica il costruttore di liste* `":"` *ad* `n` *e alla lista risultante dall'applicazione di* `nat` *al succesore di* `n`*. La funzione* `nat` *risulta indefinita in tutti i linguaggi che non prevedono valori Finitari Infiniti e, quindi non hanno meccanismi per calcolare con tali valori. Haskell invece, ha tali valori e la funzione* `nat`*, in Haskell, è definita su tutti gli interi. L'esempio mostra anche come questi valori possano essere utilizzati in computazioni con valori finiti.*

Esistono poi anche delle tecniche di generazione del codice, utilizzate nei back-end dei compilatori, note come *short circuit* (code), nate con lo scopo di ottimizzare il codice che possono essere lette come forme di valutazione esterna.

Esercizio 33 *Si estenda l'operatore prodotto definito sugli interi,* `Int`*, in un prodotto stretto e in un prodotto non stretto su* $\texttt{Int}_\perp$*. Si faccia la stesso per l'operatore somma sempre su* `Int`*, e per* `and` *sui booleani.* □

Esercizio 34 *Si dica perchè l'operatore di uguaglianza può essere esteso solo in modo stretto. Si indichino altri operatori operatori che possono essere estesi, in modo naturale, solo in modo stretto.* □

4.3.2 Espressioni Denotabili

Quando il linguaggio ha variabili o valori modificabili (ad esempio gli array in C, Pascal), la sintassi concreta contestualizza il differente uso delle variabili e dei valori modificabili.

Una variabile, ad esempio, ha come sintassi concreta, in generale, un identificatore che definisce il nome della variabile e permette di usarla nel resto del programma. La semantica di una variabile però è una coppia $(l.m) \in \texttt{Loc} \times \texttt{Mem}$, dove il primo componente, l, la locazione, inalterabile per l'intera vita della variabile, ed il secondo componente, m il valore memorizzabile, può assumere valori diversi. La variabile ha due usi distinti all'interno di un programma, e nella programmazione in generale, il primo consente di riferire il primo componente, il secondo di riferire il secondo componente. Per indicare i diversi valori di una variabile in corrispondenza ai due diversi usi, si introduce il termine di *l-value*, per il primo componente, ed *r-value* per il secondo. In C, e nei linguaggi imperativi, in generale, questo duplice uso è contenuto in scritture come `x = y`, dove l'assegnamento applicato a due variabili `x` ed `y`. Ovviamente, l'espressione `x`, a sinistra, intende esprimere il primo componente della variabile denotata da `x` mentre l'espressione `y`, a destra, intende esprimere il secondo componente della variabile denotata da `y`. Se invertiamo la scrittura, `y = x`, le sotto-epressioni `x` ed `y` sono sempre le stesse ma ciò che intendiamo esprimere ora con ciascuna delle due è completamente diverso. La sintassi concreta utilizza la stessa forma per due significati diversi e questo è reso possibile, come si è ricordato all'inizio della sezione, dalla contestualizzazione dell'uso delle espressioni: a sinistra dell'operatore devono calcolare un l-value, a destra devono calcolare un r-value. In effetti, gli l-values sono inclusi nei valori denotabili `Den`. Tutto ciò non è limitato al solo uso delle variabili, pure si ha con i valori memorizzabili. Ed ancora, non coinvolge solo l'assegnamento pure può coinvolgere la *trasmissione dei parametri* ed in generale tutti i costrutti del linguaggio che operano con valori denotabili. Per questa ragione, la sintassi astratta di un linguaggio distingue tra espressioni ed *espressioni denotabili*, quest'ulitme formano un sottoinsieme delle espressioni. Nel dominio sintattico in Table9, queste si distinguono per l'uso di un costruttore specifico, `Den`, in contapposizione al costruttore `Val`, quest'ultimo riservato alle *espressioni memorizzabili*. Nella semantica data nella stessa tabella, le due sottoclassi di espressioni si distinguono per il significato: la prima ha funzione semantica che calcola un valore denotabile, la seconda ha funzione semantica che calcola un valore memorizzabile.

Notazione 5 (L'operatore Let.) *Per esprimere le funzioni semantiche abbiamo utilizzato, in Table9, il meta operatore Let con la seguente notazione:*

Let $\{B_1\}...\{B_k\}$ *e, dove per ogni i,* $B_i \equiv p_{i,1} = e_{i,1}....p_{i,n_i} = e_{i,n_i}$

I termini $p_{i,j}$ *sono patterns che descrivono una struttura di identificatori e nella forma più semplice sono singoli identificatori da legare ai valori descritti dalle espressioni* $e_{i,j}$*. Il significato può essere ottenuto completando (nel modo ovvio) la seguente definizione induttiva:*

Let $\{x_1 = e_1....x_n = e_n\}\{B_2\}...\{B_k\}$ $e \equiv (\lambda\ x_1...x_n.\ Let\{B_2\}...\{B_k\}\ e)(e_1....e_n)$. □

Esercizio 35 *(a) Si dica cosa calcolano (i.e. valore e modifica dello stato) le seguenti due espressioni C:*

e_1 : `(x=y)|(z=w)`

e_2 : `(a=b)||(c=d)`

allorchè `y, w, b, d` *abbiano valore* `1` *e le rimanenti variabili non siano state inizializzate. (b) Si scriva un programma C che mostri quanto asserito.* □

Esercizio 36 *Si mostri che il comportamento dell'operatore* `||` *di C è esterno ed ha la semantica descritta in Table9.* □

Esercizio 37 *Si utilizzi le considerazioni fatte nei due esercizi precedenti per spiegare come si combina in C, l'uso di operatori non stretti con l'uso di espressioni con effetti laterali.* □

Esercizio 38 *In C, la dichiarazione di una variabile non inizializzata è comunque inizializzata ad un valore di default che dipende dal tipo della variabile, ad esempio 0 per int. In Pascal viceversa, la variabile è inizializzata al valore indefinito. Si modifichi la semantica data in Table9, (a) in modo da rispecchiare il linguaggio C; (b) in modo da rispecchiare il linguaggio Pascal.* □

Table9 – Semantica delle Espressioni con S.E. – 1

Domini Sintattici

$\texttt{D} ::= ... \mid \texttt{Var I;} \mid \texttt{Var I} = \texttt{E;} \mid ...$ $\quad (Dichiarazioni)$

$\texttt{E} ::= \texttt{Val(E)} \mid \texttt{Den(E)} \mid \texttt{LV} \mid op_k(\texttt{E}_1 ... \texttt{E}_k) \mid \texttt{E} = \texttt{E}$ $\quad (Espressioni)$

Domini Semantici

$\texttt{Env}, \rho, \delta \equiv ...$ $\quad (Ambienti)$

$\texttt{Store} \equiv (\texttt{Loc} \times (\texttt{Loc} \to \texttt{Mem}_\perp))$ $\quad (Memoria)$

$\texttt{Store}_\perp, \texttt{s}, \texttt{r} \equiv \texttt{Store} + \{\perp_S\}$ $\quad (Memoria\ Finita)$

operazioni di Store :

$\texttt{upd} : \texttt{Loc} \times \texttt{Mem}_\perp \times \texttt{Store}_\perp \to \texttt{Store}_\perp$

$\texttt{upd}(l, m, (L, u)) \equiv \texttt{if}((l \in [0, L]), \lambda v.\texttt{if}((v\ \texttt{eq}\ l), m, u(l)), \perp_S)$

$\texttt{look} : \texttt{Loc} \times \texttt{Store}_\perp \to \texttt{Mem}_\perp$

$\texttt{look}(l, (L, u)) \equiv \texttt{if}((l \in [0, L]), u(l), \perp_M)$

$\texttt{allocate} : \texttt{Store}_\perp \to (\texttt{Loc} \times \texttt{Store}_\perp)$

$\texttt{allocate}_k((L, u)) \equiv \texttt{if}((L > k), \perp_{LS}, (L, (L + 1, upd(L, \perp_M, u))))$

$\perp_S : \texttt{Store}_\perp$

$\perp_S \equiv (\texttt{Y}f.\lambda\texttt{x}.f(\texttt{x}))(u), \quad con\ u \in \texttt{Store}$

Funzioni Semantiche

$\mathcal{D}[\![\mathtt{D}]\!]_\rho : \mathtt{Store} \to (\mathtt{Env} \times \mathtt{Store}_\perp)$ $\quad$ (*Significato Dichiarazioni*)

$$\mathcal{D}[\![\mathtt{Var\ I};]\!]_\rho(s) = \mathtt{Let}\{(l, s_l) = allocate(s)\}\ (\mathtt{bind}(\mathtt{I}, l, \rho), s_l) \quad (Compile\ T.)$$
$$\mathcal{D}[\![\mathtt{Var\ I = E};]\!]_\rho = \mathtt{Let}\{(l, s_l) = allocate(s)\} \quad (Run\ Time)$$
$$\{(v_e, s_e) = \mathcal{E}[\![\mathtt{E}]\!]_\rho(s_l)\}$$
$$\{s_m = \mathtt{upd}(l, v_e, s_l)\}\ (\mathtt{bind}(\mathtt{I}, l, \rho), s_m)$$

$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Store} \to (\mathtt{Val}_\perp \times \mathtt{Store}_\perp)$ $\quad$ (*Significato Espressioni*)

$$\mathcal{E}[\![\mathtt{Val(E)}]\!]_\rho(s) = \begin{cases} (\mathtt{MV(s}(\rho(\mathtt{E}))), s) & \text{if } \rho(\mathtt{E}) \in \mathtt{Loc} \\ (\mathtt{DV(E)}, s) & \text{otherwise} \end{cases}$$
$$\mathcal{E}[\![\mathtt{Den(E)}]\!]_\rho(s) = \begin{cases} (\rho(\mathtt{E}), s) & \text{if } \rho(\mathtt{E}) \in \mathtt{Loc} \\ (\perp_D, s) & \text{otherwise} \end{cases}$$
$$\mathcal{E}[\![\mathtt{LV}]\!]_\rho(s) = (\mathtt{IntoVal(LV)}, s)$$
$$\mathcal{E}[\![op_k(\mathtt{E}_1 ... \mathtt{E}_k)]\!]_\rho(s) = \mathtt{Let}\{(v_1, s_1) = \mathcal{E}[\![\mathtt{E}_1]\!]_\rho(s)\} \quad (stretti)$$
$$\ldots$$
$$\{(v_k, s_k) = \mathcal{E}[\![\mathtt{E}_k]\!]_\rho(s_{k-1})\}\ (op_{\perp_S}(v_1 ... v_k), s_k)$$

$$\mathcal{E}[\![op_k(\mathtt{E}_1 ... \mathtt{E}_k)]\!]_\rho(s) = \mathtt{Let}\{(v_{i_1}, s_{i_1}) = \mathcal{E}[\![\mathtt{E}_{i_1}]\!]_\rho(s) \quad (non\ stretti)$$
$$\ldots$$
$$\{(v_{i_h}, s_{i_h}) = \mathcal{E}[\![\mathtt{E}_{i_h}]\!]_\rho(s_{i_{h-1}})\}\ (op_{\perp_{NS}}(\overline{v}_1 ... \overline{v}_k), s_{i_h})$$
$$where: (i_1 \neq ... \neq i_h \in [1..k])\ and\ (\overline{v}_{i_j} \equiv\ v_{i_j} (\forall j \in [1..h]))$$
$$\mathcal{E}[\![\mathtt{E}_l = \mathtt{E}_r]\!]_\rho(s) = \mathtt{Let}\{(v_1, s_1) = \mathcal{E}[\![\mathtt{E}_r]\!]_\rho(s)\}$$
$$\{(l_2, s_2) = \mathcal{E}[\![\mathtt{E}_l]\!]_\rho(s_1)\}\ (v_1, \mathtt{upd}(l_2, \mathtt{VM}(v_1), s_2))$$

Funzioni Ausiliarie

$\mathtt{L} : \mathtt{Loc} \to \mathtt{Den}$ $\quad$ (*iniettore, i.e. costruttore*)
$\mathtt{DV} : \mathtt{Den} \to \mathtt{Val}$
$\mathtt{VM} : \mathtt{Val} \to \mathtt{Mem}$

Esercizio 39 *La semantica, in Table9, e ovunque in queste pagine, è data facendo riferimento alla sintassi astratta. Si trascriva il programma dell'Esempio3 nella corrispondente sintassi astratta, limitatamente a quella fin'ora introdotta.* □

Esercizio 40 *Si mostrino i calcoli condotti per applicare le definizioni di Table9 alla semantica della seguente espressione (concreta) C:* `(x=7)*y=(x+2)`*. Allo scopo si assuma che l'espressione sia nello scope di* `x` *ed* `y` *entrambe dichiarate variabili intere* □

Esercizio 41 *Utilizzando i soli costrutti fin qui introdotti si mostri un'espressione che calcola indefinito e/o conduce ad uno stato indefinito (si veda la funzione semantica* $\mathcal{E}[\![\mathtt{E}]\!]_\rho$*).* □

Esercizio 42 *Come al solito per scrivere poco, siamo stati imprecisi. In Table9 abbiamo usato dei termini t che dovevano essere in realtà,* $\mathtt{L}(t)$*. Sapreste dire quali, dove e perchè?* □

4.3.3 Semantica Statica, Dinamica, Front-End e Back-End del Compilatore (Interprete)

Nella quasi totalità dei linguaggi strutturati (un caso eccezionale è la Lisp Machine realizzata intorno al 1980), l'implementazione di una macchina astratta, MA, non è una macchina concreta, bensì è realizzata su una macchina concreta (di un differente linguaggio) attraverso l'uso di un compilatore (o interprete). In questi casi, la semantica che stiamo scrivendo, come quella in Table9, è frazionata in parti distinte da utilizzare in fasi distinte del compilatore. In particolare, la semantica delle dichiarazioni è completamente trattata dal front-end in ciascuna delle sue tre principali fasi: riconoscimento degli identificatori (scanner), inserimento nella tabella dei simboli (parser), ed aggiornamento di quest'ultima con le informazioni relative a valore denotabile, tipo e altre proprietà desumibili dall'analisi della struttura del programma (analisi statica). Nel caso di una variabile, il valore denotabile è una locazione di memoria che viene allocata almeno in forma simbolica (offset rispetto ad una base di rilocamento), durante la fase di parsing o di analisi statica. Per ragioni di rilocabilità sarà rimandato, al momento del (linking e del) caricamento del programma (per la successiva esecuzione), l'allocazione fisica (cioè nella memoria effettiva). Quando operiamo a livello di front-end, ovvero in queste fasi, si dice che stiamo operando con la semantica statica. Da qui la distinzione tra semantica statica e semantica dinamica di un linguaggio. Nella semantica statica ci si concentra sugli aspetti dichiarativi e si ignora del tutto la memoria. Per questa ragione in Table9 abbiamo annotato con il termine *run time*, la semantica della dichiarazione di variabile con inizializzazione. Questo é quello che avviene quanto meno in quella forma in cui l'espressione di inizializzazione può contenere valori memorizzabili. In effetti, in alcuni linguaggi la dichiarazione con inizializzazione non è prevista o, laddove prevista, è ammessa solo con espressioni denotabili (valori costanti), il cui calcolo, in sostanza è interamente risolto al tempo di compilazione utilizzando il solo ambiente dei nomi.

Esercizio 43 *(a) Si introduca un nuovo dominio sintattico per le espressioni denotabili, esteso da quello delle espressioni. (b) sia dia una semantica per le espressioni denotabili, conforme a quella data per le espressioni, (c) si riformuli la sintassi e la semantica della dichiarazione di variabile con inizializzazione.* □

5 I Comandi

I comandi negli LP sono presenti con un unico scopo: modificare lo stato, modificando la memoria o alterando il flusso di controllo. Da questo punto di vista, l'assegnamento, trattato in Table9 come un'espressione, è per la grande quantità di linguaggi un comando. In Java l'assegnamento è entrambe le cose giacchè ogni espressione è in Java anche un comando.

Esercizio 44 *(a) Si dia una sintassi dell'assegnamento come come comando; (b) si dia una semantica dell'assegnamento come comando; (c) si mostri come potrebbe essere data la semantica denotazionale di Java limitatamente alla conversione delle espressioni come comandi.* □

Esercizio 45 *Si dica come è visto l'assegnamento in ciascuno dei seguenti linguaggi: Pascal, C, C♯, Eiffel, Haskell, F♯.* □

L'assegnamento è l'operazione elettiva per la modifica della memoria (statica), anche se per i linguaggi con allocazione dinamica non è la sola, come vedremo più avanti. I rimanenti comandi allora, operano per il controllo di sequenza:

- `Esplicito.` Includono `goto`, `break`, `continue`, `return`. Alcuni di questi (`goto`) sono non strutturati e richiedono (`goto`, `break`, `continue`) una *continuation semantics* per essere descritti in modo soddisfacente (come vedremo dopo). Altri (`return`) sono strutturati e possono essere descritti bene utilizzando le funzioni semantiche introdotte in Table3. Nel controllo esplicito includeremo anche il *comando composto*, che consente di fare il *grouping* di comandi.
- `Condizionato.` Includono condizionali `if` (a una o due vie), `case`, `cond`. `switch`, etc...
- `Iterativo.` Due forme: iteratori determinati e iteratori non determinati. Questi ultimi garantiscono, in alternativa alle procedure ricorsive, la TC del linguaggio.

5.1 I Comandi Strutturati

La forma dei costrutti, in particolare dei comandi, assume rilevanza anche semantica nella programmazione strutturata dove si chiede che i costrutti che formano il programma abbiano tutti un unico punto d'ingresso ed un unico punto di uscita che il flusso di controllo deve necessariamente attraversa se il costrutto è attraversato. Il `goto` è allora fonte di facile violazione di tale vincolo come si evince dall'esempio sotto.

```
{...while...do {...A: ...};...;goto A;...}
```

Si noti come sia difficile nel'esempio sopra, anche solo immaginare cosa il programma può mai calcolare, allorchè si completino le parti lasciate non specificate.

Esercizio 46 *Si completi un tale programma in C e lo si compili. Si discutano le osservazione sollevate dal compilatore.* □

Esercizio 47 *Si completi un tale programma in Java e lo si compili. Si discutano le osservazione sollevate dal compilatore.* □

Esercizio 48 *Si completi un tale programma in Pascal e lo si compili. Si discutano le osservazione sollevate dal compilatore.* □

Table10 – Semantica dei Comandi – 1.

Domini Sintattici

$$\begin{aligned}&\texttt{C} ::= \texttt{C C} \mid \texttt{E = E} \mid \texttt{IF E Then C Else C} \mid \texttt{Case E S}\\ &\quad \mid \texttt{For I = E To E By E Do C}\\ &\quad \mid \texttt{While E Do C} \mid \texttt{Proc I()C} \mid \texttt{Call I()} \quad (\textit{Comandi})\end{aligned}$$

Funzioni Semantiche

$$\mathcal{C}[\![\texttt{C}]\!]_\rho : \texttt{Store} \to \texttt{Store}_\perp \qquad (\textit{Comandi})$$

$$\begin{aligned}&\mathcal{M}[\![\texttt{C}_1\ \texttt{C}_2]\!]_\rho = \mathcal{M}[\![\texttt{C}_1]\!]_\rho \circ \mathcal{M}[\![\texttt{C}_2]\!]_\rho\\ &\mathcal{M}[\![\texttt{E}_1 = \texttt{E}_2]\!]_\rho = \qquad (\textit{vedi Esercizio } 49)\\ &\mathcal{M}[\![\texttt{IF E Then C}_1 \texttt{ Else C}_2]\!]_\rho = \qquad (\textit{vedi Esercizio } 50)\\ &\mathcal{M}[\![\texttt{Case E S}]\!]_\rho = \qquad (\textit{vedi Esercizio } 52)\\ &\mathcal{M}[\![\texttt{For I} = \texttt{E}_1 \texttt{ To E}_2 \texttt{ By E}_3 \texttt{ Do C}]\!]_\rho(\texttt{s}) =\\ &\qquad \texttt{Let}\{(\mathit{inizio}, \texttt{s}_1) = \mathcal{E}[\![\texttt{E}_1]\!]_\rho(\texttt{s})\}\\ &\qquad\quad \{(\mathit{fine}, \texttt{s}_2) = \mathcal{E}[\![\texttt{E}_2]\!]_\rho(\texttt{s}_1)\}\\ &\qquad\quad \{(\mathit{passo}, \texttt{s}_3) = \mathcal{E}[\![\texttt{E}_3]\!]_\rho(\texttt{s}_2)\}\\ &\qquad\quad \{n = (\mathit{fine} - \mathit{inizio} + \mathit{passo})/\mathit{passo}, f = \mathcal{M}[\![\texttt{C}]\!]_\rho\}\\ &\qquad f^n(\texttt{s}_3)\\ &\mathcal{M}[\![\texttt{While E Do C}]\!]_\rho(\texttt{s}) =\\ &\qquad \texttt{Y}f.\lambda\texttt{s}.\texttt{Let}\{(v, \texttt{s}_1) = \mathcal{E}[\![\texttt{E}]\!]_\rho(\texttt{s})\}\\ &\qquad\quad \texttt{if}(\texttt{true}(v), \texttt{s}_1, (\mathcal{M}[\![\texttt{C}]\!]_\rho \circ f)(\texttt{s}_1))\end{aligned}$$

Funzioni Ausiliarie

$$\begin{aligned}&\texttt{DV} : \texttt{Den} \to \texttt{Val}\\ &\texttt{VM} : \texttt{Val} \to \texttt{Mem}\\ &\texttt{true} : \texttt{Val} \to \texttt{Boolean}\end{aligned}$$

Esercizio 49 *Si dia la semantica del comando di assegnamento, riportato in* C. *(b) La si confronti con quella data per l'espressione di assegnamento in Table9* □

Esercizio 50 *Si dia la semantica del costrutto* IF_Then_Else*, riportato in* C. *(b) Si estenda* C *con la sintassi astratta per un costrutto condizionale di tipo one-way e si fornisca la relativa semantica;* □

Esercizio 51 *Il linguaggio C contiene il costrutto* Else. *Si dica come tale costrutto deve essere trattato nella sintassi astratta del dominio* C *di Table10. Quindi se ne dia la semantica, confrontandola con quella del condizionale two-way discusso nell'esercizio precedente.* □

Esercizio 52 *Si fornisca una definizione per il dominio ausiliario delle sequenze* S *utilizzato nel costrutto* Case *di Table10. (b) Si provveda quindi, a dare la semantica di tale costrutto.* □

Esercizio 53 *Perchè il costrutto* for *del linguaggio C è un iteratore non determinato. (b) Si dia una sintassi astratta per il* for *del C e la si corredi della semantica relativa.*□

Esercizio 54 *Perchè il costrutto* for *del linguaggio Pascal è un iteratore determinato. (b) Si dia una sintassi astratta per la variante* `downto` *del Pascal e la si corredi della semantica relativa.* □

Esercizio 55 *Si mostri come sia possibile rimpiazzare nel programma seguente:*

```
... While E do C ...
```

l'iteratore non determinato `While` *con un'invocazione* `Call` *di una procedura senza parametri che utilizzi i meccanismi fin qui introdotti e riportati nelle varie Tables. Allo scopo: (a) si consideri il nuovo programma; (b) la definizione della procedura invocata e la sua invocazione; (c) si discutano i meccanismi coinvolti; (d) si giustifichi quanto meno, che il programma ottenuto abbia lo stesso comportamento di quello originale* □

5.1.1 Tail Recursion e Iteration: Considerazioni sull'implementazione

L'esercizio precedente ci richiama una fondamentale equivalenza computazionale tra itaratore non determinato e procedure ricorsive. Questa equivalenza è la Turing Completezza, TC, ovvero ognuno dei due presi separatamente garantiscono la TC del linguaggio. Parimenti, la mancanza dei due rende il linguaggio non TC, quindi non in grado di esprimere tutte le funzioni calcolabili. Sebbene, quindi la mancanza di entrambi vada evitata, vediamo invece, che tutti i linguaggi di programmazione prevedono la presenza di entrambi. Le ragioni sono ovviamente legate non già all'esprimere ma a come una stessa funzione è espressa: gli algoritmi, le metodologie per descrivere tali algoritmi nel linguaggio, le strutture che il linguaggio fornisce per agevolare tali definizioni. Ad esempio, si rifletta sulla formulazione dell'algoritmo di quicksort per l'ordinamento di una sequenza di valori ordinabili.

Esercizio 56 *(a) Si dia una definizione ricorsiva in C di una funzione ad un argomento che calcola l'n-esimo della serie di Fibonacci; (b) Si dia una definizione iterativa in C della stessa funzione.* □

Detto questo, aggiungiamo che le implementazioni dei due meccanismi, attualmente realizzate sulle macchine concrete, sono tutt'altro che equivalenti sia dal punto di vista delle risorse utilizzate, sia dal punto di vista del tempo consumato a run-time per far funzionare tali strutture. La bilancia pesa a favore dell'iterazione. Ciò è controbilanciato tuttavia, da due fatti:
(1) Sono piú gli schemi ricorsivi di quelli iterativi (Paterson??);
(2) Esistenza della tail recursion (Linguaggio Scheme - Sussman) e sua implementazione ad un costo analogo all'iterazione (vedi testo)
Un aspetto interessante della tail recursion è che talvolta la versione tail recursive della funzione ricorsiva è più semplice e diretta di quella che altrimenti si otterrebbe cercando di fornire una versione iterativa di un algoritmo ricorsivo. Vediamo questo nell'esempio sotto.

Esempio 4 *Qui mostriamo come appare in C, una definizione tail recursive della funzione ricorsiva per il calcolo dell'esimo della serie di Fibonacci.*

```
int fibT(int n, int pre1, int pre2){//tail recursive
         if (n==0)return pre1;
         if (n==1)return pre2;
         return (fibT(n-1,pre2,pre1+pre2));}
```

Esercizio 57 *(a) Si dia una versione tail recursive della funzione ricorsiva C, seguente:*

```
int fibA(int A[], int i, int size){
          if (i>size)return 1;
          return (A[i] * fibA(A,i+2,size));}
```

(b) Le si compilino e se ne confrontino le performances, utilizzando le opportune librerie. □

C'è un modo sistematico per passare da una funzione definita ricorsiva ad una tail recursive e questo si basa (a) sulla conoscenza del *dominio ben fondato* dei valori su cui ricorre la funzione, (b) sul ricorso a funzioni come valori. Quest'ultimo è però l'aspetto più critico perchè la tail-recursion è implementabile con performance simile all'iterazione solo se non usa funzioni come valori, come è in tutti gli esempi discussi. In caso contrario, il ricorso a valori funzione richiede l'impiego di aggiuntivi AR allorchè tale funzione sia applicata (e possibilmente, un'intera catena di AR, allorchè quest valori funzioni usino a loro volta valori funzioni) Alcune considerazione sull'implementazione della tail-recursion?

Esercizio 58 *Si mostri come varia lo stack di AR nel caso della funzione Fibonacci ricorsiva e nel caso della sua corrispondente tail recursive dell'esempio sopra. Allo scopo ci si limiti alla seguente situazione: (1) lo stato del calcolo contiene un AR che richiede di applicare la funzione Fibonacci sul valore 3; (2) Abbiamo l'indirizzo del risultato di tale AR e vale "r"; (3) Abbiamo il valore del link di catena statica della funzione Fibonacci e vale "cs"; (4) Abbiamo il valore di accesso di tale AR e vale "cd". Si mostri l'evoluzione dello stack.* □

6 Le Astrazioni sul controllo

I meccanismi fin qui discussi sono sufficiente alla TC di un linguaggio di programmazione (LP). Ma ci sono ragioni estendere per gli LP con numerosi altri meccanismi, alcuni specifici e condivisi da classi di LP. Discuteremo di questi più avanti quando parleremo di paradigmi e loro specificità. Le ragioni per estendere gli LP con astrazioni sul controllo sono molteplici, di natura diversa, ed includono:

- Riuso del codice;
- Localizzazione del codice (moduli);
- Modifica e certificazione del codice (isolare dal contesto di uso);
- Meccanismo per esprimere calcoli definiti in modo induttivo (ricorsione);
- Meccanismo per esprimere valori funzionali in un programmazione higher order.

Le astrazioni sul controllo sono utili tanto per generalizzare composizioni di comandi, quanto per generalizzare composizione di espressioni. Otteniamo due forme di astrazione: le astrazioni funzionali e quelle imperative. Queste sono riportate in table11. Le due forme condividono quasi tutto a partire dai domini dei parametri (e di conseguenza, dai meccanismi di trasmissione utilizzata). Completiamo quanto fatto in Table5 per la dichiarazione e invocazione di procedura senza parametri, con la dichiarazione

e invocazione di funzione senza parametri. La struttura sintattica scelta per il corpo della funzione è quella tipica dei linguaggi funzionali (vedremo più avanti altre scelte). La semantica è quella tipica dell'astrazione che quando dichiarata lega l'identificatore di funzione ad un valore denotabile che consiste nella funzione da stato in valore stato che deve essere calcolata ad ogni invocazione. Come in Table5, continuiamo ad assumere scoping statico, pertanto il significato degli identificati occorrenti nel corpo della funzione è trovato immediatamente, ed una volta per tutte, nell'ambiente di dichiarazione. La semantica dell'invocazione quindi è il valore ottenuto applicando la funzione (che troviamo nell'ambiente di invocazione, legata all'identificatore specificato nell'invocazione) allo stato corrente. Otteniamo una coppia (valore,stato) come per ogni espressione che può generare effetti laterali. Da un punto di vista implementativo, le dichiarazioni di astrazione (sia funzionali che imperative) conducono a denotazioni chiamate *chiusure* formate da una coppia contenente il codice da eseguire (incluso prologo ed epilogo per la trasmissione dei parametri e il trasferimento di controllo al chiamante) e l'ambiente del blocco di dichiarazione. Questo ambiente (che corrisponde al ρ che vediamo nella formula $\mathtt{F}(\mathcal{E}[\![\mathtt{E}]\!]_\rho)$ di Tabella 11) è implementato nell'usuale forma di frame delle locali e puntatore di catena statica.

Table11 – Semantica delle Espressioni 2 : Le Astrazioni Funzionali

Domini Sintattici

$\mathtt{D} ::= ... \mid \mathtt{Function\ I(P_1\ I_1\ ...\ P}_n\ \mathtt{I}_n\mathtt{)\ E} \mid ...$ (*Procedure con Parametri*)
$\mathtt{E} ::= ... \mid \mathtt{A}$ (*Espressioni : Parametri Attuali*)
$\mid \mathtt{Call\ I(A_1\ ...\ A}_n\mathtt{)} \mid ...$(*Espressioni : Invocazione*)

Funzioni Semantiche

$\mathcal{D}[\![\mathtt{D}]\!]_\rho : \mathtt{Store} \to (\mathtt{Env} \times \mathtt{Store}_\perp)$ (*Significato Dichiarazioni*)

$\mathcal{D}[\![\mathtt{Function\ I()\ E}]\!]_\rho(s) = (\mathtt{bind}(\mathtt{I}, \mathtt{F}(\mathcal{E}[\![\mathtt{E}]\!]_\rho), \rho), s)$

$\mathcal{E}[\![\mathtt{E}]\!]_\rho : \mathtt{Store} \to (\mathtt{Val}_\perp \times \mathtt{Store}_\perp)$ (*Significato Espressioni*)

$\mathcal{E}[\![\mathtt{Call\ I()}]\!]_\rho(s) = \mathtt{Let}\{\mathtt{F}(f) = \rho(\mathtt{I})\}\mathtt{in}\ f(s)$

Domini Sintattici Ausiliari

$\mathtt{P} ::= \mathtt{byValue\ I} \mid \mathtt{byName\ I} \mid \mathtt{byReference\ I} \mid \mathtt{byConst\ I}$
$\mid \mathtt{byResult\ I} \mid \mathtt{byValueResult\ I}...$ (*Parametri Formali*)
$\mathtt{A} ::= \mathtt{AMem(E)} \mid \mathtt{Den(E)} \mid \mathtt{ACodeC(C)} \mid \mathtt{ACodeE(E)}$
$\mid \mathtt{AConst(E)}$ (*Parametri Attuali*)

Funzioni Ausiliarie

$\mathtt{F} : (\mathtt{State} \to (\mathtt{Val}\ \times\ \mathtt{State})) \to \mathtt{ProcFun}$ (*iniettiva, i.e. costruttore*)

Quando le astrazioni (funzionali o imperative che siano) hanno parametri la situazione è più interessante. Per alcuni versi, potremmo dire che le vere astrazioni sono quelle con parametri (vedi Esercizio 61). In effetti, la presenza dei parametri ha lo scopo di creare un ambiente per l'insieme di identificatori che sono stati generalizzati dall'astrazione e che al momento dell'invocazione devono essere legati alle strutture con cui vogliamo

istanziare il corpo della procedura o funzione. Nei linguaggi di programmazione sono stati definiti (a partire dall'ALGOL) diversi modi per realizzare questa istanziazione. Questi modi si chiamano forme di trasmissione. Mostriamo le varie forme di trasmissione dei parametri facendo riferimento al loro impiego in una procedura. Ma il loro impiego in una funzione mantiene sintassi, semantica e persegue gli stessi scopi metodologici descritti nel caso di trasmissione in una procedura.

Table11 – Semantica dei Comandi 2 : Le Astrazioni Imperative

Domini Sintattici

$\texttt{D} ::= ... \mid \texttt{Proc I}(\texttt{P}_1\ \texttt{I}_1\ ...\ \texttt{P}_n\ \texttt{I}_n)\ \texttt{C} \mid ...$ (*Procedure con Parametri*)
$\texttt{C} ::= ... \mid \texttt{Call Proc I}(\texttt{A}_1\ ...\ \texttt{A}_n) \mid ...$ (*Comandi : Invocazione*)
$\texttt{E} ::= ... \mid \texttt{A} \mid ...$ (*Espressioni : Parametri Attuali*)

Domini Sintattici Ausiliari
$\texttt{P} ::= \texttt{byValue I} \mid \texttt{byName I} \mid \texttt{byReference I} \mid \texttt{byConst I}$
$\mid \texttt{byResult I} \mid \texttt{byValueResult I}...$ (*Parametri Formali*)
$\texttt{A} ::= \texttt{AMem(E)} \mid \texttt{Den(E)} \mid \texttt{ACodeC(C)} \mid \texttt{ACodeE(E)}$
$\mid \texttt{AConst(E)}$ (*Parametri Attuali*)

6.1 Trasmissione dei Parametri nei Linguaggi di Programmazione

6.1.1 Trasmissione per Valore.

Table11.1 – Procedure 2 : Trasmissione per Valore

Funzioni Semantiche

$\mathcal{D}[\![\texttt{D}]\!]_\rho : (\texttt{Env} \times \texttt{Store}) \to (\texttt{Env} \times \texttt{Store}_\perp)$ (*Significato Dichiarazioni*)

$$\mathcal{D}[\![\texttt{Proc I(byValue I}_1\texttt{) C}]\!]_\rho = \texttt{Let}\{f = \lambda\, v\,.\lambda\, s\,.\texttt{Let}\{(l_1, s_1) = \mathit{allocate}(s)\}\{s_2 = \texttt{upd}(l_1, v, s_1),\ \rho_1 = \texttt{bind}(\texttt{I}_1, l_1, \rho)\}\ \texttt{in}\ \mathcal{M}[\![\texttt{C}]\!]_{\rho_1}(s)\ \texttt{in}\ \mathit{bind}(\texttt{I}, f, \rho)$$

$\mathcal{C}[\![\texttt{C}]\!]_\rho : \texttt{Env} \to \texttt{State} \to \texttt{State}_\perp$ (*Significato Comandi*)

$$\mathcal{C}[\![\texttt{Call I(AMem(E))}]\!]_\rho(s) = \texttt{Let}\{(v_1, s_1) =_{\perp_S} \mathcal{E}[\![\texttt{E}]\!]_\rho(s),\ f = \rho(\texttt{I})\}\ \texttt{in}\ f(\texttt{VM}(v_1))(s_1)$$

Funzioni Ausiliarie
$\texttt{DV} : \texttt{Den} \to \texttt{Val}$
$\texttt{VM} : \texttt{Val} \to \texttt{Mem}$

È una forma di trasmissione one-way in ingresso, concepita per generalizzare variabili presenti nel codice da astrarre. Tale codice conterrà possibilmente operazioni per la

modifica dei valori associati a tali variabili (vedasi, trasmissione per costante, più avanti). L'astrazione così creata può essere utilizzata in punti diversi del programma invocandola in ciascun punto coi i valori che vogliamo siano usati come valori iniziali per tali variabili in tale punto.

Esercizio 59 *Si consideri il seguente programma di un linguaggio a blocchi*

```
{int x; int y;
  void P(){
     int inx = x; int dec = 1; int i;
     y = 1; if (inx > 0) dec = -dec;
     For i = inx To 0 By dec Do y = y*i;
    }
 x = y = -3; Call P(); printf("x vale %2d; y vale %2d/n",x,y);
}
```

(a) Utilizzando la trasmissione per valore, si modifichi la definizione di P in una che eviti l'uso di variabili non locali. (b) Si dica se il programma ottenuto è equivalente a quello originario? Ovvero la semantica denotazionale di P coincide con quella della procedura modificata? Si motivi la risposta. (c) Si supponga di eseguire il programma originario e quello ottenuto dalla modifica in accordo ad una ragionevole implementazione della trasmissione per valore: I due programmi mostreranno lo stesso comportamento? E in caso contrario sapreste mostrare uno stato (stack di AR + memoria), o parte di esso, delle rispettive computazioni da cui si evinca quanto asserito.

Esercizio 60 *Si consideri il seguente programma di un linguaggio a blocchi*

```
{int x; int y;
  void P(){
     int inx = x; int dec = 1; int i;  int temp = 1;
     if (inx > 0) dec  = -dec;
     For i = inx To 0 By dec Do temp = temp*i;
     y = temp
    }
 x = y = -3; Call P(); printf("x vale %2d; y vale %2d/n", x, y);
}
```

(a) Utilizzando la trasmissione per valore, si modifichi la definizione di P in una che eviti l'uso di variabili non locali. (b) Si dica se il programma ottenuto è equivalente a quello originario? Ovvero la semantica denotazionale di P coincide con quella della procedura modificata? Si motivi la risposta. (c) Si trascriva il programma originario e il programma modificato in C.

Esercizio 61 *Si discuta come il ricorso alle variabili non locali per sopperire alla trasmissione di parametri sia fortemente limitativo. In particolare: (a) Si dica cosa calcola la procedura P() riportata nell'esercizio precedente: ovvero si mostri come il significato della procedura P() dipenda* esclusivamente *dal contesto in cui è definita. (b) Si discuta poi il riuso di P() in parti diverse del programma: ovvero si estenda il programma dell'esempio precedente in modo da includere almeno due invocazioni di P(). (c) Si riscriva la procedura P con due parametri trasmessi per valore e si considerino i punti (a) e (b) precedenti sulla nuova definizione.*

6.1.2 Trasmissione per Nome.

È una forma di trasmissione che permette di generalizzare codice presente nel codice da astrarre. Il codice generalizzato e rimpiazzato, nell'astrazione, da un parametro, può essere sia codice *comando* sia codice *espressione*, ovvero che si comporta come una sequenza di comandi oppure come un'espressione. In entrambi i casi, il codice generalizzato è associato dalla trasmissione al parametro formale e trattato come una speciale denotazione. Ad ogni invocazione, ogni occorrenza del parametro formale, nel corpo della procedura, è rimpiazzata testualmente dal codice di tale denotazione.

Table11.2 – Procedure 2 : Trasmissione per Nome

Domini Sintattici

$\texttt{C} ::= ... \mid \texttt{Call Proc I}(\texttt{A}_1\ ...\ \texttt{A}_n) \mid \texttt{Exec I} \mid ...$ $\quad (Invocazione, Esecuzione)$

Funzioni Semantiche

$\mathcal{D}[\![\texttt{D}]\!]_\rho : (\texttt{Env} \times \texttt{Store}) \to (\texttt{Env} \times \texttt{Store}_\perp)$ $\quad (Significato\ Dichiarazioni)$

$$\mathcal{D}[\![\texttt{Proc I(byName I}_1\texttt{) C}]\!]_\rho = \texttt{Let}\{f = \lambda\ v.\texttt{Let}\{\rho_1 = \texttt{bind}(\texttt{I}_1, \texttt{Z}(v), \rho)\}\ \texttt{in}\ \mathcal{M}[\![\texttt{C}]\!]_{\rho_1}\}\ \texttt{in}\ bind(\texttt{I}, f, \rho)$$

$\mathcal{C}[\![\texttt{C}]\!]_\rho : \texttt{State} \to \texttt{State}_\perp$ $\quad (Significato\ Comandi)$

$$\mathcal{C}[\![\texttt{Call I(ACodeC(C))}]\!]_\rho = \texttt{Let}\{f = \rho(\texttt{I})\}\ \texttt{in}\ f(\mathcal{M}[\![\texttt{C}]\!]_\rho)$$
$$\mathcal{C}[\![\texttt{Call I(ACodeE(E))}]\!]_\rho = \texttt{Let}\{f = \rho(\texttt{I})\}\ \texttt{in}\ f(\mathcal{E}[\![\texttt{E}]\!]_\rho)$$
$$\mathcal{C}[\![\texttt{Exec I}]\!]_\rho = \texttt{Let}\{\texttt{Z}(v) = \rho(\texttt{I})\}\ \texttt{in}\ v$$

$\mathcal{E}[\![\texttt{E}]\!]_\rho : \texttt{Store} \to (\texttt{Val} \times \texttt{Store})_\perp$ $\quad (Espressioni: Estensioni)$

$$\mathcal{E}[\![\texttt{Val(I)}]\!]_\rho(\texttt{s}) = \begin{cases} (\texttt{MV}(\texttt{s}(l), \texttt{s}) & \text{if } \rho(\texttt{I}) \equiv l, for\ l \in \texttt{Loc} \\ v(\texttt{s}) & \text{if } \rho(\texttt{I}) \equiv \texttt{Z}(v), \\ & \quad \text{for } v \in \texttt{Store} \to (\texttt{Val} \times \texttt{Store})_\perp \\ (\texttt{DV}(d), \texttt{s}) & \text{if } \rho(\texttt{I}) \equiv d, for\ d \in \texttt{LV} \end{cases}$$

$$\mathcal{E}[\![\texttt{Den(I)}]\!]_\rho(\texttt{s}) = \begin{cases} (l, \texttt{s}) & \text{if } (\rho(\texttt{I}) \equiv l)\ or\ (\rho(\texttt{I}) \equiv \texttt{Z}(l)), for\ l \in \texttt{Loc} \\ (\perp_D, \texttt{s}) & \text{otherwise} \end{cases}$$

Domini Ausiliari

$\texttt{Den} ::= \texttt{Loc} + \texttt{ProcFun} + \texttt{LV} + \texttt{Code}$ $\quad (Unione\ disgiunta)$
$\texttt{Code} ::= (\texttt{State} \to \texttt{State}) + (\texttt{State} \to (\texttt{Val} \times \texttt{State}))$ $\quad (Valori\ Codice)$

Funzioni Ausiliarie

$\texttt{Z} : \texttt{Code} \to \texttt{Den}$ $\quad (iniettiva, i.e.\ costruttore)$

L'introduzione nella trasmissione per nome, di codice come valori denotabili conduce inevitabilmente a rivedere ed estendere la semantica (e l'implementazione del meccanismo di valutazione) 1) dei comandi con un costrutto `Exec I`, e 2) delle espressioni, anche se limitatamente alle sole espressioni `Den(I)` e `Val(I)`. L'estensione è quella ovvia che permette di usare semplici variabili come codice. L'esempio sotto ne mostra le caratteristiche.

Esempio 5 *Consideriamo il seguente programma, dove abbiamo evidenziato, ricorrendo alla sintassi astratta generata dal compilatore, la vera struttura dei termini coinvolti nella trasmissione dei parametri e in altre strutture di interesse per gli scopi dell'esempio.*

```
(1) int x;
(2) Proc B(byName w){
       Den(w) = Val(w) + 5;}
(3) x = 5;
(4) Call B(ACodeE(Den(x)));
```

Mostriamo le coppie $(\rho, \mathtt{s})$*, calcolate dalla semantica dopo ogni dichiarazione o comando del blocco principale.*

(1) $\rho_1 \equiv [l_{\mathtt{x}}/\mathtt{x}]\rho$, $\mathtt{s}_1 \equiv [l_{\mathtt{x}} \leftarrow \perp_{int}]\mathtt{s}$

(2) $\rho_2 \equiv [f/\mathtt{A}]\rho_1$, $\mathtt{s}_2 \equiv \mathtt{s}_1$ *where:*

$$f \equiv \lambda v.\lambda r.\mathtt{Let}\{\delta = [\mathtt{Z}(v)/\mathtt{w}]\rho_1\} \{(u_1, r_1) = (\mathcal{E}[\![\mathtt{Val(w)}]\!]_\delta(r) + 5, r)\} \{(l_2, r_2) = (\mathcal{E}[\![\mathtt{Den(w)}]\!]_\delta(r), r)\} \mathtt{in}\ [l_2 \leftarrow u_1]r$$

(3) $\rho_3 \equiv \rho_2$, $\mathtt{s}_3 \equiv [l_{\mathtt{x}} \leftarrow 5]\mathtt{s}$

(4) $\rho_4 \equiv \rho_2$, $\mathtt{s}_4 \equiv f(l_x)(\mathtt{s}_3)$ *where:*

$$f(l_{\mathtt{x}})(\mathtt{s}_3) = \mathtt{Let}\{\delta = [\mathtt{Z}(l_{\mathtt{x}})/\mathtt{w}]\rho_1\} \{(u_1, r_1) = (\mathtt{MV}(\mathtt{s}_3(l_{\mathtt{x}}) + 5, \mathtt{s}_3)\} = (10, \mathtt{s}_3) \{(l_2, r_2) = (l_{\mathtt{x}}, \mathtt{s}_3)\} \mathtt{in}\ [l_{\mathtt{x}} \leftarrow 10]\mathtt{s}_3$$

ovvero, $\mathtt{s}_4 \equiv [l_{\mathtt{x}} \leftarrow 10]\mathtt{s}$

Il calcolo di $\mathtt{s}_4$ *conclude l'esempio che mostra come la semantica della trasmissione per nome applicata al programma genera uno stato in cui è definito l'identificatore di variabile* `x`*e questo ha associato in memoria valore 10 per effetto dell'invocazione di* `A` □

Esercizio 62 *Facciamo la stessa cosa dell'esempio, nel caso del seguente programma che ora generalizza comandi:*

```
(1) int x;
(2) Proc B(byName w){
       x=10;
       Exec w; }
(3) x = 5;
(4) Call B(ACodeC(x=x+x));
```

Si dica quanto vale x dopo l'invocazione in (4). □

6.1.3 Trasmissione per Riferimento.

È una forma di trasmissione per condivisione di memoria che può essere utilizzata per generalizzare variabili presenti nel codice da astrarre. L'astrazione così creata può essere utilizzata in punti diversi del programma invocandola in ciascun punto con le variabili che vogliamo siano condivise in tale punto del programma. L'effetto è certamente quello di una trasmissione two-way ingresso/uscita (vedi il value-result), ed anche di più. Infatti la condivisione della memoria tra formali e attuali è effettiva e per l'intera esecuzione dell'astrazione. Ciò può creare aliasing all'interno dell'astrazione ed anche effetti collaterali particolari in linguaggi di programmazione con meccanismi per la valutazione concorrente o per l'interleaving di processi, incluso il multithreading: Perchè ogni modifica su un parametro si "ripercuote istantaneamente" sul valore della variabile condivisa e sugli accessi ad essa.

Table11.3 – Procedure 2 : Trasmissione per Riferimento

Funzioni Semantiche

$\mathcal{D}[\![\texttt{D}]\!]_\rho : (\texttt{Env} \times \texttt{Store}) \to (\texttt{Env} \times \texttt{Store}_\perp)$ $\quad$ (*Significato Dichiarazioni*)

$$\mathcal{D}[\![\texttt{Proc I(byReference I}_1\texttt{) C}]\!]_\rho = \texttt{Let}\{f = \lambda\, v.\texttt{Let}\{\rho_1 = \texttt{bind}(\texttt{I}_1, v, \rho)\}\ \texttt{in}\ \mathcal{M}[\![\texttt{C}]\!]_{\rho_1}\}\ \texttt{in}\ \mathit{bind}(\texttt{I}, f, \rho)$$

$\mathcal{C}[\![\texttt{C}]\!]_\rho : \texttt{Env} \to \texttt{State} \to \texttt{State}_\perp$ $\quad$ (*Significato Comandi*)

$$\mathcal{C}[\![\texttt{Call I(Den(E))}]\!]_\rho(s) = \texttt{Let}\{(l_1, s_1) =_{\perp_S} \mathcal{E}[\![\texttt{E}]\!]_\rho(s),\ f = \rho(\texttt{I})\}\ \texttt{in}\ \texttt{if}((l_1 \in \texttt{Loc}), f(l_1)(s_1), \perp_{\texttt{Store}})$$

Funzioni Ausiliarie

$\in \texttt{Loc} : \texttt{Den} \to \texttt{TruthV}$

6.1.4 Trasmissione per Costante.

È una forma di trasmissione one-way in ingresso, concepita per generalizzare costanti presenti nel codice da astrarre. L'astrazione così creata può essere utilizzata in punti diversi del programma invocandola in ciascun punto con i valori che vogliamo siano usati per tali costanti in tale uso dell'astrazione.

Table11.4 – Procedure 2 : Trasmissione per Costante

Funzioni Semantiche

$\mathcal{D}[\![\mathtt{D}]\!]_\rho : (\mathtt{Env} \times \mathtt{Store}) \to (\mathtt{Env} \times \mathtt{Store}_\perp)$ $\quad$ (*Significato Dichiarazioni*)

$$\mathcal{D}[\![\mathtt{Proc\ I(byConst\ I_1)\ C}]\!]_\rho = \mathtt{Let}\{f = \lambda\, v\,.\mathtt{Let}\{\rho_1 = \mathtt{bind}(\mathtt{I}_1, \mathtt{L}(v), \rho)\}\ \mathtt{in}\ \mathcal{M}[\![\mathtt{C}]\!]_{\rho_1}\}\ \mathtt{in}\ \mathit{bind}(\mathtt{I}, f, \rho)$$

$\mathcal{C}[\![\mathtt{C}]\!]_\rho : \mathtt{Env} \to \mathtt{State} \to \mathtt{State}_\perp$ $\quad$ (*Significato Comandi*)

$$\mathcal{C}[\![\mathtt{Call\ I(AConst(E))}]\!]_\rho(s) = \mathtt{Let}\{(v_1, s_1) =_{\perp_S} \mathcal{E}[\![\mathtt{E}]\!]_\rho(s),\ f = \rho(\mathtt{I})\}\ \mathtt{in}\ \mathtt{if}((v_1 \in \mathtt{Den}), f(v_1)(s_1), \perp_{\mathtt{Store}})$$

Funzioni Ausiliarie

$\mathtt{L} : \mathtt{Loc} \to \mathtt{Den}$ $\quad$ (*iniettore, i.e. costruttore*)

$\in \mathtt{Loc} : \mathtt{Den} \to \mathtt{TruthV}$

6.1.5 Trasmissione per Risultato.

È una forma di trasmissione one-way in uscita, concepita per generalizzare variabili presenti nel codice da astrarre. Queste variabili sono in genere, utilizzate in tale codice come variabili di accumulazione di calcoli parziali e al termine dell'esecuzione del codice contengono il valore totale (o uno dei valori totali, se più d'uno è) calcolato dall'astrazione. L'astrazione così creata può essere utilizzata in punti diversi del programma invocandola, in ciascun punto, con le variabili nelle quali vogliamo siano copiati i valori alla fine, calcolati per le variabili generalizzate.

Table11.5 – Procedure 2 : Trasmissione per Risultato

Funzioni Semantiche

$\mathcal{D}[\![\mathtt{D}]\!]_\rho : (\mathtt{Env} \times \mathtt{Store}) \to (\mathtt{Env} \times \mathtt{Store}_\perp)$ $\quad$ (*Significato Dichiarazioni*)

$$\mathcal{D}[\![\mathtt{Proc\ I(byResult\ I_1)\ C}]\!]_\rho = \mathtt{Let}\{f = \lambda\, v\,.\lambda\, s\,.\mathtt{Let}\{(l_1, s_1) = \mathit{allocate}(s)\} \{s_2 = \mathtt{upd}(l_1, \perp_{\mathtt{Mem}}, s_1),\ \rho_1 = \mathtt{bind}(\mathtt{I}_1, l_1, \rho)\}\ \mathtt{in}\ (\mathcal{M}[\![\mathtt{C}]\!]_{\rho_1} \circ (\lambda u\,.\mathtt{upd}(v, \mathtt{look}(l_1, u), u)))(s_2)\ \mathtt{in}\ \mathit{bind}(\mathtt{I}, f, \rho)$$

$\mathcal{C}[\![\mathtt{C}]\!]_\rho : \mathtt{Env} \to \mathtt{State} \to \mathtt{State}_\perp$ $\quad$ (*Significato Comandi*)

$$\mathcal{C}[\![\mathtt{Call\ I(Den(E))}]\!]_\rho(s) = \mathtt{Let}\{(l_1, s_1) =_{\perp_S} \mathcal{E}[\![\mathtt{E}]\!]_\rho(s),\ f = \rho(\mathtt{I})\}\ \mathtt{in}\ \mathtt{if}((l_1 \in \mathtt{Loc}), f(l_1)(s_1), \perp_{\mathtt{Store}})$$

6.1.6 Trasmissione per Valore-Risultato.

Permette di combinare le caratteristiche della trasmissione per valore con quelle della trasmissione per risultato. Otteniamo il meccanismo di trasmissione two-way descritto sotto.

Table11.6 – Procedure 2 : Trasmissione per Valore – Risultato

Funzioni Semantiche

$\mathcal{D}[\![\texttt{D}]\!]_\rho : (\texttt{Env} \times \texttt{Store}) \to (\texttt{Env} \times \texttt{Store}_\perp)$ $\quad$ (*Significato Dichiarazioni*)

$$\mathcal{D}[\![\texttt{Proc I(byValueResult I}_1\texttt{) C}]\!]_\rho =$$
$$\texttt{Let}\{f = \lambda\, v\,.\lambda\, s\,.$$
$$\texttt{Let}\{(l_1, s_1) = \mathit{allocate}(s)\}$$
$$\{s_2 = \texttt{upd}(l_1, \texttt{look}(v, s_1), s_1),\ \rho_1 = \texttt{bind}(\texttt{I}_1, l_1, \rho)\}$$
$$\texttt{in}\ \ (\mathcal{M}[\![\texttt{C}]\!]_{\rho_1} \circ (\lambda u\,.\texttt{upd}(v, \texttt{look}(l_1, u), u)))(s_2)$$
$$\texttt{in}\ \mathit{bind}(\texttt{I}, f, \rho)$$

$\mathcal{C}[\![\texttt{C}]\!]_\rho : \texttt{Env} \to \texttt{State} \to \texttt{State}_\perp$ $\quad$ (*Significato Comandi*)

$$\mathcal{C}[\![\texttt{Call I(Den(E))}]\!]_\rho(s) =$$
$$\texttt{Let}\{(l_1, s_1) =_{\perp_S} \mathcal{E}[\![\texttt{E}]\!]_\rho(s),\ f = \rho(\texttt{I})\}$$
$$\texttt{in if}((l_1 \in \texttt{Loc}), f(l_1)(s_1), \perp_{\texttt{Store}})$$

Nella sezione, abbiamo limitato la presentazione della semantica della trasmissione dei parametri alle sole procedure: Dichiarazione e invocazione di procedura, ovvero astrazioni imperative. Tuttavia, alcuni linguaggi prevedono in aggiunta, astrazioni funzionali e in questo caso abbiamo dichiarazioni e invocazioni di funzioni che possono prevedere differenti tipi di trasmissione dei parametri.

Esercizio 63 *Fornire le corrispondenti semantiche per la trasmissione dei parametri in funzioni, ovvero si dia la semantica della dichiarazione ed invocazione di funzione per i diversi tipi di trasmissione esaminati. A questo scopo, si considerino dichiarazioni di funzione strutturate, ovvero definite in modo strutturato. 1) Allo scopo si usino funzioni il cui corpo sia definito come nel linguaggio Pascal. Oppure, 2) il corpo sia un blocco espressione formato da una sequenza, anche vuota di comandi, e terminante con un'espressione che calcola il valore calcolato dall'invocazione della funzione. In conclusione, non si utilizzi la forma non strutturata utilizzata dal linguaggio C (vedi Sezione 6.3)*

6.2 Higher Order: Trasmissione di Valori Funzione

Questa forma di trasmissione é tipica dei Linguaggi Funzionali, o comunque dei linguaggi che permettono programmazione Higher Order in cui le funzioni sono a tutti gli effetti valori del linguaggio e come tali utilizzabili nei programmi non solo come valori denotabili, pure come valori esprimibili, quindi calcolabili come risultato della valutazione di un'espressione e secondo i casi, perfino memorizzabili. È proprio in questo uso delle funzioni come valori esprimibili che troviamo la sostanziale differenza con un tipo di

trasmissione che abbiamo già esaminato, di cui abbiamo già dato la semantica. Parliamo ovviamente, della trasmissione per nome che ha un'espressività, e quindi un uso, nelle metodologie di programmazione, che si avvicina alla trasmissione di valori funzione ma se ne distingue perchè: 1) la struttura trasmessa è codice invece che una funzione; 2) ogni occorrenza del parametro formale è testualmente rimpiazzata dal codice, invece che sostituita da un'invocazione della funzione con, possibilmente differenti, argomenti.

Table12 – Procedure 3 : Trasmissione di Valori Funzione

Domini Sintattici

$\texttt{E} ::= ... \mid \texttt{A} \mid \texttt{Lambda}(\texttt{P}_1\ \texttt{I}_1\ ...\ \texttt{P}_n\ \texttt{I}_n)\ \texttt{E} \mid ...$ $\quad (Espressioni : Astrazione)$

Domini Sintattici Ausiliari

$\texttt{P} ::= ... \mid \texttt{Fun I}$ $\quad (Parametri\ Formali)$

$\texttt{A} ::= ... \mid \texttt{Fun(E)}$ $\quad (Parametri\ Attuali)$

I valori funzionale sono introdotti attraverso espressioni chiamate *lambda astrazioni* che estendono il dominio delle espressioni in Table12. La trasmissione di funzione è invece, indicata mediante il costrutto `Fun` utilizzato per semplicità, tanto nei parametri formali, dove è seguito dal nome del parametro[1], quanto nei parametri attuali, dove è seguito da un'espressione, possibilmente una lambda astrazione o comunque un'espressione che calcoli un valore funzionale.

Table12.1 – Trasmissione Valori Funzione : Deep Binding con Scoping Statico

Funzioni Semantiche

$\mathcal{D}[\![\texttt{D}]\!]_\rho : (\texttt{Env} \times \texttt{Store}) \to (\texttt{Env} \times \texttt{Store}_\perp)$ $\quad (Significato\ Dichiarazioni)$

$$\mathcal{D}[\![\texttt{Proc I(Fun I}_1\texttt{) C}]\!]_\rho = \texttt{Let}\{f = \lambda v.\texttt{Let}\{\rho_1 = \texttt{bind}(\texttt{I}_1, \texttt{F}(v), \rho)\}\ \texttt{in}\ \mathcal{M}[\![\texttt{C}]\!]_{\rho_1}\}\ \texttt{in}\ bind(\texttt{I}, f, \rho)$$

$\mathcal{C}[\![\texttt{C}]\!]_\rho : \texttt{Env} \to \texttt{State} \to \texttt{State}_\perp$ $\quad (Significato\ Comandi)$

$$\mathcal{C}[\![\texttt{Call I(Fun(E))}]\!]_\rho(\texttt{s}) = \texttt{Let}\{g = \mathcal{E}[\![\texttt{E}]\!]_\rho(\texttt{s}), f = \rho(\texttt{I})\}\ \texttt{in}\ \texttt{if}((g \in \texttt{Fun}), f(g)(\texttt{s}), \perp_{\texttt{State}})$$

Domini Ausiliari

$\texttt{Den} ::= \texttt{Loc} + \texttt{ProcFun} + \texttt{LV} + \texttt{Code} + \texttt{Fun}$ $\quad (Unione\ Disgiunta)$

$\texttt{Fun} ::= \texttt{State} \to \texttt{State}$ $\quad (Valori\ Funzione)$

Funzioni Ausiliarie

$\texttt{F} : \texttt{Fun} \to \texttt{Den}$ $\quad (iniettiva, i.e.\ costruttore)$

$\in \texttt{Fun} : \texttt{Val} \to \texttt{TruthV}$ $\quad (iniettiva, i.e.\ costruttore)$

[1] ricordiamo che questa è la sintassi astratta purgata dei tipi, possibilmente già controllati

Considerazioni sull'implementazione.

Table12.2 – Trasm. Valori Funzione : Shallow Binding con Scoping Dinamico

Funzioni Semantiche

$\mathcal{D}[\![\mathtt{D}]\!]_\rho : (\mathtt{Env} \times \mathtt{Store}) \to (\mathtt{Env} \times \mathtt{Store}_\perp)$ $\quad$ (*Significato Dichiarazioni*)

$$\mathcal{D}[\![\mathtt{Proc\ I(Fun\ I_1)\ C}]\!]_\rho = \mathtt{Let}\{f = \lambda\rho.\lambda v.\mathtt{Let}\{\rho_1 = \mathtt{bind}(\mathtt{I}_1, \mathtt{F}(v), \rho)\}\ \mathtt{in}\ \mathcal{M}[\![\mathtt{C}]\!]_{\rho_1}\}\ \mathtt{in}\ \mathit{bind}(\mathtt{I}, f, \rho)$$

$\mathcal{C}[\![\mathtt{C}]\!]_\rho : \mathtt{Env} \to \mathtt{State} \to \mathtt{State}_\perp$ $\quad$ (*Significato Comandi*)

$$\mathcal{C}[\![\mathtt{Call\ I(Fun(E))}]\!]_\rho(\mathtt{s}) = \mathtt{Let}\{g = \mathcal{E}[\![\mathtt{E}]\!]_\rho(\mathtt{s}), f = (\rho(\mathtt{I}))(\rho)\}\ \mathtt{in}\ \mathtt{if}((g \in \mathtt{Fun}), f(g(\rho))(\mathtt{s}), \perp_{\mathtt{State}})$$

Domini Ausiliari

$\mathtt{Den} ::= \mathtt{Loc} + \mathtt{ProcFun} + \mathtt{LV} + \mathtt{Code} + \mathtt{Fun}$ $\quad$ (*Unione Disgiunta*)

$\mathtt{Fun} ::= \mathtt{State} \to \mathtt{State}$ $\quad$ (*Valori Funzione*)

Funzioni Ausiliarie

$\mathtt{F} : \mathtt{Fun} \to \mathtt{Den}$ $\quad$ (*iniettiva, i.e. costruttore*)

$\in \mathtt{Fun} : \mathtt{Val} \to \mathtt{TruthV}$ $\quad$ (*iniettiva, i.e. costruttore*)

Esercizio 64 *Si discutano le difficoltà di definire un* deep binding *con* scoping dinamico*. In particolare (a) Si dica come potrebbe essere la denotazione di una funzione che nelle invocazioni è* chiusa *con l'ambiente in cui è invocata e nella trasmissione, come parametro, è* chiusa *con l'ambiente in cui è stata definita; (b) Se ne esprima la semantica denotazionale.*

6.3 I Comandi Non Strutturati: Continuation Based Semantics

Per formalizzare il significato dei trasferimenti di controllo espressi attraverso comandi non strutturati in un linguaggio di programmazione con(-tenente anche) comandi strutturati, ricorriamo all'uso di un ulteriore dominio semantico. Il dominio delle continuazioni con le quali possiamo esprimere funzioni che dicono come calcola il resto del programma, a cui dobbiamo trasferire il controllo, ovvero passare lo stato corrente. L'impiego di trasferimenti di controllo è molto diffuso nei linguaggi (imperativi) dove è anche impiegato nelle astrazioni funzionali per introdurre più punti di terminazione. Ad esempio, nel linguaggi C/C++, Java, il costrutto `return E` esprime il valore calcolato dall'invocazione (tale valore sarà il valore calcolato dall'espressione `E`) e la terminazione dell'invocazione stessa. Ovviamente, nel corpo di una (procedura o) funzione possono occorrere più statements `return`. La Table13 fornisce la semantica denotazionale a continuazione, nel caso di un linguaggio con comandi strutturati: La dichiarazione di procedura e funzione (e relativa invocazione), il comando composto e il comando `return` per terminare, in modo non strutturato, l'invocazione di procedura o funzione. Il corpo delle funzioni ora, ha struttura come in C, ben diversa da quella introdotta in Table11.

Table13 – Semantica con Continuazioni – 4 : return

Domini Sintattici

$\texttt{D} ::= ... \mid \texttt{Function I() C} \mid \texttt{Proc I() C} \mid ... \quad (Funzioni\ e\ Procedure)$
$\texttt{C} ::= ... \mid \texttt{C C} \mid \texttt{Call I()} \mid \texttt{return} \mid \texttt{return E} \mid ... \quad (Comandi)$
$\texttt{E} ::= ... \mid \texttt{CallE I()} \mid ... \quad (Espressioni : Invocazione)$

Domini Semantici

$\texttt{Env}, \rho, \delta, \gamma \equiv \texttt{I} \rightarrow \texttt{Den} \quad (Ambienti)$
$\texttt{Store}, \texttt{s} \equiv \texttt{Loc} \rightarrow \texttt{Mem} \quad (Memoria)$
$\texttt{State} \equiv \texttt{Store} \quad (Stato)$
$\texttt{Contn}, \psi, \eta \equiv \texttt{State} \rightarrow \texttt{State} \quad (Continuation\ per\ Comandi)$
$\texttt{Contn}_e, \psi_e, \eta_e \equiv (\texttt{Val} \times \texttt{State}) \rightarrow (\texttt{Val} \times \texttt{State})$
$(Continuation\ per\ Espressioni)$

Funzioni Semantiche

$\mathcal{D} : \texttt{D} \rightarrow \texttt{Env} \rightarrow \texttt{Env} \quad (Significato\ delle\ Dichiarazioni)$
$\mathcal{M} : \texttt{C} \rightarrow \texttt{Env} \rightarrow \texttt{Cnt} \rightarrow \texttt{State} \rightarrow \texttt{State} \quad (Significato\ dei\ Comandi)$
$\mathcal{E} : \texttt{E} \rightarrow \texttt{Env} \rightarrow \texttt{Contn}_e \rightarrow \texttt{State} \rightarrow \texttt{Val} \quad (Significato\ delle\ Espressioni)$

Definizioni

$\mathcal{D}[\![\texttt{D}]\!]_\rho : \texttt{Env} \quad (Dichiarazioni)$

$\mathcal{D}[\![\texttt{Proc I() C}]\!]_\rho = \quad (Procedura\ 0-argomenti)$
$\texttt{Let}\{f = \lambda\psi.\texttt{Let}\{\delta = \texttt{bind}(\&\texttt{Cont}, \texttt{G}(\psi), \rho)\}\texttt{in}\ \mathcal{M}[\![\texttt{C}]\!]_{\delta,\psi}\}$
$\texttt{in bind}(\texttt{I}, \texttt{Q}(f), \rho)$
$\mathcal{D}[\![\texttt{Function I() C}]\!]_\rho = \quad (Funzione\ 0-argomenti)$
$\texttt{Let}\{f = \lambda\psi_e.\texttt{Let}\{\delta = \texttt{bind}(\&\texttt{Cont}, \texttt{U}(\psi_e), \rho)\}\texttt{in}\ \mathcal{M}[\![\texttt{C}]\!]_{\delta,\texttt{U}(\psi_e)}\}$
$\texttt{in bind}(\texttt{I}, \texttt{F}(f), \rho)$

$\mathcal{C}[\![\texttt{C}]\!]_{\rho,\psi} : \texttt{State} \rightarrow \texttt{State}_\perp \quad (Comandi)$

$\mathcal{M}[\![\texttt{C}_1\ \texttt{C}_2]\!]_{\rho,\psi} = \texttt{Let}\{\eta = \mathcal{M}[\![\texttt{C}_2]\!]_{\rho,\psi}\}\ \texttt{in}\ \mathcal{M}[\![\texttt{C}_1]\!]_{\rho,\texttt{G}(\eta)}$
$\mathcal{M}[\![\texttt{Call I()}]\!]_{\rho,\psi} = \texttt{Let}\{\texttt{Q}(g) = \rho(\texttt{I})\}\ \texttt{in}\ g(\psi)$
$\mathcal{M}[\![\texttt{return}]\!]_{\rho,\psi} = \texttt{Let}\{\texttt{G}(h) = \rho(\&\texttt{Cont})\}\ \texttt{in}\ h$
$\mathcal{M}[\![\texttt{return E}]\!]_{\rho,\psi} = \texttt{Let}\{\texttt{U}(h) = \rho(\&\texttt{Cont})\}\ \texttt{in}\ \mathcal{E}[\![\texttt{E}]\!]_{\rho,h}$

$\mathcal{E}[\![\texttt{E}]\!]_{\rho,\psi_e} : \texttt{State} \rightarrow (\texttt{Val} \times \texttt{State}_\perp) \quad (Espressioni)$

$\mathcal{E}[\![\texttt{CallE I()}]\!]_{\rho,\psi_e} = \texttt{Let}\{\texttt{F}(g) = \rho(\texttt{I})\}\ \texttt{in}\ g(\psi_e)$

Domini Ausiliari

$\texttt{Cnt} ::= \texttt{Contn}_e + \texttt{Contn} \quad (Unione\ Disgiunta)$
$\texttt{Den} ::= \texttt{Loc} + \texttt{ProcFun} + \texttt{LV} + \texttt{Code} + \texttt{Fun} + \texttt{Cnt} \quad (Unione\ Disgiunta)$
$\&\texttt{Var} ::= \{\&\texttt{Cont}, ...\} \quad (Identificatori\ Riservati)$
$\texttt{GCode} ::= \texttt{State} \rightarrow \texttt{State}$

Funzioni Ausiliarie

$\texttt{U} : \texttt{Contn}_e \rightarrow \texttt{Cnt}$
$\texttt{G} : \texttt{Contn} \rightarrow \texttt{Cnt}$

Il linguaggio C/C++ non pone distinzione tra procedure e funzioni nel senso che entrambe condividono la stessa struttura sintattica e si distinguono solo per il tipo del valore calcolato (`void` nel caso di procedura) ed ovviamente per l'uso, come comando o espressione, delle rispettive invocazioni. Ciò è mostrato in Table13, dove si vede anche come procedure e funzioni, in aggiunta alla terminazione non strutturata `return`, mantengano anche la terminazione strutturata (vedi Esercizio 67)

6.3.1 Implementazione: Continuations e Link di catena dinamica

Le continuation introducono una nuova struttura semantica ma offrono spesso una chiara corrispondenza con l'implementazione della MA (macchina astratta) di un linguaggio a blocchi. Come sappiamo, l'implementazione ricorre all'uso di AR che includono il codice da eseguire (un link o un location counter ad esso), il frame delle locali, il link di catena statica per le non locali (con scoping statico), il link di catena dinamica per restituire il controllo all'AR "chiamante" quando il corrente AR ha terminato di eseguire il proprio codice. Il termine "chiamante" indica AR di blocchi in-line o di procedure o funzioni che contengono invocazioni di procedure o funzioni, oppure contengono altri blocchi o costrutti la cui esecuzione richiede che la MA crei un nuovo AR. L'effettiva struttura dell'AR di una MA dipende comunque dalla effettiva scelta fatta per la realizzazione della MA, ovvero per la MC, macchina concreta, che farà da supporto alla MA. Sappiamo le tre scelte possibili: compilazione pura, interpretazione pura, mista (compilazione su una macchina virtuale, supportata da un interprete su una MC). Le scelte incidono essenzialmente sul ciclo di interpretazione della MA e sulla conseguente struttura del suo codice.

- **Compilazione**. Nel caso di compilazione pura, il codice richiede un ciclo di interpretazione estremamente semplice. L'assegnamento $\mathtt{E}_1 = \mathtt{E}_2$, ad esempio, è già stato, come sappiamo, tradotto in una sequenza di operazioni a 3 indirizzi, ad esempio, contenente la traduzione dell'espressione denotabile $\mathtt{E}_1$, seguita dalla traduzione dell'espressione memorizzabile $\mathtt{E}_2$, seguita dall'operazione di update della locazione ottenuta dalla traduzione di $\mathtt{E}_1$ (e corrispondente alla denotazione calcolata dall'esecuzione della traduzione di $\mathtt{E}_1$) con il contenuto della locazione ottenuta dalla traduzione di $\mathtt{E}_2$ (e corrispondente al valore calcolato dall'esecuzione della traduzione di $\mathtt{E}_2$). Di contro, essa utilizza AR solo per separare ambienti (di identificatori) diversi: blocchi in-line o di procedure/funzioni o comunque di strutture che introducono ambienti (di identificatori) (ad esempio, oggetti di un linguaggio OO). Questo significa che le continuazioni sono implementate come link di catena dinamica solo quando occorrono (1) nell'accesso di un nuovo blocco in-line, (2) nell'invocazione di procedura/funzione, (3) istanziazione di struttura che introduce nuovo ambiente. In tutti gli altri casi, la continuazione è implementata come il location counter del corrente AR.

- **Interpretazione**. Nel caso di interpretazione pura, il codice è quello del linguaggio sorgente che, se ad alto livello, richiede un ciclo di interpretazione che può essere estremamente complicato. Nei linguaggi strutturati, uno statement può avere una struttura che richiama altri statements, anche dello stesso tipo (ad esempio, un `while` può contenere altri `while` annidati nel suo comando) ed avere una struttura di profondità arbitraria. La profondità della struttura si riflette sulla profondità del ciclo di interpretazione e sulla necessit di mantenere

informazioni sul controllo di cosa stiamo interpretando e come questo è collegato all'intepretazione in corso e di profondità inferiore. Tutto questo conduce ad utilizzare gli AR per gestire il controllo del ciclo di intepretazione ovvero richiede di generare un nuovo AR anche nell'accesso a operazioni "primitive": il nuovo AR controllerà che le strutture componenti sia interpretate e adeguatamente valutate prima di procedere all'applicazione dell'operazionela. Questo significa che le continuazioni sono implementate come link di catena dinamica praticamente sempre: ai punti (1)-(3) della compilazione, aggiungiamo il seguente: (4) accesso a operazioni del linguaggio.

Un esempio della diversa struttura degli AR nel caso di compilazione e di interpretazione si può trovare nell'Esempio 6. In tale esempio si mostrano (vedi 3.1-3.4) le continuazioni nel caso della semantica dell'assegnamento. Ovviamente nel caso di compilazione, tali continuazioni sono implementate come modifiche del location counter dell'AR in cui occorre (il codice compilato per) tale assegnamento. Nel caso di interpretazione pura invece, ciascuna delle continuazione riportate da origine ad un link di catena dinamica ad un AR contenente come codice da eseguire (quindi interpretare) il codice di cui la continuazione è la semantica.

Esempio 6 *Consideriamo il seguente frammento, dove abbiamo evidenziato, ricorrendo alla sintassi astratta generata dal compilatore, la vera struttura dei termini coinvolti nelle strutture di interesse per gli scopi dell'esempio. Mostriamo le coppie* $(\rho, \mathtt{s})$*, calcolate dalla semantica dopo ogni dichiarazione o comando del blocco principale.*
Applichiamo la semantica al seguente frammento di programma:

```
(1)   {... int x; int y;
(2)      Function A(){
          if (x==0) return y;
          if (y==0) return x;
          x=x+y;
          return x;
          }
(3)   x=3; y=2;
(4)   Den(x)=Val(x)+A(); ...
```

Mostriamo le triple $(\rho, \psi, \mathtt{s})$*, calcolate dalla semantica dopo ogni dichiarazione o comando del blocco principale.*
(1) $\rho_1 \equiv [l_{\mathtt{y}}/\mathtt{y}][l_{\mathtt{x}}/\mathtt{x}]\rho$, $\psi_1 \equiv \psi$, $\mathtt{s}_1 \equiv [l_{\mathtt{y}} \leftarrow \perp_{int}][l_{\mathtt{x}} \leftarrow \perp_{int}]\mathtt{s}$
(2) $\rho_2 \equiv [f/\mathtt{A}]\rho_1$, $\psi_2 \equiv \psi$, $\mathtt{s}_2 \equiv \mathtt{s}_1$ *where:*

$$f \equiv \lambda\psi_e.\mathtt{Let}\{\delta = [\mathtt{Q}(\psi_e)/\&\mathtt{Count}]\rho_1\}$$
$$\{\eta_1 = \mathtt{Let}\{\eta_2 = \mathtt{Let}\{\eta_3 = \mathcal{M}[\![\mathtt{return\ x}]\!]_{\delta,\psi_e}\}$$
$$\mathtt{in}\ \mathcal{M}[\![\mathtt{x = x + y}]\!]_{\delta,\eta_3}\}$$
$$\mathtt{in}\ \mathcal{M}[\![\mathtt{if(y == 0)return\ x}]\!]_{\delta,\eta_2}\}$$
$$\mathtt{in}\ \mathcal{M}[\![\mathtt{if(x == 0)return\ y}]\!]_{\delta,\eta_1}$$

(3) $\rho_3 \equiv \rho_2$, $\psi_3 \equiv \mathcal{M}[\![\mathtt{x = x + CallE\ A()}]\!]_{\rho_2,\psi}$, $\mathtt{s}_3 \equiv [l_{\mathtt{y}} \leftarrow 2][l_{\mathtt{x}} \leftarrow 3]\mathtt{s}$
(4) $\rho_4 \equiv \rho_2$, $\psi_4 \equiv \psi$, $\mathtt{s}_4 \equiv [l_{\mathtt{y}} \leftarrow 2][l_{\mathtt{x}} \leftarrow 8]\mathtt{s}$
Il calcolo di $\mathtt{s}_4$ *conclude l'esempio che mostra l'uso delle continuazioni nel significato dei vari costrutti. In particolare, nell'implementazione dei linguaggi, le continuazioni sono i link di catena dinamica degli AR. A questo scopo completiamo l'esempio, mostrando*

l'uso delle continuazioni nella semantica $\mathcal{M}[\![\mathtt{x} = \mathtt{x} + \mathtt{CallE\ A}()]\!]_{\rho_2,\psi}$ *nel punto (3) sopra. Esaminiamo la "catena" di continuazioni che è creata dalla semantica: Allo scopo usiamo ancora le triple e provvediamo ad avere ridefinito, sotto forma di continuation semantics, la semantica dell'assegnamento data in Table 9-10 (si veda l'Esercizio 65 punto (b)):*
Da $\mathcal{M}[\![\mathtt{x} = \mathtt{x} + \mathtt{CallE\ A}()]\!]_{\rho_2,\psi}$, *otteniamo:*
(3.1) $\rho_{3.1} \equiv \rho_2$, $\psi_{3.1} \equiv \mathcal{E}[\![\mathtt{Den(x)}]\!]_{\rho_2,\psi_{3.2}}$, $\mathtt{s}_{3.1} \equiv \mathtt{s}_3$
(3.2) $\rho_{3.2} \equiv \rho_2$, $\psi_{3.2} \equiv \lambda(l_{3.2}, \mathtt{s}_{3.2}).\mathcal{E}[\![\mathtt{Val(x)}]\!]_{\rho_2,\psi_{3.3}}(\mathtt{s}_{3.2})$, $\mathtt{s}_{3.2} \equiv \mathtt{s}_3$
(3.3) $\rho_{3.3} \equiv \rho_2$, $\psi_{3.3} \equiv \lambda(v_{3.3}, \mathtt{s}_{3.3}).\mathcal{E}[\![\mathtt{CallE\ A}()]\!]_{\rho_2,\psi_{3.4}}()$, $\mathtt{s}_{3.3} \equiv \mathtt{s}_3$
(3.4) $\rho_{3.4} \equiv \rho_2$, $\psi_{3.4} \equiv \lambda(v_{3.4}, \mathtt{s}_{3.4}).\psi(\mathtt{upd}(l_{3.2}, v_{3.3}+v_{3.4}, \mathtt{s}_{3.4}))$, $\mathtt{s}_{3.4} \equiv [l_{\mathtt{y}} \leftarrow 2][l_{\mathtt{x}} \leftarrow 5]\mathtt{s}$
□

Esercizio 65 *Si dia la semantica denotazionale a continuazioni del comando di assegnamento: (a) La semantica data segua quella data in Table9-10 dove prima si valuta l'espressione memorizzabile e solo dopo, quella denotabile; (b) La semantica data proceda, come in C, da sinistra a destra, valutando prima l'espressione denotabile, poi l'espressione memorizzabile, ed infine applichi l'update. (c) Si mostri un frammento di programma dove le due semantiche, date sopra, differiscono tra loro, discutendo dove e perchè.* □

Esercizio 66 *Si consideri il programma dell'Esempio 6. Si mostrino le triple* $(\rho, \psi, \mathtt{s})$ *definite dalla semantica durante la valutazione del corpo della procedura* `A` *nell'invocazione de punto (4)* □
$\rho_3 \equiv [f/\mathtt{A}][l_{\mathtt{y}}/\mathtt{y}][l_{\mathtt{x}}/\mathtt{x}]\rho$, $\psi_3 \equiv \mathcal{M}[\![\mathtt{x} = \mathtt{x} + u_4]\!]_{\rho_2,\psi}$, $\mathtt{s}_3 \equiv [l_{\mathtt{y}} \leftarrow 2][l_{\mathtt{x}} \leftarrow 3]\mathtt{s}$,
dove: $(u_4, \mathtt{s}_4) = \mathcal{E}[\![\mathtt{CallE\ A}()]\!]_{\rho_3,\eta_1}$

Esercizio 67 *Osservando la definizione di* $\mathcal{D}[\![\mathtt{Proc\ I()\ C}]\!]_\rho$ *si dica dove è mantenuta l'informazione per la terminazione non strutturata e dove quella per la terminazione struttura dell'invocazione di procedura.*

Esercizio 68 *Osservando la definizione di* $\mathcal{M}[\![\mathtt{C}_1\ \mathtt{C}_2]\!]_{\rho,\psi}$ *e confrontandola con quella data in Table10, si dica (a) cosa esprime la continuazione in questa semantica, (b - difficile) come potremmo mostrare che le due semantiche sono equivalenti allorchè il linguaggio contenga solo costrutti strutturati.*

Potremmo estendere la semantica data in Table13 al caso di comandi etichettati e al comando `goto`, e allo stesso tempo semplificare l'uso di continuazioni, prevedendo un solo tipo di continuazioni e parimenti un solo tipo di codice. Questo è fatto in Table13.1.

Table13.1 – Semantica con continuazioni dei Comandi – 5 : Goto

Domini Sintattici

$\texttt{D} ::= ... \mid \texttt{Function I() C} \mid \texttt{Proc I() C} \mid ...$ $(Funzioni\ e\ Procedure)$
$\texttt{C} ::= \texttt{C C} \mid \texttt{Call I()} \mid \texttt{Label(I)} : \texttt{C} \mid \texttt{goto Label(I)} \mid$
$\quad \mid \texttt{E} = \texttt{E} \mid \texttt{return} \mid ...$ $(Comandi)$

Domini Semantici

$\texttt{Env}, \rho, \delta, \gamma \equiv \texttt{I} \rightarrow \texttt{Den}$ $(Ambienti)$
$\texttt{Store}, \texttt{s} \equiv \texttt{Loc} \rightarrow \texttt{Mem}$ $(Memoria)$
$\texttt{State} \equiv \texttt{Store}$ $(Stato)$
$\texttt{Contn}, \psi, \eta \equiv \texttt{Env} \rightarrow \texttt{State} \rightarrow \texttt{State}$ $(Continuation)$

Funzioni Semantiche

$\mathcal{D} : \texttt{D} \rightarrow \texttt{Env} \rightarrow \texttt{Env}$ $(Significato\ delle\ Dichiarazioni)$
$\mathcal{M} : \texttt{C} \rightarrow \texttt{Env} \rightarrow \texttt{Contn} \rightarrow \texttt{State} \rightarrow \texttt{State}$ $(Significato\ dei\ Comandi)$
$\mathcal{E} : \texttt{E} \rightarrow \texttt{Env} \rightarrow \texttt{Contn} \rightarrow \texttt{State} \rightarrow \texttt{Val}$ $(Significato\ delle\ Espressioni)$

Definizioni

$\mathcal{D}[\![\texttt{D}]\!]_\rho : \texttt{Env}$ $(Dichiarazioni)$

$\mathcal{D}[\![\texttt{Proc I() C}]\!]_\rho =$ $(Procedura\ 0-argomenti)$
$\quad \texttt{Let}\{f = \lambda\psi.\texttt{Let}\{\delta = \texttt{bind}(\&\texttt{Cont}, \texttt{H}(\psi(\rho)), \rho)\}\texttt{in}\ \mathcal{M}[\![\texttt{C}]\!]_{\delta,\psi}\}$
$\quad \texttt{in bind}(\texttt{I}, f, \rho)$

$\mathcal{C}[\![\texttt{C}]\!]_{\rho,\psi} : \texttt{State} \rightarrow \texttt{State}_\perp$ $(Comandi)$

$\mathcal{M}[\![\texttt{C}_1\ \texttt{C}_2]\!]_{\rho,\psi} = \texttt{Let}\{\eta = \lambda\delta.\mathcal{M}[\![\texttt{C}_2]\!]_{\delta,\psi}\}\ \texttt{in}\ \mathcal{M}[\![\texttt{C}_1]\!]_{\rho,\eta}$
$\mathcal{M}[\![\texttt{Label(I)} : \texttt{C}]\!]_{\rho,\psi} = \texttt{Let}\{\gamma = \texttt{Y}\delta.\texttt{bind}(\texttt{I}, \texttt{H}(\mathcal{M}[\![\texttt{C}]\!]_{\delta\circ\rho,\psi}), \rho)\}\ \texttt{in}\ \mathcal{M}[\![\texttt{C}]\!]_{\gamma,\psi}$
$\mathcal{M}[\![\texttt{goto Label(I)}]\!]_{\rho,\psi} = \texttt{Let}\{\texttt{H}(g) = \rho(\texttt{I})\}\ \texttt{in}\ g$
$\mathcal{M}[\![\texttt{Call I()}]\!]_{\rho,\psi} = \texttt{Let}\{\texttt{Q}(g) = \rho(\texttt{I})\}\ \texttt{in}\ g$
$\mathcal{M}[\![\texttt{return}]\!]_{\rho,\psi} = \texttt{Let}\{\texttt{H}(g) = \rho(\&\texttt{Cont})\}\ \texttt{in}\ g$

Domini Ausiliari

$\texttt{Den} ::= \texttt{Loc} + \texttt{ProcFun} + \texttt{LV} + \texttt{Code} + \texttt{Fun} + \texttt{GCode}$ $(Unione\ Disgiunta)$
$\&\texttt{Var} ::= \{\&\texttt{Cont}, ...\}$ $(Identificatori\ Riservati)$
$\texttt{GCode} ::= \texttt{State} \rightarrow \texttt{State}$

Funzioni Ausiliarie

$\texttt{H} : (\texttt{State} \rightarrow \texttt{State}) \rightarrow \texttt{GCode}$

Esercizio 69 *Si dica se la semantica della dichiarazione di procedura data in Table13.1: (a) permetta di esprimere procedure definite ricorsive. (b) Si giustifichi la risposta. (c) In caso di risposta negativa si dica allora: 1) come dovrebbe essere modificata tale semantica; 2) se devono essere apporatte modifiche all'invocazione, e quali.* □

Esercizio 70 *Si estenda il dominio dei comandi con l'assegnamento e si forniscano le relative estensioni delle funzioni semantiche* □

Esercizio 71 *Si dia la semantica della dichiarazione di funzione C-like, omessa in Table13.1. Si ricordi che le funzioni C utilizzano il comando* `return` *per terminare un invocazione, trasferire il controllo al chiamante, restituire il valore calcolato* □

Esercizio 72 *Si applichino le funzioni semantiche di Table13.1 e le estensioni date nell'esercizio precedete alla semantica del seguente frammento di programma:*

```
function F(){
       if (x > y) return x;
       return y
```

Esercizio 73 *(a) Si estenda il dominio sintattico dei comandi con il blocco. (b) si dia allora, la semantica di tale costrutto.* □

Esercizio 74 *Si commenti la definizione del costrutto* `goto`*. In particolare si dica (a) perchè l'ambiente ρ non è utilizzato in g; (b) perchè la continuazione ψ non è utilizzata in g.*

Esercizio 75 *(a) Si discuta l'uso del comando* `continue` *nei linguaggi C-like. (b) Si estendano domini sintattici e funzioni semantiche per tale comando. (c) Si mostri come deve essere modificata la semantica del comando* `while`*, data in Table10, nel caso di linguaggi C-like (d) Si discuta come cambia l'implementazione del costrutto while rispetto a quella di Table10 (si rifletta sull'implementazione delle continuazioni)* □

Esercizio 76 *Si estendano domini sintattici e funzioni semantiche per il comando* `case` □ □

Esercizio 77 *(a) Si discuta l'uso del comando* `break` *nei linguaggi C-like; (b) Si estendano domini sintattici e funzioni semantiche per tale comando; (c) Si discutano le estensioni date nel precedente esercizio nel caso del linguaggio C. In particolare si dica se tale semantica è compatibile con quella richiesta dai linguaggi C-like e dalla sua interazione con il comando* `break`*; (d) e in caso negativo, la si modifichi a tale scopo* □

Esercizio 78 *Si discuta come la semantica del comando etichettato, data in Table13.1, permetta di trasferire il controllo solo all'indietro, in particolare si confronti la semantica ottenuta in ciascuno dei seguenti tre casi:*

```
caso1: ... A: x=7; y=x; goto A;...
caso2: A: (x=7; y=x; goto A);...
caso3: ...goto A; x=7; A: y=x...
```

Nel caso2, `A` *etichetta un comando composto qui, identificato, utilizzando le parentesi.*□

In Table13.1 utiliziamo un unico ambiente per gli identificatori, inclusi quelli definenti etichette. Questo va veramente bene se tutti questi identificatori hanno stesso comportamento che nel caso di identificatori significa: hanno stesse regole di visibilità, ovvero di scope. Gli identificatori di tipo, di costante, di variabile, di procedura, di

funzione seguono lo scope statico o quello dinamico. Nel caso delle etichette nessuno dei due è soddisfacente. Ma cè di più, se per gli altri identificatori l'introduzione avviene attraverso dichiarazioni, nel casi di etichette queste sono introdotte attraverso i comandi e possono occorrere ovunque occorra un comando. Questo potrebbe richiamarci alla mente lo stile delle dichiarazioni dei linguaggi C-like, dove le dichiarazioni possono occorrere ovunque occorra un comando. Ma le analogie si perdono qui. Nel caso del linguaggio C infatti, se applicassimo la regola di scope degli identificatori alle etichette, avremmo che il linguaggio permetterebbe solo salti all'indietro come discusso anche nell'esercizio sopra e come è dato dalla semantica in Table13.1. Dobbiamo quindi ammettere che i linguaggi con comandi etichettati hanno per le etichette regole di scope specifiche. Un prima possibilità è data da una regola che stabilisca che:

- (a) Lo scope di un etichetta è nel blocco in cui essa occorra come etichetta di comando, e nei blocchi annidati che non la ri-definiscano;
- (b) Lo scope di un etichetta è l'intero programma (o modulo, se i moduli sono previsti). In un programma non possono occorre più comandi etichettati da una stessa etichetta. E, in questo senso, le etichette sono globali (al modulo) sebbene siano definite ovunque nel programma. Procedure e funzioni sono trattate come moduli: Non è possibile trasferire controllo da un comando ad un comando del corpo di una procedura o funzione. Ed anche il contrario: non è possibile da un comando di una procedura o funzione traserire il conrollo ad un comando del programma.

La regola (a) è certamente più blanda, ma di difficile uso nella programmazione non strutturata come quella voluta dalla presenza dei costrutti `goto`. A titolo di esempio, questa regola impedisce il salto incondizionato da un comando di un blocco a comandi di un blocco annidato. Da un punto di vista semantico non pone particolari problemi in più rispetto alla regola (b). In tutti i casi, è richiesto il ricorso ad un ambiente specifico per le etichette. Introdurremo quindi un nuovo dominio semantico che chiameremo `Lab` e che ha una struttura che è in tutto una copia di quella dell'ambiente `Env` (vedi Table2), ma comportamento diverso nella semantica del linguaggio. Questo diverso comportamento bene evidenziato dalle differenti funzioni semantiche in cui i due tipi di ambiente sono costruiti, ovvero $\mathcal{L}$ e $\mathcal{D}$, e dal modo in cui sono coinvolti nelle altre funzioni semantiche, $\mathcal{M}$ e $\mathcal{E}$. La Table13.2 fornisce la semantica nel caso di scope delle etichette con regola (b). Allo scopo abbiamo riportato i domini delle dichiarazione di procedura e funzione (sebbene, per semplicità, senza parametri), e i domini dei comandi includenti sequenzializzazione, blocco, comando etichettato, `goto`, e vari tipi di trasferimenti non strutturati (che vedremo anche negli esercizi). I domini semantici includono il nuovo ambiente, `Lab`. Le funzioni semantiche ora includono la nuova funzione $\mathcal{L}$ ed estendono il dominio della funzione $\mathcal{M}$ ad includere il `Lab`. Commentiamo la funzione $\mathcal{L}$: Applicata, ad un comando (pissibilmente, il corpo delle'intero programma) raccoglie tutte le coppie *etichetta - codice etichettato*. Il codice etichettato deve essere *chiuso*, ovvero tuuti gli identificatori devono essere legati nell'ambiente ρ. Tutto questo si vede dalla definizione di $\mathcal{L}$, data in Table13.2.

Table13.2 – Semantica dei Comandi – 6 : Goto e i suoi Fratelli

Domini Sintattici

$\texttt{D} ::= \ldots \mid \texttt{Function I() C} \mid \texttt{Proc I() C} \mid \ldots \qquad (Funzioni\ e\ Procedure)$
$\texttt{C} ::= \texttt{C C} \mid \{\texttt{D C}\} \mid \texttt{Call I()} \mid \texttt{Label(I)} : \texttt{C} \mid \ldots \mid \texttt{goto Label(I)} \mid$
$\mid \texttt{return} \mid \texttt{return E} \mid \texttt{continue} \mid \texttt{break} \qquad (Comandi)$

Domini Semantici

$\texttt{Env}, \rho, \delta, \gamma \equiv \texttt{I} \to \texttt{Den} \qquad (Ambienti)$
$\texttt{Store}, \texttt{s} \equiv \texttt{Loc} \to \texttt{Mem} \qquad (Memoria)$
$\texttt{State} \equiv \texttt{Store} \qquad (Stato)$
$\texttt{Lab}, \alpha, \beta \equiv \texttt{I} \to \texttt{GCode} \qquad (Ambienti\ di\ Labels)$
$\texttt{Contn}, \psi, \eta \equiv \texttt{State} \to \texttt{State} \qquad (Continuation)$

Funzioni Semantiche

$\mathcal{D} : \texttt{D} \to \texttt{Env} \to \texttt{Env} \qquad (Significato\ delle\ Dichiarazioni)$
$\mathcal{L} : \texttt{C} \to \texttt{Env} \to \texttt{Lab} \to \texttt{Lab} \qquad (Significato\ delle\ Labels\ nei\ comandi)$
$\mathcal{M} : \texttt{C} \to \texttt{Env} \to \texttt{Lab} \to \texttt{Contn} \to \texttt{State} \to \texttt{State} \qquad (Sign.\ dei\ Comandi)$
$\mathcal{E} : \texttt{E} \to \texttt{Env} \to \texttt{Contn} \to \texttt{State} \to \texttt{Val} \qquad (Significato\ delle\ Espressioni)$

Definizioni

$\mathcal{D}[\![\texttt{D}]\!]_\rho : \texttt{Env} \qquad (Dichiarazioni)$

$\mathcal{D}[\![\texttt{Proc I() C}]\!]_\rho = \qquad (Procedura\ 0 - argomenti)$
$\texttt{Let}\{f = \lambda\psi.\texttt{Let}\{\delta = \texttt{bind}(\&\texttt{Cont}, \texttt{H}(\psi), \rho)\}\texttt{in}\ \mathcal{M}[\![\texttt{C}]\!]_{\delta,\texttt{empty},\psi}\}$
$\texttt{in bind}(\texttt{I}, f, \rho)$

$\mathcal{L}[\![\texttt{C}]\!]_{\rho,\beta} : \texttt{Lab} \qquad (Ambiente\ di\ labels : Costruzione)$

$\mathcal{L}[\![\texttt{C}_1\ \texttt{C}_2]\!]_{\rho,\beta} = \texttt{Y}\alpha.\mathcal{L}[\![\texttt{C}_1]\!]_{\rho,\alpha} \circ \mathcal{L}[\![\texttt{C}_2]\!]_{\rho,\alpha} \circ \beta$
$\mathcal{L}[\![\{\texttt{D C}\}]\!]_{\rho,\beta} = \mathcal{L}[\![\texttt{C}]\!]_{\rho,\beta}$
$\mathcal{L}[\![\texttt{Label(I)} : \texttt{C}]\!]_{\rho,\beta} = \texttt{Y}\alpha.\texttt{bind}(\texttt{I}, \texttt{H}(\mathcal{M}[\![\texttt{C}]\!]_{\rho,\alpha}), \beta)$
$\mathcal{L}[\![\texttt{goto Label(I)}]\!]_{\rho,\beta} = \beta$

$\mathcal{C}[\![\texttt{C}]\!]_{\rho,\beta,\psi} : \texttt{State} \to \texttt{State}_\perp \qquad (Comandi)$

$\mathcal{M}[\![\texttt{C}_1\ \texttt{C}_2]\!]_{\rho,\beta,\psi} = \texttt{Let}\{\alpha = \mathcal{L}[\![\texttt{C}_1\ \texttt{C}_2]\!]_{\rho,\beta}\}; \{\eta = \mathcal{M}[\![\texttt{C}_2]\!]_{\rho,\alpha,\psi}\}$
$\texttt{in}\ \mathcal{M}[\![\texttt{C}_1]\!]_{\rho,\alpha,\eta}$
$\mathcal{M}[\![\{\texttt{D C}\}]\!]_{\rho,\beta,\psi} = \texttt{Let}\{\delta = \mathcal{D}[\![\texttt{D}]\!]_\rho\}\ \texttt{in}\ \mathcal{M}[\![\texttt{C}]\!]_{\delta,\beta,\psi}$
$\mathcal{M}[\![\texttt{Label(I) : C}]\!]_{\rho,\beta,\psi} = \texttt{Let}\{\alpha = \mathcal{L}[\![\texttt{Label(I) : C}]\!]_{\rho,\beta}\}\ \texttt{in}\ \mathcal{M}[\![\texttt{C}]\!]_{\rho,\alpha,\psi}$
$\mathcal{M}[\![\texttt{goto Label(I)}]\!]_{\rho,\beta,\psi} = \texttt{Let}\{\texttt{H}(g) = \beta(\texttt{I})\}\ \texttt{in}\ g$
$\mathcal{M}[\![\texttt{Call I()}]\!]_{\rho,\beta,\psi} = \texttt{Let}\{\texttt{Q}(g) = \rho(\texttt{I})\}\ \texttt{in}\ g$
$\mathcal{M}[\![\texttt{return}]\!]_{\rho,\beta,\psi} = \texttt{Let}\{\texttt{H}(g) = \rho(\&\texttt{Cont})\}\ \texttt{in}\ g$

Domini Ausiliari

$\texttt{Den} ::= \texttt{Loc} + \texttt{ProcFun} + \texttt{LV} + \texttt{Code} + \texttt{Fun} + \texttt{GCode} \quad (Unione\ Disgiunta)$
$\&\texttt{Var} ::= \{\&\texttt{Cont}, ...\} \quad (Identificatori\ Riservati)$
$\texttt{GCode} ::= \texttt{State} \to \texttt{State}$

Funzioni Ausiliarie

$\texttt{H} : (\texttt{State} \to \texttt{State}) \to \texttt{GCode}$

Esercizio 79 *Si spieghi perchè in Table13.2, il dominio* `Contn` *non utilizza più il dominio* `Env` □

Esercizio 80 *Si confronti la semantica della dichiarazione di procedura data in Table13.1 con quella data in Table13.2, commentando e giustificando le differenze trovate* □

Esercizio 81 *Table13.2 ha definito i domini, in modo tale che procedure e funzioni non possano usare al loro interno etichette definite esternamente. Ed anche il contrario. Si giustifichi l'affermazione (in entrambi i versi) e si commenti la semantica definita allorchè il corpo di procedure e funzioni usi etichette. In particolare si dica cosa succede allorchè occorra nel corpo di una procedura un comando* `goto` *con target un etichetta definita esternamente alla procedura* □

Esercizio 82 *Table13.2 riporta la definizione della funzione $\mathcal{D}$ per il solo caso di dichiarazione di procedura. Si mostri la definizione di $\mathcal{D}$ nel caso di dichiarazioni di funzione* □

Esercizio 83 *Si completi la definizione della funzione semantica $\mathcal{L}$ nei casi omessi in Table13.2, motivano per ciascun caso, la formula scritta* `Y` □

Esercizio 84 *Si commenti la definizione della funzione semantica $\mathcal{L}$ nel caso della sequenza di comandi. In particolare si spieghi la presenza dell'operatore* `Y` □

Esempio 7 *Applichiamo le funzioni semantiche di Table13.2 al seguente frammento formato da una sequenza ed un blocco.*

```
(1)      Label(A): goto Label(B);
          {
(2)       Label(B): goto Label(A);
           }
```

Calcoliamo la semantica $\mathcal{M}$ in un ambiente di identificatori $\rho = \texttt{empty}$, un ambiente di etichette $\beta = \texttt{empty}$, la continuazione identica $\psi = \lambda\texttt{x.x}$, uno stato s*. Inizialmente dobbiamo applicare M alla sequenza:*

$\mathcal{M}[\![\texttt{C}_1\ \texttt{C}_2]\!]_{\rho,\beta,\psi}$
$= \texttt{Let}\{\alpha = \mathcal{L}[\![\texttt{C}_1\ \texttt{C}_2]\!]_{\rho,\beta}\}; \{\eta = \mathcal{M}[\![\texttt{C}_2]\!]_{\rho,\alpha,\psi}\}$
$\texttt{in}\ \mathcal{M}[\![\texttt{C}_1]\!]_{\rho,\alpha,\eta}$

dove $\texttt{C}_1 = \texttt{Label A: goto B}$, *e* $\texttt{C}_2 = \{\texttt{Label B: goto A}\}$.
Calcoliamo α:

$$\begin{aligned}\alpha &= \mathcal{L}[\![\texttt{C}_1\ \texttt{C}_2]\!]_{\rho,\beta}\\ &= \texttt{Y}\alpha.\mathcal{L}[\![\texttt{C}_1]\!]_{\rho,\alpha} \circ \mathcal{L}[\![\texttt{C}_2]\!]_{\rho,\alpha} \circ \beta\\ &= \texttt{Y}\alpha.[\texttt{H}(\texttt{Let H}(g1) = \alpha(B)\ \texttt{in}\ g1)/A; \texttt{H}(\texttt{Let H}(g2) = \alpha(A)\ \texttt{in}\ g2)/B]\end{aligned}$$

Calcoliamo η:

$\eta = \texttt{Let}\{\alpha_1 = \mathcal{L}[\![\texttt{Label B : goto A}]\!]_{\rho,\alpha}\}\ \texttt{in}\ \mathcal{M}[\![\texttt{goto A}]\!]_{\rho,\alpha_1,\psi}$
Calcoliamo α_1:
$\alpha_1 = \alpha$
$= \texttt{Let H}(g1) = \alpha(B)\ \texttt{in}\ g1$

Otteniamo:
$= \texttt{Let H}(g2) = \alpha(A)\ \texttt{in}\ g2$

che calcola indefinito, giacchè nell'ambiente α, $g2$ è definito ricorrendo a $g1$ e questo a sua volta è definito ricorrendo a $g2$. □

Esercizio 85 *(a) Si estenda il dominio dei comandi con l'assegnamento; (b) Si estendano le funzioni semantiche date in Table13.2 all'assegnamento; (c) Si confronti la semantica data con quella di Table10 (possibilmente, dopo aver svolto il relativo esercizio)* □

Esercizio 86 *Si consideri il seguente frammento formato ancora da una sequenza ed un blocco.*

```
(1)      Label(A): goto Label(B);
          {
(2)       Label(B): x = x +1;
(3)       goto Label(A);
           }
```

Si dica in cosa differisce la sua semantica da quella descritta nell'esempio sopra □

Esercizio 87 *Si applichino le funzioni semantiche di Table13.2 al frammento dell'esercizio precedente* □

Esercizio 88 *Si estenda la semantica data in 13.2 al dominio sintattico $\mathcal{P}$ dei programmi, nelli'ipotesi che un programma sia:*

- *(a) Come in Pascal, una sequenza di dichiarazioni seguita da una sequenza di comandi da valutare;*
- *(b) Come in C, una sequenza di dichiarazioni contenenti una procedura* `main` *da invocare* □

6.4 Astrazioni di Controllo: Le Eccezioni

Sono necessariamente trasferimenti di controllo esprimibili attraverso costrutti non strutturati. Coinvolgono:

- una sezione di codice detta *sezione critica*;
- un insieme di valori detti *eccezioni*;
- un meccanismo per segnalare anomalie attraverso il *sollevamento di eccezioni*;
- un meccanismo per *intercettare e gestire* le eccezioni.

Una sezione critica è un codice che può generare stati di calcolo *impredicibili*, ovvero con proprietà inattese (e certamente, non volute). La criticità del codice sta nel fatto che l'esecuzione del programma, una volta raggiunti tali stati inattesi, potrebbe continuare fino a raggiungere stati di errore non recuperabile (detti anche *stucks*). Useremo il costrutto `try _ catch _ _` di Java per indicare la sezione critica (primo argomento), intercettare un' eccezione (secondo argomento), indicare il codice da eseguire per la relativa gestione (terzo argomento). Useremo il costrutto `throw _` per indicare un'eccezione (primo ed unico argomento) da sollevare.

Table14 – Semantica dei Comandi – 5 : Eccezioni

Domini Sintattici

$\texttt{C} ::= ... \mid \texttt{throw X} \mid \texttt{try C catch X C} \qquad (Comandi)$

Domini Semantici

$\texttt{Env}, \rho, \delta \equiv \texttt{I} \rightarrow \texttt{Den} \qquad (Ambienti)$
$\texttt{Store}, \texttt{s} \equiv \texttt{Loc} \rightarrow \texttt{Mem} \qquad (Memoria)$
$\texttt{State} \equiv \texttt{Store} \qquad (Stato)$
$\texttt{Contn}, \psi, \eta \equiv \texttt{Env} \rightarrow \texttt{State} \rightarrow \texttt{State} \qquad (Continuation)$

Funzioni Semantiche

$\mathcal{D} : \texttt{D} \rightarrow \texttt{Env} \rightarrow \texttt{Env} \qquad (Significato\ delle\ Dichiarazioni)$
$\mathcal{M} : \texttt{C} \rightarrow \texttt{Env} \rightarrow \texttt{Contn} \rightarrow \texttt{State} \rightarrow \texttt{State} \quad (Significato\ dei\ Comandi)$
$\mathcal{E} : \texttt{E} \rightarrow \texttt{Env} \rightarrow \texttt{Contn} \rightarrow \texttt{State} \rightarrow \texttt{Val} \quad (Significato\ delle\ Espressioni$

Definizioni

$\mathcal{C}[\![\texttt{C}]\!]_\rho : \texttt{State} \rightarrow \texttt{State}_\perp \qquad (Comandi)$

$$\mathcal{M}[\![\texttt{try C}_p\ \texttt{catch X C}_h]\!]_{\rho,\psi} =$$
$$\texttt{Let}\{f = \mathcal{M}[\![\texttt{C}_h]\!]\rho, \psi\}$$
$$\{\delta = \texttt{bind}(\texttt{X}, \texttt{K}(f), \rho)\}\ \texttt{in}\ \mathcal{M}[\![\texttt{C}_p]\!]_{\delta,\psi}$$
$$\mathcal{M}[\![\texttt{throw X}]\!]_{\rho,\psi} = \texttt{Let}\{\texttt{K}(g) = \rho(\texttt{X})\}\ \texttt{in}\ g$$

Domini Ausiliari

$\texttt{Den} ::= \texttt{Loc} + \texttt{ProcFun} + \texttt{LV} + \texttt{Code} + \texttt{Fun} + \texttt{GCode} + \texttt{ECode}$
$(Unione\ Disgiunta)$
$\texttt{X} ::= ... \qquad (DominiodelleEccezioni)$
$\texttt{ECode} ::= \texttt{State} \rightarrow \texttt{State}$

Funzioni Ausiliarie

$\texttt{K} : (\texttt{State} \rightarrow \texttt{State}) \rightarrow \texttt{ECode}$