Anno Accademico 77/78

UNIVERSITA' DEGLI STUDI DI PISA

Facolta' di Scienze Matematiche Fisiche e Naturali

Corso di Laurea in SCIENZE DELL'INFORMAZIONE

TESI DI LAUREA

Una Teoria dei Linguaggi Applicativi

Relatore

(Gianfranco Prini)

.....

Candidato

(Luca Cardelli)

.....

CONTENUTI

II: Il linguaggio TAU: note sulla semantica
  II.0: Notazioni
    II.0.0: Notazioni definitorie
    II.0.1: Notazioni metacircolari
  II.1: Costanti
    II.1.0: Semantica delle costanti
    II.1.1: Costanti di tipo null
    II.1.2: Costanti di tipo bool
    II.1.3: Costanti di tipo num
    II.1.4: Costanti di tipo string
    II.1.5: Costanti di tipo dom
    II.1.6: Costanti funzione
  II.2: Variabili
    II.2.0: Semantica delle variabili
    II.2.1: Scoping delle variabili
    II.2.2: Scoping statico
    II.2.3: Scoping dinamico
    II.2.4: Scoping e binders
    II.2.5: Scoping e ambienti
    II.2.6: Accesso delle var. nello scoping indecidibile
    II.2.7: Accesso delle var. nello scoping decidibile
    II.2.8: La scelta dello scoping statico
    II.2.9: Teoria dello scoping
  II.3: Liste e arrays
    II.3.0: Liste
    II.3.1: Arrays
  II.4: Condizionale
    II.4.0: Semantica del condizionale
  II.5: Applicazione e λ-astrazione
    II.5.0: Semantica della λ-astrazione
    II.5.1: Semantica dell'applicazione
    II.5.2: Call-by-need
    II.5.3: Uguaglianza tra funzioni
  II.6: μ-astrazione
    II.6.0: Semantica della μ-astrazione
    II.6.1: Stampa di strutture circolari
    II.6.2: Uguaglianza di strutture circolari
  II.7: Let e letrec
    II.7.0: Let
    II.7.1: Letrec
  II.8: Continuations
    II.8.0: Semantica delle continuazioni
  II.9: Blocchi
    II.9.0: Semantica dei blocchi
  II.10: Supertipi
    II.10.0: Sistemi di supertipi
    II.10.1: Unione disgiunta
    II.10.2: Unione congiunta
    II.10.3: Prodotto cartesiano
    II.10.4: Domini funzionali
    II.10.5: Domini potenza
    II.10.6: Insiemi
    II.10.7: Classi
    II.10.8: Supertipi per il TAU
  II.11: Tecniche di supporto
    II.11.0: Memoria soffice

III: Il linguaggio TAU: semantica
  III.0: Semantica a' la Scott-Strachey
    III.0.0: Domini sintattici
    III.0.1: Domini semantici
    III.0.2: Valutazione
    III.0.3: Funzioni elementari
  III.1: Interprete metacircolare
    III.1.0: Organizzazione generale
    III.1.1: Rappresentazione esterna
    III.1.2: Rappresentazione interna
    III.1.3: Parser
    III.1.4: Eval
  III.2: Interprete iterativo
    III.2.0: Rappresentazione interna
    III.2.1: Insieme dei valori
    III.2.2: Eval
    III.2.3: Funzioni elementari

0: PREMESSE

0.0: Introduzione

I linguaggi di programmazione sono ormai diventati cosi' vari e numerosi che e' necessario un certo sforzo di classificazione prima di poter affrontare qualsiasi discorso approfondito su di essi. La prima distinzione, basata sul flusso del controllo, viene generalmente fatta tra linguaggi procedurali e linguaggi non procedurali. I linguaggi procedurali, sui quali accentreremo l'attenzione, sono fondati sulle operazioni di "chiamata di procedura" e di "sequenza di istruzioni". I linguaggi procedurali che privilegiano il costrutto "sequenza di istruzioni" vengono detti imperativi, mentre quelli che privilegiano la "chiamata di procedura" sono detti applicativi. Tranne casi estremi (es. assembler - LISP puro) la linea di confine tra linguaggi applicativi e imperativi e' in qualche misura arbitraria. Vedremo anche che tecniche sviluppate per fornire la semantica di linguaggi applicativi si adattano facilmente a linguaggi imperativi e viceversa.

I linguaggi di programmazione, oltre ad essere gia' molto numerosi, continuano a crescere rapidamente di numero. Per vari motivi non sembra possibile porre freno a questo processo di proliferazione, al punto che il vecchio mito del linguaggio "general purpose" buono per la maggior parte degli usi e degli utenti e' definitivamente crollato. Oggi, specialmente nel campo del software di sviluppo, si assiste ad una torre di Babele in cui molte comunita' di programmatori preferiscono sviluppare un proprio linguaggio ("general purpose"!), invece di utilizzare quelli gia'

abbondantemente presenti sul mercato. Lo stimolo di questa proliferazione e' l'introduzione di sempre nuovi e perfezionati costrutti e di migliori caratteristiche, destinate a soddisfare (o creare) negli utenti nuove esigenze, in una specie di mercato dei consumi dei linguaggi di programmazione.

L'evoluzione dei linguaggi tuttavia non procede a caso: gli orientamenti dei progettisti si stanno stabilizzando attorno a certi standards quasi universalmente accettati. Si puo' notare come pochi caratteri siano alla base della maggior parte dei linguaggi e ne forniscano una rapida chiave di comprensione e di uso, e come il punto cruciale sia il modo in cui questi costrutti interagiscono tra loro per formare linguaggi piu' o meno ben definiti. Si cade cosi' nel problema della semantica dei linguaggi, la cui recente soluzione ha portato nuove concezioni su come un "bel" linguaggio dovrebbe essere fatto e, analogamente alla soluzione del problema della sintassi, ha prodotto una nuova generazione di linguaggi e relative caratteristiche.

La progettazione di un linguaggio si delinea oggi come l'arte di mettere insieme costrutti potenti e quanto piu' possibile ortogonali, in modo che la comprensione delle singole parti e dei meccanismi di composizione eguivalga alla comprensione del tutto. Quanto segue e' una esposizione delle tecniche di definizione della semantica dei linguaggi, insieme ad una analisi di alcune caratteristiche rilevanti che un linguaggio <u>puo</u>' avere e di come queste caratteristiche possono combinarsi (o escludersi) per non ostacolarsi a vicenda. A scopo

illustrativo la maggior parte di queste caratteristiche vengono integrate e discusse nell'ambito di un linguaggio (TAU) che fornisce al contempo un esempio di definizione formale e implementazione di un linguaggio programmativo.

0.1: Estratto

0.1.0: Semantica dei linguaggi [I]

Dopo una breve introduzione ai linguaggi ed al λ-calcolo [I.0] [I.1] viene esposta la teoria delle funzioni continue su Pw [I.2.0] che fornisce il fondamento teorico per lo studio della semantica dei linguaggi. Su questa base viene definito un linguaggio (LAMBDA) [I.2.2] simile al λ-calcolo che, attraverso meccanismi di interpretazione, servira' da collegamento tra un qualsiasi linguaggio formale e la sua semantica su Pw. Nel resto del capitolo [I.2] vengono definiti in LAMBDA alcuni concetti familiari dei linguaggi di programmazione, fra cui un ricco assortimento di strutture dati, denominate domini.

Nel capitolo [I.3] viene sviluppata una tecnica di definizione della semantica dei linguaggi, originata dal lavoro di Scott e Strachey, fondata sul linguaggio LAMBDA. Con questa tecnica viene definita nuovamente la semantica di LAMBDA e di alcuni linguaggi affini. In particolare si mostra come ricavare la semantica del goto con l'introduzione delle continuazioni [I.3.5] e la semantica dell'assegnamento con l'introduzione delle memorie [I.3.6]. Il risultato principale di questo capitolo e' mostrare che la semantica di un qualsiasi linguaggio puo' essere data sotto forma di un interprete per quel linguaggio, scritto in LAMBDA.

Il capitolo [I.4] illustra come si passa algoritmicamente da un interprete scritto in LAMBDA ad un interprete in

linguaggio assemblativo. Cio' si ottiene, detto L il linguaggio in oggetto, scrivendo un interprete metacircolare per L [I.4.0] (cioe' un interprete per L scritto in L), ed eliminando da questo interprete con trasformazioni successive, tutte le caratteristiche che normalmente non sono presenti in un linguaggio assemblativo. Il risultato e' un interprete iterativo per L; cioe' un interprete ancora scritto in L, ma che non fa uso di ricorsione, che non passa funzioni come argomenti e che non ritorna funzioni come valori di altre funzioni. Il passaggio da un interprete iterativo ad un interprete in linguaggio assemblativo e' poi un procedimento estremamente lineare.

L'interprete cosi' ottenuto e' piu' efficiente di quelli scritti con le tecniche tradizionali. L'uso delle continuazioni porta ad una implementazione in cui tutte le procedure falsamente ricorsive [I.4.1] vengono automaticamente eseguite in modo iterativo (cioe' senza crescita dello stack di controllo).

0.1.1: Il linguaggio TAU: note sulla semantica [II]

I tre interpreti per il TAU riportati nella terza parte [III] (un LAMBDA-interprete, un interprete metacircolare ed un interprete iterativo) vengono descritti e confrontati dettagliatamente, costrutto per costrutto. Vengono inoltre discusse le scelte (arbitrarie e motivate) che sono state fatte a livello di progetto del linguaggio.

Nel capitolo [II.1] si parla dei vari tipi di costanti e delle loro rappresentazioni. In [II.2] si affronta il problema dello scoping delle variabili e viene dimostrato un teoreme che mette in stretta relazione il tipo di scoping presente in un linguaggio con la struttura degli ambienti durante l'esecuzione. Sulla base di questo risultato si giustificano alcune tecniche di accesso alle variabili in regime di scoping statico. In [II.3] si illustra il ruolo delle sospensioni nella costruzione delle strutture dati ed in [II.6] e [II.7] si mostra come questa tecnica consenta di costrutire e utilizzare strutture dati circolari e infinite. In [II.5] viene trattato il problema del passaggio dei parametri alle procedure, specialmente in riferimento alla call-by-need. In [II.8] si descrive un costrutto (escape) che rende disponibili le continuazioni dell'interprete a livello di linguaggio. In [II.10] si da' una descrizione del sistema di tipi del TAU e si fornisce un quadro generale per la strutturazione di vari sistemi di tipi procedurali. In [II.11] infine si accenna ad una tecnica di gestione della memoria che giustifica l'uso delle strutture dati necessarie all'implementazione del TAU.

0.1.2: Il linguaggio TAU: semantica [III]

Con l'aiuto delle tecniche per la definizione della semantica illustrate nella prima parte [I] e sulla base delle discussioni condotte nella seconda [II] viene definita in modo formale la semantica de l TAU.

In [III.0] si da' la semantica standard per mezzo di un LAMBDA-interprete. In [III.1] si fornisce un interprete metacircolare utilizzabile per le usuali procedure di bootstrapping. In [III.2] si presenta il nucleo di un interprete iterativo adatto ad una trascrizione diretta in linguaggio macchina.

0.2: Indice dei simboli

=>	implicazione logica
&	and logico
v	or logico
¬	negazione logica
<=>	equivalenza logica
∀P(x).Q(x)	per ogni x, P(x) implica Q(x)
∃P(x).Q(x)	esiste un x tale che P(x) e Q(x)
=	uguale
≠	diverso
{ ... }	insiemi
∅	insieme vuoto
∈	appartenenza
∉	non appartenenza
u	unione di due insiemi
n	intersezione di due insiemi
U	unione di una famiglia di insiemi
N	intersezione di una famiglia di insiemi
⊆	inclusione tra insiemi
⊈	non inclusione tra insiemi
->	spazio delle funzioni tra due insiemi
f:A->B	f e' una funzione da A in B
x	prodotto cartesiano
*	potenza (iterazione del prodotto cartesiano)
$a^0, a^1, \ldots, a[n]$	espressioni indicizzate
$(I^1, \ldots, I[n])$	n-upla di insiemi
...	omissis
..	omissis formato da sole parentesi
w	insieme degli interi
Pw	insieme delle parti degli interi
+	somma di interi
-	differenza di interi
x	prodotto di interi
/	quoziente di interi
*	elevazione a potenza di interi
<	relazione di minore tra interi
≤	relazione di non maggiore tra interi
S(a≤i≤b) ...	sommatoria
@	alfa
ß	beta
y	gamma
n	eta
l	lambda
µ	mu
w	omega
Teorema ... □	teorema
Prop ... □	proposizione
Dim	dimostrazione
Def ... □	definizione

0.3: Bibliografia

[Aiello 76] L. Aiello, D. Buggiani, G. Prini; "On the implementation of call-by-need"; Note Scientifiche S-76-14; Universita' di Pisa.

[Chomsky 56] N. Chomsky; "Three models for the description of languages"; IRE Transaction on Information Theory, 3, pp 113-124.

[Church 41] A. Church: "The calculi of λ-conversion"; Annals of Mathematical Studies Number 6; Princeton University Press; ristampa: Klaus Reprint Corporation (New York 1965).

[Curry 58] H.B. Curry, R. Feys; "Combinatory logic"; North Holland Publishing Company.

[Friedman 76] D.P. Friedman, D.S. Wise; "Cons should not evaluate its arguments": Proceedings of the Second Symposium in Programming, Paris.

[Gries 71] D. Gries; "Compiler construction for digital computers"; Wiley.

[Milne 71] R.E. Milne; "The mathematical semantics of PAL"; manuscript; PRG, university of Oxford.

[Milne 72] R.E. Milne; "The mathematical semantics of ALGOL68"; manuscript; University of Oxford.

[Milne 76] R.E. Milne, C. Strachey; "A theory of programming languages semantics"; Chapman and Hall.

[Naur 60] P. Naur (editor); "Report on the algorithmic language ALGOL60"; Comm. A.C.M. 3,5 pp 299-314.

[Naur 63] P. Naur (editor); "Revised report on the algorithmic language ALGOL60"; Comm. A.C.M. 6,1 pp 1-17.

[Reynolds 72] J.C. Reynolds; "definitional Interpreters for Higer Order Programming Languages"; Proceedings of the A.C.M. Conference 1972.

[Reynolds 74] J.C. Reynolds; "On the relation between direct and continuation semantics"; Proceedings of the Second Colloquium on Automata, Languages and Programming: Saarbricken; Springer-Verlag.

[Rogers 67] H. Rogers; "Theory of Recursive Functions and Effective Computability"; McGraw-Hill.

[Scott 71] D. Scott; "A lattice oretic model for the the λ-calculus" IV International congress for Logic; Bucarest.

[Scott 72] D. Scott; "Continuous lattices"; in Toposes, algebraic geometry and logic; Spring Verlag; Berlino.

[Scott 73] D. Scott; "A simplified construction for λ-calculus models"; unpublished.

[Scott 75] D. Scott; "Data types as lattices"; in Lecture notes in mathematics n499; Logic conference Kiel 1974; Springer Verlag; Berlino.

[Steele 76a] G.L.Jr. Steele, G.J. Sussman≤ }LAMBDA: the ultimate imperative"; AI Lab memo 353 Cambridge.

[Steele 76b] G.L.Jr. Steele; "LAMBDA: the ultimate declarative"; AI Lab memo 379; Cambridge.

[Strachey 74] C. Strachey, C.P. Wadsworth; "continuations: a Mathematical Semantics for handling full jumps"; Technical monograph; PRG-11 University of Oxford.

[Wadsworth 71] C.P. Wadsworth; "Semantics and pragmatics of the Lambda Calculus" (Ph.D. thesis) University of Oxford.

I: SEMANTICA DEI LINGUAGGI

I.0: Sintassi

I.0.0: Backus Naur form

Il problema teorico della definizione della sintassi di un linguaggio puo' considerarsi risolto fin dai tempi dell'introduzione delle grammatiche generative [Chomsky 56] e della loro applicazione nella descrizione dell'ALGOL [Naur 60,63]. La maggior parte degli studi successivi in questo campo si sono rivolti alla elaborazione di tecniche di traduzione, note come compilazione e parsing, e alla soluzione di vari problemi sintattici conessi con l'implementazione dei linguaggi sul calcolatore. Tutta quest'area di conoscenza puo' considerarsi, almeno nelle sue linee generali, acquisita [Gries 71], e sara' necessario qui ricordare solo uno strumento tecnico usato nel seguito come notazione: la Backus Naur form (BNF). Cominciamo col definire un linguaggio formale come un insieme di stringhe finite ottenute giustapponendo gli elementi di un altro insieme detto alfabeto terminale. Una grammatica G=(AT,AN,S,P) e' un formalismo che permette di esprimere linguaggi. AT e' l'alfabeto terminale, AN e' un alfabato detto alfabeto non terminale, S e' un elemento di AN (simbolo iniziale) e P e' un insieme di produzioni del tipo a->b dove a e' una stringa non vuota costruita su A=(AT u AN) e b e' una stringa (eventualmente vuota) costruita su A. Il linguaggio L(G) generato da una grammatica G e' l'insieme delle stringhe di simboli terminali derivabili da S. Una stringa a' e' derivabile da una stringa a[n] se esiste una

sequenza di stringhe a^1, a^2, ... ,a[n] tale che a[i] e' direttamente derivabile da a[i+1] per $1 \leq i \leq n-1$. Una stringa a infine e' direttamente derivabile da una stringa b quando b=b'Mb" con $M \in AN$ e a=b'a'b" con $M \to a' \in P$.

La BNF e' un modo conciso di esprimere le produzioni appartenenti a P. Ogni simbolo non terminale viene racchiuso tra parentesi angolate: ≤≥ mentre i simboli terminali vengono semplicemente giustapposti. Una espressione del tipo

≤p≥ := ab≤q≥c≤r≥ | ≤s≥≤t≥a | bcb

rappresenta l'insieme di produzioni:

{ ≤p≥ -> ab≤q≥c≤r≥ , ≤p≥ -> ≤s≥≤t≥a , ≤p≥ -> bcb }

Le barre verticali possono essere omesse quando le parti destre delle produzioni sono allineate verticalmente:

```
≤p≥ := ab≤q≥c≤r≥
       ≤s≥≤t≥a
       bcb
```

Ecco una sintassi BNF che definisce il linguaggio dei numeri interi privi di zeri non significativi:

```
≤succ≥ := 1|2|3|4|5|6|7|8|9
≤num≥ := 0 | ≤succ≥ | ≤succ≥≤num≥ | ≤succ≥0≤num≥
```

Una derivazione della grammatica

G=({0,1,2,3,4,5,6,7,8,9},{≤num≥,≤succ≥},≤num≥,P)

(dove P e' stato espresso sopra in BNF) e':

≤num≥ -> ≤succ≥0≤num≥ -> 30≤num≥ -> 30≤succ≥ -> 305

I.1: λ-calcolo

I.1.0: Sintassi del λ-calcolo

Il λ-calcolo [Church 41] puo' essere visto come un prototipo di linguaggio contenente, in una sintassi molto semplice, tutti i concetti essenziali dei linguaggi di programmazione. Questa sua caratteristica lo ha reso "il" linguaggio per eccellenza, non per facilita' di uso o espressivita', ma per la potenza con cui riesce a descrivere situazioni complicate a partire da pochi concetti base. Questo capitolo [I.1] trattera' degli aspetti sintattici del λ-calcolo, mentre il prossimo [I.2] sara' dedicato alla semantica di una sua estensione. La sintassi BNF del λ-calcolo e' la seguente:

```
≤term≥ := ≤var≥                      variabili
          ≤term≥(≤term≥)             applicazioni
          (λ≤var≥.≤term≥)            λ-astrazioni
```

dove ≤var≥ e' un insieme numerabile di identificatori (a,b,c, ... ,aa,ab,ac, ...). Ecco alcuni esempi di stringhe del λ-calcolo (che verranno d'ora in poi chiamate λ-espressioni, λ-termini o semplicemente termini)

x, x(y), (λx.x), (λx.x)(y), x((λy.x(z))),
(λf.(λx.f(x)))((λx.f(x))))

Le lettere a,b,c,d,x,y,w,z verranno usate (a seconda del contesto) come variabili, o per denotare generiche variabili; le lettere A,B,C,D,M,N,P,Q,R,S denoteranno generiche λ-espressioni:

x	denota qualsiasi variabile a,b,x,y, ...
M	denota qualsiasi λ-espressione x, (λx.x), f(x(x)), ...
(λx.M(N))	denota qualsiasi λ-astrazione contenente una applicazione

(λx.x(y)), (λa.(λx.a)(b)(c)),
(λf.(λx.f(x(x)))((λx.f(x(x))))), ...

In una applicazione M(N) il termine M viene detto funzione e il termine N argomento dell'applicazione. In una λ-astrazione (λx.M) la variabile x e' il binder e il termine M e' il corpo della λ-astrazione. Allo scopo di snellire la sintassi vengono introdotte le abbreviazioni:

(λx¹ ... x[n].M) sta per
(λx¹.(λx².(... (λx[n].M) ...))) n≥1
MN sta per
M(N) quando N e' gia' tra parentesi (es: M((λx.P)))
M(N¹, ... ,N[n]) sta per
M(N¹) ... (N[n]) n≥2
(let x¹<-M¹ ... x[n]<-M[n] in M) sta per
(λx¹ ... x[n].M)(M¹) ... (M[n]) n≥1

Inoltre le eventuali parentesi piu' esterne di una λ-espressione e quelle che racchiudono il corpo di una λ-astrazione, o il corpo di un let (il termine che segue in) possono essere omesse:

(λx.(λy.x((λz.y)))) viene abbreviato in λx y.x(λz.y)

Le parentesi attorno ai termini devono essere mantenute in caso di ambiguita' non risolte. Questi casi si presenteranno anche in successive estensioni della sintassi e non verranno evidenziati.

I.1.1: Sostituzione

Le occorrenze di una variabile in una λ-espressione si dividono in binders, quelle immediatamente precedute dal simbolo λ; occorrenze libere, quelle che non sono contenute in alcuna λ-astrazione con binder di identico nome; occorrenze legate, quelle che invece sono contenute in almeno una λ-astrazione con binder dello stesso nome. Le funzioni FV e FB, definite ricorsivamente sulla struttura dei termini, estraggono da un λ-espressione l'insieme delle variabili con occorrenze libere e l'insieme dei binders. Quest'ultimo include l'insieme delle variabili con occorrenze legate.

$FV(x) = \{x\}$
$FV(M(N)) = FV(M) \cup FV(N)$
$FV(\lambda x.M) = FV(M) - \{x\}$

$BV(x) = \emptyset$
$BV(M(N)) = BV(M) \cup BV(N)$
$BV(\lambda x.M) = BV(M) \cup \{x\}$

se $Q = \lambda a.(\lambda b.f(a(b)))(\lambda f.g(f))$ allora

$FV(Q) = \{f,g\}$ $\qquad$ $BV(Q) = \{a,b,f\}$

La variabile f ha in Q occorrenze libere e legate. Vedremo che questi due aspetti di f possono essere considerati come occorrenze in Q di due variabili distinte aventi lo stesso nome. Le occorrenze libere di un termine possono sempre essere legate premettendo opportune λ-astrazioni: $\lambda f\ g.Q$ non ha occorrenze libere.

La principale operazione che si effettua sulle λ-espressioni e' la sostituzione di una variabile con un termine: SUBST(M,N,x) (abbreviato M[N/x]) e' la sostituzione del termine N al posto di tutte le occorrenze

libere della variabile x nel termine M. Le espressioni M[N/x] verranno liberamente immerse nelle λ-espressioni intendendo pero' che : "(... M[N/x] ...)" abbrevia "(... M' ...) dove M'=SUBST(M,N,x)".

Sostituzione:

(S.1) x[N/x] = N
(S.2) y[N/x] = y se y ≠ x
(S.3) A(B)[N/x] = A[N/x](B[N/x])
(S.4) (λx.M)[N/x] = (λx.M)
(S.5) (λy.M)[N/x] = (λz.M[z/y])[N/x]
se y≠x e y∉FV(N) e dove z∉FV(N)
(S.6) (λy.M)[N/x] = (λy.M[N/x]) se y≠x e y∉FV(N)

La sostituzione e' definita in modo che le occorrenze legate di x in M non vengono mai sostituite in M[N/x] (S.4). Inoltre le variabili libere in N non vengono mai legate durante una sostituzione (S.5) cosi' che FV(M[N/x]) = (FV(M) - {x}) u FV(N). Ecco alcuni esempi:

(S.1) a[(λx.x)/a] = (λx.x)
(S.2) b[(λx.x)/a] = b
(S.3) a(b)[(λx.a)/a] = (λb.b)
(S.4) (λb.b)[(λx.a)/b] = (λb.b)
(S.5) (λa.a(b))[(λx.a)/b] = (λy.y(b))[(λx.a)/b]
(S.6) (λy.y(b))[(λx.a)/b] = (λy.y(λx.a))

I.1.2: Riduzioni

Un termine della forma $\lambda x.M$ e' detto @-redex e uno della forma $(\lambda x.M)(N)$ e' detto β-redex. A questi due tipi di redex corrispondono due conversioni (il simbolo ":≥" si legge "si riduce a") :

(@) $(\lambda x.M) :\geq (\lambda y.M[y/x])$ se $y \notin FV(M)$
(β) $(\lambda x.M)(N) :\geq M[N/x]$

L'@-conversione e' un cambiamento di nome di un binder, con rispettiva ridenominazione del corpo della λ-espressione, ed e' attuabile solo se nessuna occorrenza libera in M viene legata durante il processo. La conversione di un @-redex produce ancora un @-redex.

La β-conversione modella l'applicazione di una funzione ad un argomento: il termine N viene testualmente sostituito alle occorrenze libere di x nel corpo della λ-astrazione, M. La β-conversione comporta la scomparsa di un β-redex ma cio' non implica che in assoluto i β-redex del termine in questione diminuiscano:

$(\lambda x.x(x))(\lambda x.x(x)) :\geq (\lambda x.x(x))(\lambda x.x(x))$
$(\lambda x.x(x(x)))(\lambda x.x(x(x))) :\geq$
$(\lambda x.x(x(x)))(\lambda x.x(x(x)))(\lambda x.x(x(x)))$

La β-conversione, per come e' definita la sostituzione, puo' coinvolgere automaticamente delle @-conversioni (quando ad es. M sia una λ-astrazione $(\lambda y.P)$ e $y \in FV(N)$) per evitare la cattura di occorrenze libere. Perche' questo non accada e' sufficiente la condizione $FV(N) \cap BV(M) = \emptyset$:

$(\lambda a.(\lambda b.a(b)))(b) :\geq (\lambda c.b(c))$ e non $:\geq (\lambda b.b(b))$

Ecco altri esempi di conversioni:

(@) $\lambda x.x :\geq \lambda y.y :\geq \lambda z.z$
$\lambda x.x(y) :\geq \lambda z.z(y)$ (ma non $:\geq \lambda y.y(y)$)
$\lambda f\ x.f(x(x)) :\geq \lambda f\ y.f(y(y))$

(ß) $(\lambda x.x)(y) :\geq y$
$(\lambda a.a(b))(\lambda x.a) :\geq (\lambda x.a)(b)$
$(\lambda b.(\lambda b.b))(y) :\geq (\lambda b.b)$

Un altro tipo di conversione e' la

(n) $\lambda x.M(x) :\geq M$

che puo' o no essere inserita nella teoria.

Le á e ß-conversioni possono essere liberamente applicate all'interno di λ-termini in qualsiasi ordine, generando un processo di trasformazione che si chiame riduzione. Piu' precisamente una riduzione e' una successione di λ-termini M¹ ... M[n] in cui M[i+1] e' ottenuto da M[i] applicando una conversione. Quando una riduzione conduce da un termine M ad un termine N privo di ß-redex, si dice che la riduzione converge, altrimenti che diverge. Nel primo caso N e' un termine in forma normale e M ha una forma normale. Un esempio di riduzione e' il seguente:

$(\lambda x\ y.(\lambda y\ z.x(z))(x(x)))((\lambda z\ y.z)(y)(y)))(\lambda x.x)(a) :\geq$
$(\lambda y\ z.(\lambda x.x)(z))((\lambda x.x)(\lambda x.x))((\lambda z\ y.z)(a)(a)) :\geq$
$(\lambda y\ z.(\lambda x.x)(z))(\lambda x.x)(a) :\geq$
$(\lambda x.x)(a) :\geq$
a

L'uguaglianza tra termini e' definita come la minima relazione di equivalenza contenente "≤". Per essa vale la seguente importante proprieta':

Teorema (Church-Rosser) Se M=N, esiste un P tale che M:≥P e N:≥P □

Una dimostrazione si puo' trovare in [Barendregt 71]. Questo teorema vale con o senza la n-conversione.

Teorema Il λ-calcolo (con o senza n-conversione) e' consistente.

Dim Sia I=λx.x e K=λx y.x. E' impossibile dimostrare che I=K perche se cosi' fosse esisterebbe un P tale che I:≥P e

K:≥P. Ma cio' e' assurdo perche' I e K sono in forma normale

□

Il fatto che esistano due termini sicuramente diversi (cioe' che esistano delle uguaglianze non derivabili) garantisce che il λ-calcolo sia consistente. Per definizione infatti una teoria si dice consistente se non tutte le sue formule sono teoremi.

I.2: Modelli del λ-calcolo

I.2.0: Funzioni continue su Pw

Pw = {x|x≤w} e' l'insieme delle parti dell'insieme w dei numeri interi. Le lettere i,j,k,l,m,n verranno usate per indicare elementi di w, mentre le lettere u,v,x,y,z denoteranno elementi di Pw. $\emptyset$ e w come elementi di Pw saranno indicati con $\perp$ e $\top$ (bottom e top). Una funzione f:Pw->Pw verra' detta: strict (o anche strict su $\perp$) se $f(\perp)=\perp$, strict su $\top$ se $f(\top)=\top$ e doppiamente strict se $f(\perp)=\perp$ e $f(\top)=\top$.

Cominciamo col fissare una enumerazione (codifica) di tutti gli insiemi finiti di interi: e[n] (con n∈w) sara' l'n-esimo insieme finito. L'enumerazione deve essere ricorsiva (cioe' k∈e[n] deve essere un predicato ricorsivo in k e n) e ben fondata, cioe' k∈e[n] => k<n (ovvero non deve esistere alcun ciclo del tipo $k^0 \in e[k^1]$, $k^1 \in e[k^2]$, ... , $k[m] \in e[k^0]$ con m≥0). Se e[n]={k^0, ... ,k[m]} con k^0< ... <k[m] si puo' assumere la codifica

$$S(0 \le i \le m).\ 2^{k[i]}$$

che soddisfa le proprieta' richieste. L'enumerazione degli insiemi finiti torna utile in vari processi di approssimazione in quanto ogni insieme di interi puo' essere espresso come unione dei suoi sottinsiemi finiti:

$$\forall x \in Pw.\quad x = \bigcup\{e[n] \mid e[n] \le x\}$$

Si assume inoltre una enumerazione (codifica) ricorsiva delle coppie ordinate di interi (n,m), anch'essa ben fondata grazie alla proprieta': n≤(n,m), m≤(n,m). Una codifica

appropriata e' ad esempio quella derivante dal cosiddetto ordinamento diagonale:

$$(n,m) = ((n+m)(n+m+1)+m)/2$$

Per le funzioni f:Pw->Pw si usera' una λ-notazione del tipo: (λx∈Pw. ...) che non dovra' essere confusa ad esempio con il λ-calcolo visto come linguaggio. (λx∈Pw. ...) e' una funzione Pw->Pw, mentre (λx. ...) e' una stringa.

Passiamo ora ad esaminare le funzioni continue f:Pw->Pw [Scott 75]. La nozione di continuita' puo' essere definita sulla base di una certa topologia indotta dall'ordinamento parziale "≤" su Pw. Questo tipo di trattazione non e' tuttavia strettamente necessario e si puo' partire, anziche' dalla definizione della topologia, direttamente dalla definizione di continuita'; una completa trattazione topologica si puo' trovare in [Scott 72].

Def 1 Una funzione f:Pw->Pw e' continua sse

$$\forall x \in Pw.\ f(x) = \bigcup\{f(e[n]) \mid e[n] \leq x\} \qquad \square$$

L'insieme delle funzioni continue ha la cardinalita' del continuo (a differenza ad esempio dell'insieme delle funzioni monotone che ha cardinalita' maggiore) e puo' essere immerso in modo naturale in Pw, come verra' mostrato piu' avanti. Ogni elemento di Pw puo' allora essere visto come un insieme, oppure come una funzione continua f:Pw->Pw.

Le funzioni continue costituiscono un primo passo verso la caratterizzazione delle funzioni calcolabili (che risulteranno essere tutte continue). Questo si puo' intuire anche dalla Def 1, basata sulla raggiungibilita' (al limite) degli argomenti e dei valori delle funzioni a partire da quantita' finite. Il senso della definizione e' infatti il

seguente: dato un insieme x il valore di f non dipende semplicemente dagli elementi di x (altrimenti avremmo potuto scrivere la piu' restrittiva: f(x) = U{f({k})|k∈x}) ma neppure e' permesso agli elementi di x di influenzarsi in modo troppo complesso (cioe' infinitamente complesso) nel determinare il valore di f; le influenze reciproche non si estendono mai al di la' dei sottinsiemi finiti di x, da cui la definizione.

Prop_1 (Scott) Le funzioni continue sono monotone rispetto all'ordinamento "≤":

f continua => ∀x,y∈Pw. x≤y => f(x)≤f(y) □

Delle funzioni continue si puo' dare anche una definizione del tipo "per ogni epsilon esiste un delta ..." utile nella dimostrazione di molti teoremi:

Prop_2 (Scott) Una funzione f:Pw->Pw e' continua sse

∀x∈Pw. ∀m∈w. e[m]≤f(x) <=> ∃e[n]≤x. e[m]≤f(e[n]) □

Le funzioni vengono spesso considerate come insiemi di coppie ordinate argomento-valore. Una funzione f:Pw->Pw e' allora un insieme di coppie di elementi di Pw: un insieme molto complesso, composto da una infinita' non numerabile di elementi, ognuno dei quali e' una coppia di insiemi non enumerabili. Limitandosi pero' alle funzioni continue f:Pw->Pw c'e' un modo di caratterizzare ogni funzione con insiemi piu' semplici. Per la definizione di continuita', conoscendo il valore di f sugli insiemi finiti si puo' calcolare f su qualsiasi insieme di Pw. E' sufficiente allora una quantita' numerabile di coppie $\{(e^0,y^0),(e^1,y^1),(e^2,y^2), \ldots\}$ per definire completamente $y = f(x) = U\{f(e[n]) \mid e[n]\leq x\}$.

Niente garantisce che y[n]=f(e[n]), sia un insieme finito (generalmente non lo e') pero' y[n]≤w, e la coppia (e[n],y[n]) puo' essere effettivamente caratterizzata dalle coppie {(e[n],m)|m∈y[n]}. Unendo questi insiemi per tutti gli e[n] si determina interamente f.

In questo modo ogni funzione continua Pw->Pw e' individuata da un insieme numerabile di coppie di interi e quindi, con una codifica delle coppie, da un elemento di Pw.

Prop 3 (Scott) Ogni funzione continua f:Pw->Pw e' determinata univocamente dall'insieme:

graph(f) = {(n,m)| m∈f(e[n])} □

Prop 4 (Scott) Ogni insieme u∈Pw determina una funzione continua tramite la formula:

fun(u)(x) = {m|∃e[n]≤x. (n,m)∈u}

Inoltre:

fun(graph(f)) = f
u ≤ graph(fun(u))
u = graph(fun(u)) sse ((k,m)∈u & e[k]≤e[n])=>(n,m)∈u □

Ad ogni insieme u di Pw corrisponde quindi una funzione continua f:Pw->Pw tale che f=fun(u). Ad ogni funzione continua f:Pw->Pw corrispondono invece molti insiemi u∈Pw tali che fun(u)=f ed il massimo di questi e' graph(f), il grafico di f.

Non tutti gli insiemi di Pw sono grafici di funzioni continue: ad esempio {0} = {(0,0)} e' il grafico di una funzione p(x)=({0} se x=∅, ∅ altrimenti) che non e' continua e neppure monotona (infatti ∅≤{0} ma p(∅)≰p({0})). fun({0}) e' comunque una funzione continua; la minima estensione continua di {0}.

fun({0}) = (λx∈Pw.{m|∃e[n]≤x. (n,m)∈{0}}) = (λx∈Pw.{0})
= {(n,0)|n∈w} = {n(n+1)/2|n∈w}

Def_2 Una funzione f:Pw*k->Pw di k variabili e' continua se e' continua in ognuna delle sue variabili separatamente. Cioe': f(x¹, ... ,x[k]) e' continua sse

$$\forall 0 \le i \le k.\quad f(x^1, \ldots, x[k]) = \bigcup\{f(x^1, \ldots, e[n], \ldots, x[k]) \mid e[n] \le x[i]\}$$ □

Prop_5 L'insieme delle funzioni continue e' chiuso rispetto alla sostituzione (composizione generalizzata). Se f(x¹, ... ,x[k]) e g[k](x[k,1], ... ,x[k,n[k]]) sono continue allora

f(g¹(x¹[1], ... ,x¹[n¹]), ... ,g[k](x[k,1], ... ,x[k,n[k]]))

e' continua □

Prop_6 Ogni funzione continua f:Pw->Pw ha un minimo punto fisso m dato dalla formula:

$$m = \bigcup\{f[n](\emptyset) \mid n \in w\}$$

dove $f^0(x)=x$ e $f[n+1](x)=f(f[n](x))$ □

Il principale risultato di questo paragrafo e' che ogni elemento di Pw puo' essere visto come un insieme di interi, o come una funzione Pw->Pw. Si parlera' quindi liberamente di applicare un insieme, o di selezionare elementi da una funzione, ogni volta sottintendendo le appropriate trasformazioni fun e graph.

I.2.1: Il linguaggio LAMBDA e la sua semantica

Il linguaggio LAMBDA [Scott 75] e' una estensione del λ-calcolo: la sua sintassi (a parte le normali abbreviazioni e parentesizzazioni) e' data da:

```
≤term≥ := ≤var≥
          0
          ≤term≥+1
          ≤term≥-1
          ≤term≥ :> ≤term≥ , ≤term≥
          ≤term≥(≤term≥)
          λ≤var≥.≤term≥
```

LAMBDA sara' usato come strumento di definizione della semantica di vari linguaggi. Dare la semantica di un linguaggio L significa fissare un oggetto matematico D di cui si sia dimostrata l'esistenza e definire una funzione calcolabile E:L->D, detta valutazione, che associa ad ogni stringa di L un elemento di D. Si dice che ogni stringa denota un elemento di D e che questo elemento e' il suo valore. Il dominio D viene detto modello di L. Modelli banali ovviamente non rivestono alcun interesse (ad esempio il λ-calcolo ha per modello l'insieme {0} sotto una valutazione che manda ogni λ-espressione in 0). Ci proponiamo adesso di mostrare che Pw e' un modello (non banale) di LAMBDA, e che quindi LAMBDA "ha senso" perche' parla senza contraddizioni di un oggetto reale assai complesso. La semantica di LAMBDA viene data in modo informale con una serie di equazioni aventi a sinistra uno schema di sintassi e a destra l'elemento di Pw corrispondente. Metodi piu' rigorosi di definizione della semantica verranno elaborati piu' avanti.

(L.1) $0 = \{0\}$
(L.2) $x+1 = \{n+1 \mid n \in x\}$
(L.3) $x-1 = \{n \mid n-1 \in x\}$
(L.4) $z\text{->}x,y = \{n \in x \mid 0 \in z\} \cup \{m \in y \mid \neg k.k+1 \in z\}$
(L.5) $u(x) = \{m \mid \neg e[n] \leq x.(n,m) \in u\}$
(L.6) $\lambda x.M = \{(n,m) \mid m \in M[e[n]/x]\}$

Le prime tre equazioni mostrano come l'insieme dei numeri interi del LAMBDA viene valutato all'insieme dei singoletti $\{\{n\} \mid n \in w\}$. (L.2) definisce l'operazione "successore di un insieme" nel senso che succ{3,5,7}={4,6,8}. Analogamente per pred in (L.3) notando che pred{0} = $\perp$. (L.4) definisce il condizionale che si puo' scrivere anche:

$$z\text{:>}x,y = \begin{cases} \perp \text{ se } z=\perp \\ x \text{ se } z=\{0\} \\ y \text{ se } 0 \notin z \text{ con } z \neq \perp \\ x \cup y \text{ se } 0 \in z \text{ con } z \neq \{0\} \end{cases}$$

e che funziona essenzialmente come un test su zero con qualche complicazione se z non e' un singoletto. L'uso normale e' di prendere {0} come vero e {1} come falso. L'applicazione u(x) in (L.5) viene definita come fun(u)(x) interpretando u come una funzione. Similmente la λ-astrazione in (L.6) e' definita come graph($\lambda a \in Pw.M[a/x]$). Resta da specificare la semantica di un termine M con variabili libere $a^1 \ldots a[k]$. Un tale termine definisce una funzione delle sue variabili libere:

$M=f(a^1, \ldots, a[k]):Pw*k\text{->}Pw$ dove $\{a^1, \ldots, a[k]\}=FV(M)$

Def_3 Una funzione $f:Pw*k\text{->}Pw$ e' LAMBDA-definibile se esiste un LAMBDA-termine M che la definisce come funzione delle sue variabili libere □

Prop_7 (Scott) Tutte le funzioni LAMBDA-definibili sono continue □

La n-conversione [I.1.2] non vale per LAMBDA in Pw. Essa pero' implica (e non e' implicata da) un principio di

estensionalita' che risulta verificato:

(n°) $\lambda a.M = \lambda a.N \iff \forall x \in Pw.\ M[x/a] = N[x/a]$

Prop 8 (Scott) Le proprieta' (@),(ß),(n°) sono valide in Pw, cioe':

(@) $\lambda a.M = \lambda b.M[b/a]$ se $b \notin FV(M)$
(ß) $(\lambda a.M)(N) = M[N/a]$
(n°) $\lambda a.M = \lambda a.N \iff \forall x \in Pw.\ M[x/a] = N[x/a]$ □

Queste proposizioni assicurano che Pw e' veramente un modello di LAMBDA perche' le espressioni interconvertibili (secondo una definizione puramente sintattica) risultano uguali sul modello.

Prop 9 Per ogni funzione continua $f:Pw*k->Pw$ c'e' un elemento $u \in Pw$ tale che:

$f(x^1, \ldots, x[k]) = u(x^1) \ldots (x[k])$ □

Questo u non e' nient'altro che $\lambda x^1 \ldots x[k].f(x^1, \ldots, x[k])$ dove f viene aggiunta come una nuova costante del linguaggio, nel caso che non sia gia' LAMBDA-esprimibile. In questo modo non solo le funzioni Pw->Pw, ma anche quelle Pw*k->Pw sono rappresentabili in Pw come funzioni Pw->Pw-> ... ->Pw.

Prop 10 (Scott) Se u=graph(f) per qualche f:Pw->Pw continua, allora Y(u) e' il minimo punto fisso di f, dove

$Y = \lambda f.(\lambda x.f(x(x)))(\lambda x.f(x(x)))$ □

Def 4 Una funzione $f:Pw*k->Pw$ e' calcolabile sse la relazione $m \in f(e[n^1]) \ldots (e[n[k]])$ e' ricorsivamente enumerabile in $m,n^1, \ldots, n[k]$ □

Prop 11 (Scott) Per ogni funzione continua $f:Pw*k->Pw$ le seguenti proposizioni sono equivalenti:

a) f e' calcolabile
b) $\lambda x^1 \ldots x[k].f(x^1) \ldots (x[k])$ e' un insieme r.e.
c) $\lambda x^1 \ldots x[k].f(x^1) \ldots (x[k])$ e' LAMBDA-definibile □

Si puo' facilmente vedere che la nozione di calcolabilita' qui definita coincide con quella intuitiva di "avere un algoritmo". Nel caso di una funzione di una variabile f:Pw->Pw, per calcolare y=f(x) con x r.e. si procede enumerando i sottinsiemi finiti e[n]≤x. Per ognuno di questi si possiede un procedimento f[n](e[n]) che enumera gli elementi di f(e[n]) (per l'ipotesi che m∈f(e[n]) e' r.e.). Facendo partire "in parallelo" questi procedimenti f[n] (ad esempio con la ben nota tecnica diagonale) si riescono ad accumulare tutti gli m∈f(e[n]) per qualche e[n], approssimandosi a y quanto si vuole. Se viceversa m∈f(e[n]) non e' r.e., non c'e' nessun procedimento per enumerare f(e[n]) e quindi nessun algoritmo per approssimare y.

La Prop 11 suggerisce un altro modello per LAMBDA: RE={x∈Pw|x e' r.e.}≤Pw la collezione di tutti gli insiemi r.e., che e' chiusa rispetto all'applicazione ed alla λ-astrazione di funzioni calcolabili. Questo risultato deriva direttamente dal fatto che x∈RE e' r.e. sse e' LAMBDA-definibile. Molti altri modelli di LAMBDA esistono certamente in Pw, e similmente esistono molti modelli del λ-calcolo, come si vedra' nel seguito.

I.2.2: Alcune definizioni

Un risultato del paragrafo precedente e' che tutte le funzioni calcolabili e tutti gli insiemi ricorsivamente enumerabili sono LAMBDA-definibili. Vediamo alcune di queste LAMBDA-definizioni. Si usera' la sintassi $\mu f.M = Y(\lambda f.M)$ per le funzioni ricorsive, e le abbreviazioni 1=0+1, 2=1+1, ... per i numeri.

(1) $\bot = (\lambda x.x(x))(\lambda x.x(x))$ insieme vuoto $\emptyset$

(2) $x \cup y = \{\lambda z.0\} :> x,y$ unione di insiemi r.e.
notare che $(\lambda z.0) = \{(n,0) \mid n \in w\} = \{n(n+1)/2 \mid n \leq w\}$ e quindi $0,1 \in (\lambda z.0)$

(3) $\tau = \mu x.0 \cup x+1$ insieme degli interi w

(4) $x \cap y = \{\mu f.\lambda x\ y.x :> (y :> 0, \bot), f(x-1)(y-1)+1\}(x)(y)$
Intersezione di insiemi r.e.. Si scrive anche:

$$x \cap y := \begin{cases} \{0\} \cup (x-1 \cap y-1)+1 & \text{se } 0<x \text{ e } 0<y \\ (x-1 \cap y-1)+1 & \text{altrimenti} \end{cases}$$

(5) $z \to x,y = z :> (z :> x, \tau), (z :> \tau, y)$
condizionale doppiamente strict.

$$z \to x,y = \begin{cases} \bot & \text{se } z=\bot \\ x & \text{se } z=\{0\} \\ y & \text{se } 0 \notin z \text{ con } z \neq \bot \\ \tau & \text{se } 0 \in z \text{ con } z \neq \{0\} \end{cases}$$

(6) $x==y = \{\mu f.\lambda x\ y.x \to (y \to 0,1), y \to 1, f(x-1)(y-1)\}(x)(y)$
uguaglianza sui singoletti. Si puo' scrivere

$$x==y = \begin{cases} \bot & \text{se } x=\bot \text{ o } y=\bot \\ \{0\} & \text{se } \neg\exists n \in w.\ x=y=\{n\} \\ \{1\} & \text{se } \neg\exists n,m \in w.\ x=\{n\},\ y=\{m\},\ n \neq m \\ \tau & \text{altrimenti} \end{cases}$$

(7) $\text{double} = \mu f.\lambda x.x :> 0, f(x-1)+1+1$
$\text{double}(\{0,1,2,3\}) = \{0,2,4,6\}$

(8) $\text{half} = \mu f.\lambda x.x :> 0, f(x-1-1)+1$
$\text{half}(\{0,1,2,3\}) = \{0,1\}$

(9) $\langle a,b \rangle = \text{pair}(a,b)$
$\text{pair} = \lambda x\ y.\text{double}(x) \cup \text{double}(y)+1$
$\text{left} = \text{half}$
$\text{right} = \lambda x.\text{half}(x-1)$
Alcune proprieta' delle coppie sono:
$\forall x \in Pw.\ x = \langle \text{left}(x), \text{right}(x) \rangle$
$\forall x,y \in Pw.\ \text{left}(\langle x,y \rangle) = x \quad \text{right}(\langle x,y \rangle) = y$
$\forall a,b,a',b' \in Pw.\ a' \leq a,\ b' \leq b \Rightarrow \langle a',b' \rangle \leq \langle a,b \rangle$

(10) Sia D un insieme discriminabile di elementi di Pw, cioe' uno per il quale esista una funzione calcolabile biunivoca

num-of: D->I dove I≤w

Allora e' possibile definire una funzione di uguaglianza su D, indicata con equal(D), utilizzando l'uguaglianza sui singoletti definita in (6)

equal(D) = λx y.num-of(x)==num-of(y)

Si usera' la sintassi

M=N = equal(D)(M)(N)

per indicare l'uguaglianza su qualche insieme D di cui si suppone sia stata dimostrata la discriminabilita'.

(11) x°y = λz.x(y(z))

(12) any = i = λx.x

I.2.3: Retrazioni, retratti, iniezioni

Def_5 Una retrazione e' una funzione r:Pw->Pw per la quale vale la proprieta' di idempotenza:

$r = r \circ r$ □

Alcune semplici retrazioni sono:

$\perp$

$\top$

bool = $\lambda x.x \to 0, \top+1$

num = $\mu f.\lambda n.n \to 0, f(n-1)+1$

fun = $\lambda f\ x.f(x)$

i = $\lambda x.x$

Def_6 L'insieme dei punti fissi di una retrazione r e' il retratto di r (indicato con R(r)):

$R(r) = \{x \in Pw \mid r(x)=x\}$ □

Notare che lo stesso retratto puo' essere generato da molte retrazioni diverse. Si vede facilmente che:

$R(\perp) = \{\perp\}$

$R(\top) = \{\top\}$

$R(bool) = \{\top, \perp, \{0\}, \top+1\}$

$R(num) = \{\top, \perp\} \cup \{\{n\} \mid n \in w\}$

$R(fun) = \{u \in Pw \mid graph(fun(u)) = u\}$

$R(i) = Pw$

Dalla proprieta' di idempotenza discende che se r e' una retrazione, allora:

$x \in R(r) \iff r(x) = x$

cioe' una retrazione e' una funzione r:Pw->R(r) che coincide con l'identita' ($\lambda x \in Pw.x$) su R(r) e solo su R(r).

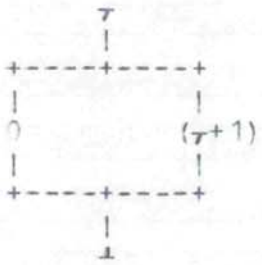
I retratti hanno una struttura ben definita, come viene stabilito dal teorema:

Prop_12 L'insieme dei punti fissi di una funzione continua forma un lattice completo sotto l'ordinamento "≤" □

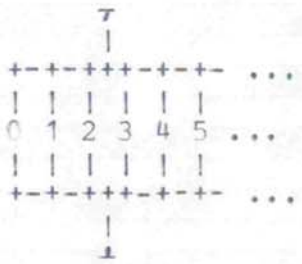
Un retratto verra' quindi spesso disegnato con un diagramma che ne mette in evidenza le relazioni di

ordinamento:

R(bool) =



R(num) =



Def 7 Una iniezione e' una retrazione c tale che i≤c. Una proiezione e' una retrazione p tale che p≤i □

Una iniezione puo' essere vista come una operazione di chiusura, cioe' come una operazione che da un insieme x∈Pw e da una funzione f definita sui sottinsiemi finiti di x, consente di ottenere il minimo insieme c(x) contenente x e chiuso sotto l'applicazione di f. Tralasciando la particolare funzione f che genera c(x) (e che in genere, fissati x e c(x), non e' unica), i requisiti di una operazione di chiusura sono:

i) che x≤c(x) : un insieme e' contenuto nella sua chiusura
ii) che c(c(x))=c(x) : la chiusura di una chiusura equivale alla chiusura

Ma la condizione (i) equivale ad i≤c perche' i≤c <=> ∀x∈Pw.x≤c(x), e la condizione (ii) equivale a dire che c e' una retrazione. Questo da' il senso della Def 7, in base alla quale si puo' vedere immediatamente che i e τ sono iniezioni, mentre ⊥ e' una proiezione. Inoltre:

Prop 13 bool, num, fun sono iniezioni □

I.2.4: Domini

Def_8 (i) I domini sono gli insiemi dei punti fissi (i retratti) delle iniezioni.

(ii) Il predicato "x:d" ("x e' di tipo d") per ogni iniezione d e' cosi' definito:

$$\forall x \in Pw.\ \ x:d \iff x \in R(d) \iff d(x)=x$$

(iii) L'insieme di tutte le iniezioni viene indicato con D e "d<D" viene scritto "d:D" □

La notazione definita in (iii) sara' giustificata mostrando che anche D e' esprimibile con una iniezione, rientrando cosi' nel caso (ii).

Cominciamo col definire vari tipi di domini, ricordando che alcuni domini elementari sono gia' stati visti in [I.2.3] (cioe' $R(\tau)$, R(bool), R(num), R(fun), R(i)).

Def_9 prodotti cartesiani

$$a \otimes b = \lambda x.\langle a(left(x)), b(right(x))\rangle$$ □

Prop_14 (Scott)

$$a,b:D \Rightarrow a \otimes b:D$$ □

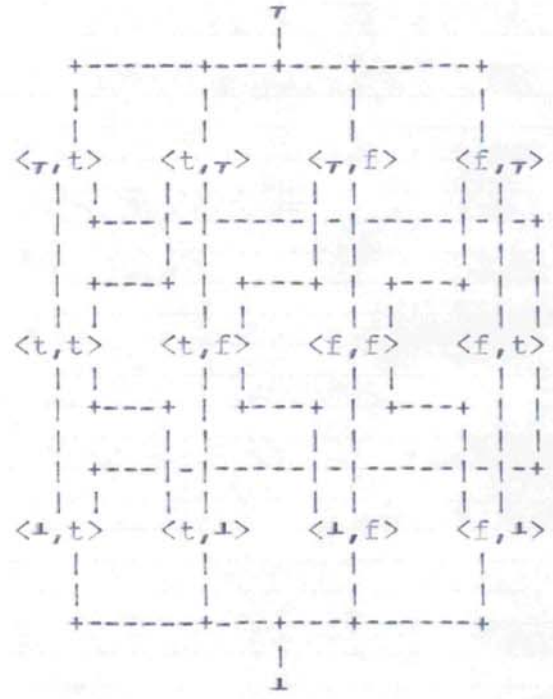
Quindi "⊗" (cross) e' una funzione $\otimes: D\times D \to D$. L'iniezione $(a\otimes b)(z)$ estrae le parti destre e sinistre di z e le retrae rispettivamente sui domini R(a) e R(b). Il risultato e' di retrarre z sull'insieme $R(a\otimes b)$ di tutte le coppie ordinate formate da elementi di R(a) (a sinistra) e R(b) (a destra). I punti fissi di $(a\otimes b)$ sono allora tutti del tipo $\langle x,y\rangle$ con x:a e y:b.

$$x:(a\otimes b) \iff left(x):a \ \&\ right(x):b$$

$R(a\otimes b)$, come tutti i retratti, e' un lattice completo ed i suoi elementi sono cosi' ordinati:

$\forall x,y:(a \otimes b).\ x \leq y \iff left(x) \leq left(y)\ \&\ right(x) \leq right(y)$

Ad esempio R(bool⊗bool) e' il dominio:



dove t=0 e f=$_{\top}$+1 (notare che $_{\top}$=<⊤,⊤> e ⊥=<⊥,⊥>).

Def 10 domini funzionali

$a \otimes b = \lambda x.b \circ x \circ a$ □

Prop 15 (Scott)

$a,b:D \Rightarrow a \otimes b:D$ □

Quindi "⊗" (arrow) e' una funzione ⊗:DxD->D. La retrazione (a⊗b)(f) restringe ogni funzione f:Pw->Pw ad una funzione f:R(a)->R(b). Infatti

$(a \otimes b)(f) = b \circ f \circ a = \lambda x.b(f(a(x)))$

(a⊗b)(f) e' una funzione che prende un argomento x, lo retrae su R(a), lo applica a f e retrae il risultato su R(b); cioe' (a⊗b)(f) vede solo argomenti di R(a) e produce solo risultati in R(b), qualunque sia f. Il risultato e' di restringere f al dominio R(a⊗b) di tutte le funzioni continue da R(a) in R(b).

$f:(a \triangleright b) \iff \forall x:a.\ f(x):b$

Intuitivamente retrarre una funzione su un dominio $R(a \triangleright b)$ significa fare un test sul tipo dell'argomento e del risultato e applicare, quando e' necessario, una conversione di tipo. Si puo' introdurre una sintassi per queste funzioni tipizzate:

$$\lambda(x:a)\text{->}b.M = (a \triangleright b)(\lambda x.M)$$
$$\lambda(x^1:a^1, \ldots, x[n]:a[n])\text{->}b.M = \lambda(x^1:a^1)\text{->any}. \ldots \lambda(x[n]:a[n])\text{->}b.M$$

Si verifica facilmente che $\lambda(x:any)\text{->any}.M = \lambda x.M$.

L'ordinamento introdotto da "$\leq$" sul lattice completo $(a \triangleright b)$ e':

$$\forall f,g:(a \triangleright b).\ f \leq g \iff \forall x:any.\ f(x) \leq g(x)$$

dove $\tau = \lambda x.\tau$ e $\bot = \lambda x.\bot$.

Def 11 unioni disgiunte

$$a \oplus b = \lambda x.((left(x):>0,0)\ u\ (right(x):>1,1))\text{->} \langle a'(left(x)),\bot\rangle,\langle\bot,b'(right(x))\rangle$$

dove $a' = \lambda x.0\ u\ a(x-1)+1$ e $b' = \lambda x.0\ u\ b(x-1)+1$ □

Prop 16 (Scott)

$$a,b:D \Rightarrow a \oplus b:D$$ □

Anche "$\oplus$" (union) e' una funzione $\oplus:D \times D \to D$ e puo' essere scritta:

$$(a \oplus b)(z) = \begin{cases} \langle a'(x),\bot\rangle & \text{se } z=\langle x,\bot\rangle \text{ con } x \neq \bot \\ \langle\bot,b'(y)\rangle & \text{se } z=\langle\bot,y\rangle \text{ con } y \neq \bot \\ \bot & \text{se } z=\langle\bot,\bot\rangle=\bot \\ \tau & \text{se } z=\langle x,y\rangle \text{ con } x,y \neq \bot \end{cases}$$

Questa operazione viene detta unione disgiunta perche' gli elementi di $R(a \oplus b)$ sono quelli di $R(a)$ piu' quelli di $R(b)$ codificati in modo da riconoscere la loro originaria appartenenza a $R(a)$ o $R(b)$. La codifica scelta e':

$$x:a \iff \langle 0\ u\ x+1,\bot\rangle:a \oplus b$$
$$y:b \iff \langle\bot,0\ u\ y+1\rangle:a \oplus b$$

Il predicato which:any$\triangleright$bool cosi' definito:

```
which = λz.((left(z):>0,0) u (right(z):>1,1))->0,⊤+1
```

$$\text{which}(z) = \begin{cases} 0 & \text{se } z=\langle x,\bot\rangle \text{ con } x\neq\bot \\ \top+1 & \text{se } z=\langle \bot,y\rangle \text{ con } y\neq\bot \\ \bot & \text{se } z=\langle \bot,\bot\rangle=\bot \\ \top & \text{se } z=\langle x,y\rangle \text{ con } x,y\neq\bot \end{cases}$$

consente di determinare l'originaria appartenenza della decodifica di z a R(a) o a R(b).

Il motivo per cui si scrive $\langle 0 \text{ u } x+1,\bot\rangle$ anziche' semplicemente $\langle x,\bot\rangle$ e' che puo' valere $\bot$:a e in tal caso which non e' piu' utilmente applicabile perche' $\bot$:a diviene indistinguibile in a⊕b da un eventuale $\bot$:b (in tal caso si ha infatti $\langle x,\bot\rangle=\langle\bot,\bot\rangle=\langle\bot,y\rangle$). Allora $\bot$:a viene precodificato con 0 e ogni altro x:a con x+1. Questo spiega l'usc di a' e b' nella Def 11. Lo stesso problema non si presenta con $\top$ perche $\langle\top,\bot\rangle\neq\langle\bot,\top\rangle$.

Altre funzioni standard per la manipolazione delle unioni disgiunte sono:

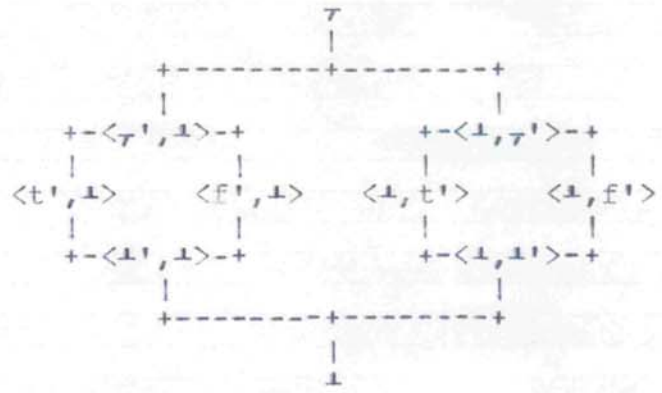
```
inleft = λr x.r(<x,⊥>)
inright = λr x.r(<⊥,x>)
outleft = left
outright = right
out = λx.which(x)->outleft(x),outright(x)
```

Queste possono essere ristrette a unioni specifiche scrivendo ad esempio outleft' = λ(x:a⊕b)->a.left(x). Adesso outleft':(a⊕b)⊃a.

L'ordinamento del lattice completo R(a⊕b) e':

$$\forall x,y:a\oplus b.\ x\leq y \iff \text{which}(x)=\text{which}(y)\ \&\ \text{out}(x)\leq\text{out}(y)$$

Ad esempio R(bool⊕bool) e' il lattice:



dove x'=bool'(x)=(0 u bool(x-1)+1) per $x=\bot,\top,t,f$ con $t=0$ e $f=\top+1$.

Def_12 iniezioni

$$D = \lambda f\, x.\flat y.x \sqcup f(y) \qquad \square$$

D e' una retrazione che manda ogni funzione continua f in una iniezione $D(f)=\lambda x.\flat y.x \sqcup f(y)$ ottenuta chiudendo tutti gli insiemi $x\in Pw$ sotto l'applicazione di f (cioe' $D(f)(x) = x \sqcup f(x) \sqcup f(f(x)) \sqcup \ldots$):

$$D(f)(x) = N\{y \mid x\leq y \ \&\ f(y)\leq y\}$$

Il fatto notevole e' che D e' qualcosa di piu' di una retrazione, come stabilisce la seguente:

Prop_17 (Scott) La funzione D e' una iniezione e l'insieme dei punti fissi di D e' esattamente l'insieme di tutte le iniezioni $\square$

Questo giustifica la notazione x:D fin qui usata per indicare che x e' una iniezione, e giustifica anche in assoluto il concetto di iniezione. La funzione D sembra incarnare il concetto di "dominio di tutti i domini", la cui esistenza e' stata ipotizzata in molti linguaggi di programmazione; ha senso infatti scrivere D:D e anche $D\otimes D:D$, $D\oplus D:D$, $D\otimes D:D$. Un risultato di questo genere non poteva essere ottenuto considerando semplicemente le retrazioni,

perche' l'insieme di tutte le retrazioni non e' un retratto.

Resta da dimostrare la possibilita' di costruire domini a partire da equazioni ricorsive del tipo:

tree = bool⊕(tree⊗tree)

Prop 18 (Scott)

f:D⊗D => Yf:D □

Il dominio R(tree) puo' quindi essere definito come retratto di tree = μd.bool⊕(d⊗d).

La Prop 18 consente di risolvere semplicemente un problema finora lasciato aperto, e cioe' la costruzione di un modello del λ-calcolo puro. Un tale modello e' dato dalla scluzicne dell'equazione

d = d⊗d

cioe' dal minimo punto fisso della funzione f=λ(d:D)->D.d⊗d che puo' essere ottenuto per la Prop 6 come limite della sequenza:

f(⊥), f(f(⊥)), ... ossia
D(⊥)⊗D(⊥), (D(⊥)⊗D(⊥))⊗(D(⊥)⊗D(⊥)), ...

o anche, essendo D(⊥)=i

i⊗i, (i⊗i)⊗(i⊗i), ...

Ogni elemento di questa sequenza e' una funzione doppiamente strict e quindi Yf ha almeno due punti fissi: ⊥ e $\top$. Questo e' sufficiente a mostrare che d non e' un modello banale. Le proprieta' (@), (β) e (η) si possono poi facilmente verificare.

I.2.5 Ancora domini

Per gli scopi dei capitoli seguenti e' opportuno ridefinire alcune delle nozioni introdotte nell'ultimo paragrafo [I.2.4]. Il problema e' puramente tecnico e riguarda solo la complessita' delle espressioni che coinvolgono domini, e non della loro sostanza. Per le nuove definizioni si usera' la stessa notazione delle vecchie. In questo paragrafo la distinzione sara' chiara dal contesto, mentre nei capitoli seguenti si intendera' di usare sempre le nuove definizioni.

Def_13 liste

$\{a^0, \ldots, a[n]\} = \langle a^0, \ldots \langle a[n], \tau \rangle \ldots \rangle \quad n \geq 0$ □

Per selezionare gli elementi di una lista si usera' la notazione:

$a.n = n \to left(a), right(a).(n-1)$

Def_14 prodotti multipli

$a^0 \otimes \ldots \otimes a[n] = (a^0 \otimes \ldots (a[n] \otimes_\tau) \ldots)$ □

Prop_19

$a^0, \ldots, a[n] : D \Rightarrow a^0 \otimes \ldots \otimes a[n] : D$ □

Vale la relazione

$\{a^0, \ldots, a[n]\} : b^0 \otimes \ldots \otimes b[n] \Leftrightarrow \forall 0 \leq i \leq n . a[i] : b[i]$

In caso di prodotti di prodotti, l'estensione dei prodotti multipli viene determinata dall'uso esplicito delle parentesi: $a^0 \otimes (a^1 \otimes a^2 \otimes a^3)$ e' il prodotto di due iniezioni di cui la seconda e' il prodotto di tre iniezioni.

Def_15 unioni multiple

$a^0 \oplus \ldots \oplus a[n] = (a^0 \oplus \ldots (a[n] \oplus_\tau) \ldots) \quad n \geq 0$ □

Prop_20

$a^0, \ldots, a[n] : D \Rightarrow a^0 \oplus \ldots \oplus a[n] : D$ □

L'estensione delle unioni multiple viene determinato dall'uso delle parentesi come in Def 14. Le unioni multiple vengono usate insieme alle funzioni:

```
in(r,n,x) = r((λf./n x.n->⟨x,⊥⟩,⟨⊥,f(n-1,x)⟩)(n,x))
is(n,x) = n->which(x),is(n-1,right(x))
out(x) = which(x)->left(x),out(right(x))
```

Valgono le proprieta':

$$in(a^0 \oplus \ldots \oplus a[n],i,x) = \{\bot, \ldots, x[i], \ldots, \bot\} \Leftrightarrow x:a[i] \quad \forall 0 \le i \le n$$

$$is(i,\{\bot, \ldots, x[j], \ldots, \bot\}) = (i=j) \quad \forall 0 \le i,j \le n$$

$$out(\{\bot, \ldots, x[i], \ldots, \bot\}) = x[i] \quad \forall 0 \le i \le n$$

$$\{\bot, \ldots, x[i], \ldots, \bot\} : a^0 \oplus \ldots \oplus a[n] \Leftrightarrow x:a[i] \quad \forall 0 \le i \le n$$

Def_16 iterazioni

$$a^* = \lambda f./x.\langle a(left(x)),f(right(x))\rangle \quad \square$$

Prop_21

$$a:D \Rightarrow a^*:D \quad \square$$

I punti fissi di a^* sono del tipo $\{a^0, \ldots, a[n]\}$ oppure $\{a^0, \ldots, a[n], \ldots\}$. A tal fine si puo' notare che $_\tau:a^*$ (infatti $a^*:D \Rightarrow {}_\tau \le a^*({}_\tau)$). Gli elementi di $R(a^*)$ sono cioe' le liste finite o infinite di elementi di $R(a)$.

$$x:a^* \Leftrightarrow left(x):a \ \&\ right(x):a^*$$

Gli elementi del lattice completo $R(a^*)$ sono cosi' ordinati:

$$\forall x,y:a^*.\ x \le y \Leftrightarrow left(x) \le left(y) \ \&\ right(x) \le right(y)$$

Def_17 insiemi

```
set(a) = (λf./a x.x=left(a)->x,f(right(a),x))(a)   □
```

Dove a e' interpretata come una lista di elementi di Pw appartenenti ad un insieme discriminabile (per la precisione esistono varie versioni di set, una per ogni insieme discriminabile dal quale vogliamo costruire domini insieme. Questa distinzione non verra' comunque fatta esplicitamente ed ogni volta si supporra' di usare l'adeguata versione di

set). Si usera' la sintassi (da non confondere con quella usata per i veri insiemi):

$\{a^0, \ldots, a[n]\} = set(\langle a^0, \ldots, a[n]\rangle)$

Prop_22

$\forall a^0, \ldots, a[n] \in Pw.\ \{a^0, \ldots, a[n]\} : D$ □

Vale la proprieta'

$a[i] : x \iff x = \langle a^0, \ldots, a[i], \ldots, a[n]\rangle \quad \forall 0 \le i \le n$

I.3: Il formalismo di Scott-Strachey

I.3.0: Semantica dei linguaggi

In [I.2], LAMBDA appare come un linguaggio privilegiato perche' viene definito in modo da possedere (una volta sviluppata la teoria delle funzioni continue) una semantica semplice su Pw. In generale, dato un qualsiasi linguaggio L, e' sempre ragionevole andare a cercare un modello che rispecchi da vicino la struttura del linguaggio, cosi' che la semantica (cioe' la funzione di valutazione) si riduca ad una traduzione quasi lineare tra due formalismi diversi: quello del linguaggio (nel nostro caso le stringhe di LAMBDA) e quello del modello (le espressioni di teoria degli insiemi).

Il problema e' dove andare a cercare tutti questi modelli: occorre sviluppare ogni volta una teoria simile a quella delle funzioni continue su Pw? La risposta fortunatamente e' no, perche' possiamo trovare proprio in Pw tutti i modelli di cui abbiamo bisogno, costruendoli per mezzo di iniezioni.

L'altro problema e' come esprimere le valutazioni. In [I.2] e' stato usato un metodo (informale ma facilmente formalizzabile) per cui ogni stringa di LAMBDA viene proiettata su un opportuno valore in Pw. Per parlare di Pw abbiamo adesso a disposizione un linguaggio ad alto livello (LAMBDA) implementato sul meno leggibile (ma semanticamente ben noto) linguaggio macchina di teoria degli insiemi. Anche LAMBDA e' adesso semanticamente ben noto, e con un passo di astrazione possiamo considerarlo avente una semantica

propria, dimenticandoci del modello sottostante. La semantica di un linguaggio L allora non viene data

In termini di elementi di Pw, ma (indirettamente) in termini di LAMBDA-espressioni. La funzione di valutazione di L si configura come un interprete per L scritto in LAMBDA, come viene mostrato in dettaglio nei prossimi paragrafi, dove il linguaggio L scelto come esempio e' (casualmente) lo stesso LAMBDA.

Il principale vantaggio di questo procedimento e' che l'esistenza delle valutazioni e dei modelli (che sono spesso assai complessi e poco intuitivi) non deve essere dimostrata perche' e' garantita dal teorema del punto fisso (Prop 6) e dalle proprieta' di chiusura delle iniezioni (Prop 18).

I.3.1: Domini sintattici

Per poter definire le valutazioni e i modelli in Pw, anche i linguaggi devono essere rappresentati in qualche modo in Pw. Fissato un dominio di caratteri char (ad esempio char = {0,1, ... ,n}, il dominio dei primi n+1 numeri interi) una stringa del linguaggio sara' un elemento

x:string dove string = char*

A questo punto abbiamo bisogno di un LAMBDA-termine

parse : string⊗exp

che trasforma una stringa dalla sintassi abituale in sintassi astratta. Mettiamo a confronto questi due tipi di sintassi per il linguaggio LAMBDA:

≤var≥ := $a^0 \mid a^1 \mid \ldots$	var = num
≤term≥ := ≤var≥	exp = var ⊕
0	τ ⊕
≤term≥+1	exp ⊕
≤term≥-1	exp ⊕
≤term≥:>≤term≥,≤term≥	(exp⊗exp⊗exp) ⊕
≤term≥(≤term≥)	(exp⊗exp) ⊕
/≤var≥.≤term≥	(var⊗exp)

Il dominio exp viene detto dominio sintattico del linguaggio LAMBDA. Le variabili vengono rappresentate da numeri e lo zero dal dominio $R(\tau)=\{\tau\}$. Un x:exp rappresenta poi (ad esempio) un ≤term≥+1 o un ≤term≥-1 a seconda della sua posizione nella unione disgiunta.

La funzione parse non viene di solito definita esplicitamente, ma vengono fornite delle proprieta' che la caratterizzano e da cui puo' facilmente essere ricavata. Usando la notazione

⟦s⟧ = parse(s)

e indicando l'n-esima variabile con a ed i termini con M,N,P (tutti elementi di R(string)) possiamo scrivere:

$$\llbracket a \rrbracket = in(exp, 0, n)$$
$$\llbracket 0 \rrbracket = in(exp, 1, \tau)$$
$$\llbracket M+1 \rrbracket = in(exp, 2, \llbracket M \rrbracket)$$
$$\llbracket M-1 \rrbracket = in(exp, 3, \llbracket M \rrbracket)$$
$$\llbracket P:>M,N \rrbracket = in(exp, 4, \langle \llbracket P \rrbracket, \llbracket M \rrbracket, \llbracket N \rrbracket \rangle)$$
$$\llbracket M(N) \rrbracket = in(exp, 5, \langle \llbracket M \rrbracket, \llbracket N \rrbracket \rangle)$$
$$\llbracket \lambda a.M \rrbracket = in(exp, 6, \langle n, \llbracket M \rrbracket \rangle)$$

Un dominio sintattico e' quindi un dominio usato per codificare un linguaggio. Questo rende piu' agevole la successiva opera di valutazione del linguaggio stesso.

I.3.2: Domini semantici

Un dominio semantico e' un dominio usato come modello per la semantica di un linguaggio. Nel caso del LAMBDA possiamo fissare il modello:

val = any

cio' e' ovvio perche' un modello di LAMBDA e' proprio Pw=R(any). I modelli hanno in genere aspetti piu' complessi; ad esempio un linguaggio strictLAMBDA in cui l'unico condizionale disponibile sia il condizionale doppiamente strict "->" [I.2.2], ha come modello:

val = num⊕(val⊗val)

in cui i valori sono i singoletti e le gerarchie di funzioni su di essi. I domini semantici dei linguaggi di programmazione sono generalmente assai piu' complessi di quelli qui mostrati. Un esempio non banale si trova in [III.2.1] ed e' il dominio semantico del linguaggio TAU.

I.3.3 Ambienti

Un ambiente e' una funzione che associa ad ogni variabile un valore del modello e viene usata dalla funzione di valutazione per memorizzare le variabili incontrate e i loro valori.

env = var∅val

Sugli ambienti si definisce una operazione di estensione, cioe' l'associazione di una nuova coppia variabile-valore che si aggiunge ed eventualmente si sovrappone a quelle gia' esistenti

```
extend = λ(n:num,v:val,e:env)->env.
                λm.m=n:>v,e(m)
```

Si usa la notazione:

e[v/n] = extend(n,v,e)

Come ambiente vuoto, da cui verranno costruiti tutti gli ambienti per successive estensioni, si usa:

arid = τ

Notare che τ:var∅val.

I.3.4: Valutazioni

Una valutazione e' una funzione:

evaluate : exp→val

che associa ad ogni stringa del linguaggio un valore nel modello. Cominciamo col vedere una semplice funzione di valutazione per LAMBDA:

evaluate = λ(x:exp)->val. eval(close(x),arid)

dove close:exp→exp e' una iniezione che trasforma tutte le espressioni in espressioni chiuse, eventualmente racchiudendole in opportune λ-astrazioni che legano le variabili libere. eval:exp→(env→val) puo' essere scritta:

```
eval = λ(x:exp,e:env)->val.
        is(0,x)->e(out(x)),
        is(1,x)->0,
        is(2,x)->eval(out(x),e)+1,
        is(3,x)->eval(out(x),e)-1,
        is(4,x)->eval(out(x).0,e):>
                  eval(out(x).1,e),eval(out(x).2,e),
        is(5,x)->eval(out(x).0,e)(eval(out(x).1,e)),
        is(6,x)->λa.eval(out(x).1,e[a/out(x).0]),⊥
```

E' comunque di uso corrente adottare una notazione meno procedurale che racchiude in se' la definizione di eval e di parse. La semantica di un linguaggio verra' d'ora in poi presentata in questa forma conosciuta come <u>notazione di Scott-Strachey</u>, dove eval⟦M⟧e = eval(⟦M⟧,e):

```
eval⟦a⟧e = e⟦a⟧
eval⟦0⟧e = 0
eval⟦M+1⟧e = eval⟦M⟧e+1
eval⟦M-1⟧e = eval⟦M⟧e-1
eval⟦P:>M,N⟧e = eval⟦P⟧e:>eval⟦M⟧e,eval⟦N⟧e
eval⟦M(N)⟧e = eval⟦M⟧e(eval⟦N⟧e)
eval⟦λa.M⟧e = λa.eval⟦M⟧e[a/⟦a⟧]
```

Nella semantica di linguaggi diversi da LAMBDA si presenta un problema tecnico a proposito dell'immersione dei valori negli appropriati domini semantici. Ad esempio nel gia'

citato strictLAMBDA [I.3.2] con modello val=num⊕(val⊗val):

```
eval⟦M+1⟧e = let m<-eval⟦M⟧e
             in is(0,m)->in(val,0,out(m)+1),⊥
```

cioe' ⟦M⟧ deve valutare ad un numero (0-esimo addendo dell'unione disgiunta val) ed in tal caso il valore di M viene incrementato di 1 e iniettato in val con l'operazione in. In certi casi puo' essere problematico ricordarsi che 0 corrisponde all'addendo num, 1 all'addendo (val⊗val) ecc. Si converra' allora di etichettare gli elementi delle unioni disgiunte e si utilizzeranno queste etichette nelle funzioni is e in al posto dei numeri d'ordine. Ecco come esempio la semantica del linguaggio strictLAMBDA:

```
val = num::num⊕fun::(val⊗val)

eval⟦a⟧e = e⟦a⟧
eval⟦0⟧e = in(val,num,0)
eval⟦M+1⟧e = let m<-eval⟦M⟧e
             in is(num,m)->in(val,num,out(m)+1),⊥
eval⟦M-1⟧e = let m<-eval⟦M⟧e
             in is(num,m)->in(val,num,out(m)-1),⊥
eval⟦P->M,N⟧e = let p<-eval⟦P⟧e
                in is(num,p)->out(p)->
                     eval⟦M⟧e,eval⟦N⟧e,⊥
eval⟦M(N)⟧e = let m<-eval⟦M⟧e
              in is(fun,m)->out(m)(eval⟦N⟧e),⊥
eval⟦λa.M⟧e = in(val,fun,(λa.eval⟦M⟧e[a/⟦a⟧]))
```

Una analoga convenzione si usera' per i prodotti: se x:(a⁰::d⁰⊗ ... ⊗a[n]::d[n]) allora per ogni 0≤i≤n si definisce

```
x.a[i] = x.i
```

E' bene sottolineare che "a::d" e "x.a" non sono operazioni, ne' rappresentano LAMBDA-termini, ma sono solo notazioni che permettono di usare nomi per numeri, allo scopo di migliorare la leggibilita' delle espressioni.

I.3.5: Continuazioni

Le continuazioni sono state introdotte come uno strumento per la descrizione della semantica del goto e di vari regimi di controllo non applicativi [Reynolds 74] [Strachey 74]. La discussione verra' condotta sulla base di un esempio: il linguaggio LAMBDA+goto di cui viene data la seguente semantica:

```
val = any
env = var⊗val ; arid = τ
cont = val⊗val ; stop = λ(x:val)->val.x

evaluate = λ(x:exp)->val. eval(close(x),arid,stop)

eval⟦a⟧ek = k(e⟦a⟧)
eval⟦0⟧ek = k(0)
eval⟦M+1⟧ek = eval⟦M⟧e(λv.k(v+1))
eval⟦M-1⟧ek = eval⟦M⟧e(λv.k(v-1))
eval⟦P:>M,N⟧ek = eval⟦P⟧e(λv.v:>eval⟦M⟧ek,eval⟦N⟧ek)
eval⟦M(N)⟧ek = eval⟦M⟧e(λv.k(v(eval⟦N⟧e stop)))
eval⟦λa.M⟧ek = k(λa.eval⟦M⟧e[a/⟦a⟧]stop)
eval⟦a:M⟧ek = eval⟦a⟧(λe'.e[eval⟦M⟧e' stop/⟦a⟧])k
eval⟦goto a⟧ek = eval⟦a⟧e stop
```

Come si vede eval ha un argomento in piu': una continuazione k:val⊗val. eval⟦M⟧ek ha il significato di valutare ⟦M⟧ nell'ambiente e, e poi passare il valore v di ⟦M⟧ a k effettuando l'applicazione k(v) (infatti M valutera' sempre in fin dei conti ad una variabile, a zero o ad una astrazione, e la semantica di questi costrutti e' esattamente del tipo k(...), mentre per gli altri costrutti si ha una chiamata ricorsiva ad eval). La continuazione ha il senso di "cio' che si fa dopo la valutazione di ⟦M⟧" e impone una rigida sequenzializzazione delle valutazioni. Ad esempio cio' che si fa dopo la valutazione di ⟦M⟧ in ⟦M+1⟧ e' (λv.k(v+1)), cioe' si prende il valore v di ⟦M⟧, lo si incrementa e poi si fa cio' che

rimaneva da fare dopo la valutazione di ⟦M+1⟧, cioe' k. Alla fine rimarra' da fare solo stop=λv.v , la continuazione finale, e se v e' il valore calcolato fino a quel momento, stop(v)=v e otteniamo il risultato.

```
evaluate(⟦0+1⟧) = eval⟦0+1⟧arid stop
                = eval⟦0⟧arid(λv.stop(v+1))
                = (λv.stop(v+1))(0)
                = stop(0+1)
                = 1
```

Il condizionale ⟦P:>M,N⟧ viene trattato valutando ⟦P⟧ e lasciando alla continuazione il compito di scegliere tra valutare ⟦M⟧ o ⟦N⟧ a secondo del valore di ⟦P⟧. Nell'applicazione ⟦M(N)⟧ si valuta ⟦M⟧ e si passa il suo valore alla continuazione, la quale lo applica al valore di ⟦N⟧. Nella λ-astrazione ⟦λa.M⟧ si passa alla continuazione una funzione che valuta ⟦M⟧ con la continuazione stop.

In ⟦a:M⟧ il valore della label (variabile) ⟦a⟧ viene definito come il valore dell'espressione ad essa associata ⟦M⟧. Il valore di ⟦M⟧ va pero' calcolato tenendo conto della eventuale presenza di goto nel suo interno, e il valore di un goto e' definito come il valore della sua label. Se questa label e' ⟦a⟧, siamo di fronte ad una definizione circolare e per risolverla occorre un punto fisso

```
eval⟦a:M⟧ek = eval⟦M⟧(μe'.e[eval⟦M⟧e' stop/⟦a⟧])k
```

cioe' ⟦M⟧ viene valutato in un ambiente in cui ad ⟦a⟧ e' associato il valore di ⟦M⟧ (valutato in un ambiente in cui ad ⟦a⟧ e' associato il valore di ⟦M⟧ ... ecc.) come se ⟦M⟧ fosse gia' stato valutato.

Quando in ⟦M⟧ si incontra un ⟦goto a⟧ il valore del goto e' il valore di ⟦a⟧, ignorando la continuazione:

```
eval⟦goto a⟧ek = eval⟦a⟧e stop
```

Incontrare un ⟪goto a⟫ non e' la stessa cosa che incontrare semplicemente ⟪a⟫. Questo succede perche' la continuazione da utilizzare dopo il salto e' quella di ⟪a:M⟫ e non quella del goto.

Nel modo illustrato si possono effettuare solo salti all'indietro ad una etichetta gia' incontrata, ma il costrutto e' adesso facilmente generalizzabile:

```
eval⟪[a⁰:M⁰, ... ,a[n]:M[n]]⟫ek =
  eval⟪a⁰⟫(λe'.e[eval⟪M⁰⟫e' stop/⟪a⁰⟫] ...
                [eval⟪M[n]⟫e' stop/⟪a[n]⟫])k
```

I.3.6: Memorie

Il linguaggio LAMBDA che stiamo adoperando per dare la semantica dei linguaggi e' totalmente privo di side-effects. Come e' possibile allora dare ad esempio la semantica dell'assegnamento? Per poterlo fare occorre complicare un po' la struttura degli ambienti e delle continuazioni e introdurre i domini delle locazioni e degli stores.

```
loc = num
env = var⊗loc
arid = τ
store = free::loc⊗val::(loc⊗val)
void = {0,τ}
cont = val⊗(store⊗val)
stop = λ(x:val,y:store)->val. x
```

Un ambiente e' adesso una funzione che fa passare da variabili a locazioni; le locazioni contengono poi i valori delle variabili. Uno store e' una coppia costituita dalla prima locazione ancora libera (cioe' dal suo numero d'ordine) e da una funzione che associa ad ogni locazione il suo valore. Le continuazioni hanno un secondo argomento che e' uno store, e che viene passato da una continuazione all'altra per preservare le modifiche (cioe' le estensioni) che vi sono state operate. Si useranno le funzioni di estensione:

```
extend = λ(n:num,l:loc,e:env)->env.
           λm.m=n->l,e(m)
assign = λ(n:num,v:val,s:store)->store.
           {s.free,λm.m=n->v,s.val(m)}
allocate = λ(v:val,s:store)->store.
             {s.free+1,assign(s.free,v,s)}
```

con l'usuale abbreviazione e[l/n] = extend(n,l,e). Assign e' l'analogo di extend sugli stores. Allocate invece mette un valore in una locazione nuova, mai usata prima, e incrementa il puntatore alla prima locazione libera, s.free.

Vediamo quindi la semantica del linguaggio LAMBDA+assign:

```
evaluate = /(x:exp)->val.
             eval(close(x),arid,void,stop)

(1) eval⟦a⟧esk = k(s.val(e⟦a⟧),s)
(2) eval⟦0⟧esk = k(0,s)
(3) eval⟦M+1⟧esk = eval⟦M⟧es(/v s.k(v+1,s))
(4) eval⟦M-1⟧esk = eval⟦M⟧es(/v s.k(v-1,s))
(5) eval⟦P:>M,N⟧esk =
       eval⟦P⟧es(/v s.v:>eval⟦M⟧esk,eval⟦N⟧esk)
(6) eval⟦M(N)⟧esk =
      eval⟦M⟧es(/v s.eval⟦N⟧es(/v's'.k(v(v',s'),s')))
(7) eval⟦/a.M⟧esk =
       k(/a s.eval⟦M⟧e[s.free/⟦a⟧]allocate(a,s)stop)
(8) eval⟦a<-M⟧esk = eval⟦M⟧es(/v s.k(v,assign(e⟦a⟧,v,s)))
(9) eval⟦[M⁰, ... ,M[n]]⟧esk =
      eval⟦M⁰⟧es(/v⁰s⁰.eval⟦M¹⟧es(/v¹s¹. ...
                       eval⟦M[n]⟧esk .. ))
```

Eval ha ancora un argomento in piu': uno store. eval⟦M⟧esk significa adesso valutare ⟦M⟧ nell'ambiente e e nello store s, e passare il valore v di ⟦M⟧ ed il nuovo store s' (modificato dalla valutazione di ⟦M⟧) alla continuazione k.

Gli stores vengono modificati per semplice estensione, come gli ambienti. La differenza sta nel modo in cui i primi vengono usati, passandoli sempre attraverso le continuazioni. In questo modo una volta che uno store e' stato modificato dalla valutazione di un assegnamento ⟦a<-M⟧ non c'e' modo di recuperare il vecchio valore di ⟦a⟧, in nessuna parte del programma. Questo contrasta col tipico funzionamento a stack degli ambienti, per cui una variabile globale puo' essere coperta da una variabile locale, ma puo' tornare poi accessibile.

Commentiamo le equazioni semantiche. Nella (1) il valore di una variabile si ottiene ricavando prima la locazione in cui questa e' contenuta con e⟦a⟧, e poi applicando alla locazione la parte funzione dello store. Le (2) (3) (4) (5) si limitano a passare oltre lo store. La (9) e' una

valutazione sequenziale di n termini, il cui valore e' l'ultimo termine. La (8) implementa l'assegnamento passando alla continuazione k uno store esteso col nuovo valore della locazione e⟦a⟧. La variabile ⟦a⟧ si riferisce ora alla stessa locazione, ma ad un diverso valore. Nella (7) si puo' vedere che gli stores vengono passati alle funzioni al momento dell'applicazione, insieme all'argomento. In questo modo la stessa funzione puo' essere valutata in stores diversi, con diversi valori (ma stesse locazioni) per le variabili libere. Per ogni variabile locale e per ogni attivazione di funzione viene generata una nuova locazione in cui viene posto il valore dell'argomento. La (6) mostra un meccanismo di chiamata per valore: prima si valuta l'argomento ⟦N⟧ e poi si fa l'applicazione nello store generato dalla valutazione di ⟦N⟧. Durante il passaggio dei parametri gli argomenti vengono "copiati" (cioe' il passaggio non e' per locazione): a causa della generazione di sempre nuove locazioni per i parametri formali, un assegnamento ad una variabile locale non puo' ripercuotersi in alcun modo nella funzione chiamante:

$$(\lambda a.[(\lambda x.x\leftarrow 2)(a), a])(3) = 3$$

Una chiamata per nome puo' essere realizzata secondo la tecnica dei thunks dell'ALGOL:

(1') $\mathrm{eval}⟦a⟧esk = k(s.\mathrm{val}(e⟦a⟧)(s), s)$

(6') $\mathrm{eval}⟦M(N)⟧esk =$
$\quad \mathrm{eval}⟦M⟧es(\lambda v\ s.k(v(\lambda s.\mathrm{eval}⟦N⟧es\ \mathrm{stop}), s))$

Tutti i parametri attuali in (6') vengono trasformati in funzioni (thunks) che aspettano uno store nel quale valutare il loro corpo (che e' il parametro attuale vero e proprio). I parametri formali, cioe' le variabili, vengono valutate ed

il loro valore (un thunk) viene applicato allo store corrente. In questo modo gli assegnamenti presenti nei parametri attuali vengono eseguiti ogni volta che viene valutato il parametro formale corrispondente nel corpo della funzione:

$(\lambda a.(\lambda x.[x,x,x])(a<-a+1))(0) = 3$

Una ulteriore discussione sul passaggio dei parametri si trova in [II.5].

I.4: Trasformazioni di interpreti

I.4.0: Interpreti metacircolari

A questo punto dovrebbe essere chiara la potenza del metodo di definizione della semantica illustrato nel capitolo precedente, e di come esso riesca ad esprimere tutti i concetti teorici ed implementativi coinvolti nei linguaggi di programmazione. In esso e' possibile definire vari tipi di semantica per lo stesso linguaggio (una semantica astratta, un interprete, un compilatore, un loader, un linker ecc.) e dimostrare la loro equivalenza [Milne 76]. Il metodo si e' rivelato in grado di fornire la semantica di linguaggi reali come il Pal [Milne 71] e l'Algol68 [Milne 72] e fornisce una completa soluzione teorica e pratica al problema della semantica dei linguaggi programmativi.

Tuttavia l'argomento non si esaurisce qui. Non si puo' non notare ad esempio una certa pesantezza della notazione che puo' contribuire ad oscurare, invece che a chiarire, il significato di certi costrutti. Inoltre una descrizione della semantica deve servire anche come base per una implementazione, e non deve precludere possibilita' di bootstrapping (una LAMBDA-machine e' molto difficile da implementare ed in ogni caso costituisce uno strato di software aggiuntivo tra un linguaggio e la sua macchina).

Per questi motivi faremo un salto di astrazione simile a quello compiuto in [I.3.0]: se abbiamo dato in LAMBDA la semantica di un linguaggio L, adesso L e' ben noto e

possiamo dimenticarci di LAMBDA utilizzando sempre L. La prima cosa che faremo in L e' scrivere un interprete per L, analogamente a quanto abbiamo fatto in [I.3.4] per LAMBDA. Un interprete scritto in se stesso viene detto metacircolare e l'utilita' di questa tecnica e' nota ad esempio in vari problemi di bootstrapping.

Il passo successivo consiste nell'introdurre trasformazioni successive nell'interprete metacircolare per semplificarlo e renderlo sempre piu' simile ad un interprete in linguaggio macchina. Il risultato e' un interprete iterativo scritto ancora in L, ma che non utilizza caratteristiche complesse di L (come la ricorsione, i ritorni funzionali ecc.), e che puo' essere tradotto uno a uno in un linguaggio macchina. Le trasformazioni da effettuare sono standard e sono valide praticamente per ogni linguaggio procedurale. In [III.0] e [III.1] viene seguita questa traccia per il linguaggio TAU, mentre in questo capitolo viene illustrato il procedimento ponendo L = strictLAMBDA.

Prima di poter scrivere un interprete metacircolare per il linguaggio strictLAMBDA (di cui abbiamo visto in [I.3.4] un LAMBDA-interprete) occorre in qualche modo fornire strictLAMBDA di domini e relative operazioni, analogamente a quanto abbiamo fatto in [I.2] per LAMBDA. Estendiamo quindi strictLAMBDA con le seguenti (strictLAMBDA-) definizioni:

(1) $\perp$ = $(\lambda x.x(x))(\lambda x.x(x))$
(2) x=y = $(\mu f.\lambda x\ y.x\to(y\to 0,1),y\to 1,f(x-1,y-1))(x,y)$
(3) $\langle a,b\rangle$ = $\lambda x.x\to a,b$
(4) left = $\lambda a.a(0)$
(5) right = $\lambda a.a(1)$
(6) $a\circ b$ = $\lambda x.a(b(x))$
(7) any = i = $\lambda x.x$
(8) bool = $\lambda x.x\to 0,1$

```
(9)  num = μf.λn.n->0,f(n-1)+1
(10) fun = λf x.f(x)
(11) a⊗b = λx.b°x°a
(12) λ(x:a)->b.M = (a⊗b)(λx.M)
     λ(x°:a°, ... ,x[n]:a[n])->b.M =
        λ(x°:a°)->any. ... λ(x[n]:a[n])->b.M
(13) {a°, ... ,a[n]} = <a°, ... <a[n],⊥> .. >   n≥0
(14) a.n = n->left(a),right(a).(n-1)
(15) a°⊗ ... ⊗a[n] = λx.{a°(x.0), ... ,a[n](x.n)}
(16) a°⊕ ... ⊕a[n] =
       λx.left(x)-> <0,a°(right(x))>, ...
          left(x-n)-> <n,a[n](right(x))>,⊥
(17) in(r,n,x) = r(<n,x>)
(18) is(n,x) = (n=left(x))
(19) out(x) = right(x)
(20) a* = μf.λx.<a(left(x)),f(right(x))>
(21) set(a) = (μf.λa x.x=left(a)->x,f(right(a),x)))(a)
(22) {a°, ... ,a[n]} = set({a°, ... ,a[n]})
(23) e[v/n] = λm.m=num(n)->val(v),env(e)(m)
```

Le differenze tra queste strictLAMBDA-definizioni e le cmonime LAMBDA-definizioni risiedono soprattutto nel fatto che in strictLAMBDA non sono esprimibili τ e u. Per quanto riguarda i domini l'uso di queste definizioni in [I.2] avrebbe portato a delle retrazioni, ma non a delle iniezioni.

Tutte le notazioni metasintattiche introdotte per il LAMBDA saranno applicate anche a strictLAMBDA.

I.4.1: Eliminazione della ricorsione

Cominciamo quindi con lo scrivere l'interprete metacircolare del linguaggio strictLAMBDA:

Interprete A

```
var = num
exp = var::var ⊕
      zero::⊥ ⊕
      succ::exp ⊕
      pred::exp ⊕
      cond::(if::exp⊗then::exp⊗else::exp) ⊕
      appl::(fun::exp⊗arg::exp) ⊕
      lamb::(binder::var⊗body::exp)

val = any

env = var⊗val
arid = λ(x:var)->val.⊥

evaluate = λ(x:exp)->val. eval(close(x),arid)
```

```
(1a) eval⟦a⟧e = e⟦a⟧
(2a) eval⟦0⟧e = 0
(3a) eval⟦M+1⟧e = eval⟦M⟧e+1
(4a) eval⟦M-1⟧e = eval⟦M⟧e-1
(5a) eval⟦P->M,N⟧e = eval⟦P⟧e->eval⟦M⟧e,eval⟦N⟧e
(6a) eval⟦M(N)⟧e = eval⟦M⟧e(eval⟦N⟧e)
(7a) eval⟦λa.M⟧e = λa.eval⟦M⟧e[a/⟦a⟧]
```

Questo interprete e' strettamente ricorsivo nel senso della seguente definizione:

Def Una funzione f=(λx. ... f ...) e' strettamente ricorsiva se almeno una delle chiamate ricorsive a f nel corpo di f:

i) compare come argomento di qualche funzione g diversa dal condizionale, oppure:

ii) compare come primo argomento di un condizionale □

In (3a) ad esmpio eval e' argomento della funzione +, in (5a) e' primo argomento di un condizionale e in (6a) e' argomento di se stessa.

Una funzione non strettamente ricorsiva prende il nome di

iterativa, perche' in questo caso le chiamate ricorsive possono essere implementate semplicemente con una riassegnazione dei parametri formali con i nuovi parametri attuali, e con un salto all'inizio della funzione.

La definizione e' intuitivamente generalizzabile alle funzioni mutuamente ricorsive. Essa e' inoltre una definizione sintattica: possono esistere due rappresentazioni della stessa funzione, una strettamente ricorsiva e una iterativa. Un modo sistematico di trasformare una rappresentazione strettamente ricorsiva in una iterativa e' quello di introdurre le continuazioni.

Interprete B

```
cont = val→val
stop = /(x:val)->val. x

evaluate = /(x:exp,k:cont)->val.
             eval(close(x),arid,stop)

(1b) eval⟦a⟧ek = k(e⟦a⟧)
(2b) eval⟦0⟧ek = k(0)
(3b) eval⟦M+1⟧ek = eval⟦M⟧e(/v.k(v+1))
(4b) eval⟦M-1⟧ek = eval⟦M⟧e(/v.k(v-1))
(5b) eval⟦P->M,N⟧ek =
        eval⟦P⟧e(/v.v->eval⟦M⟧ek,eval⟦N⟧ek)
(6b) eval⟦M(N)⟧ek = eval⟦M⟧e(/v.k(v(eval⟦N⟧e stop)))
(7b) eval⟦/a.M⟧ek = k(/a.eval⟦M⟧e[a/⟦a⟧]stop)
```

Adesso la (3b) e la (4b) sono diventate iterative, e questo e' gia' un passo avanti. Nelle (5b) (6b) (7b) ci sono ancora delle chiamate strettamente ricorsive, ma queste scompariranno nel corso di successive trasformazioni, effettuate per altri motivi. Il merito andra' comunque alle continuazioni.

I.4.2: Eliminazione degli argomenti funzionali

Sempre nel tentativo di ricondurci a interpreti via via piu' semplici, cerchiamo questa volta di eliminare quelle funzioni passate come argomenti ad altre funzioni. Nell'interprete B, queste sono gli ambienti e:var⊗val e le continuazioni k:val⊗val. La λ-astrazione in (7b) non viene considerata, perche' in realta' e' un ritorno funzionale (confrontare con (7a)) e sara' oggetto del prossimo paragrafo. Cominciamo con gli ambienti. Un modo di eliminare una funzione e' quello di sostituirla con un dato che la rappresenti. Ridefiniamo quindi env nel seguente modo:

```
arid = in(env,arid,⊥)
env = arid::⊥ ⊗
      assoc::(var::var⊗val::val⊗env::env)
extend = λ(n:var,v:val,e:env)->env.
           in(env,assoc,⟨n,v,e⟩)
```

Adesso env e' una lista di coppie variabile-valore. In (1b) gli ambienti vengono applicati come funzioni e dobbiamo in qualche modo simulare questa applicazione. Se indichiamo con env[old] la vecchia definizione (funzionale) di env, e con env[new] la nuova, abbiamo bisogno di una funzione f tale che

$$\forall a.\ env[old](a) = f(env[new],a)$$

Una funzione che soddisfa questa condizione e'

```
lookup = λ(x:var,e:env)->val.
           x=out(e).var→out(e).val,
                        lookup(x,out(e).env)
```

Lookup e' iterativa e quindi non abbiamo peggiorato la situazione precedente. Adesso possiamo scrivere un interprete con un solo tipo di argomenti funzionali (le

continuazioni), che e' identico al precedente tranne che in (1b):

(1b') eval⟦a⟧ek = k(lookup(⟦a⟧,e)) e dove in (7b) "e[a/⟦a⟧]" abbrevia ora la nuova definizione di extend.

L'eliminazione delle continuazioni e' un po' piu' complessa perche' ve ne sono di quattro tipi diversi in (3b) (4b) (5b) (6b). Inoltre se vogliamo sostituire la continuazione in (5b) con un dato, dobbiamo memorizzare in questo dato anche ⟦M⟧,⟦N⟧,e,k che sono variabili libere nella funzione di continuazione e variano da una chiamata all'altra (da un dato all'altro). Ridefiniamo quindi le continuazioni in questo modo:

```
stop = in(cont,stop,⊥)
cont = stop::⊥ ⊕
       succcont::cont ⊕
       predcont::cont ⊕
       condcont::(then::exp⊗else::exp⊗
                  env::env⊗cont::cont) ⊕
       applcont::(arg::exp⊗env::env⊗cont::cont)
```

La funzione che si preoccupa di simulare l'applicazione delle continuazioni e' allora:

```
capply = λ(k:cont,v:val)->val.
           is(stop,k)->v,
           is(succcont,k)->capply(out(k),v+1),
           is(predcont,k)->capply(out(k),v-1),
           is(condcont,k)->
               eval(v->out(k).then,out(k).else,
                    out(k).env,out(k).cont),
           is(applcont,k)->
               capply(out(k).cont,
                      v(eval(out(k).arg,
                             out(k).env,stop))),⊥
```

capply fallisce di essere iterativa solo per la chiamata ad

eval nell'ultima riga. L'interprete diventa:

Interprete C

```
(1c) eval⟦a⟧ek = capply(k,lookup(⟦a⟧,e))
(2c) eval⟦0⟧ek = capply(k,0)
(3c) eval⟦M+1⟧ek = eval⟦M⟧e in(cont,succcont,k)
(4c) eval⟦M-1⟧ek = eval⟦M⟧e in(cont,predcont,k)
(5c) eval⟦P->M,N⟧ek =
        eval⟦P⟧e in(cont,condcont,⟨⟦M⟧,⟦N⟧,e,k⟩)
(6c) eval⟦M(N)⟧ek = eval⟦M⟧e in(cont,applcont,⟨⟦N⟧,e,k⟩)
(7c) eval⟦λa.m⟧ek = capply(k,(λa.eval⟦M⟧e[a/⟦a⟧]stop))
```

I.4.3: Eliminazione dei ritorni funzionali

Notiamo che in (7a) viene ritornata come valore una funzione, o equivalentemente che in (7b) e in (7c) viene passata una funzione ad una continuazione. Questa e' una prerogativa di non molti linguaggi (tutti gli algol-like la escludono) e in genere non possiamo sperare che un linguaggio macchina la possegga o la possa facilmente implementare. Il nostro compito sara' quindi quello di eliminare i ritorni funzionali facendo ritornare, anziche' una funzione, un dato che li rappresenti.

Poiche' stiamo compiendo una trasformazione sui valori della valutazione, il nostro modello, che finora era val=any, diventa:

```
val = num::num ⊕ closure::closure
closure = text::exp⊗env::env
```

Un valore e' un numero, oppure una chiusura (una coppia testo-ambiente) rappresentante una funzione. Occorre come al solito una funzione che simuli l'applicazione di una chiusura ad un argomento:

```
apply = λ(f:closure,a:val,k:cont)->val.
          eval(out(out(f).fun).body,
               out(f).env[a/out(out(f).fun).binder],
               k)
```

Dove se f=⟨⟦λa.M⟧,e⟩ allora eval(... , ... ,k) equivale a scrivere eval⟦M⟧e[a/⟦a⟧]k. Notare che anche apply e' una funzione iterativa.

Ecco il nuovo interprete che presenta modifiche in (2d) in

(7d) e in capply:

Interprete D

```
(1d) eval⟦a⟧ek = capply(k,lookup(⟦a⟧,e))
(2d) eval⟦0⟧ek = capply(k,in(val,num,0))
(3d) eval⟦M+1⟧ek = eval⟦M⟧e in(cont,succcont,k)
(4d) eval⟦M-1⟧ek = eval⟦M⟧e in(cont,predcont,k)
(5d) eval⟦P->M,N⟧ek =
        eval⟦P⟧e in(cont,condcont,⟨⟦M⟧,⟦N⟧,e,k⟩)
(6d) eval⟦M(N)⟧ek = eval⟦M⟧e in(cont,applcont,⟨⟦N⟧,e,k⟩)
(7d) eval⟦λa.M⟧ek = capply(k,in(val,closure,⟨⟦λa.M⟧,e⟩))

capply = λ(k:cont,v:val)->val.
          is(stop,k)->v,
          is(succcont,k)->is(num,v)->
            capply(out(k),in(val,num,out(v)+1)),⊥,
          is(predcont,k)->is(num,v)->
            capply(out(k),in(val,num,out(v)-1)),⊥
          is(condcont,k)->is(num,v)->
            eval(out(v)->out(k).then,out(k).else,
                 out(k).env,out(k).cont),⊥,
          is(applcont,k)->
            apply(v,eval(out(k).arg,out(k).env,stop),
                  out(k).cont),⊥
```

I.4.4: Interprete iterativo

L'ultimo interprete presentato fallisce di essere iterativo solo per la chiamata ad eval effettuata nell'ultima riga di capply. Quella chiamata e' provocata dal meccanismo di call-by-name di passaggio dei parametri, per il quale prima si effettua l'applicazione, e poi si valutano i parametri attuali se e quando si incontrano i parametri formali corrispondenti. In un linguaggio con call-by-value questa difficolta' non si presenta ed il relativo interprete sarebbe a questo punto completamente iterativo.

Il problema viene risolto sostituendo la chiamata ad eval con una struttura dati che la rappresenta, detta sospensione, contenente il testo da valutare e l'ambiente in cui valutarlo. Le sospensioni vengono sempre legate a parametri formali da apply, e lookup ha il compito di forzare la valutazione delle sospensioni richiamando (iterativamente) eval.

Ecco quindi finalmente un interprete completamente iterativo:

Interprete_I

```
var = num
exp = var::var ⊕
      zero::1 ⊕
      succ::exp ⊕
      pred::exp ⊕
      cond::(if::exp&then::exp&else::exp) ⊕
      appl::(fun::exp&arg::exp) ⊕
      lamb::(binder::var&body::exp)
```

```
env = arid::⊥ ⊕
      assoc::(var::var⊗susp::susp⊗env::env)
arid = in(env,arid,⊥)
susp = exp::exp⊗env::env
extend = λ(n:var,v:val,e:env)->env.
           in(env,assoc,⟨n,v,e⟩)
lookup = λ(x:var,e:env)->val.
          x=out(e).var->
            eval(out(e).susp.exp,out(e).susp.env,stop),
            lookup(x,out(e).env)

cont = stop::⊥ ⊕
       succcont::cont ⊕
       predcont::cont ⊕
       condcont::(then::exp⊗else::exp⊗
                  env::env⊗cont::cont) ⊕
       applcont::(arg::exp⊗env::env⊗cont::cont)
stop = in(cont,stop,⊥)

val = num::num ⊕ closure::closure
closure = text::exp⊗env::env

apply = λ(f:closure,a:susp,k:cont)->val.
         eval(out(out(f).fun).body,
              out(f).env[a/out(out(f).fun).binder],
              k)

(1e) eval⟦a⟧ek = capply(k,lookup(⟦a⟧,e))
(2e) eval⟦0⟧ek = capply(k,in(val,num,0))
(3e) eval⟦M+1⟧ek = eval⟦M⟧e in(cont,succcont,k)
(4e) eval⟦M-1⟧ek = eval⟦M⟧e in(cont,predcont,k)
(5e) eval⟦P->M,N⟧ek =
       eval⟦P⟧e in(cont,condcont,⟨⟦M⟧,⟦N⟧,e,k⟩)
(6e) eval⟦M(N)⟧ek = eval⟦M⟧e in(cont,applcont,⟨⟦N⟧,e,k⟩)
(7e) eval⟦λa.M⟧ek = capply(k,in(val,closure,⟨⟦λa.M⟧,e⟩))

capply = λ(k:cont,v:val)->val.
          is(stop,k)->v,
          is(succcont,k)->is(num,v)->
            capply(out(k),in(val,num,out(v)+1)),⊥,
          is(predcont,k)->is(num,v)->
            capply(out(k),in(val,num,out(k)-1)),⊥,
          is(condcont,k)->is(num,v)->
            eval(out(v)->out(k).then,out(k).else,
                 out(k).env,out(k).cont),⊥,
          is(applcont,k)->
            apply(v,⟨out(k).arg,out(k).env⟩,out(k).cont),⊥
```

In questo interprete le continuazioni corrispondono grosso modo allo stack di controllo negli interpreti per linguaggi di programmazione. Vi e' pero' una differenza fondamentale nel meccanismo di chiamata delle funzioni. Consideriamo il

programma:

let f<-/x.x in f(0)

L'esecuzione di f(0) in un normale interprete provocherebbe la crescita dello stack di controllo, sul quale verrebbe aggiunto un punto di ritorno per la valutazione del corpo di f. Vediamo invece che cosa accade con le continuazioni sviluppando la semantica di f(0):

```
eval⟦(/x.x)(0)⟧arid stop =
eval⟦/x.x⟧arid in⟨cont,applcont,⟨⟦0⟧,arid,stop⟩⟩ =
capply(in⟨cont,applcont,⟨⟦0⟧,arid,stop⟩⟩,
       in⟨val,closure,⟨⟦/x.x⟧,arid⟩⟩) =
apply(in⟨val,closure,⟨⟦/x.x⟧,arid⟩⟩,
      ⟨⟦0⟧,arid⟩,stop)  =
eval⟦x⟧arid[⟨⟦0⟧,arid⟩/⟦x⟧]stop
```

Come si vede nell'ultima riga, nessun punto di ritorno e' stato aggiunto allo stack di controllo (che e' ancora stop) dopo l'applicazione di f(0). Cio' accade perche' il puntatore di ritorno di f(0) e' lo stesso di x (corpo di f) e sara' la valutazione di x a preoccuparsi di far ritornare il valore finale al punto giusto:

```
eval⟦x⟧arid[⟨⟦0⟧,arid⟩/⟦x⟧]stop =
capply(stop,lookup(⟦x⟧,arid[⟨⟦0⟧,arid⟩/⟦x⟧])) =
lookup(⟦x⟧,arid[⟨⟦0⟧,arid⟩/⟦x⟧]) =
eval⟦0⟧arid stop =
capply(stop,in⟨val,num,0⟩)=
in⟨val,num,0⟩
```

Lo stack di controllo viene quindi alzato solo quando c'e' strettamente bisogno di un punto di ritorno per svolgere una computazione subordinata, e non quando la seconda computazione e' semplicemente successiva alla prima. In quest'ultimo caso rientrano tutte le funzioni iterative (non strettamente ricorsive [I.4.1]) che anche se espresse ricorsivamente, vengono implementate iterativamente (cioe' senza espansione dello stack di controllo).

Una importante conseguenza di questo fatto e' che un

linguaggio di programmazione non ha bisogno di costrutti primitivi per l'iterazione (for, while, loop ecc.) perche' questi possono essere implementati ricorsivamente a software senza perdita di efficienza.

II: IL LINGUAGGIO TAU: NOTE SULLA SEMANTICA

II.0: Notazioni

II.0.0: Notazioni definitorie

Nel seguito per meglio parlare delle singole occorrenze delle variabili, queste verranno indicizzate progressivamente da sinistra a destra:

$$\lambda a^1.(\lambda b^2.f^3(a^4)(b^5))(\lambda f^6.g^7(f^8))$$

dove a^1, b^2 e f^6 sono binders, f^3 e g^7 sono occorrenze libere e a^4, b^5 e f^8 sono occorrenze legate.

Come notazione si introduce anche la sostituzione di un termine al posto di una occorrenza di una variabile; ad esempio $M[N/x^3]$ con significato ovvio e purche' x^3 non sia un binder. La numerazione delle variabili di M e di N viene rinormalizzata dopo una sostituzione:

$$(\lambda x^1.f^2(x^3(x^4)))[(\lambda a^1.a^2)/x^3] = (\lambda x^1.f^2((\lambda a^3.a^4)(x^5)))$$

Infine se M e' un termine con occorrenze numerate si puo' anche scrivere

$$\lambda x^0.M$$

e in questo caso la numerazione di M resta invariata. Ove non ci sia bisogno di eccessivo rigore, la numerazione delle variabili puo' essere parziale.

I domini semantici del TAU sono indicati da lettere maiuscole (V,B,N,S, ecc.) [II.0.1].

II.0.1: Notazioni metacircolari

Per non cadere in un circolo vizioso e' necessario dare una spiegazione informale di alcune caratteristiche del TAU che verranno usate metacircolarmente per spiegare il TAU. Queste caratteristiche si riducono essenzialmente alle strutture dati, di cui viene data in [II.3] ed in [II.10] un'ampia descrizione.

Questa seconda parte dovrebbe essere letta in parallelo alla terza, in quanto i frequenti riferimenti non sono sempre autosufficienti.

I domini elementari del TAU sono:

    void (indicato anche con "{}") dominio vuoto
    null ("<>") dominio contenente solo nil ("()")
    bool dominio dei valori di verita' true e false
    num dominio dei numeri
    string dominio delle stringhe di caratteri
    dom dominio di tutti i domini
    any dominio di tutti i valori I predicati (infissi) di appartenenza a domini sono is ((()is<>, true is bool, ecc. che non deve essere confuso con l'omonimo is di LAMBDA) e has = λx y.y is x. I domini si possono comporre in vari modi per ottenere altri domini strutturati:

    <D¹; ... ;D[n]> prodotto cartesiano, dominio delle
        liste (a¹; ... ;a[n]) dove D[i] has a[i]
    D¹|D² unione congiunta:
        D¹|D² has a <=> (D¹ has a) v (D² has a)
    D¹->D² spazio delle funzioni:
        D¹->D² has λ[[x:D¹]->D². ...]
    D* dominio potenza:
        D* has D{a¹; ... ;a[n]) <=> D has a[i]
    {a¹; ... ;a[n]} dominio insieme:
        {a¹; ... ;a[n]} has a[i]

Le classi vengono costruite a partire dai domini, usando la notazione:

    c <= D

La freccia "<-" verra' invece riservata per i semplici

assegnamenti (in "c<-D", c non e' una classe, ma solo un altro nome per D).

Le classi sono anch'esse domini. Gli elementi di una classe vengono creati con l'operazione "[:]" (make) ed hanno la proprieta':

c has c[a:] <=> D has a dove c <= D

In particolare se D e' un prodotto cartesiano si possono assegnare nomi ai fattori di D:

c' <= <s¹:D¹; ... ;s[n]:D[n]>

Si possono allora costruire records:

r <- c'[a¹; ... ;a[n]:] dove D[i] has a[i]

e selezionare i campi dei records per nome, con la notazione:

r.s[i] = a[i]

Nella costruzione degli interpreti metacircolare ed iterativo per il TAU si dara' una codifica delle stringhe del TAU in TAU. Per evitare conflitti di nomi si converra' di premettere il carattere "n" (letto "coded") ai nomi delle codifiche: ad esempio nbool sara' una struttura dati TAU rappresentante il dominio bool del TAU.

II.1: Costanti

II.1.0: Semantica delle costanti

Per costanti si intendono quegli identificatori il cui valore non dipende dall'ambiente in cui vengono valutati. Il valore di una costante e' fissato al momento della definizione del linguaggio. Le equazioni semantiche per le costanti saranno quindi:

eval⟦c⟧ek = k(decode⟦c⟧)

dove decode e' una funzione che manda l'oggetto sintattico c nell'oggetto semantico corrispondente. Nell'interprete metacircolare ed iterativo si ha rispettivamente:

```
term is nconst -> term.decode
term is nconst -> capply[cont;term.decode]
```

dove la funzione decode e' implementata dalla selezione del campo decode.

II.1.1: Costanti di tipo null

Con null si indica il dominio <>: prodotto cartesiano di zero domini, il cui unico elemento e' nil, indicato con (), la lista di zero elementi. La sola costante di tipo null e' quindi nil e la sua semantica e':

decode⟦()⟧ = in(V,null,out⟦()⟧) = in(V,null,τ)

Nell'interprete metacircolare, ⟦()⟧ e' tradotto dal parser in

⌐null⟦()⟧ dove ⌐null <= <decode:<>>

per cui si ha, come e' lecito aspettarsi:

eval[⌐null⟦()⟧;env] = ⌐null⟦()⟧.decode = ()

Nell'interprete iterativo la situazione e' piu' complessa. Come abbiamo detto lo scopo di questo interprete e di fornire uno schema per la traduzione lineare del suo codice in qualche linguaggio a basso livello, il quale in generale non avra' immediatamente disponibile (hardware) un valore nil, ma dovra' codificarlo a partire dai valori a sua disposizione, ad esempio utilizzando l'indirizzo zero. Questa operazione di codifica viene rappresentata nel seguente modo: ⌐null contiene, non direttamente nil, ma ancora una codifica di nil detta nullval:

```
⌐null <= <decode:nullval>
nullval <= <value:<>>
```

Adesso ⟦()⟧ e' tradotto in ⌐null⟦nullval⟦()⟧⟧ e

```
eval[⌐null⟦nullval⟦()⟧⟧;env;cont] =
     capply[cont;nullval⟦()⟧]
```

nullval⟦()⟧ appartiene quindi all'insieme dei valori prodotti dalla valutazione dei programmi TAU.

Lo stesso meccanismo di doppia codifica si applica a

tutti gli altri tipi di costanti.

II.1.2: Costanti di tipo bool

Le costanti di tipo bool sono true e false:

```
decode⟦true⟧ = in(V,bool,out⟦true⟧) = in(V,bool,0)
decode⟦false⟧ = in(V,bool,out⟦false⟧) = in(V,bool,1)
```

Per l'interprete metacircolare:

```
eval[⌜bool⟦true⟧⌝] = true
⌜bool <= <decode:bool>
```

Per l'interprete iterativo:

```
eval[⌜bool⟦boolval⟦true⟧⟧⌝;env;cont] =
     capply[cont;boolval⟦true⟧]
⌜bool <= <decode:boolval>
boolval <= <value:bool>
```

II.1.3: Costanti di tipo num

Le costanti di tipo num sono i numeri interi. Per semplicita' non vengono considerati i numeri reali e complessi (che pure hanno una sistemazione standard in quasi tutti i linguaggi), visto che i concetti coinvolti nella trattazione degli interi si possono estendere linearmente anche agli altri tipi di numeri.

La semantica degli interi e':

$$decode\llbracket n \rrbracket = in(V, num, out\llbracket n \rrbracket)$$

dove $\llbracket n \rrbracket$ e' una rappresentazione che viene proiettata in un valore n appartenente al dominio semantico N, definito come num [III.0.0].

Su N sono definite le usuali operazioni sui numeri interi (+,-,x,/,*,ecc.) che sono tutte strict. Cio' consente di definire la semantica di espressioni del tipo $\llbracket a+b \rrbracket$ come $eval\llbracket a+b \rrbracket e = eval\llbracket a \rrbracket e + eval\llbracket b \rrbracket e$, ovvero:

$$eval\llbracket a+b \rrbracket ek = eval\llbracket a \rrbracket e(\lambda a.eval\llbracket b \rrbracket e(\lambda b.k(a+b)))$$

Per l'interprete metacircolare:

```
eval[ nnum[ 3 ]; env ] = 3
nnum <= <decode:num>
```

per l'interprete iterativo:

```
eval[ nnum[ numval[ 3 ] ]; env ] = numval[ 3 ]
nnum <= <decode:numval>
numval <= <value:num>
```

II.1.4: Costanti di tipo string

Le costanti di tipo string sono le stringhe di caratteri ("a", "abc", "a d", ecc.) e la loro semantica e':

decode⟦s⟧ = in(V,string,out⟦s⟧)

dove ⟦"abc"⟧ viene proiettato su un valore n∈S che lo codifica. Su S sono definite le usuali operazioni su stringhe che forniscono la semantica per le corrispondenti operazioni in TAU. Queste operazioni sono tutte strict, ad esempio:

eval⟦length["a"]⟧e = length(eval⟦"a"⟧e)

Le rappresentazioni delle stringhe negli interpreti metacircolare ed iterativo sono ormai intuibili, e rispettivamente:

```
nstring <= <decode:string>

nstring <= <decode:stringval>
stringval <= <value:string>
```

II.1.5: Costanti di tipo dom

Le costanti di tipo dom sono i sette domini elementari void, null, bool, num, string, dom, any; che sono rispettivamente i domini contenenti nessun valore, nil, true e false, i numeri, le stringhe, tutti i domini, tutti i valori.

La semantica dei domini elementari e':

decode⟦d⟧ = in(V,dom,in(D,elemdom,out⟦d⟧))

dove ad esempio eval⟦any⟧e=decode⟦any⟧, ed ⟦any⟧ appartiene ad un dominio-insieme D° contenente le codifiche numeriche dei sette valori void, null, bool, num, string, dom, any.

D e' il dominio di tutti i domini ed ha la forma D=D°⊕..., mentre V=...⊕D⊕... e questo spiega la doppia iniezione necessaria nell'equazione semantica.

Le principali operazioni sui domini elamentari (e sui domini in generale) sono: prodotto, unione, freccia, stella, insieme, classe; esse sono tutte non strict e verranno trattarte dettagliatamente in seguito.

La rappresentazione dei domini elementari e' nell'interprete metacircolare:

```
nelemdom = <decode:{{};<>;bool;num;string;dom;any}>
```

e nell'interprete iterativo:

```
nelemdom <= <decode:elemdomval>
elemdomval <= <value:{{};<>;bool;num;string;dom;any}
```

II.1.6: Costanti funzione

Le funzioni primitive in TAU sono delle costanti, ed hanno quindi senso non solo quando vengono applicate (plus[a;b]) ma anche quando sono isolate ([x->plus,times][a;b]) e possono essere partapplicate (plus[a]=/[[x];plus[a;x]]) e iperapplicate (plus[a;b;c]=[plus[a;b]][c] anche se in questo caso applicare un numero ad un argomento non ha alcun senso).

La semantica dell'insieme delle funzioni primitive e' quindi un dominio-insieme F° contenente delle opportune funzioni, ognuna delle quali fornisce al semantica della corrispondente funzione TAU.

decode⟦f⟧ = in⟦V,elemfun,out⟦f⟧⟧

Nell'interprete metacircolare:

```
nelemfun <= <decode:{plus;times; ... }>
```

Nell'interprete iterativo:

```
nelemfun <= <decode:elemfunval>
elemfunval <= <value:{nplus;ntimes; ... }>
```

Le funzioni nplus, ntimes, ecc. sono poi definite come delle TAU-espressioni che attraverso l'interpretazione agiscono sull'insieme dei valori prodotto dall'interprete iterativo, cosi' come plus, times, ecc. agiscono sull'insieme dei veri valori del TAU.

II.2: Variabili

II.2.0: Semantica delle variabili

La valutazione di una variabile consiste nella ricerca del valore associato a quella variabile nell'ambiente di valutazione:

$$eval\llbracket a \rrbracket ek = k(e\llbracket a \rrbracket)$$

La semantica delle variabili investe tutto un insieme di problemi (dallo scoping, al passaggio dei parametri, alla struttura degli ambienti), che vengono trattati anche in altri capitoli. In questo capitolo si approfondiscono soprattutto i problemi riguardanti lo scoping e le tecniche di accesso alle variabili in ambienti variamente strutturati.

II.2.1: Scoping delle variabili

In prima approssimazione si puo' definire lo scoping delle variabili come l'insieme di variabili di ugual nome che hanno in ogni istante lo stesso valore. Ad esempio in:

$$\lambda f^1.(\lambda x^2.f^3(x^4(x^5)))(\lambda x^6.f^7(x^8(x^9)))$$

lo scoping di f^3 e' $\{f^3,f^7\}$, quello di x^4 e' $\{x^4,x^5\}$ (e non $\{x^4,x^5,x^8,x^9\}$) e quello di x^8 e' $\{x^8,x^9\}$. I binders sfuggono a questi insiemi perche' non sono valutabili.

Tuttavia questa definizione non e' precisa perche' gli insiemi di variabili cosi' costruiti possono essere piu' grandi del necessario; ad esempio si avrebbe che lo scoping di x^3 in $(\lambda x^1.(\lambda x^2.x^3)(x^4))$ e' $\{x^3,x^4\}$ mentre nel programma equivalente $(\lambda y^1.(\lambda x^2.x^3)(y^4))$ e' $\{x^3\}$.

Nel caso del λ-calcolo si puo' normalizzare il programma, trasformandolo con @-conversioni in uno equivalente che presenti il massimo numero di nomi diversi di variabili, e poi applicare la definizione precedente. Parlando di linguaggi qualsiasi e' difficile dire se questa definizione di scoping sia sempre effettivamente applicabile. Essa viene introdotta solo per illustrare il concetto, mentre una definizione precisa verra' data piu' avanti.

Nel seguito si dira' comunemente che una variabile sta nello scoping di un'altra quando le due variabili hanno lo stesso scoping.

II.2.2: Scoping statico

Si dice che un linguaggio ha scoping decidibile se lo scoping delle sue variabili puo' essere determinato con una funzione primitiva ricorsiva, cioe' semplicemente ispezionando il testo del programma, senza coinvolgere particolari valutazioni dei programmi stessi.

Il λ-calcolo ad esempio ha scoping decidibile e la regola che permette di determinare lo scoping di una variabile e' la seguente: "in $\lambda x.M$ lo scoping di x e' dato da tutte le occorrenze di x libere in M".

Per il momento faremo coincidere la definizione di scoping statico con quella di scoping decidibile; piu' avanti verra' data una formulazione definitiva.

Il tipo piu' comune di scoping statico e' quello usato nei linguaggi a blocchi (λ-calcolo, SCHEME, ALGOL, ecc.) e la sua semantica e':

$$\text{eval}\llbracket\lambda a.M\rrbracket e = \lambda a.\text{eval}\llbracket M\rrbracket e[a/\llbracket a\rrbracket]$$
$$\text{eval}\llbracket A(B)\rrbracket e' = (\text{eval}\llbracket A\rrbracket e')(\text{eval}\llbracket B\rrbracket e')$$

dove il corpo della funzione $\llbracket M\rrbracket$ viene valutato (al momento dell'applicazione), non nell'ambiente dell'applicazione (e'), ma nell'ambiente in cui era stata valutata la funzione (e). Ogni funzione si porta con se' l'ambiente in cui era stata valutata, e usa questo ambiente (esteso con le associazioni parametri formali-attuali) al momento dell'applicazione per valutare il suo corpo.

Da un punto di vista implementativo cio' significa che la valutazione di una funzione produce una chiusura composta dal testo della funzione e dall'ambiente attuale. L'applicazione di una chiusura ad un argomento comporta poi

la valutazione del corpo della funzione nell'ambiente della chiusura, esteso con il valore dell'argomento.

II.2.3: Scoping dinamico

Con scoping <u>dinamico</u> si intende tutto cio' che non e' scoping statico. Il tipo piu' comune di scoping dinamico e' quello realizzato dal LISP puro, e deve la sua ragione di esistere al fatto di essere piu' semplice da implementare rispetto allo scoping statico del λ-calcolo. La semantica di questo scoping e' la seguente (immaginando di dare scoping dinamico alla sintassi del λ-calcolo):

$$\text{eval}[\![\lambda a.M]\!]e = \lambda e'\ a.\ \text{eval}[\![M]\!]e'[a/[\![a]\!]]$$
$$\text{eval}[\![A(B)]\!]e' = (\text{eval}[\![A]\!]e')(e', \text{eval}[\![B]\!]e')$$

In questo caso il corpo della funzione $[\![M]\!]$ viene valutato nell'ambiente dell'applicazione (e') che viene passato come parametro.

Dal punto di vista dell'implementazione si puo' dire che le funzioni valutano a <u>testi</u> e che l'applicazione di un testo ad un argomento comporta la valutazione del testo nell'ambiente dell'applicazione (e') (esteso col valore dell'argomento) dimenticandosi dell'ambiente nel quale la funzione era stata valutata.

Si ha in questo modo l'effetto spiacevole che diverse applicazioni della stessa funzione comportano la valutazione delle variabili libere della funzione in ambienti diversi, in modo che lo scoping di una variabile con occorrenze libere in M non puo' essere determinato se non in relazione all'ambiente di applicazione.

Altri tipi di scoping dinamico si ottengono ad esempio ammettendo espressioni che valutano a nomi di variabili, o permettendo di richiamare esplicitamente l'interprete, come accade in tutte le implementazioni LISP.

II.2.4: Scoping e binders

Alla ricerca di una definizione piu' precisa di scoping statico e dinamico, e' interessante osservare le relazioni tra scoping e binders. Vediamo un esempio:

```
let x¹<-3
 in let f<-λa.a(x²)
        g<-λx³.f(x⁴)
    in <f(λb.b),g(λb.b)>
```

In scoping statico x^2 cade sempre nello scoping di x^1, cioe' $x^2=3$ ed il risultato e' <3,3>.

In scoping dinamico invece, in seguito alla chiamata f(λb.b), x^2 cade nello scoping di x^1, ma in seguito alla chiamata g(λb.b), x^2 cade nello scoping di x^3 con valore (λb.b). Percio' adesso il risultato e' <3,(λb.b)>.

In caso di scoping dinamico quindi l'occorrenza x^2 cade nello scoping di due binders diversi: x^1 e x^3. Questo fenomeno e' cosi' caratteristico che puo' essere preso come base di una definizione che non ricorre alla esistenza o meno di una procedura in grado di trovare lo scoping di una variabile.

Def Si definiscono linguaggi con scoping statico quelli in cui ogni occorrenza di variabile e' nello scoping di al piu' un binder, e linguaggi con scoping dinamico quelli che ammettono almeno un programma con una occorrenza di variabile nello scoping di piu' di un binder □

II.2.5: Scoping e ambienti

Vediamo adesso come devono essere strutturati gli ambienti in relazione al tipo di scoping presente in un linguaggio. Il modo piu' naturale di implementare gli ambienti e' di utilizzare una struttura a liste del tipo:

```
a-list <- arid|environ
arid <= <>
environ <= <var:nvar;val:val;env:a-list>
```

dove arid implementa l'ambiente vuoto e environ implementa l'operazione di estensione e[v/a]. Viene poi utilizzata una funzione lookup che ricerca il valore di una variabile in un ambiente:

```
lookup<-/[[x:nvar;env:a-list]->val;
          env is arid -> error[],
          x=env.var -> env.val,
          lookup[x;env.env]]
```

In scoping statico un'organizzazione a liste di questo genere (dove nessuna informazione puo' essere distrutta, ma solo aggiunta) e' indispensabile perche' in ogni istante si possono utilizzare ambienti costruiti in istanti diversi. Ad esempio valutando un'applicazione occorrono l'ambiente corrente, per valutare gli argomenti, e l'ambiente di valutazione della funzione (memorizzato nella chiusura), per valutare il corpo della funzione.

Nello scoping dinamico del LISP puro invece l'unico ambiente che occorre in ogni istante e' l'ambiente corrente: tutti gli altri ambienti possono essere dimenticati o distrutti. In questo caso la a-list puo' essere implementata come uno stack di coppie variabile-valore. Nel corso di una valutazione lo stack cresce e decresce periodicamente, e ogni volta che cresce distrugge le informazioni

precedentemente memorizzate e relative ad un altro ambiente.

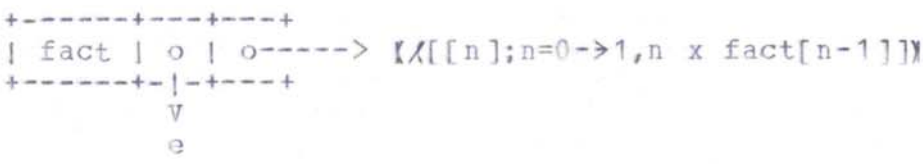
Il discorso puo' anche essere rovesciato: con una a-list implementata a stack, si puo' avere un linguaggio con scoping statico? Il punto cruciale e' che le *chiusure* memorizzano ambienti, ed evidentemente non devono memorizzare ambienti che poi verranno distrutti. Cio' puo' essere ottenuto (es: ALGOL) permettendo la creazione di chiusure, ma vincolandone l'utilizzazione negli ambienti che sono semplici espansioni di quelli delle chiusure, in modo che l'ambiente di una chiusura puo' essere semplicemente trovato piu' in basso sullo stack rispetto all'ambiente attuale. Questa condizione e' soddisfatta se si impedisce alle chiusure di essere ritornate come valori di funzioni: ogni chiusura viene cosi' utilizzata solo nella funzione in cui e' stata creata e in ogni istante l'ambiente della chiusura e' incluso nell'ambiente corrente.

II.2.6: Accesso delle variabili nello scoping indecidibile

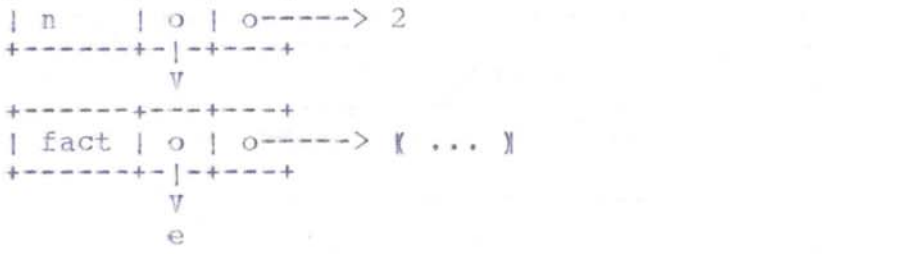
Lo scoping indecidibile e' caratterizzato dalla crescita imprevedibile degli ambienti di valutazione. Ecco un esempio in cui viene utilizzato lo scoping dinamico del LISP puro che (come tutti gli scoping dinamici) e' indecidibile [II.2.9]. Lo scoping qui sottinteso e' dunque quello illustrato in [II.2.3]

```
let[[fact<-ƛ[[n];n=0->1,n x fact[n-1]]];
    fact[2]]
```

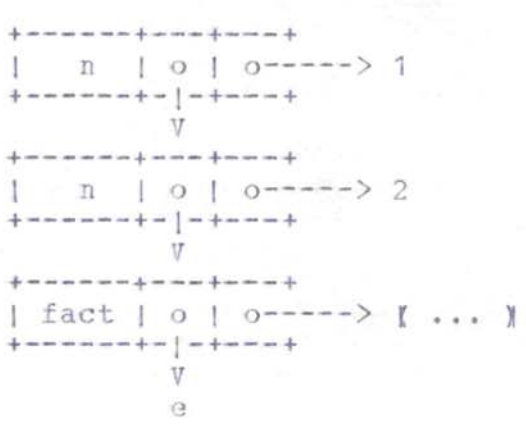
valutiamo la precedente espressione in un ambiente e. Per prima cosa viene valutata la definizione di fact che produce:



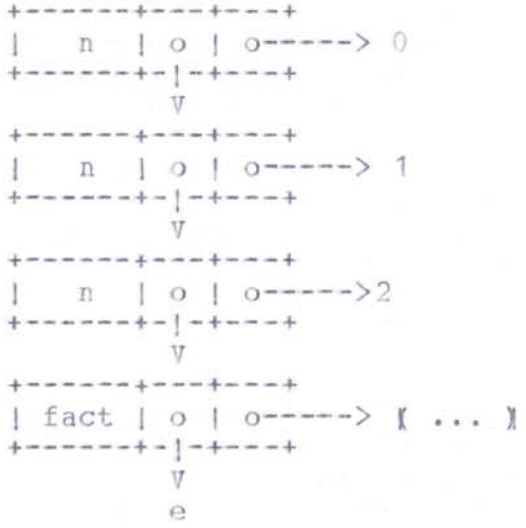
fact[2] provoca la creazione del legame tra n e 2:



A questo punto viene richiamata fact[1] nell'ambiente attuale (vedi la semantica dell'applicazione in [II.2.2) e si ha un nuovo legame di n con 1:



La successiva chiamata di fact[0] valuta a 1, fact[1] valuta a 1 e fact[2] valuta a 2.



Osserviamo come l'accesso alla variabile fact sia disomogeneo: alla prima chiamata (fact[2]) la variabile fact si trova subito sotto il top dell'ambiente di valutazione. Alla seconda chiamata (fact[1]) si trova invece sotto due variabili n. Alla terza chiamata infine (fact[0]) si trova tre celle sotto il top dell'ambiente.

In generale in scoping dinamico non si potra' mai prevedere a che profondita' si trovi una variabile globale di una λ-espressione, perche' questa profondita' cambia durante la valutazione. L'unico modo di accedere alle variabili libere e' allora quello di compiere una ricerca

associativa, confrontando cioe' i nomi delle variabili nell'ambiente finche' non si trovi quella giusta.

Questa tecnica di accesso e' notevolmente meno efficiente di quelle disponibili in scoping decidibile e costituisce uno dei maggiori svantaggi dei linguaggi con scoping dinamico.

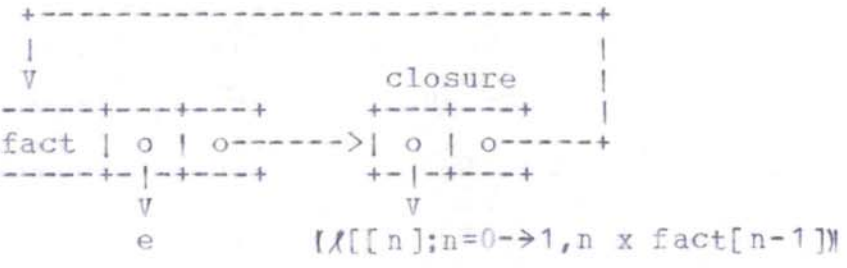
II.2.7: Accesso delle variabili nello scoping decidibile

Vediamo adesso come crescono gli ambienti nello scoping statico dei linguaggi a blocchi [II.2.2] che e' uno scoping decidibile. In [II.2.9] verra' dimostrato che questo e' il modo caratteristico di crescita degli ambienti in tutti i linguaggi con scoping statico.

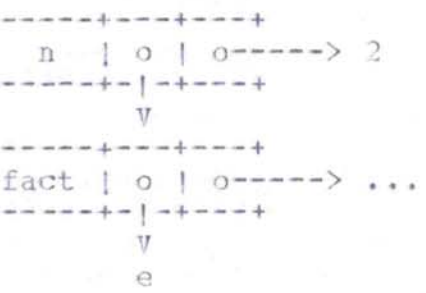
Valutiamo in un ambiente (e) l'espressione:

```
letrec[[fact<-λ[[n];n=0->1,n x fact[n-1]]];
       fact[2]]
```

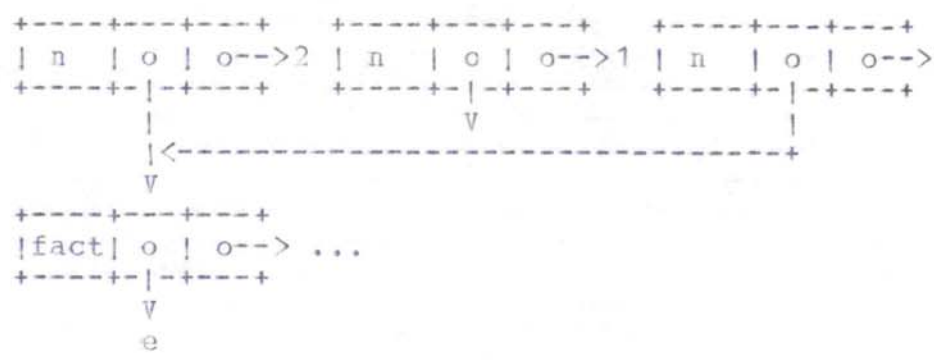
la definizione di fact produce una chiusura del testo di fact nel nuovo ambiente (e'):



La valutazione di fact[2] in questo nuovo ambiente aggiunge a (e') un legame per n:



La valutazione del corpo di fact con n=2 provoca una valutazione fact[1] nell'ambiente della chiusura e non, come in scoping dinamico, nell'ultimo ambiente costruito. Anche fact[1] aggiunge quindi un nuovo legame a (e') e cosi' pure fa fact[0]:



Ad ogni chiamata, fact si trova sempre alla stessa profondita'. Questa proprieta' e' valida per tutti i linguaggi con scoping statico (vedi [II.2.9]). Se questa profondita' puo' essere prevista (cioe' se lo scoping e' decidibile oltre che statico) si possono utilizzare tecniche di accesso alle variabili piu' efficienti di quella illustrata nel precedente paragrafo.

i) Accesso per profondita'.

In una prima passata sul testo (svolta generalmente dal parser) si associa ad ogni occorrenza valutabile di una variabile, la profondita' con cui questa verra' trovata nell'ambiente a run-time.

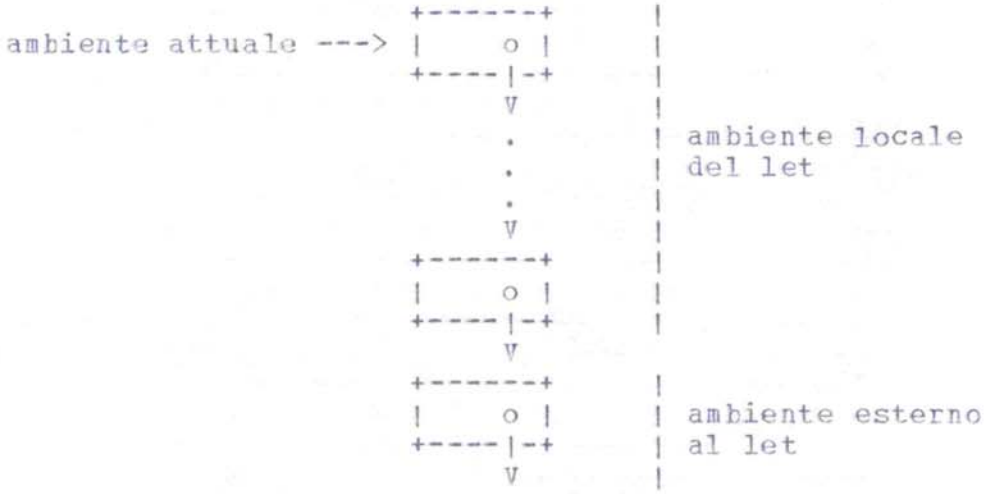
Durante l'esecuzione la variabile viene reperita direttamente tramite la sua profondita', senza dover confrontare nomi di variabili.

Se l'ambiente e' implementato come una lista il reperimento dell'n-esima variabile dal top puo' ancora essere inefficiente, specialmente per variabili libere molto profonde come possono essere le definizioni di funzioni. In questo caso sono applicabili tecniche di display comuni nei compilatori, che consentono un accesso quasi diretto a tutte le variabili.

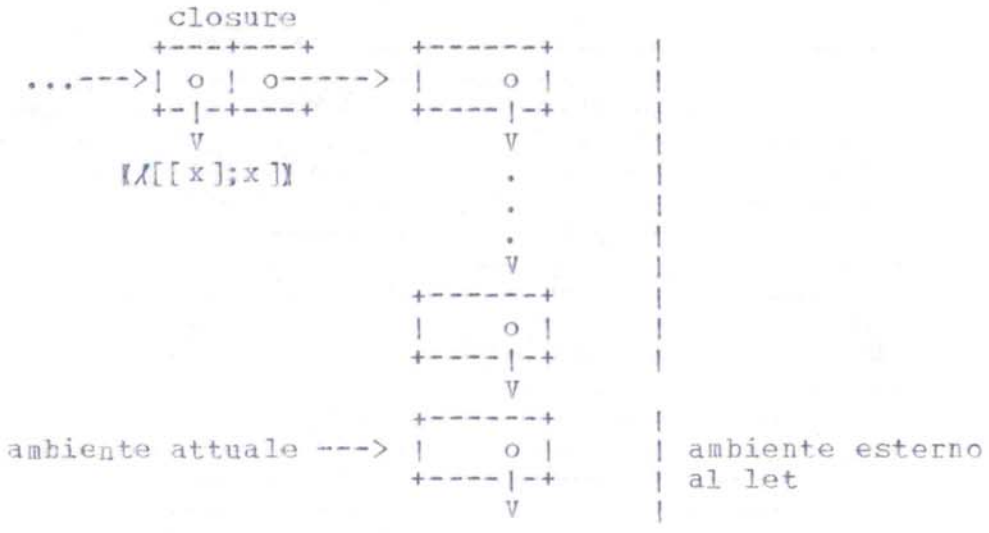
ii) Ambienti distribuiti.

Quando in scoping statico un ambiente e' implementato come una lista si possono avere grossi sprechi di memoria a causa dei puntatori agli ambienti contenuti nelle chiusure.

Supponiamo ad esempio di trovarci in un blocco let con molte variabili locali:



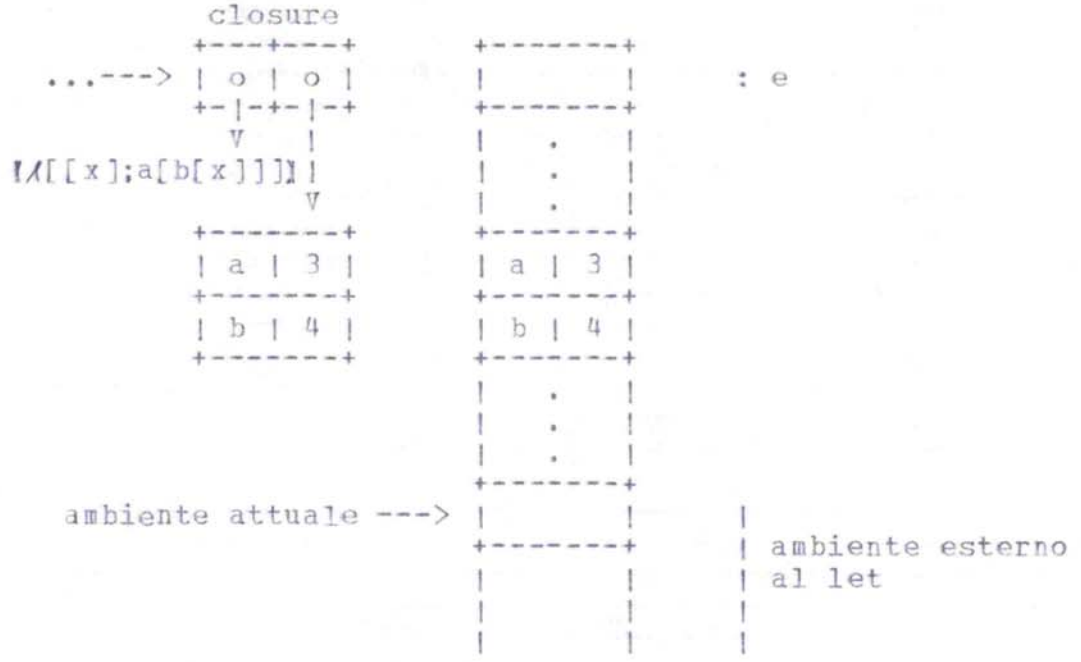
e supponiamo anche che il valore dell'intero blocco let sia la funzione λ[[x];x]:



Adesso le celle al di sopra dell'ambiente attuale sono completamente inutili, ma non possono essere liberate perche' sono puntate da una chiusura; essa del resto non ne fara' mai uso perche' λ[[x];x] non ha variabili libere.

Il problema si risolve mettendo nelle chiusure, non il puntatore all'ambiente di definizione, bensi' il puntatore ad un piccolo ambiente appositamente costruito, contenente solo le associazioni per le variabili libere della λ-espressione (queste possono essere fornite dal parser).

Gli ambienti vengono adesso implementati come stacks: le informazioni utili al di sopra dell'ambiente attuale sono state duplicate negli ambienti delle chiusure e possono anche essere distrutte.



Al momento dell'applicazione della chiusura, il corpo della sua λ-espressione viene valutato in un ambiente locale formato dalle associazioni parametri formali-parametri attuali e dalle altre variabili locali, e in un ambiente globale che e' quello della chiusura e contiene i valori delle variabili libere. Nell'esempio precedente l'ambiente (e) puo' essere considerato come l'ambiente locale di qualche λ-astrazione contenente il blocco let di cui si parlava sopra.

```
Es:   λ[[x;y];
        [let[[a<-1];a];
         let[[b<-2;c<-3];b]]]
```

In questo esempio l'ambiente locale e' formato da quattro celle: x(1°) e y(2°) per i parametri formali; a(3°) per la valutazione del primo let; b(ancora 3°) e c(4°) per la valutazione del secondo let. L'accesso alle variabili puo' ancora essere fatto per profondita' e senza seguire liste di puntatori. Il tutto si presta quindi ad una interpretazione e ad una compilazione molto efficiente. Tutto il lavoro di analisi (determinazione delle variabili libere, dimensione degli ambienti locali, profondita' delle variabili nei vari ambienti) puo' essere svolto dal parser.

II.2.8: La scelta dello scoping statico

I due precedenti paragrafi hanno mostrato come uno scoping decidibile sia da preferirsi da un punto di vista implementativo ad uno indecidibile, che risulta meno efficiente.

Anche dal lato della programmazione i vantaggi sono indubbi: in scoping decidibile il programmatore e' in grado di dire a prima vista dove viene legata una variabile libera, senza dover ricostruire la sequenza delle chiamate e tener conto di tutti i parametri formali incontrati in esecuzione.

La scelta dello scoping decidibile coincide poi a tutti gli effetti pratici con la scelta dello scoping statico, per due ragioni: prima di tutto lo scoping dinamico e' sempre indecidibile (la dimostrazione e' in [II.2.9]), inoltre lo scoping statico indecidibile esiste, ma deve essere introdotto ad hoc e non ha nessuna utilita'.

Consideriamo ad esempio un linguaggio simile al λ-calcolo, in cui pero' la semantica dell'espressione $\lambda[[x;x];M]$ sia:

$$eval\langle\lambda[[x;x];M]\rangle e = \lambda a\ b.eval\langle M\rangle e[?\to a,b/\langle x\rangle]$$

dove ? sia una funzione (calcolabile) di cui non si conosca alcun algoritmo. Questo linguaggio ha scoping statico (le x libere in M cadono nello scoping di un solo binder, anche se non si sa quale) ed ha scoping indecidibile.

Il principale tipo di scoping statico utilizzato in pratica e' comunque quello del λ-calcolo, (ed oggi si tende sempre di piu' a modellare i linguaggi applicativi sul λ-calcolo) e questo e' del tutto decidibile.

II.2.9: Teoria dello scoping

In questo paragrafo si cerca di dare una definizione rigorosa di scoping e di dimostrare alcune relazioni interessanti tra lo scoping e gli ambienti.

Iniziamo col definire un formalismo, il formalismo degli interpreti (che coincide con il formalismo di Scott-Strachey), in modo da poter parlare con una certa generalita' di quelle proprieta' degli interpreti valide per tutti i linguaggi.

Un interprete I e' una tripla {L,V,E} con le proprieta':

i) L e' un linguaggio su un alfabeto AT. L e' il dominio sintattico da interpretare, il linguaggio di cui l'interprete fornisce una semantica. L ha un sottinsieme ricorsivo K detto insieme delle costanti sintattiche. Var e' un altro sottinsieme ricorsivo di L disgiunto da K, detto insieme delle variabili, con la proprieta' che in ogni stringa di L si puo' cambiare il nome delle variabili in essa contenute ottenendo sempre un'altra stringa di L. Cioe':

$$\forall a,b \in Var.\ \forall s \in L.\ s[b/a[j]] \in L$$

ii) V e' l'insieme ricorsivo dei valori, il dominio semantico contenente gli oggetti denotati dalle stringhe di L. VK e' l'insieme dei valori denotati dagli elementi di K tramite la funzione decode:K->VK ed e' detto insieme delle costanti semantiche.

iii) E e' una funzione calcolabile da L in V, detta interprete, che manda stringhe di L (programmi) in elementi

di V (valori) fornendo cosi' la semantica del linguaggio L. Le proprieta' di E sono espresse attraverso le proprieta' di una funzione eval:LxEnv->V, cosi' definita:

$\forall s \in L.\ E(s) = eval(s)\,arid$

L'insieme Env:(Var x V)*, insieme degli ambienti, e' un insieme di associazioni variabili-valori su cui e' definita una funzione extend che permette di aggiungere una nuova associazione ad un vecchio ambiente, ed una funzione lookup che estrae da un ambiente il valore associato ad una data variabile. Queste funzioni vengono anche abbreviate cosi':

$\forall e \in Env; v \in V; a \in Var.\quad e[v/a] = extend[a,v,e]$
$\forall e \in Env; a \in var.\quad e.a = lookup[a,e]$

Le proprieta' di Env sono:

$arid \in Env$
$\forall e \in Env; v \in V; a \in Var.\ e[v/a] \in Env$
niente altro appartiene ad Env
$\forall a \in Var.$ arid.a e' indefinito
$\forall e \in Env; a,b \in Var; v \in V.\quad e[v/a].a = v$
$e[v/a].b = e.b \qquad$ con $a \neq b$

La funzione eval ha le proprieta':

$eval:(L \times Env) \to V$
$\forall e \in Env; k \in K.\quad eval(k)e = decode(k)$
$\forall e \in Env; a \in Var.\quad eval(a,e) = e.a$
$\forall s \in L-K-Var.\quad \neg f[i]$ calcolabile. $\forall e \in Env.$
$eval(s,e) = f[i](s,e)$

cioe' eval(s,e) puo' essere scomposta in un insieme di funzioni calcolabili {f', ... ,f[n]}. Ad ogni s∈L e' associata una ed una sola di queste f[i], e sulla scelta di i non influisce il particolare ambiente e. Si puo' anche scrivere che:

$eval(s,e) = f[i](s,e) \Rightarrow eval(s,e') = f[i](s,e')$

Questa viene chiamata nel seguito <u>proprieta' di sostituzione degli ambienti</u>, ed ha il significato intuitivo che le f[i], e quindi eval, non fanno test sulla struttura

degli ambienti (del tipo: se e.a e' definito per piu' di 5 variabili allora ...).

Una stringa (s) che valutata in (e) provoca una espansione e[v/a] di (e) viene detta blocco e la variabile (a) e' il binder del blocco. Un binder (a) di (s) puo' comparire in (s) oppure no, ma in ogni caso deve essere possibile trasformare un blocco (s) di binder (a) in un blocco (s') di binder (b), per ogni variabile (b). Questa condizione consente di trattare correttamente anche quei linguaggi che generano dinamicamente binders nel corso della valutazione (come ad esempio tutte le implementazioni LISP che consentono la chiamata esplicita di eval da programma).

Prop Il formalismo degli interpreti e' in grado di esprimere tutte e sole le funzioni calcolabili.

Dim infatti e' in grado di esprimere il λ-calcolo (vedi l'esempio seguente) e puo' essere espresso in termini di λ-calcolo (vedi il modo in cui il formalismo di Scott-Strachey viene espresso in λ-calcolo) □

Es: un interprete per il λ-calcolo.

$\leq Var \geq$:= a|b|c| ...
$\leq L \geq$:= $\leq Var \geq$ | ($\lambda \leq Var \geq . \leq L \geq$) | $\leq L \geq (\leq L \geq)$
K = $\emptyset$
VK = $\emptyset$
V = P_W
$eval[\![a]\!]e = e.a$
$eval[\![(\lambda a.M)]\!]e = f^1([\![(\lambda a.M)]\!],e) = \lambda a.eval[\![M]\!]e[a/[\![a]\!]]$
$eval[\![M(N)]\!]e = f^2([\![M(N)]\!],e) = (eval[\![M]\!]e)\,(eval[\![N]\!]e)$

Def Un interprete I e' un λ-interprete sse:

i) Per ogni coppia di stringhe $f, x \in L$ esiste una stringa di L, indicata con f(x), tale che:

eval(f(x),e) = (eval(f,e))(eval(x,e))

e l'applicazione nel membro destro e' quella del λ-calcolo.

ii) Per ogni stringa m∈L e per ogni variabile a∈Var esiste una stringa di L, indicata con λa.m tale che :

eval(λa.m,e) = λx.eval(m,e[x/a])

dove la λ-astrazione nel membro destro e' quella del λ-calcolo □

Es: un interprete che non e' un λ-interprete

```
≤Var≥ := a|b|c| ...
≤K≥ := 0|1|2| ...
≤L≥ := ≤Var≥ | let[[≤Var≥<-≤L≥];≤L≥] | ≤K≥
VK = {0,1,2, ... }
V = VK
eval⟦k⟧e = const⟦k⟧
eval⟦a⟧e = e.a
eval⟦let[[a<-N];M]⟧e = eval⟦M⟧e[eval⟦N⟧e/a]
```

Nello studio del comportamento degli ambienti non e' sufficiente in generale sapere come un ambiente appare all'esterno (cioe' quali sono le variabili definite e qual e' il loro valore) ma occorre anche conoscere come l'ambiente e' stato costruito.

A tal scopo si introduce una relazione di ordinamento parziale tra ambienti ("espansione di") cosi' definita:

Def 1 ∀c,d,e∈Env. ∀v,t∈V.

```
e≤e                 (e) e' espansione di se stesso
e≤e[v/a]            e[v/a] e' espansione di (e)
e[v/a]≤e[t/a]       "≤" non si cura dei valori
                    associati alle variabili
e≤d & d≤c => e≤c    l'espansione di una espansione
                      di (e) e' una espansione di (e)   □
```

Def 2 Due ambienti d,e∈Env sono simili sse:

∀a∈Var. d.a e' definito <=> e.a e' definito □

Def 3 Due ambienti d,e∈Env sono equivalenti sse:

∀a∈Var. d.a = e.a □

Def_4 Due ambienti d,e∈Env sono strutturalmente simili sse:

d≤e & e≤d □

Def_5 Una a-estensione di un ambiente (d) e' un ambiente (d') tale che:

d' = d[v1/b1] ... [v[n]/b[n]] (n≥1)

con b[i]=a per qualche i. Una non-a-estensione di (d)e' un ambiente d'≥d che non e' una a-estensione di (d) □

Def_6 Sia s un blocco con binder a. Lo scoping di a e' l'insieme delle a[i] che vengono valutate in qualche non-a-estensione dell'ambiente a-esteso dalla valutazione di s □

Def_7 In un λ-interprete una occorrenza a[i] e' libera in s se a[i] cade nello scoping di a° in λa°.s. Altrimenti a[i] si dice legata □

Def_8 Un λ-interprete ha scoping statico se ∀s∈L . ∀a[i]∈Var. a[i] e' nello scoping di al piu' un binder in λa°.s. In caso contrario l'interprete ha scoping dinamico □

Prop_1 In un λ-interprete sia a[i] libera in s. Allora in E(s) viene effettuata una valutazione e.a[i] per qualche e$\in$Env, tale che e.a[i] e' indefinito.

Dim a[i] e' libera in s (Hp). a[i] cade nello scoping di a^0 in $\lambda a^0.s$ (Def 7). a[i] viene valutata in qualche non-a-estensione di arid[v/a] nella valutazione $E(\lambda a^0.s)=\lambda v.eval(s)arid[v/a]$ (Def 6). Quindi in $E(s)=eval(s)arid$, a[i] viene valutata in qualche non-a-estensione e di arid (proprieta' di sostituzione degli ambienti). Allora e.a[i] e' indefinito (Def 5) □

Prop_2 In un λ-interprete con scoping statico, sia a[i] libera in s. Allora in E(s) tutte le valutazioni e.a[i] per qualche e$\in$Env sono indefinite.

Dim a[i] e' libera in s (Hp). a[i] cade solo nello scoping di a^0 in $\lambda a^0.s$ (Def 7,Def 8). a[i] viene sempre valutata in qualche non-a-estensione di arid[v/a] nella valutazione $E(\lambda a^0.s)=\lambda v.eval(s)arid[v/a]$ (Def 6). Quindi in $E(s) = eval(s)arid$, a[i] viene sempre valutata in qualche non-a-estensione e di arid (proprieta' di sostituzione degli ambienti). Allora e.a[i] e' sempre indefinito (Def 5).

Prop_3 In un λ-interprete sia a[i] legata in s. Allora in E(s) vengono effettuate delle valutazioni e.a[i] per qualche e$\in$Env, e per tutte queste e.a[i] e' definito.

Dim a[i] e' legata in s (Hp). a[i] non cade nello scoping di a^0 in $\lambda a^0.s$ (Def 7). a[i] viene sempre valutata in qualche a-estensione di arid[v/a] nella valutazione $E(\lambda a^0.s)=\lambda v.eval(s)arid[v/a]$ (Def 6). Quindi in $E(s)=eval(s)arid$, a[i] viene sempre valutata in qualche a-estensione di arid (proprieta' di sostituzione degli

ambienti). Allora e.a[i] e' sempre definito (Def 5) □

Teorema Un λ-interprete I ha scoping statico se e solo se tutte le valutazioni di a[i] avvengono in ambienti strutturalmente simili.

Dim

=>) Se I ha scoping statico allora tutte le valutazioni di a[i] avvengono in ambienti strutturalmente simili.

Supponiamo per assurdo che a[i] venga valutato in due ambienti non strutturalmente simili d,e. Siano b,c i primi due binders che si incontrano esplorando d,e dall'alto verso il basso, tali che: (i) $b \neq c$; (ii) b in d e' alla stessa profondita' di c in e; (iii) b e' generato dal blocco B[b] di binder b, e c dal blocco C[c] di binder c di s. Consideriamo la stringa s' ottenuta da s sostituendo a[i] con una nuova variabile z non inclusa in s, e sostituendo i blocchi B[b] e C[c] con due blocchi B[z] e C[z] che generano due binders z. Allora in s', la variabile z sostituita ad a[i] viene a cadere nello scoping di due binders, generati da B[z] e C[z]. Cio' contro l'ipotesi di scoping statico.

<=) Se tutte le valutazioni di a[i] avvengono in ambienti strutturalmente simili allora I ha scoping statico.

Supponiamo per assurdo che I abbia scoping dinamico. Allora esistono $s \in L$ e $a[i] \in Var$ tali che a[i] e' nello scoping di almeno due binders a generati dai blocchi A e B di s. a[i] viene quindi valutata in due ambienti e[A] ed e[B] la cui ultima a-estensione e' dovuta rispettivamente ad A e B (sia B non incluso in A). Se e[A] e' strutturalmente simile a e[B], allora sia s' ottenuta da s sostituendo A con $(\lambda z.A)(3)$. Allora nella valutazione di s' gli ambienti e'[A] ed e'[B] (corrispondenti a e[A] ed e[B] nella

valutazione di s) non sono strutturalmente simili perche' e'[A] ha un binder z in piu' □

Corollario Lo scoping dinamico e' sempre indecidibile

Dim discende dalla prova (<=) del teorema □

Corollario Lo scoping decidibile e' sempre statico

Dim negazione del precedente corollario □

II.3: Liste e arrays

II.3.0: Liste

Quelle che nella sintassi esterna del TAU sono liste (a¹; ... ;a[n]) vengono trasformate dal parser in applicazioni della funzione elementare pair:

```
⟦(a¹; ... ;a[n])⟧ = ⟦pair[a¹; ... pair[a[n];()] .. ]⟧
```

La semantica di pair e':

```
eval⟦pair[a;b]⟧ek =
     k(pair(eval⟦a⟧e stop,eval⟦b⟧e stop))
```

Bisogna qui assumere l'esistenza di un dominio semantico delle liste:

```
L = first::V ⊗ rest::(L ⊕ H)
```

dove H e' il dominio che contiene solo nil e pair e' la funzione

```
pair = λa b.in(V,pair,⟨a,b⟩)
```

La funzione pair del TAU e' un costruttore, cioe' e' una funzione che da uno o piu' valori ricava un nuovo valore da cui si possono nuovamente estrarre i valori originari tramite appositi selettori. First e rest sono i selettori associati a pair:

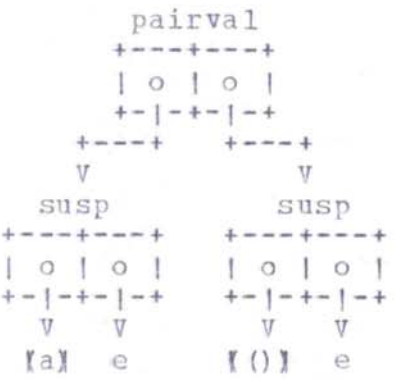
```
eval⟦first[a]⟧ek = eval⟦a⟧e(λa.k(first(a)))
eval⟦rest[a]⟧ek = eval⟦a⟧e(λa.k(rest(a)))
first = λa.is(pair,a)->out(a).first,⊥
rest = λa.is(pair,a)->out(out(a).rest),⊥
```

In TAU tutti i selettori sono strict e tutti i costruttori sono non strict. Questo si esprime anche dicendo che pair non valuta, oppure che sospende i suoi argomenti, mentre first e rest valutano o non sospendono gli argomenti.

Nell'interprete iterativo la valutazione di un costruttore

viene fatta allocando una opportuna cella di memoria (in questo caso una cella pairval) e memorizzando le sospensioni degli argomenti:

eval⟦pair[a;()]⟧e =



Le sospensioni servono a dilazionare le valutazioni il piu' a lungo possibile: esse sono formate da un termine da valutare e da un ambiente di valutazione e contengono quindi tutte le informazioni necessarie per valutare una espressione in qualsiasi istante lo si desideri. Con questa implementazione la funzione pair e' veramente non-strict, perche' converge anche se i suoi argomenti non terminano.

La valutazione degli argomenti di un costruttore puo' essere ritardata finche' non si usi un selettore per recuperare il valore dell'argomento. Il selettore first deve quindi controllare che la parte sinistra del suo argomento pairval non sia una sospensione. Se lo e', deve valutarla e rimpiazzare il risultato in pairval al posto della sospensione. In questo modo ogni successiva first dello stesso pairval trovera' il valore gia' pronto.

Le sospensioni, oltre al vantaggio di rendere non strict le funzioni, sono anche molto utili per la costruzione di dati circolari, cioe' di dati contenenti cicli di puntatori. Esempi di cio' si trovano in [II.6].

II.3.1: Arrays

Gli arrays sono liste omogenee di elementi, cioe' liste di elementi tutti appartenenti allo stesso dominio:

A(a¹; ... ;a[n]) e' un array <=> ∀i. A has a[i]

La sintassi esterna A(a¹; ... ;a[n]) viene trasformata dal parser nell'applicazione :

array[A;(a¹; ... ;a[n])]

dove array e' una funzione primitiva.

```
eval(array[A;(a¹; ... ;a[n])])ek =
     k(array(eval(a)e stop,
       pair(eval(a¹)e stop,
            ...
            pair(eval(a[n])e stop,eval(())e stop)..)))
```

Il dominio degli arrays e' A = D ⊗ L e:

array = λa b.in(V,array,(a,b))

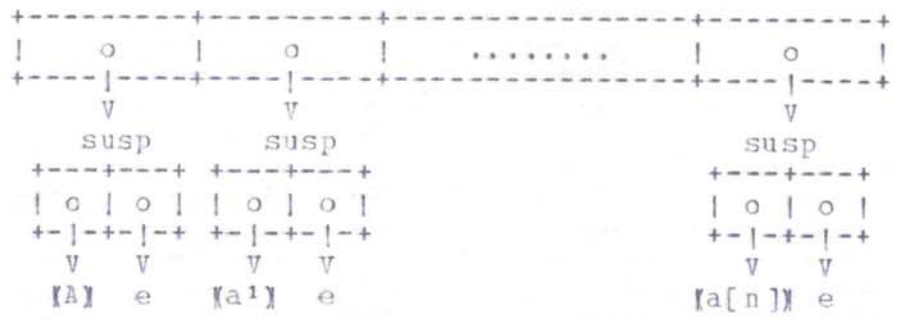
Anche gli arrays sono costruttori non strict. Il selettore corrispondente e' item:

eval(item[a;n]) = eval(a)e(λa.eval(n)e(λn.item(a,n)))

Dove item e' una funzione che ha la proprieta' di estrarre l'i-esimo elemento di un array a:

item = λa n.is(array,a)->is(num,n)->out(a).out(n),τ,τ

Nell'interprete iterativo gli arrays vengono implementati come vettori di celle di memoria, ognuna delle quali contiene inizialmente la sospensione di un elemento dell'array. La prima cella contiene la sospensione del tipo dell'array:



Man mano che vengono selezionate, alle sospensioni viene sostituito il loro valore effettivo.

II.4: Condizionale

II.4.0: Semantica del condizionale

Il condizionale e' una funzione strict nel primo argomento (boolean) e non-strict nel secondo e nel terzo argomento:

```
eval[a->b,c]ek =
     (/a.is(bool,a)->out(a)->eval[b]ek,eval[c]ek,⊥)
```

Nell'interprete metacircolare:

```
term is ncond -> eval[ term.if;env ->
                       eval[ term.then;env ],
                       eval[ term.else;env ]]
```

dove ncond <= <if:nterm;then:nterm;else:nterm>

Nell'interprete iterativo:

```
term is ncond ->
     eval[ term.if;env;condcont[ term;env;cont ]]

 ...
cont is condcont ->
     value is boolval ->
          eval[ boolval->cont.cond.then,cont.cond.else;
               cont,env;cont.cont ],
     error[ cont ]
  ...
```

dove condcont <= <cond:nterm;env:a-list;cont:c-list>

II.5: Applicazione e λ-astrazione

II.5.0: Semantica della λ-astrazione

La semantica della λ-astrazione in regime di scoping statico e' data da:

```
eval⟦λ[[x];M]⟧ek = k(in(V,fun,λx.eval⟦M⟧e[x/⟦x⟧]stop))
```

Una λ-astrazione valuta quindi ad una funzione. Questa valutazione e' "sicura", cioe' non e' mai indefinita, e si puo' dire che il corpo M viene in realta' valutato solo al momento di un'applicazione.

Per vedere come cio' accade, supponiamo che la valutazione di M sia indefinita, cioe' che provochi un errore o che non termini, e valutiamo [λ[[x];M];N] (La semantica di [a;b] e' eval⟦[a;b]⟧ek = eval⟦a⟧e(λa.eval⟦b⟧ek) , vedi [II.9]):

```
eval⟦[λ[[x];M];N]⟧ek =
eval⟦λ[[x];M]⟧e(λa.eval⟦N⟧ek) =
(λa.eval⟦N⟧ek)(in(V,fun,λx.eval⟦M⟧e[x/⟦x⟧]stop)) =
eval⟦N⟧ek
```

che puo' essere ben definito, indipendentemente da M.

In TAU e' consentito esprimere il tipo degli argomenti e del risultato di una funzione, ma ci si puo' ricondurre al caso precedente tramite le seguenti riduzioni successive:

```
λ'[[P¹;P²; ... ;P[n]]->R;M] =
     λ'[[P¹];λ'[[P²]; ... λ'[[P[n]]->R;M]
λ'[[P];M] = λ'[[P]->any;M]
λ'[[x]->R;M] = λ'[[x:any]->R;M]
λ'[[x:D]->R;M] =
     λ[[d];λ[[r];
       λ[[x];[[λ[[x];M][x▽d]]▽r]
       ]][D][R]
```

Dove λ' indica le λ-astrazioni non ancora completamente ridotte e ▽ = λ[[v;d];d has v->v,error[]] e' la retrazione di un valore su un dominio. Notare che D e R sono valutati

nell'ambiente esterno alla λ-astrazione.

In TAU le λ-astrazioni possono anche avere parametri e argomenti vuoti. I casi possibili sono:

```
λ[[ ];M][ ]  =  M
λ[[x];M][ ]  =  λ[[x];M]
λ[[ ];M][A] = M[A]
```

I casi precedenti sono le situazioni limite della partapplicazione e della iperapplicazione, cosi' esemplificate:

```
λ[[x;y];M][A]  =  λ[[y];λ[[x;y];M][A;y]]
λ[[x];M][A;B]  =  [λ[[x];M][A]][B]
```

Limitandoci a λ-espressioni di un solo parametro, ma con tipi espressi, si ha per l'interprete metacircolare:

```
term is nlamb ->
  λ[[x:eval[term.pars.kind;env]]->eval[term.range;env];
       eval[term.form;
            environ[term.pars.var;
                    x;
                    env]]]
```

e per l'interprete iterativo:

```
term is nlamb -> capply[cont;closure[term;env]]
```

dove closure <= <fun:nterm;env:a-list>

In questo caso la valutazione si riduce a memorizzare la λ-astrazione non valutata (e senza quindi valutarne il corpo) in una chiusura insieme al suo ambiente, e rimandando ogni valutazione al momento dell'applicazione.

II.5.1: Semantica dell'applicazione

Supponiamo di dover valutare l'espressione TAU

λ[[x];M][N]

il problema e' se e quando vogliamo valutare gli argomenti, cioe' se vogliamo ccnsiderare le λ-espressioni come funzioni strict o non strict.

Possiamo dare principalmente due tipi di semantica dell'applicazione:

call-by-value:
eval⟦f[a]⟧ek = eval⟦f⟧e(λf.eval⟦a⟧e(λa.k(f(a))))

call-by-name:
eval⟦f[a]⟧ek = eval⟦f⟧e(λf.k(f(eval⟦a⟧e stop)))

In entrambe i casi viene prima valutata la funzione. Poi nella call by value viene valutato l'argomento, applicato alla funzione e il risultato vien passato alla continuazione. Nella call-by-name invece viene prima effettuata l'applicazione, poi viene valutato l'argomento e infine viene applicata la continuazione.

Vediamo piu' in dettaglio cosa succede ponendo f=λ[[x];M]

call-by-value:
eval⟦λ[[x];M][a]⟧ek =
eval⟦λ[[x];M]⟧e(λf.eval⟦a⟧e(λa.k(f(a)))) =
(λf.eval⟦a⟧e(λa.k(f(a))))(λx.eval⟦M⟧e[x/⟦x⟧]stop) =
eval⟦a⟧e(λa.k(eval⟦M⟧e[a/⟦x⟧]stop)) =
eval⟦a⟧e(λa.eval⟦M⟧e[a/⟦x⟧]k)

Cioe': viene valutato l'argomento e poi viene valutato il corpo della funzione in un ambiente esteso col valore dell'argomento. Infine si applica la continuazione.

call-by-name:
eval⟦λ[[x];M][a]⟧ek =
eval⟦λ[[x];M]⟧e(λf.k(f(eval⟦a⟧e stop)))
(λf.k(f(eval⟦a⟧e stop)))(λx.eval⟦M⟧e[x/⟦x⟧]stop) =
k((λx.eval⟦M⟧e[x/⟦x⟧]stop)(eval⟦a⟧e stop))
k(eval⟦M⟧e[eval⟦a⟧e stop/⟦x⟧]stop)
eval⟦M⟧e[eval⟦a⟧e stop/⟦x⟧]k

Cioe' viene valutato il corpo della funzione in un ambiente in cui per ottenere il valore di x, occorre valutare l'argomento. Al tutto viene applicata la continuazione.

Vediamo cosa succede se la valutazione di N in ⟦λ[x];M⟧[N] e' indefinita (cioe' se provoca un errore o se non termina).

Con la call-by-value tutta la valutazione e' indefinita, perche' ⟦a⟧ e' la prima cosa che viene valutata: la funzione denotata da ⟦λ[x];M⟧ e' strict. Con la call-by-name invece l'intera valutazione e' indefinita solo se x viene valutato durante la valutazione di M, Infatti:

```
Se M=x: eval⟦x⟧e[eval⟦a⟧e stop/⟦x⟧]k =
        k(e[eval⟦a⟧e stop/⟦x⟧](x)) =
        k(eval⟦a⟧e stop) =
        eval⟦a⟧ek                    che e' indefinito.

Se M=3: eval⟦3⟧e[eval⟦a⟧e stop/⟦x⟧]stop =
        stop(3) =
        3                            che e' ben definito.
```

Quindi con la call-by-name la funzione denotata da ⟦λ[x];M⟧ non e' strict.

II.5.2: Call-by-need

Prendiamo in esame adesso la call-by-value e la call-by-name da un punto di vista implementeativo e consideriamo l'espressione:

$\lambda[[x;y];x][3;4]$

Nella call-by-value si procede valutando prima gli argomenti, e poi valutando il corpo della funzione in un ambiente esteso con le associazioni:

x = 3 ; y = 4

Nella call-by-name invece gli argomenti non devono essere valutati e si utilizzano le sospensioni, in cui vengono racchiusi gli argomenti ed i relativi ambienti di valutazione. Il corpo di una funzione viene quindi valutato in un ambiente esteso con le associazioni:

x = susp⟦3;env⟧ ; y = susp⟦4;env⟧

Quando la funzione lookup va a cercare il valore di x, essa forza in quell'istante la valutazione della sospensione, fornendo il valore corretto. Si noti che nell'esempio l'argomento 4 non viene mai valutato.

In assenza di side-effects la call-by-name puo' essere implementata in modo molto piu' efficiente con un meccanismo di chiamata detto call-by-need [Wadsworth 71] [Aiello 76].

A differenza della call-by-name, la procedura lookup non solo forza la valutazione delle sospensioni, ma rimpiazza anche il valore calcolato nell'ambiente al posto della sospensione. In questo modo ogni argomento viene valutato una sola volta, come nella call-by-value, ma solo se e' necessario, come nella call-by-name.

Il corpo della funzione viene quindi valutato in un ambiente esteso con le associazioni

x = susp[|3|;env] ; y = susp[|4|;env]

ma dopo la valutazione di x nell'ambiente si ha:

x = 3 ; y = susp[|4|;env]

in modo che ad una successiva valutazione di x il valore 3 e' gia' pronto.

In un linguaggio con side-effects la call-by-need non e' piu' equivalente alla call-by-name, e non ne costituisce quindi una implementazione, ma diviene un diverso regime di passaggio dei parametri. Per vedere la differenza basta considerare l'espressione (dove := e' l'assegnamento e la semantica e' una semantica con stores):

```
let[[a<-(1;2)];
    /[[x;y];[y;first[x]:=3;y]][a;first[a]]]
```

Se il passaggio dei parametri e' call-by-name il risultato e' 3, mentre se e' call-by-need e' 1.

Il passaggio dei parametri in TAU e' call-by-need. Questa scelta e' stata fatta perche':

i) La call-by-need e' per efficienza l'algoritmo ottimale di passaggio dei parametri in quanto provoca le valutazioni degli argomenti solo quando sono strettamente necessarie.

ii) La call-by-need in assenza di side-effects implementa la call-by-name, la quale ha una semantica piu' semplice e generale della call-by-value (corrisponde infatti al semplice rimpiazzamento del testo degli argomenti al posto dei parametri formali) ed ha la proprieta' di mantenere non strict i costruttori (cioe' /[[x;y];pair[x;y]] risulta esattamente equivalente a pair, in modo che l'n-conversione

rimane valida).

Le sospensioni sono composte di tre parti, in modo da contenere anche il tipo del valore sospeso:

susp <= <term:nterm;range:val|susp;env:a-list>

L'appicazione λ[[x:a]->b;M][N] genera la sospensione:

susp[N;susp[a;elemdomval[dom];env];env]

Prima viene valutata la λ-espressione e viene costruita una chiusura con l'ambiente corrente, env. Poi viene effettuata l'applicazione nel seguente modo: si espande env con la variabile x a cui viene legata la sospensione susp[N,a,env], ottenendo newenv. Si valuta M in newenv; se in questa valutazione si incontra x allora la procedura lookup si preoccupa di valutare N in env, di retrarlo sulla valutazione di a in env (che deve essere un dominio) e di rimpiazzarlo come nuovo valore non sospeso di x in newenv. Ogni successiva valutazione di x accede a questo valore gia' calcolato. Alla fine della valutazione di M si ottiene il valore m. Questo viene retratto sulla valutazione di b in env ed il risultato e' infine m.

II.5.3: Uguaglianza tra funzioni

L'uguaglianza di due funzioni e' un problema indecidibile, e la prima tentazione e' di lasciare totalmente indefinita l'espressione:

equal[λ[[x];x];λ[[y];y]]

Questa soluzione ha l'effetto di lasciare indefinita l'uguaglianza su qualsisi struttura dati contenente λ-espressioni. Soluzioni alternative (tutte ugualmente arbitrarie) sono ad esempio:

i) due funzioni son uguali se lo sono le loro rappresentazioni (LISP)
ii) due funzioni qualsiasi sono sempre uguali
iii) due funzioni qualsiasi sono sempre diverse

In questi casi l'uguaglianza diventa rispettivamente:

i) non estensionale
(∀x.equal[a[x];b[x]] ≠> equal[a;b])
ii) non sostitutiva
(∀x.equal[a;b] ≠> equal[M[a/x];M[b/x]])
iii) non estensionale

In TAU l'uguaglianza tra funzioni e' indefinita. Il problema di memorizzare funzioni in strutture dati e poi discriminare su di esse puo' essere risolto in molti casi utilizzando le classi ed il predicato has, al posto di equal.

II.6: μ-astrazione

II.6.0: Semantica della μ-astrazione

La semantica della μ-astrazione puo' essere data in termini dell'operatore minimo punto fisso:

```
eval⟦μ[[x];M]⟧ek =
     eval⟦x⟧(λe'.e[eval⟦M⟧e'/⟦x⟧])k
```

La μ-astrazione permette di definire funzioni ricorsive e dati circolari, come verra' mostrato negli esempi. In TAU esiste una forma di μ-astrazione tipizzata che puo' essere ricondotta alla precedente con le seguenti riduzioni successive:

```
μ'[[x];M] = μ'[[x:any];M]
μ'[[x:D];M] = μ[[x];M↓D]
```

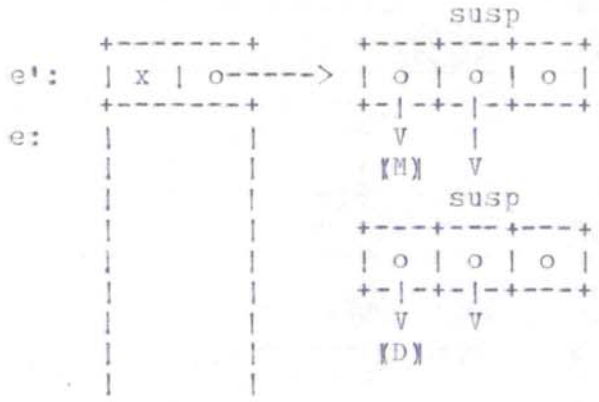
dove μ' indica le μ-astrazioni non ancora completamente ridotte, e ↓ rappresenta la retrazione di un valore su un dominio. Notare che D viene valutato all'interno della μ-astrazione, e quindi conosce x.

Nell'interprete metacircolare:

```
term is mmu ->
     μ[[x:eval[term.range;extend[term.var;x;env]]];
       eval[term.form;extend[term.var;x;env]]]
```

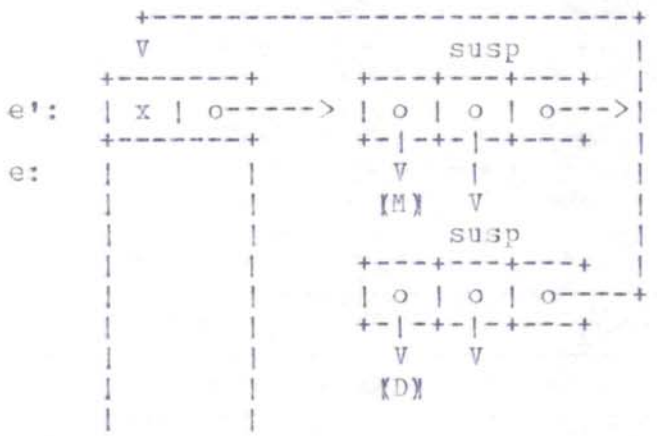
Nell'interprete iterativo la valutazione di μ[[x:D];M] in (e) avviene nel seguente modo:

i) L'ambiente (e) viene esteso con la sospensione tipizzata di M in arid:

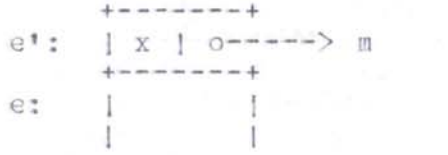


chiamiamo (e') il nuovo ambiente.

ii) Nelle sospensioni appena costruite, arid viene sostituito da (e'):



iii) Viene forzata la valutazione delle sospensioni, ed il risultato m viene posto come valore di x:

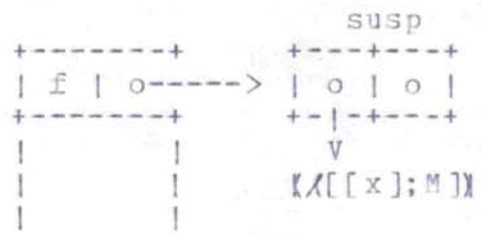


Il risultato della μ-astrazione e' m.

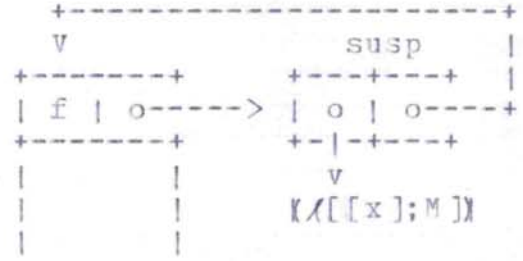
Per semplicita' negli esempi che seguono si usano solo sospensioni non tipizzate.

Esempio 1. $\mu[[f];\lambda[[x];M]]$

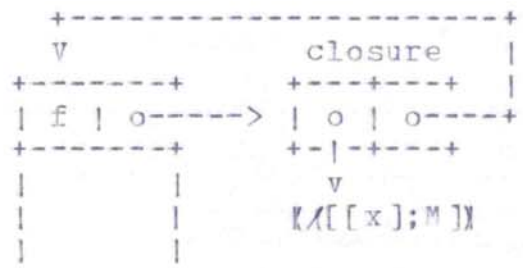
i)



ii)



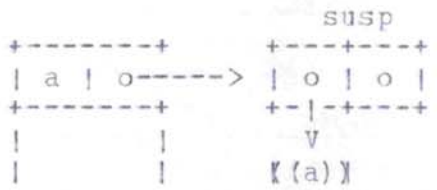
iii)



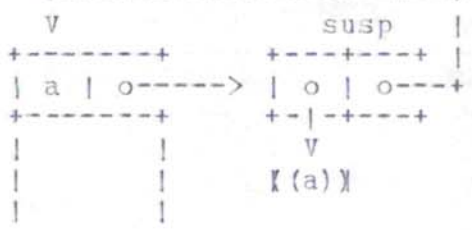
Notare come ad ogni chiamata ricorsiva di f in M viene reperita la giusta λ-definizione.

Esempio 2. μ[[a];(a)]

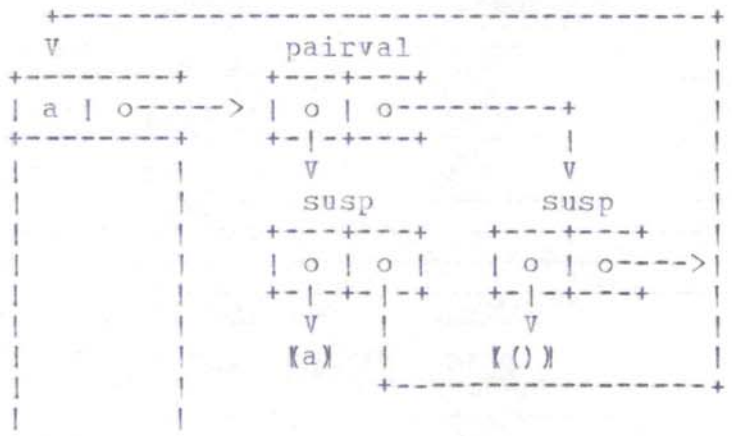
i)



ii)

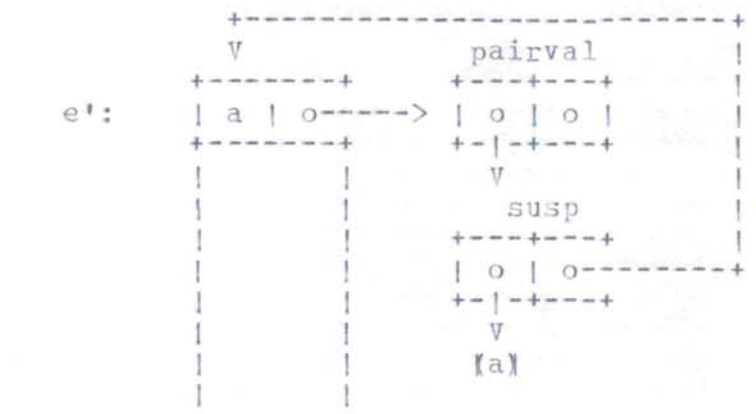


iii)

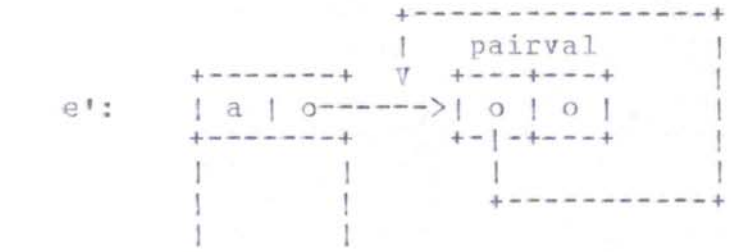


Si noti che se il costruttore pair valutasse gli argomenti, anziche' sospenderli, la valutazione non terminerebbe.

Una successiva valutazione di right[a] in (e') forza la valutazione della sospensione di destra, ed il rimpiazzamento del valore cosi' calcolato (nil) nella parte destra di pairval:

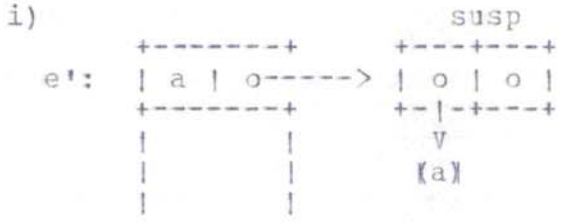


Infine valutando anche left[a] si ottiene:

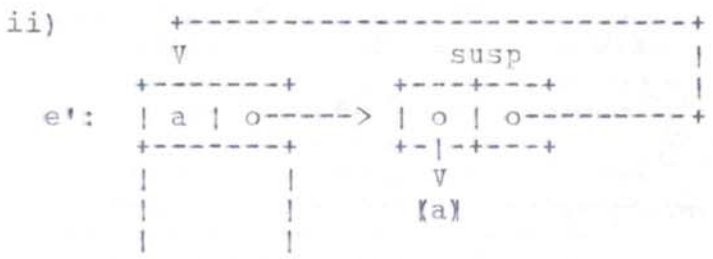


Esempio 3. μ[[a];a]

i)



ii)



iii) dobbiamo adesso valutare (a) in (e'), ossia lookup[(a);e'], ma poiche' (a) e' legata ad una sospensione, lookup provoca automaticamente la sua valutazione. Ma la valutazione della sospensione provoca ancora lookup[(a);e'] e cosi' via all'infinito. La valutazione non termina.

Questo risultato e' corretto: infatti μ[[a];a] e' il minimo punto fisso della funzione identita' i=λ[[a];a]. Si cerca cioe' il minimo x tale che x=i[x]. Poiche' questa uguaglianza e' vera per ogni x, il minimo e' certamente ⊥.

Esempio 4. w = μ[[f];λ[[x];(x;f[x+1])]]

w[0] = (0;(1;(2;(3;(4; ...

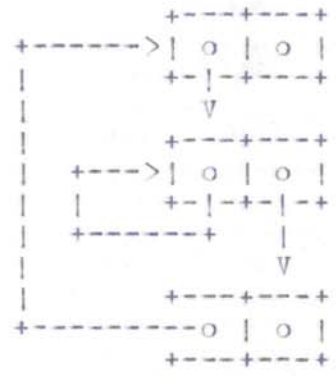
w[0] e' la lista infinita di tutti i numeri, e come tale puo' essere trattato a tutti gli effetti:

```
first[w[0]] = 0
first[rest[w[0]]] = 1
first[rest[rest[w[0]]]] = 2
 ...
```

Anche qui la valutazione converge grazie alle sospensioni.

II.6.1: Stampa di strutture circolari

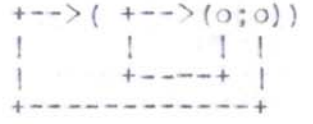
Come abbiamo visto il costrutto μ e' in grado di creare strutture circolari del tipo:



Per queste strutture il normale algoritmo di stampa cade in loop e fallisce quindi il suo scopo. Si vorrebbe infatti che la struttura precedente fosse stampata cosi':

μ[[a];(μ[[b];(b;a)])]

o forse anche cosi':



Un algoritmo di stampa di questo genere richiede due passate.

La prima passata percorre in ordine anticipato la struttura da stampare mantenendo uno stack delle sottostrutture formanti un cammino tra la radice ed il punto corrente. Ogni volta che si accede ad un sottostruttura si controlla se questa e' gia' presente sullo stack. In caso affermativo si e' scoperto un ciclo nella struttura e si evita di proseguire in questa direzione. Il risultato della prima passata e' un elenco dei punti di circolarita'.

La seconda passata stampa le strutture controllando che

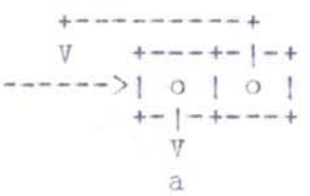
queste non siano punti di circolarita'. Se lo sono, la prima volta che le incontra stampa "#[[a];" (a e' una variabile mai prima usata che viene associata a quel particolare punto di circolarita') e prosegue a stampare le sottostrutture della struttura circolare. In questo processo quando incontra di nuovo lo stesso punto di circolarita', stampa semplicemente "a" senza entrare di nuovo nella struttura. Alla fine della stampa si chiude la parentesi "]" del #.

II.6.2: Uguaglianza di strutture circolari

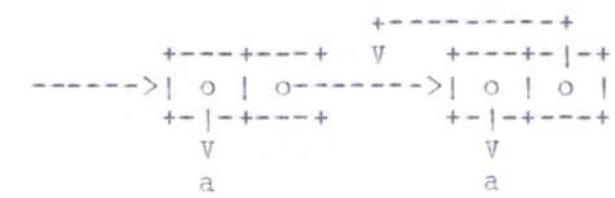
Il principio su cui si basa l'algoritmo di stampa si adatta anche al problema dell'uguaglianza di strutture circolari. E' sufficiente una sola passata in cui si confrontano in parallelo le due strutture mantenendo due stacks per rilevare i punti di circolarita'. Notare che i punti di circolarita' possono non corrispondere (vedi s1 e s2 sotto), ma quando corrispondono devono risultare ugualmente profondi sugli stacks.

Consideriamo le strutture:

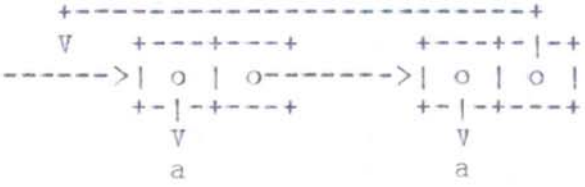
s1 = μ[[b];(a;b)]



s2 = (a;μ[[b];(a;b)])



s3 = μ[[b];(a;(a;b))]



s1=s2 e' vero, mentre s1=s3 e s2=s3 sono falsi. Infatti:

s1 = μ[[b];(a;b)]
= /[[b];(a;b)][μ[[b];(a;b)]]
= (a;μ[[b];(a;b)]
= s2

II.7: Let e letrec

II.7.0: Let

La semantica del costrutto let e' la seguente:

$$eval⟦let[[x^1<-A^1; \ldots ;x[n]<-A[n]];M]⟧ek =$$
$$eval⟦A^1⟧e(\lambda a^1. \ldots$$
$$eval⟦A[n]⟧e(\lambda a[n].eval⟦M⟧e[a^1/⟦x^1⟧] \ldots$$
$$[a[n]/⟦x[n]⟧k] ..)$$

A[i] viene valutato nell'ambiente esterno a let e legato con x[i]. M viene valutato nell'ambiente esterno esteso con i nuovi valori A[i] degli x[i], ed il suo valore e' il valore di tutto il let.

La sintassi del TAU prevede anche la possibilita':

let[[x<-A then y<-B];M] = let[[x<-A]; let[[y<-B];M]]

Si vede facilmente che let[[x^1<-A^1; ... ;x[n]<-A[n]];M] equivale a

λ[[x^1; ... ;x[n]];M][A^1; ... ;A[n]]

con call-by-value.

In un let o in un letrec puo' comparire la freccia <= al posto della <- . Questa e' una abbreviazione sintattica per la definizione delle classi: vedi [II.10.8].

II.7.1: Letrec

La semantica del costrutto letrec e':

```
eval⟦letrec[[x¹<-A¹; ... ;x[n]<-A[n]];M]⟧ek =
     let e'<-⟦e'.e[eval⟦A¹⟧e' stop/⟦x¹⟧] ...
                  [eval⟦A[n]⟧e' stop/⟦x[n]⟧]
      in eval⟦A¹⟧e'(λa¹. ...
             eval⟦A[n]⟧e'(λa[n].eval⟦M⟧e'k) .. )
```

A¹ ... A[n] vengono valutati in un ambiente (e') ottenuto estendendo l'ambiente esterno (e) con i valori di A¹ ... A[n] (valutati in un ambiente (e') ottenuto estendendo l'ambiente esterno (e) con i valori A¹ ... A[n] ecc.) come se A¹ ... A[n] fossero gia' stati valutati in (e'). Poi M viene valutato nell'ambiente (e') fornendo il risultato finale. L'effetto di questo meccanismo circolare di valutazione e' che tutti gli x[i] sono conosciuti all'interno di ogni A[i].

Cio' permette ad esempio di costruire funzioni ricorsive:

```
let[[fact'<-λ[[n];0]];
    let[[fact"<-λ[[n];n=0->1,n x fact'[n-1]]];
        fact"[2]]] = 0

let[[fact'<-λ[[n];0]];
    letrec[[fact"<-λ[[n];n=0->1,n x fact"[n_1]]];
           fact"[2]] = 2
```

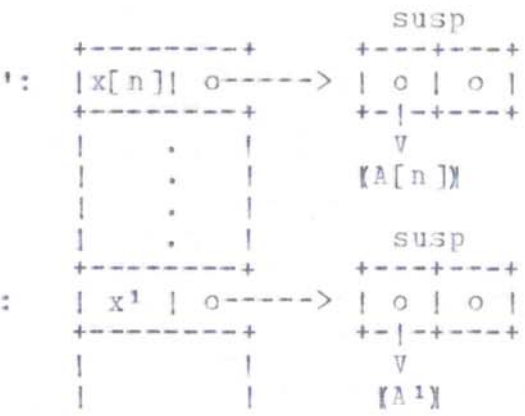
dove fact' e fact" sono due variabili diverse aventi lo stessc ncme. Il fact di fact[n-1] e' quello esterno (fact') nel primo caso, e quello nello stesso ambiente (fact") nel secondo caso, di modo che il risultato differisce. Notare che ⟦μf.M⟧ e' equivalente a ⟦letrec[[f<-M];f]⟧.

Nell'interprete iterativo la valutazione

```
eval⟦letrec[[x¹<-A¹; ... ;x[n]<-A[n]];M]⟧ek
```

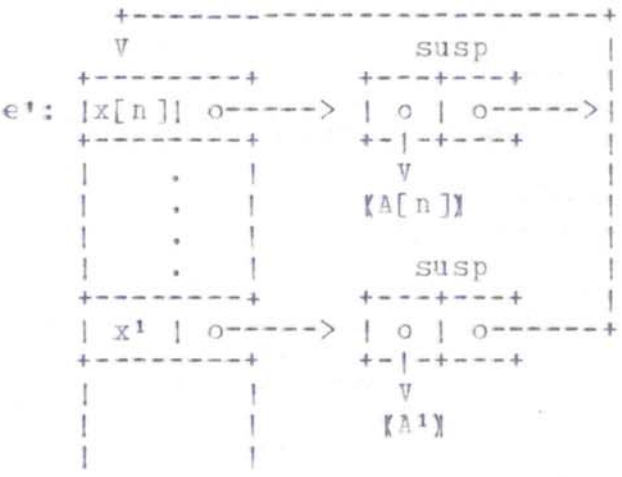
avviene nel seguente modo:

i) l'ambiente (e) viene esteso con le sospensioni degli A[i] in arid:



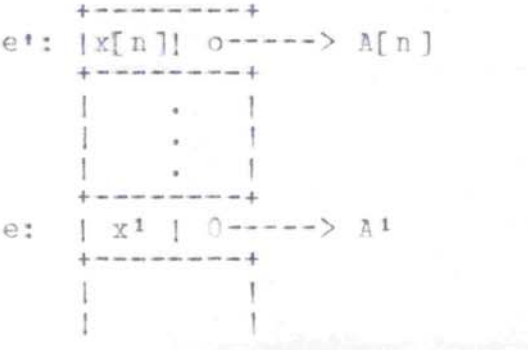
ottenendo cosi' un ambiente (e').

ii) nelle sospensioni appena costruite, arid viene sostituito con (e'):



iii) vengono valutate le sospensioni, nell'ordine x¹ ... x[n], ed i valori calcolati vengono sostituiti nell'ambiente al posto delle sospensioni.

Indichiamo con A[i] la valutazione di ⟦A[i]⟧ in (e'):



iv) nell'ambiente (e') cosi' ottenuto viene valutato ⟦M⟧.

II.8: Continuazioni

II.8.0: Semantica delle continuazioni

Le continuazioni sono state introdotte in [I.3.5] come tecnica di espressione della semantica dei linguaggi, ma esse possono anche essere messe a disposizione dell'utente per mezzo di opportuni costrutti. Il piu' comune e' l'escape:

$$\text{eval}\llbracket \mathcal{Y}[[x];M]\rrbracket ek = \text{eval}\llbracket M\rrbracket e[\text{in}(V,\text{cont},k)/\llbracket x\rrbracket]k$$
$$\text{eval}\llbracket A[B]\rrbracket ek = \text{eval}\llbracket A\rrbracket e(\lambda a.\text{eval}\llbracket B\rrbracket e\ \text{out}(a))$$

La prima equazione dice che la continuazione di una certa espressione M puo' essere conservata come funzione in una variabile x e resa accessibile in M. La seconda equazione mostra come si passa un valore B ad una continuazione A, assumendo che A valuti ad una continuazione.

II.9: Blocchi

II.9.0: Semantica dei blocchi

Un blocco e' una sequanza di espressioni che vengono valutate di seguito. L'ultima di queste espressioni da' il valore di tutto il blocco:

$$eval[\![[a^1; \ldots ; a[n]]]\!]ek =$$

$$eval[\![a^1]\!]e(\lambda a^1. \ldots eval[\![a[n]]\!]ek \ldots)$$

II.10: Supertipi

II.10.0: Sistemi di supertipi

Per supertipi si intendono qui le gerarchie complesse di tipi presenti in linguaggi come l'EL1, l'ALGO68, ecc. Cio' che distingue un sistema di supertipi da un sistema di tipi piu' convenzionale e' l'esistenza, a fianco ad esempio dei tipi: num (numeri), bool (booleani), ecc., del tipo: dom (tipi). Cioe' se 0 e' un num e true e' un bool, si ha anche che num e' un dom, bool e' un dom, dom e' un dom.

Dei sitemi di supertipi si puo' dare una formulazione molto generale come segue:

a) A, B, C, ... $\in$ Dom sono <u>domini</u>

b) a, b, c, ... $\in$ Val sono <u>valori</u>

c) Dom $\leq$ Val tutti i domini sono valori

d) esiste una operazione di <u>retrazione</u> di un valore su un dominio:

$\Downarrow$: Dom x Val -> Val

che ha la proprieta':

$\Downarrow(A) \circ \Downarrow(A) = \Downarrow(A)$

L'operazione di retrazione consente di specificare quali valori appartengono a quali domini e quali siano le gerarchie tra i domini.

e) esiste un dominio any$\in$Dom tale che:

$\forall a \in Val.\ a\Downarrow any = a$

cioe' $\Downarrow$(any) e' l'identita' su Val.

f) esiste un dominio dom$\in$Dom tale che:

$\forall A \in Dom.\ A\Downarrow dom = A$

questa condizione e' quella che caratterizza i sistemi di supertipi.

g) altri valori per i quali a∀A = a sono caratteristici dei particolari sistemi di supertipi.

h) sulla base della definizione di ∀ si possono definire tre predicati utili:

∀a∈Val;A∈Dom. a is A <=> a∀A = a

∀a∈Val;A,B∈Dom. A in B <=> (a is A => a is B)

∀A,B∈Dom. A=B <=> (A in B) & (B in A)

i) Di tutte le funzioni ∀ che soddisfano le condizioni precedenti viene scelta quella corrispondente al minimo predicato is (minimo nel senso di "piu' falso" sotto l'ordinamento parziale true≤true; false≤false; false≤true; ∀A∈Dcm. f(A)≤g(A) <=> ∀a∈Val. f(A,a)≤g(A,a)).

Alcune proprieta' di is e in possono essere immediatamente ricavate dalle definizioni:

∀a∈Val. a is any
∀A∈Dom. A is dom
∀A∈Dom. A in any
∀A∈Dom. A in A
∀A,B,C∈Dom. A in B & B in C => A in C

Proprieta' interessanti di dom ed any si ottengono istanziando le precedenti:

any is any	any in any
dom is any	dom in any
dom is dom	dom in dom
any is dom	

Nei sistemi di supertipi non e' possibile definire a questo livello di generalita' una funzione type(a) che da un valore ricava un dominio che lo contiene. La definizione piu' naturale di type sembrerebbe la seguente:

type(a) = min{A|(a is A)}

dove il minimo e' fatto rispetto al predicato in.

Questa definizione pero' non da' i risultati aspettati quando ad esempio si includono gli insiemi ({a,b,c}) tra i tipi strutturati. In tal caso si ha sempre type(a) = {a} che non fornisce nessuna informazione su a.

La funzione type deve quindi essere definita caso per caso nei sistemi di supertipi nel modo che risulta piu' conveniente. Nel caso precedente una definizione soddisfacente puo' essere:

type(a) = min{A|a is A e A non e' un insieme}

II.10.1: Unione disgiunta

Con il termine unione si intende una operazione tra domini con il significato intuitivo che A|B (A unito B) e' un dominio i cui elementi sono l'unione di quelli di A con quelli di B e che se un elemento appartiene ad A, esso appartiene in modo piu' o meno diretto anche a A|B. L'unione tra domini puo' essere realizzata principalmente in due modi, e in questo paragrafo esaminiamo l'unione disgiunta. Essa vuole modellare l'unione disgiunta definita in [I.2.4].

Definiamo un sitema di supertipi:

i) bool,dom,any∈Dom
   A,B∈Dom => A|B∈Dom unione disgiunta

ii) true,false∈Bool sono valori (Bool≤Val)
    i domini sono valori (Dom≤Val)
    a∈Val => a^{ι}∈Val,$a^{\jmath}$∈Val inserzione sinistra e destra

iii) ∀ e' individuata dal minimo predicato is che soddisfa le condizioni:
   ∀v∈Val. v is any
   ∀d∈Dom. d is dom
   ∀b∈Bool. b is bool
   ∀a∈Val;A,B∈Dom.
      a^{ι} is A|B <=> a is A
      $a^{\jmath}$ is A|B <=> a is B

Come si vede, se (a is A) non e' vero che sia (a is A|B) ma a deve prima essere inserito nell'unione disgiunta con l'operazione a^{ι}. Il termine unione disgiunta deriva dal fatto che gli elementi di A e B non si mischiano indistinguibilmente come nell'unione tra insiemi, ma mantengono l'informazione sulla loro originale appartenenza ad A o B. Inoltre se simultaneamente (a is A) e (a is B), nell'unione disgiunta A|B si trovano due distinte occorrenze di a: a^{ι} e $a^{\jmath}$, anziche' una sola come nell'unione tra insiemi.

II.10.2: Unione congiunta

L'unicne congiunta intende modellare l'unione tra insiemi, per cui un valore appartiene all'unione congiunta A|B semplicemente se appartiene ad A o appartiene a B.

Il sistema di supertipi che ne deriva e' il seguente:

i) bool,dom,any∈Dom
   A,B∈Dom => A|B∈Dom unione congiunta

ii) true,false∈Bool sono valori (Bool≤Val)
    i domini sono valori (Dom≤Val)

iii) ∀ e' individuata dal minimo predicato is che soddisfa le condizioni:
   ∀v∈Val. v is any
   ∀d∈Dom. d is dom
   ∀b∈Bool. b is bool
   ∀a∈Val;A,B∈Dom. a is A|B <=> (a is A) v (a is B)

II.10.3: Prodotto cartesiano

Il prodotto cartesiano tra domini e' l'analogo del prodotto multiplo definito in [I.2.5]. Si usa la rappresentazione:

$\langle A^1; \ldots ; A[n] \rangle \qquad n \geq 0$

Il prodotto di zero domini <> e' detto anche null. Se $a^1, \ldots , a[n]$ sono elementi dei domini $A^1, \ldots , A[n]$ allora $(a^1; \ldots ; a[n])$ e' un elemento di $\langle A^1; \ldots ; A[n] \rangle$ ed e' detto genericamente lista. La lista di zero elementi () e' detta anche nil ed e' l'unico elemento di null.

Un sistema di supertipi con prodotto cartesiano e' ad esempio:

i) bool,dom,any$\in$Dom
$A^1, \ldots , A[n] \in$ Dom => $\langle A^1; \ldots ; A[n] \rangle \in$ Dom

ii) true,false$\in$Bool sono valori (Bool$\leq$Val)
i domini sono valori (Dom$\leq$Val)
$a^1, \ldots , a[n] \in$ Val => $(a^1; \ldots ; a[n]) \in$ Val

iii) $\forall$ e' individuata dal minimo predicato is che soddisfa le condizioni:
$\forall v \in$Val. v is any
$\forall d \in$Dom. d is dom
$\forall b \in$Bool. b is bool
$\forall a^1, \ldots , a[n] \in$Val; $A^1, \ldots , A[n] \in$Dom.
$(a^1; \ldots ; a[n])$ is $\langle A^1; \ldots ; A[n] \rangle$ <=>
$\forall i$. a[i] is A[i]

[I.10.4: Domini funzionali

I domini funzionali intendono modellare l'operazione definita in [I.2.4]. Un elemento di un dominio A->B e' una funzione che prende un elemento di A e restituisce un elemento di B. Il modo piu' adatto di rappresentare funzioni di questo genere e' mediante λ-espressioni tipizzate: λ[[x:A]->B;M].

Il sistema di supertipi che si ottiene e' il seguente:

i) bool,dom,any$\in$Dom
   A,B$\in$Dom => A->B$\in$Dom

ii) true,false$\in$Bool sono valori (Bool$\leq$Val)
    i domini sono valori (Dom$\leq$Val)
    a$\in$Val; A,B$\in$Dom => λ[[x:A]->B;a] $\in$ Val

iii) $\forall$ e' individuata dal minimo predicato is che soddisfa le condizioni:
   $\forall$v$\in$Val. v is any
   $\forall$d$\in$Dom. d is dom
   $\forall$b$\in$Bool. b is bool
   $\forall$A,B,A',B'$\in$Dom;a$\in$Val.
      λ[[x:A]->B;a] is A'->B' <=> A->B in A'->B'

II.10.5: Domini potenza

I domini potenza modellano l'operazione * definita in [I.2.5]. Un elemento di A* e' una lista (a[1]; ... ;a[n]) di elementi di A (con n≥0) indicata con A(a[1]; ... ;a[n]). Un sistema di supertipi con domini potenza e' il seguente:

i) bool,dom,any∈Dom
   A∈Dom => A*∈Dom

ii) true,false∈Bool sono valori (Bool≤Val)
    i domini sono valori (Dom≤Val)
    A∈Dom; a[1], ... ,a[n] ∈ Val =>
        A(a[1]; ... ;a[n]) ∈ Val <=> ∀i. a[i] is A

iii) ∀ e' individuata dal minimo predicato is che soddisfa le condizioni:
   ∀v∈Val. v is any
   ∀d∈Dom. d is dom
   ∀b∈Bool. b is bool
   ∀A(a[1]; ... ;a[n])∈Val; B*∈Dom.
        A(a[1]; ... ;a[n]) is B* <=> A in B

II.10.6: Insiemi

Gli insiemi modellano i domini-insieme definiti in [I.2.5]. Un gruppo qualsiasi di valori $a^1, \ldots, a[n] \in Val$ puo' essere raccolto in un insieme $\{a^1; \ldots; a[n]\}$ $(n \geq 0)$. L'insieme vuoto $\{\}$ e' detto anche void. Un sistema di supertipi con insiemi e' il seguente:

i) bool,dom,any$\in$Dom
$a^1, \ldots, a[n] \in Val \Rightarrow \{a^1; \ldots; a[n]\} \in Dom$

ii) true,false$\in$Bool sono valori (Bool$\leq$Val)
i domini sono valori (Dom$\leq$Val)

iii) $\forall$ e' individuata dal minimo predicato is che soddisfa le condizioni:
$\forall v \in Val.$ v is any
$\forall d \in Dom.$ d is dom
$\forall b \in Bool.$ b is bool
$\forall a^1, \ldots, a[n] \in Val.$
$\forall i.$ a[i] is $\{a^1;, \ldots, a[n]\}$

II.10.7: Classi

Il concetto di classe corrisponde alla possibilita' di creare domini dotati, nelle intenzioni, delle seguenti caratteristiche:

a) Un dominio classe (o semplicemente classe) A' viene costruito sulla base di un dominio preesistente A.

b) Piu' classi diverse A', A", ... possono essere costruite sulla base dello stesso dominio A.

c) Per ogni classe A' esiste una funzione di iniezione up(A'):A->A' biunivoca che immerge gli elementi di A in A'. La funzione inversa e' una proiezione down(A'):A'->A. La notazione che verra' usata in seguito e':

```
a up A'  =  up(A')(a)
down a   =  down(A')(a)      se  a is A'
```

d) a is A <=> (a up A') is A'

e) un elemento di una classe e' diverso da qualsiasi valore che non sia elemento di quella stessa classe. In particolare elementi di classi diverse sono diversi.

Le classi consentono di costruire domini "astratti" formati da valori "nuovi", esclusivi della classe a cui appartengono. Con cio' si raggiungono due obbiettivi. In primo luogo una strutturazione dei dati, in quanto si puo' nascondere sotto una classe A', che appare all'uso simile ad un dominio elementare, un dominio A eventualmente molto complesso. In secondo luogo si ottiene una discreta protezione dei dati, in quanto le funzioni scritte per lavorare su A' non funzionano se applicate a dati diversi.

L'unione disgiunta si avvicina abbastanza, ma non del tutto, a implementare le classi.

La costruzione di una classe a partire da un dominio A puo' essere fatta con l'unione A|void, dove void e' il dominio senza elementi. Allora la iniezione (a up A|void) diventa l'operazione a$^{\perp}$.

Purtroppo pero' non vengono soddisfatti i punti (b) ed (e) che sono essenziali per applicazioni di una certa complessita'.

Un modo di realizzare integralmente le classi e' quello di introdurre i domini etichettati i/A, dove A e' un dominio e i∈Lab una etichetta. Se i/A e' una classe allora: (a up i/A) e' un elemento della classe i/A sse (a is A). In questo modo i punti (a) ... (e) risultano soddisfatti. Un esempio di sistema di supertipi con classi e' il seguente:

i) bool,dom,any∈Dom
   A∈Dom => ∀i∈Lab. i/A∈Dom

ii) true,false∈Bool sono valori (Bool≤Val)
    i domini sono valori (Dom∈Val)
    a∈Val => (a up i/A)∈Val

iii) ∀ e' individuato dal minimo predicato is che soddisfa le condizioni:
   ∀v∈Val. v is any
   ∀d∈Dom. d is dom
   ∀b∈Bool. b is bool
   ∀a∈Val; ∀A∈Dom; ∀i∈Lab.
       (a up i/A) is i/A <=> a is A

II.10.8: Supertipi per il TAU

Nel costruire un sistema di supertipi la prima scelta da fare riguarda l'operazione di unione. L'unione disgiunta, insieme al prodotto cartesiano e ai domini funzionali forma un sistema completo e ben sperimentato di supertipi. Se a questo si vogliono pero' aggiungere le classi le caratteristiche di strutturazione dei dati dell'unione disgiunta diventano ridondanti e puo' essere piu' conveniente e naturale utilizzare l'unione congiunta.

In TAU l'unione congiunta viene affiancata al prodotto cartesiano, ai domini funzionali, ai domini potenza, agli insiemi e alle classi.

Il prodotto cartesiano presenta una variazione rispetto a quello gia' discusso: ogni elemento di un prodotto di domini viene associato ad una etichetta:

$\langle x^1:A^1; \ldots ;x[n]:A[n]\rangle$

Due prodotti sono uguali se hanno uguali domini ed etichette. Per tutte le altre operazioni sui prodotti, le etichette sono del tutto trasparenti. Questi prodotti etichettati combinati con le classi consentono l'uso della dot-notation nella selezione dei campi di un record. (Qui per record si intende un elemento di una classe costruita su di un prodotto cartesiano)

La dot-notation viene definita cosi': sia $item[(a^1; \ldots ;a[n]);i]=a[i]$ la funzione che estrae l'i-esimo elemento da una lista, e sia

$record=(a^1; \ldots ;a[n])\ up\ i/\langle x^1:A^1; \ldots ;x[n]:A[n]\rangle$

allora:

record.x[i] = item[down record;i]

Anche le classi in TAU presentano alcune particolarita':

i) le labels di classe sono inaccessibili all'utente e vengono automaticamente generate, sempre diverse, ogni volta che viene creata una classe. Ogni nuova classe e' quindi automaticamente diversa da tutte le precedenti (cio' consente una implementazione dell'uguaglianza tra classi basata , in terminologia LISP, su una eq anziche' su una equal).

ii) un elemento di una classe costruita su un dominio funzionale puo' essere applicato:

[f up i/[A->B]][x] = f[x]

iii) l'operazione (a up A) che costruisce gli elementi di una classe viene scritta:

A⁅a¹; ... ;a[n]⁆

se A=i/<x¹:A¹; ... ;x[n]:A[n]> e a=(a¹; ... ;a[n])

(la notazione corretta sarebbe A⁅(a¹; ... a[n])⁆, ma viene usata questa per comodita'); altrimenti:

A⁅a⁆

iv) l'operazione che genera le classi e'

s::d

dove d e' il dominio su cui viene creata la nuova classe i/d e s e' una stringa usata per motivi di stampa. Ad esempio la stampa di

"pair"::<left:any;right:any>⁅3;5⁆

e' semplicemente "pair⁅3≤5⁆. Se s="" viene invece stampata l'intera struttura di d.

v) l'espressione:

let[[a<=d]; ...]

e' una abbreviazione sintattica di:

```
let[[a<-"a"::d]; ... ]
```

e analogamente per il costrutto letrec.

Vediamo adesso il sistema di supertipi del TAU:

i) void,bool,num,string,dom,any∈Dom
A,A[i],B∈Dom; a[i]∈Val; i∈Lab =>

$\langle x^1:A^1; \ldots ;x[n]:A[n]\rangle$	∈ Dom
A|B	∈ Dom
A->B	∈ Dom
A*	∈ Dom
$\{a^1; \ldots ;a[n]\}$	∈ Dom
i/A	∈ Dom

ii) A∈Dom; a[i]∈Val; i∈Lab =>
0,1,2, ... ∈ N; N≤Val
true,false ∈ B; B≤Val
" ... " ∈ S; S≤Val
void,null,bool,num,dom,any ∈ Dom; Dom≤Val
$(a^1; \ldots ;a[n]) \in$ Val
A$(a^1; \ldots ;a[n])$ ∈ Val purche' ∀i. A has a[i]
a up i/A ∈ Val purche' A has a

iii) ∀v∈Val. v is any
∀d∈Dom. d is dom
∀b∈B. b is bool
∀n∈N. n is num
∀s∈S. s is string
∀a,a[i],b∈Val; A,A[i],B∈Dom; i∈Lab.
$(a^1; \ldots ;a[n])$ is $\langle x^1:A^1; \ldots ;x[n]:A[n]\rangle$
<=> ∀i. a[i] is A[i]
a is A|B <=> (a is A) v (a is B)
ƛ[[x:A]->B; ...] is A'->B'
<=> (A in A') & (B in B')
A$(a^1; \ldots ;a[n])$ is B* <=> A in B
∀i. a[i] is $\{a^1; \ldots ;a[n]\}$
a up i/A is i/A

II.11: Tecniche di supporto

II.11.0: Memoria soffice

L'uso spregiudicato che e' stato fatto in vari paragrafi di strutture dati fortemente dinamiche sia nel tempo che nello spazio ([II.2.7] ambienti distribuiti; [II.5.2] chiusure; [II.3] [II.5.2] [II.7] sospensioni; [II.8] continuazioni) ha la sua giustificazione nell'esistenza di un algoritmo di garbage collection in grado di trattare in modo ampiamente soddisfacente strutture di questo tipo [Hewitt-Backer].

L'algoritmo si basa sulla ricopiatura e compattificazione delle strutture attive da un'area di memoria ad un'altra di uguali dimensioni. Questa idea di base puo' essere sviluppata fino ad ottenere un garbage collector incrementale (che raccoglie le strutture attive in parallelo con l'elaborazione) oppure uno che lavora su una memoria partizionata in n parti (anziche' in due), o combinazioni delle due cose.

Se ad un garbage collector di questo tipo si affianca un meccanismo di definizione dei formati dei dati da raccogliere, procedure generali di manipolazione dei dati cosi' definiti, e alcuni tipi di dati built-in (tipicamente interi, booleani e stringhe con le rispettive operazioni elementari) si ottiene un sistema di gestione della memoria capace di servire da supporto per l'implementazione di qualsiasi linguaggio. Lo sforzo di programmazione si riduce in questo modo di almeno un'ordine di grandezza.

III: IL LINGUAGGIO TAU: SEMANTICA

III.0: Semantica a' la Scott-Strachey

III.0.0: Domini sintattici

I domini sintattici vengono definiti a partire da una sintassi ristretta, dalla quale per successive definizioni si puo' ottenere la sintassi completa [III.1.1]. Il processo di estensione e' trattato per ogni singolo costrutto nel corso della seconda parte [II] ed e' riassunto in [III.1.3].

≤var≥ := a|b| ...

≤term≥ := ≤var≥
≤null≥
≤bool≥
≤num≥
≤string≥
≤elemdom≥
≤elemfun≥
≤term≥->≤term≥,≤term≥
≤term≥[≤term≥]
λ[[≤var≥:≤term≥]->≤term≥;≤term≥]
μ[[≤var≥];≤term≥]
Y[[≤var≥];≤term≥]
let[[≤deflist≥];≤term≥]
letrec[[≤deflist≥];≤term≥]
[≤termlist≥]

≤termlist≥ := ≤term≥|≤term≥;≤termlist≥

≤deflist := |≤deflist1≥
≤deflist1≥ := ≤def≥|≤def≥;≤deflist1≥

≤def≥ := ≤var≥ <- ≤term≥

```
var = num

exp = var::var ⊕
      null::⊤ ⊕
      bool::{0,1} ⊕
      num::num ⊕
      string::(length::num⊗chars::num*) ⊕
      elemdom::{0,1,2,3,4,5,6} ⊕
      elemfun::{0, ... ,n} ⊕
      cond::(if::exp⊗then::exp⊗else::exp) ⊕
      appl::(fun::exp⊗arg::exp) ⊕
      lamb::(binder::var⊗body::exp⊗
             domain::exp⊗range::exp) ⊕
      mu::(binder::var⊗body::exp) ⊕
      escape::(binder::var⊗body::exp) ⊕
      let::(defs::def*⊗body::exp) ⊕
      letrec::(defs::def*⊗body::exp) ⊕
      block::exp*

def = binder::var⊗value::exp
```

Il parser opera le traduzioni:

⟦a⟧ = in(exp,var,n) se ⟦a⟧ e' l'n-esima variabile
⟦()⟧ = in(exp,null,τ)
⟦true⟧ = in(exp,bool,0)
⟦false⟧ = in(exp,bool,1)
⟦n⟧ = in(exp,num,n)
⟦"c¹ ... c[m]"⟧ = in(exp,string,⟨m,⟨n¹, ... ,n[m]⟩⟩)
se n[i] e' la codifica del carattere c[i]
⟦d⟧ = in(exp,elemdom,n) se ⟦d⟧ e' l'n-esimo elemdom
⟦f⟧ = in(exp,elemfun,n) se ⟦f⟧ e' l'n-esima elemfun
⟦P->M,N⟧ = in(exp,cond,⟨⟦P⟧,⟦M⟧,⟦N⟧⟩)
⟦M[N]⟧ = in(exp,appl,⟨⟦M⟧,⟦N⟧⟩)
⟦λ[[a:D]->R;M]⟧ = in(exp,lamb,⟨⟦a⟧,⟦M⟧,⟦D⟧,⟦R⟧⟩)
⟦μ[[a];M]⟧ = in(exp,mu,⟨⟦a⟧,⟦M⟧⟩)
⟦Y[[a];M]⟧ = in(exp,escape,⟨⟦a⟧,⟦M⟧⟩)
⟦let[[a¹<-N¹; ... ;a[n]<-N[n]];M⟧ =
in(exp,let,⟨⟨⟨⟦a¹⟧,⟦N¹⟧⟩, ...
,⟨⟦a[n]⟧,⟦N[n]⟧⟩⟩,⟦M⟧⟩)
⟦letrec[[a¹<-N¹; ... ;a[n]<-N[n]];M]⟧ =
in(exp,letrec,⟨⟨⟨⟦a¹⟧,⟦N¹⟧⟩, ...
,⟨⟦a[n]⟧,⟦N[n]⟧⟩⟩,⟦M⟧⟩)
⟦M¹; ... ;M[n]⟧ = in(exp,block,⟨⟦M¹⟧, ... ,⟦M[n]⟧⟩)

III.0.1: Domini semantici

```
V = null::H ⊕
    bool::B ⊕
    num::N ⊕
    string::S ⊕
    pair::L ⊕
    array::A ⊕
    cont::K ⊕
    elemfun::F° ⊕
    fun::F ⊕
    dom::D ⊕
    datum::I

H = ⊥

B = {0,1}

N = num

S = length::num⊗chars::num*

L = first::V⊗rest::(H⊕L)

A = kind::D⊗items::V*

K = V⊗V

F° = {0, ... ,n}

F = kind::D⊗fun::V⊗V

D = elemdom::ED ⊕
    union::UD ⊕
    cross::PD ⊕
    star::ID ⊕
    arrow::AD ⊕
    set::SD ⊕
    class::CD

ED = {0,1,2,3,4,5,6}

UD = left::D⊗right::D

PD = D*

ID = D

AD = domain::D⊗range::D

SD = V*

CD = label::num⊗down::D

I = type::CD⊗proj::V
```

III.0.2: Valutazione

In guanto segue si indicheranno con "a" (a[i]) le variabili ≤var≥, con "()" l'unica costante ≤null≥, con "b" le costanti ≤bool≥ ("true" e "false"), con "n" i ≤num≥, con "s" le ≤string≥, con "d" gli ≤elemdom≥, con "f" le ≤elemfun≥ e con "M","N","P","D","R" i ≤term≥.

```
evaluate = λ(x:exp)->V. eval(close(x),arid,stop)

eval⟦a⟧ek = k(e⟦a⟧)
eval⟦()⟧ek = k(in(V,null,out⟦()⟧))
eval⟦b⟧ek = k(in(V,bool,out⟦b⟧))
eval⟦n⟧ek = k(in(V,num,out⟦n⟧))
eval⟦s⟧ek = k(in(V,string,out⟦s⟧))
eval⟦d⟧ek = k(in(V,dom,in(D,elemdom,out⟦d⟧)))
eval⟦f⟧ek = k(in(V,elemfun,out⟦f⟧))
eval⟦P->M,N⟧ek =
     eval⟦P⟧e(λp.is(bool,p)->out(p)->
          eval⟦M⟧ek,eval⟦N⟧ek,⊥)
eval⟦M[N]⟧ek =
     eval⟦M⟧e(λf.
        is(elemfun,f)->k(call(f,eval⟦N⟧e stop)),
        is(fun,f)->k(apply(out(f),eval⟦N⟧e stop),
        is(cont,f)->eval⟦N⟧e out(f),⊥)
eval⟦λ[[a:D]->R;M]⟧ek =
     k(in(V,fun,⟨eval⟦D->R⟧e stop,
                λa.eval⟦M⟧e[a/⟦a⟧]stop⟩))
eval⟦μ[[a];M]⟧ek = eval⟦a⟧(μe'.e[eval⟦M⟧e' stop/⟦a⟧])k
eval⟦γ[[a];M]⟧ek = eval⟦a⟧e[in(V,cont,k)/⟦a⟧]k
eval⟦let[[a¹<-N¹; ... ;a[n]<-N[n]];M]⟧ek =
     eval⟦N¹⟧e(λa¹. ... eval⟦N[n]⟧e(λa[n].
          eval⟦M⟧e[a¹/⟦a¹⟧] ... [a[n]/⟦a[n]⟧]k) .. )
eval⟦letrec[[a¹<-N¹; ... ;a[n]<-N[n]];M⟧ =
     let e'<-μe'.e[eval⟦N¹⟧e' stop/⟦a¹⟧] ...
                  [eval⟦N[n]⟧e' stop/⟦a[n]⟧]
     in eval⟦N¹⟧e'(λa¹. ... eval⟦N[n]⟧e'(λa[n].
               eval⟦M⟧e'k) .. )
eval⟦[N¹; ... ;N[n]⟧ek =
     eval⟦N¹⟧e(λa¹. ... eval⟦N[n]⟧ek .. )

apply = λ(f:F,a:V)->V.
          has(out(f.kind).domain,a)->
            let v<-f.fun(a)
            in has(out(f.kind).range,v)->v,⊥,⊥

call = λ(f:F°,a:V)->V.
            out(f)->in(V,fun,union(a)),
            out(f)-1->cross(a),
            out(f)-2->star(a),
            ... ,⊥
```

III.0.3: Funzioni elementari

```
union = λa b.in(V,dom,in(D,union,{a,b}))

cross = λa.in(V,dom,in(D,cross,a))

star = λa.in(V,dom,in(D,star,a))

arrow = λa b.in(V,dom,in(D,arrow,{a,b}))

set = λa.in(V,dom,in(D,set,a))

class = λa b.in(V,dom,in(D,class,{a,b}))

pair = λa b.in(V,pair,{a,b})

array = λa.in(V,array,a)

make = λa b.in(V,datum,{a,b})

first = λa.is(pair,a)->out(a).first,T

rest = λa.is(pair,a)->out(out(a).rest),T

kind = λa.is(array,a)->out(a).kind,T

item = λa n.is(array,a)->is(num,n)->
          out(a).items.out(n),T,T

proj = λa.is(datum,a)->out(a).proj,T

left = λa.is(dom,a)->is(union,out(a))->
          out(out(a)).left,T,T

right = λais(dom,a)->is(union,out(a))->
          out(out(a)).right,T,T

factor = λa n.is(dom,a)->is(cross,out(a))->
          is(num,n)->out(out(a)).out(n),T,T,T

base = λa.is(dom,n)->is(star,out(a))->out(out(a)),T,T

domain = λa.is(dom,a)->is(arrow,out(a))->
          out(out(a)).domain,T,T

range = λa.is(dom,a)->is(arrow,out(a))->
          out(out(a)).range,T,T

elem = λa n.is(dom,a)->is(set,out(a))--is(num,n)->
          out(out(a)).out(n),T,T,T

down = λa.is(dom,a)->is(class,out(a))->
          out(out(a)).down,T,T

not = λa.is(bool,a)->in(V,bool,out(a)->1,0),T
```

```
succ = λa.is(num,a->in(V,num,out(a)+1),τ

pred = λa.is(num,a)->in(V,num,out(a)-1),τ

length = λa.is(string,a)->out(a).length,τ

char = λa n.is(string,a)->is(num,n)->
          in(V,string,{1,{out(a).chars.out(n)}}),τ,τ

conc = λa b.is(string,a)->is(string,b)-> ... ,τ,τ

type = λa.is(null,a)->in(V,dom,in(D,elemdom,τ)),
         is(bool,a)->in(V,dom,in(D,elemdom,1)),
         is(num,a)->in(V,dom,in(D,elemdom,2)),
         is(string,a)->in(V,dom,in(D,elemdom,3)),
         is(pair,a)->in(V,dom,in(D,cross, ... )),
         is(array,a)->in(V,dom,in(D,star,out(a).kind)),
         is(cont,a)-> ... ,
         is(elemfun,a)-> ... ,
         is(fun,a)->out(a).kind,
         is(dom,a)->in(V,dom,in(D,elemdom,4)),
         is(datum,a)->out(a).type,τ

has = λa b.
       let true<-in(V,bool,0)
           false<-in(V,bool,1)
       in is(dom,a)->
            is(elemdom,out(a))->
              out(out(a))=0->is(null,b)->true,false,
              out(out(a))=1->is(bool,b)->true,false,
              out(out(a))=2->is(num,b)->true,false,
              out(out(a))=3-> is(string,b)->true,false,
              out(out(a))=4->is(dom,b)->true,false,
              out(out(a))=5->true,τ
            is(union,out(a))-> ... ,
            is(cross,out(a))-> ... ,
            is(star,out(a))-> ... ,
            is(arrow,out(a))-> ... ,
            is(set,out(a))-> ... ,
            is(class,out(a))->is(datum,b)->
              (out(out(a))=b.type)->true,false,false,
            τ,
          τ
```

III.1: Interprete metacircolare

III.1.0: Organizzazione generale

Si distinguono tre livelli disgiunti di strutture dati:

*) rappresentazioni esterne: Rext

*) rappresentazioni interne: Rint

*) insieme dei valori: val

L'interprete e' costituito dalle funzioni:

*) parse: Rext -> Rint

*) eval: Rint -> val

*) print: val -> Rext

III.1.1: Rappresentazione esterna

```
≤term≥ := ≤const≥
          ≤var≥
          (≤termlist≥)
          ≤term≥(≤termlist≥)
          ≤term≥.≤ident≥
          ≤term≥->≤term≥,≤term≥
          ≤term≥[≤termlist≥]
          /[[≤parlist≥]≤range≥;≤term≥]
          ϕ[[≤par≥];≤term≥]
          Y[[≤var≥];≤term≥]
          let[[≤letdeflist≥];≤term≥]
          letrec[[≤deflist≥];≤term≥]
          <≤fieldeflist≥>
          ≤term≥|≤term≥
          ≤term≥->≤term≥
          ≤term≥*
          {≤termlist≥}
          ≤term≥::≤term≥
          ≤term≥[≤termlist≥:]
          [≤blocktermlist≥]
          ≤monop≥≤term≥
          ≤term≥≤dyop≥≤term≥

≤const≥ := ≤null≥|≤bool≥|≤num≥|≤string≥|≤elemdom≥
           |≤elemfun≥

≤null≥ := ()

≤bool≥ := true|false
≤num≥ := 0|1|2| ...
≤string≥ := ""|"a"|"b"| ... |"aa"|"ab"| ...
≤elemdom≥ := void|null|bool|num|string|dom|any
≤elemfun≥ := first|rest|left|right| ...

≤var≥ := ogni ≤ident≥ che non sia una costante
         o una keyword
≤ident≥ := ogni sequenza di caratteri

≤par≥ := ≤var≥|≤var≥:≤term≥
≤parlist≥ := |≤parlist1≥
≤parlist1≥ := ≤par≥|≤par≥;≤parlist1≥
≤range≥ := |->≤term≥
≤def≥ := ≤var≥<-≤term≥|≤var≥<=≤term≥
≤deflist≥ := |≤deflist1≥
≤deflist1≥ := ≤def≥|≤def≥;≤deflist1≥
≤letdeflist≥ := |≤letdeflist1≥
≤letdeflist1≥ := ≤def≥|≤def≥;≤letdeflist1≥
                 |≤def≥then≤letdeflist1≥
≤fieldef≥ := ≤term≥|≤ident≥:≤term≥
≤fieldeflist≥ := |≤fieldeflist1≥
≤fieldeflist1≥ := ≤field≥|≤field≥;≤fieldeflist1≥
≤termlist≥ := ≤term≥|≤term≥;≤termlist≥
≤monop≥ := +|-| ...
≤dyop≥ := +|-|x|/|=| ...
```

III.1.2: Rappresentazione interna

```
nterm <- nconst|nvar|ncond|nappl|nlamb|nmu|nlet|nletrec
        |ncont|nblock

nconst <- nnull|nbool|nnum|nstring|nelemdom|nelemfun

nnull <= <decode:<>>
nbool <= <decode:bool>
nnum <= <decode:num>
nstring <= <decode:string>
nelemdom <= <decode:elemdom>
nelemfun <= <decode:elemfun>

nvar <= <name:string>
ncond <= <if:nterm;then:nterm;else:nterm>
nappl <= <fun:nterm;args:ntermlist>
nlamb <= <pars:nparlist;range:nterms;form:nterm>
nmu <= <var:nvar;range:nterm;form:nterm>
nlet <= <defs:ndeflist;body:nterm>
nletrec <= <defs:ndeflist;body:nterm>
ncont <= <var:nvar;body:nterm>
nblock <= <terms:nterms>

ntermlist <- empty|nterms
nparlist <- empty|npars
ndeflist <- empty|ndef
empty <= <>
nterms <= <first:nterm;rest:ntermlist>
npars <= <var:nvar;kind:nterm;rest:nparlist>
ndef <= <var:nvar;val:nterm;rest:ndeflist>

elemdom <- {{};<>;bool;num;string;dom;any}
elemfun <- {first;rest;left;right; ... }
```

III.1.3: Parser

Il parser traduce le rappresentazioni esterne in rappresentazioni interne, rispettando le specifiche sotto elencate.

```
⟦()⟧ = nnull⟦()⟧
⟦b⟧ = nbool⟦b⟧
⟦n⟧ = nnum⟦n⟧
⟦s⟧ = nstring⟦s⟧
⟦d⟧ = nelemdom⟦d⟧
⟦f⟧ = nelemfun⟦f⟧
⟦a⟧ = nvar⟦"a"⟧
⟦(A¹; ... ;A[n])⟧ = ⟦pair[A¹; ... pair[A[n];()] .. ]⟧
⟦A(A¹; ... ;A[n])⟧ = ⟦array[A;(A¹; ... ;A[n])]⟧
⟦A.a⟧ = ⟦dot[A;"a"]⟧
⟦P->M,N⟧ = ncond⟦⟦P⟧;⟦M⟧;⟦N⟧⟧
⟦M[N¹; ... ;N[n]]⟧ =
     nappl⟦⟦M⟧;nterms⟦⟦N¹⟧; ...
                  nterms[⟦N[n]⟧;empty] .. ⟧⟧
⟦λ[[a¹:D¹; ... ;a[n]:D[n]]->R;M]⟧ =
     nlamb⟦nparse⟦⟦a¹⟧;⟦D¹⟧; ...
               nparse⟦⟦a[n]⟧;⟦D[n]⟧;empty] .. ⟧;
          ⟦R⟧;⟦M⟧⟧
⟦μ[[a:D];M]⟧ = nmu⟦⟦a⟧;⟦D⟧;⟦M⟧⟧
⟦γ[[a];M]⟧ = ncont⟦⟦a⟧;⟦M⟧⟧
⟦let[[a¹<-A¹; ... ;a[n]<-A[n]];M]⟧ =
     nlet⟦ndef⟦⟦a¹⟧;⟦A¹⟧; ...
           ndef⟦⟦a[n]⟧;D[n]⟧;empty] .. ⟧;
        ⟦M⟧⟧
⟦letrec[[a¹<-A¹; ... ;a[n]<-A[n]];M]⟧ =
     nletrec⟦ndef⟦⟦a¹⟧⟦A¹⟧; ...
                ndef⟦⟦a[n]⟧;⟦A[n]⟧;empty] .. ⟧;
             ⟦M⟧⟧
⟦<a¹:D¹; ... ≤a[n]:D[n]>⟧ =
     ⟦cross[("a¹";D¹); ...
        cross[("a[n]";D[n]);<>] .. ]⟧
⟦M|N⟧ = ⟦union[A;B]⟧
⟦M->N⟧ = ⟦arrow[A;B]⟧
⟦A*⟧ = ⟦star[a]⟧
⟦{A¹; ... ;A[n]}⟧ = set[A¹; ... set[A[n];{}] .. ]⟧
⟦A::D⟧ = ⟦class[A;D]⟧
⟦A⟦A¹; ... ;A[n]⟧⟧ = ⟦make[A;(A¹; ... ;A[n])]⟧
⟦[A¹; ... ;A[n]]⟧ =
     nblock⟦nterms⟦⟦A¹⟧; ...
               nterms⟦⟦A[n]⟧;empty] .. ⟧⟧
⟦pA⟧ = ⟦p[A]⟧
⟦ApB⟧ = ⟦p[A;B]⟧
```

III.1.4: Eval

```
val <- any

evaluate <-
  λ[[ ];
    letrec[[loop<-
             λ[[dummy];
               loop[print[eval[read[ ];arid]]]];
             arid<-a-list‡λ[[x:nvar]->val;error[ ]]‡];
            loop[() ]]]

a-list <= nvar->val

extend <-
  λ[[var:nvar;value:val;env:list]->a-list;
    a-list‡λ[[x:nvar]->val;
             equal[x;var]->value,env[var]]‡]

eval <-
  λ[[term:nterm;env:a-list]->val;
    term is nconst -> term.decode,
    term is nvar -> env[term],
    term is ncond ->
      eval[term.if;env]->eval[term.then;env],
                         eval[term.else;env],
    term is nappl ->
      apply[eval[term.fun;env];term.args;env],
    term is nlamb ->
      abstr[term.pars;term.range;term.form;env],
    term is nmu ->
      μ[[x:eval[term.range;extend[term.var;x;env]]];
        eval[term.form;extend[term.var;x;env]]],
    term is nlet ->
      eval[term.body;let-extend[term.defs;env]],
    term is nletrec ->
      eval[term.body;letrec-extend[term.defs;env]],
    term is ncont ->
      Y[[x];eval[term.body;extend[term.var;x;env]]],
    term is nblock -> evalist[term.terms;env],
    error[ ]]
```

```
apply <-
  λ[[fun:any->any;args:ntermlist;env:list]->val;
     args is empty ->fun,
     apply[fun[eval[args.first;env]];
           args.rest;
           env]]

 abstr <-
   λ[[pars:nparlist;range:nterm;
      form:nterm;env:list]->any->any;
     letrec[[abstr1<-
             λ[[pars:nparlist;newenv:list]->any->any;
               pars is empty->
                 λ[[ ]->eval[range;env];
                   eval[form;newenv]],
                 λ[[x:eval[pars.kind;env]]->val;
                   abstr1[pars.rest;
                          extend[pars.var;x;newenv]]]]];
           abstr1[pars;env]]]

 let-extend <-
   λ[[deflist:ndeflist;env:a-list]->a-list;
     expand[deflist;env;env]]

 letrec-extend <-
   λ[[deflist:ndeflist;env:a-list]->a-list;
     ℵ[[recenv:list];expand[deflist;env;recenv]]]

 expand <-
   λ[[deflist:ndeflist;newenv:a-list;
      suspenv:a-list]->a-list;
     deflist is empty -> newenv,
     expand[deflist.rest;
            extend[deflist.var;
                   eval[deflist.val;suspenv];
                   newenv];
          suspenv]]

 evlist <-
   λ[[terms:nterms;env:a-list]->val;
     terms.rest is empty-> eval[terms.first;env],
       [eval[terms.first;env];
        evalist[terms.rest;env]]]
```

III.2: Interprete iterativo

III.2.0: Rappresentazione interna

```
nterm <- nconst|nvar|ncond|nappl|nlamb|nmu|nlet|nletrec
         |ncont|nblock

nconst <- nnull|nbool|nnum|nstring|nelemdom|nelemfun

nnull <= <decode:nullval>
nbool <= <decode:boolval>
nnum <= <decode:numval>
nstring <= <decode:stringval>
nelemdom <= <decode:elemdomval>
nelemfun <= <decode:elemfunval>

nval <= <name:string>
ncond <= <if:nterm;then:nterm;else:nterm>
nappl <= <fun:nterm;args:ntermlist>
nlamb <= <pars:nparlist;range:nterm;form:nterm>
nmu <= <var:nvar;range:nterm;form:nterm>
nlet <= <defs:ndeflist;body:nterm>
nletrec <= <defs:ndeflist;body:nterm>
ncont <= <var:nvar;body:nterm>
nblock <= <terms:nterms>

ntermlist <- empty|nterms
nparlist <- empty|npars
ndeflist <- empty|ndef
empty <= <>
nterms <= <first:nterm;rest:ntermlist>
npars <= <var:nvar;range:nterm;rest:nparlist>
ndef <= <var:nvar;val:nterm;rest:ndeflist>
```

III.2.1: Insieme dei valori

```
val <- const|pairval|vectorval|closure|compdomval|
       |datum|c-list

const <- nullval|boolval|numval|stringval|elemdomval
         |elemfunval

dcmval <- elemdomval|compdomval

funval <- elemfunval|closure|datum|c-list

nullval <= <value:<>>
boolval <= <value:bool>
numval <= <value:num>
stringval <= <value:num>
elemdomval <= <value:elemdom>
elfunval <- niladic|monadic|dyadic

list <- nullval|pairval
pairval <= <first:val|susp;rest:list|susp>
vectorval <= <range:domval|susp;tuple:[val|susp]*>
closure <= <fun:nterm;env:a-list>
datum <= <range:classval|susp;down:val|susp>
compdomval <- crossval|unionval|arrowval|arrayval
              |fullset|classval
crossval <- emptycross|crosspair
emptycross <= <>
crosspair <= <name:string|susp;first:domval|susp;
              rest:crossval|susp>

unionval <= <left:domval|susp;right:domval|susp>
arrowval <= <domain:domval|susp;range:domval|susp>
arrayval <= <base:domval|susp>
setval <- emptyset|fullset
emptyset <= <>
fullset <= <take:val|susp;drop:setval|susp>
crossval <= <pname:string|susp;base:domval|susp>

elemdom <- {[];<>;bool;num;string;dom;any}
niladic <= <value:{ ... }>
monadic <- strict-monadic|notstrict-monadic
dyadic <- strict dyadic|notstrict dyadic
strict-monadic <= <value:{ ... }>
notstrict-monadic <= <value:{ ... }>
strict-dyadic <= <value:{ ... }>
notstrict-dyadic <= <value:{ ... }>
```

```
c-list <- condcont|applcont|applistcont|letcont
        |letreccont|letrecdefcont|letrecbodycont
        |blockcont|niladiccont|monadiccont|dyadiccont
        |dyadicargcont|evalsuspcont
        |evalsusprangecont|evalsuspiscont
        |retractcont|retractrangecont
        |retractiscont

condcont <= <cond:nterm;env:a-list;cont:c-list>
applcont <= <args:ntermlist;env:a-list;cont:c-list>
applistcont <= <args:ntermlist;env:a-list;cont:c-list>
evalsuspcont <= <record:any;field:string;cont:c-list>
evalsusprangecont <= <datum:val;record:any;
                      field:string;cont:c-list>
evalsuspiscont <= <datum:val;record:any;
                   field:string;cont:c-list>
letcont <= <deflist:ndeflist;body:nterm;env:a-list;
           newenv:a-list;cont:c-list>
letrccont <= <deflist:ndeflist;cont:c-list>
letrecdefcont <= <env:a-list;cont:c-list>
letrecbodycont <= <body:nterm;env:a-list;cont:c-list>
blockcont <= <terms:ndeflist;env:a-list;cont:c-list>
niladiccont <= <args:ntermlist;env:a-list;cont:c-list>
monadiccont <= <fun:strict-monadic;args:ntermlist;
                env:a-list;cont:c-list>
dyadiccont <= <fun:strict-dyadic;args:ntermlist;
               env:a-list;cont:c-list>
dyadicargcont <= <fun:strict-dyadic;arg1:val;
                  args:ntermlist;env:a-list;
                  cont:c-list>
retractcont <= <range:nterm;env:a-list;cont:c-list>
retractrangecont <= <datum:val;cont:c-list>
retractiscont <= <result:val;cont:c-list>
```

III.2.2: Eval

L'interprete iterativo per il TAU e' scritto in un linguaggio TAU+assign di cui non viene esplicitamente data la semantica. Essa e' comunque ricavabile da quanto detto in [I.3.6]. L'assegnamento viene realizzato da una funzione assign[a;b] che mette il valore di b nella locazione di a. Il valore di assign[a;b] e' b. L'assegnamento viene usato solo per rimpiazzare una sospensione col suo valore, nella call-by-need, nel letrec e nel μ.

```
a-list <- arid|environ
arid <= <>
environ <= <var:nvar;val:val|susp;env:a-list>

susp <= <term:nterm;range:val|susp;env:a-list>
```

```
eval <-
  /[[term:nterm;env:a-list;cont:c-list]->val;
    term is nconst -> capply[cont;term.decode],
    term is nvar -> lookup[term;env;cont],
    term is ncond ->
      eval[term.if;env;condcont[term;env;cont]],
    term is nappl ->
      eval[term.fun;env;applcont[term.args;env;cont]],
    term is nlamb ->
      capply[cont;closure[term;env]],
    term is nmu ->
      eval[term.var;mu-extend[term;env];cont],
    term id nlet ->
      term.deflist is empty -> eval[term.body;env;cont],
      eval[term.deflist.val;env;
           letcont[term.deflist;term.body;env;env;cont]],
    term is nletrec ->
      let[[newenv<-letrec-extend[term.deflist;env]];
          capply[letreccont[term.deflist;
                            letrecbodycont[term.body;
                                           newenv;
                                           cont]];
                 newenv],
    term is ncont ->
      eval[term.body;environ[term.var;cont;env];cont],
    term is nblock -> evlist[term.terms;env;cont],
    error[cont]]
```

```
lookup <-
  λ[[var:nvar;env:a-list;cont:c-list]->val;
    env is arid -> error[cont],
      var=env.var -> evalsusp[env;"val";cont],
      lookup[var;env.env;cont]]

evalsusp <-
  λ[[record:any;field:string;cont:c-list]->val;
   let[[sus<-dot[record;field]];
       sus is susp ->
          eval[sus.term;sus.env;
               evalsuspcont[record;field;cont]],
       capply[cont;sus]]]

mu-extend <-
 λ[[term:nmu;env:a-list]->a-list;
   let[[newenv<-
         environ[term.var;
                 susp[term.form;
                      susp[term.range;elemdomval[dom];
                           arid[]];
                      arid[]];
                 env]];
       [assign[newenv.val.env;newenv];
        assign[newenv.val.range.env;newenv]]]]
```

```
apply <-
 λ[[fun:funval;args:ntermlist;env:a-list;
    cont:c-list]->val;
   fun is elemfunval -> call[fun.args;env;cont],
   fun is closure ->
     letrec[[applist<-
            λ[[parlist:nparlist;arglist:ntermlist;
               newenv:a-list;cont:c-list]->val;
              parlist is empty ->
                eval[fun.fun.form;newenv;
                     retractcont[fun.fun.range;env;
                       applistcont[arglist;env;cont]]],
              arglist is empty ->
                capply[cont;
                       closure[nlamb[parlist;
                                     fun.fun.range;
                                     fun.fun.form];
                               newenv]],
              applist[parlist.rest;arglist.rest;
                      environ[parlist.var;
                              susp[arglist.first;
                                   susp[parlist.range;
                                        elemdomval[dom];
                                        fun.env];
                                   env];
                              newenv]
                      cont]]];
            applist[fun.fun.pars;args;fun.env;cont]],
   fun is datum ->
     ndown[fun;applcont[args;env;cont]],
   fun is c-list ->
     args is empty -> fun,
     eval[args.first;env;fun],
   error[cont]]
```

```
letrec-extend <-
 /[[deflist:ndeflist;env:a-list]->a-list;
   let[[recenv<-suspextend[deflist;env]];
       rewind[env;recenv;recenv]]]

suspextend <-
 /[[deflist:ndeflist;env:a-list]->a-list;
   deflist is empty -> env,
   suspextend[deflist.rest;
              environ[deflist.var;
                      susp[deflist.term;
                           elemdomval[any];
                           arid[]];
                      env]]]

rewind <-
 /[[oldenv:a-list;recenv:a-list;env:a-list]->a-list;
   oldenv=env -> recenv,
   rewind[oldenv;
          recenv;
          [assign[env.val.env;recenv];env.env]]]

evlist <-
 /[[terms:nterms;env:a-list;cont:c-list]->val;
   eval[terms.first;env;blockcont[terms.rest;env;cont]]]
```

```
call <-
 λ[[fun:elemfun;args:ntermlist;env:a-list;
    cont:c-list]->val;
   fun is niladic ->
     call-niladic[fun;niladiccont[args;env;cont]],
   args is empty -> capply[cont;fun],
   fun is strict-monadic ->
     eval[args.first;env;
          monadiccont[fun;args.rest;env;cont]],
   fun is strict-dyadic ->
     args.rest is empty ->
       capply[cont;part-dyadic[fun;args.first;env]],
     eval[args.first;env;
          dyadiccont[fun;args.rest;env;cont]],
   fun is notstrict-monadic ->
     call-monadic[fun;
                  susp[args.first;elemdomval[any];env];
                  niladiccont[args.rest;env;cont]]
   fun is notstrict-dyadic ->
     args.rest is empty ->
       capply[cont;part-dyadic[fun;args.first;env]],
     call-dyadic[fun;
                 susp[args.first;elemdomval[any];env];
                 susp[args.rest.first;
                      elemdomval[any];env];
                 niladiccont[args.rest;env;cont]],
   error[cont]]
```

```
part-dyadic <-
 /[[fun:elemfunval;arg:nterm;env:a-list]->closure;
   closure[nlamb[npars[nvar["x"];
                      nelemdom[elemdomval[any]];
                      empty[]];
                nelemdom[elemdomval[any]];
                nappl[fun;
                      nterms[arg;
                             nterms[nvar["x"];
                                    empty[]]]]];
           env]]

call-niladic <-
 /[[fun:niladic;cont:c-list]->val;
   fun.value[cont]]

call-monadic <-
 /[[fun:monadic;arg:val|susp;cont:c-list]->val;
   fun.value[arg;cont]]

call-dyadic <-
 /[[fun:dyadic;arg1:val|susp;arg2:val|susp;
    cont:c-list]->val;
   fun.value[arg1;arg2;cont]]
```

```
capply <-
 /[[cont:c-list;value:val]->;
   cont is condcont ->
     value is boolval ->
       eval[boolval.value->cont.cond.then,cont.cond.else;
            cont.env;cont.cont],
     error[cont],
   cont is applcont ->
     apply[value;cont.args;cont.env;cont.cont],
   cont is applistcont ->
     cont.args is empty -> capply[cont.cont;value],
       apply[value;cont.args;cont.env;cont.cont],
   cont is letcont ->
     cont.deflist.rest is empty ->
       eval[cont.body;
            environ[cont.deflist.var;value;cont.newenv];
            cont.cont],
     eval[cont.deflist.rest.val;cont.env;
          letcont[cont.deflist.rest;cont.body;cont.env;
                  environ[cont.deflist.var;value;
                          cont.newenv];
                  cont.cont]],
   cont is letreccont ->
     cont.deflist is empty -> capply[cont.cont;()],
     capply[letreccont[cont.deflist.rest;
                       letrecdefcont[value;cont.cont]];
            value.env],
   cont is letrecdefcont ->
     evalsusp[cont.env;"val";cont.cont],
   cont is letrecbodycont ->
     eval[cont.body;cont.env;cont.cont],
   cont is blockcont ->
     cont.terms is empty -> capply[cont.cont;value],
     eval[cont.terms.first;cont.env;
          blockcont[cont.terms.rest;cont.env;cont.cont]],
   cont is niladiccont _>
     cont.args is empty -> capply[cont.cont;value],
     apply[value;cont.args;cont.env;cont.cont],
   cont is monadiccont ->
     call-monadic[cont.fun;value;
                  niladiccont[cont.args;cont.env;
                              cont.cont]],
   cont is dyadiccont ->
     eval[cont.args.firs;cont.env;
          dyadicargcont[cont.fun;value;cont.args.rest;
                        cont.env;cont.cont]],
   cont is dyadicargcont ->
     call-dyadic[cont.fun;cont.arg1;value;
                 niladiccont[cont.args;cont.env;
                             cont.cont]],
   cont is evalsuspcont ->
     evalsusp[dot[cont.record;cont.field];"range";
              evalsusprangecont[value;cont.record;
                                cont.field;cont.cont]],
   cont is evalsusprangecont ->
     nis[cont.datum;value;
         evalsuspiscont[value;cont.record;cont.field;
                        cont.cont]],
```

```
cont is evalsuspiscont ->
  value -> capply[cont.cont;
                  assign[dot[cont.record;cont.field];
                         cont.datum]],
  error[cont],
cont is retractcont ->
eval[cont.range;cont.env;
     retractrangecont[value;cont.cont]],
cont is retractrangecont ->
  mis[cont.datum;value;
      retractiscont[cont.datum;cont.cont]],
cont is retractiscont ->
  value -> capply[cont.cont;cont.result],
  error[cont],
error[cont]]
```

I.2.6: Note tecniche

Prova della Prop 1

$x \leq y \Rightarrow \{e[n] \mid e[n] \leq x\} \leq \{e[n] \mid e[n] \leq y\}$
$\Rightarrow \{f(e[n]) \mid e[n] \leq x\} \leq \{f(e[n]) \mid e[n] \leq y\}$
$\Rightarrow \cup\{f(e[n]) \mid e[n] \leq x\} \leq \cup\{f(e[n]) \mid e[n] \leq y\}$
$\Rightarrow f(x) \leq f(y)$

Prova della Prop 2

a) $f(x) = \cup\{f(e[n]) \mid e[n] \leq x\}$ per ipotesi.
   a.1) $e[n] \leq x$; $e[m] \leq f(e[n])$:
   per la monotonicita' $f(e[n]) \leq f(x)$
   quindi $e[m] \leq f(x)$
   a.2) $e[m] \leq f(x)$:
   $e[m] \leq \cup\{f(e[n]) \mid e[n] \leq x\}$
   cioe' $\forall k \in e[m].\ \exists e[n[k]] \leq x.\ k \in f(e[n[k]])$
   consideriamo $e[n] = \cup\{e[n[k]] \mid k \in e[m]\}$
   ogni $e[n[k]] \leq x$ e quindi anche $e[n] \leq x$
   inoltre $e[m] \leq f(e[n])$ perche':
   $\forall k \in e[m].\ k \in f(e[n[k]])$
   $e[n[k]] \leq e[n] \Rightarrow f(e[n[k]]) \leq f(e[n])$
   $k \in f(e[n])$

b) $e[m] \leq f(x) \iff \exists e[n] \leq x.\ e[m] \leq f(e[n])$ per ipotesi.
questa proprieta' vale per qualsiasi $e[m]$, ed
in particolare varra' anche per gli $e[m[k]] = \{k\}$
$\{k\} \leq f(x) \iff \exists e[n] \leq x.\ \{k\} \leq f(e[n])$ cioe':
$k \in f(x) \iff \exists e[n] \leq x.\ k \in f(e[n])$
$k \in f(x) \iff k \in \cup\{f(e[n]) \mid e[n] \leq x\}$
$f(x) = \cup\{f(e[n]) \mid e[n] \leq x\}$

Prova della Prop 3

discende dalla discussione che segue la Prop 2
in [I.2.0].

Prova della Prop 4

a) Dimostriamo che $f = fun(u)$ e' continua, cioe' che
$f(x) = \cup\{f(e[n]) \mid e[n] \leq x\}$:
se $m \in f(x)$ per definizione di $fun(u)$:
$\exists e[n] \leq x.\ (n,m) \in u$
ponendo $x = e[n]$ otteniamo anche che $m \in f(e[n])$
e quindi $m \in \cup\{f(e[n]) \mid e[n] \leq x\}$.
Viceversa se $m \in \cup\{f(e[n]) \mid e[n] \leq x\}$ allora
$\exists e[n] \leq x.\ m \in f(e[n])$ e per monotonicita' $m \in f(x)$

b) $fun(graph(f))(x) =$
$\{m \mid \exists e[n] \leq x.\ (n,m) \in \{(n,m) \mid m \in f(e[n])\}\} =$
$\{m \mid \exists e[n] \leq x.\ m \in f(e[n])\} =$
$\{m \mid m \in \cup\{f(e[n]) \mid e[n] \leq x\}\} =$
$\{m \mid m \in f(x)\} =$
$f(x)$

c) $graph(fun(u)) =$
$\{(n,m) \mid m \in fun(u)(e[n])\} =$
$\{(n,m) \mid m \in \{m \mid \exists e[k] \leq e[n].\ (k,m) \in u\}\} =$

$\{(n,m) \mid \exists e[k] \leq e[n].\ (k,m) \in u\} \geq$
$\{(n,m) \mid (n,m) \in u\} =$
u

d) come il punto (c) sostituendo '=' a '≥' in virtu' dell'ipotesi che $\exists e[n] \leq e[m].\ (k,m) \in u \Rightarrow (n,m) \in u$.

Prova della Prop 5

La sostituzione e' la composizione generalizzata. Questa proposizione afferma che $p(w,x,y,z) = f(g(x,w),h(x,y,z))$ e' continua se f,g e h lo sono. Per far questo basta mostrare che

1) $p'(w,x,x',y,z) = f(g(x,w),h(x',y,z))$ e' continua (poiche' ogni variabile e' trattata separatamente e' sufficiente far vedere ad es. che $a(b(w))$ e' continua se a e b lo sono, con $b(w)=g(x^0,w)$ e $a(v)=f(v,h(x^0,y^0,z^0))$
2) $p'(w,x,x,y,z) = p(w,x,y,z)$ e' continua (e per questo basta fissare le variabili w,y,z e verificare che $a(x,x)$ e' continua se $a(x,y)$ lo e', con $a(u,v)=p'(w^0,u,v,y^0,z^0)$)

Per il primo caso abbiamo:

$e[m] \leq f(g(x)) \iff \exists e[n] \leq g(x).\ e[m] \leq f(e[n])$
[f continua; Prop 2]
$\iff \exists n,k.\ (e[k] \leq x) \& (e[n] \leq g(e[k])) \& (e[m] \leq f(e[n]))$
[g continua; Prop 2]
$\iff \exists e[k] \leq x.\ (\exists e[n] \leq g(e[k]).\ e[m] \leq f(e[n]))$
$\iff \exists e[k] \leq x.\ e[m] \leq f(g(e[k]))$ [f continua; Prop 2]

Per il secondo:

$e[m] \leq f(x,x) \iff \exists e[n] \leq x.\ e[m] \leq f(e[n],x)$
[f continua; Prop 2]
$\iff \exists n,k.\ (e[n] \leq x) \& (e[k] \leq x) \& (e[m] \leq f(e[n],e[k]))$
[f continua; Prop 2]
$\iff \exists e[j] \leq x.\ e[m] \leq f(e[j],e[j])$

dove si pone $e[j] = e[k]\ u\ e[n]$. Infatti per la monotonicita' di f: $e[m] \leq f(e[n],e[k]) \leq f(e[j],e[j])$

Prova della Prop 6

a) Dimostriamo che x e' un punto fisso: $x=f(x)$

z.1) $U\{f[n](\emptyset) \mid n \in w\} \leq x \Rightarrow \forall n \in w.\ f[n](\emptyset) \leq x$
$\Rightarrow \forall n \in w.\ f[n+1](\emptyset) \leq f(x)$
$\Rightarrow \forall n \in w.\ f[n](\emptyset) \leq f(x)$
$\Rightarrow U\{f[n](\emptyset) \mid n \in w\} \leq f(x)$
$\Rightarrow x \leq f(x)$

a.2) Sia $e[m] \leq f(x)$, allora $\exists e[k] \leq x.\ e[m] \leq f(e[k])$
Poiche' $e[k] \leq x = U\{f[n](\emptyset) \mid n \in w\}$ e $f[i](\emptyset) \leq f[i+1](\emptyset)$ allora $e[k] \leq f[m](\emptyset)$ per qualche n (perche' $e[k]$ e' finito).
Ma allora $e[m] \leq f(e[k]) \leq f[n+1](\emptyset) \leq x$.
Cioe' $e[m] \leq f(x) \Rightarrow e[m] \leq x$ e quindi $f(x) \leq x$.

b) Dimostriamo che x e' il minimo punto fisso.
Sia $y=f(y)$:
$\emptyset \leq y \Rightarrow f(\emptyset) \leq f(y) = y$
$\Rightarrow \forall n \in w.\ f[n](\emptyset) \leq y$
$\Rightarrow x \leq y$

Lemma alla Prop 7

Per ogni predicato P, f continua =>
U{{m|P(m,f(e[n]))}|e[n]≤x} =
{m|P(m,U{f(e[n]}|e[n]≤x})} =
{m|∃e[n]≤x.P(m,f(e[n]))}

Prova della Prop 7

Se f e' LAMBDA-definibile allora esiste un LAMBDA-termine M che definisce f come funzione delle sue variabili libere.

a) se M=0, M definisce una funzione z=(λx∈Pw.{0}) che e' continua:
U{&(e[n])|e[n]≤x} = U{{0}|e[n]≤x} = {0} = &(x)

b) se M=a la funzione
Id = (λx∈Pw.a[x/a]) = (λx∈Pw.x) e' continua:
U{Id(e[n])|e[n]≤x} = Ue[n]|e[n]≤x} = x = Id(x)

c) se M=N+1 e N e' un termine che definisce una funzione continua g(x) (possiamo considerare una variabile libera alla volta fissando le rimanenti) allora M definisce una funzione succ(g(x)) che e' continua:
U{succ(g(x))|e[n]≤x} = U{{m+1|m∈g(e[n])|e[n]≤x}
= {m+1|m∈U{g(e[n])|e[n]≤x}}
= {m+1|m∈g(x)}
= succ(g(x))

d) se M=N-1 e N definisce una funzione continua g: analogamente al punto (c)

e) se M=N:>P,Q e N,P,Q definiscono funzioni continue h,f,g allora M definisce una funzione continua:

e.1) continuita' rispetto alle variabili libere di h
U{h(e[n]):>f,g|e[n]≤x} =
U{{n∈f|0∈h(e[n])}u{m∈g|∃k.k+1∈h(e[n])}|e[n]≤x} =
{n∈f|0∈U{h(e[n])|e[n]≤x}} u
{m∈g|∃k.k+1∈U{h(e[n])|e[n]≤x}} =
{n∈f|0∈h(x)}u{m∈g|∃k.k+1∈h(x)} =
h(x):>f,g

e.2) continuita' rispetto alle variabili libere di f
U{h:>f(e[n]),g|e[n]≤x} =
U{{n∈f(e[n]|0∈h}u{m∈g|∃k.k+1∈h}|e[n]≤x} =
{n∈U{f(e[n])|e[n]≤x}|0∈h}u{m∈g|∃k.k+1∈h} =
{n∈f(x)|0∈h}u{m∈g|∃k.k+1∈h}
=h:>f(x),g

e.3) continuita' rispetto alle variabili libere di g
U{h:>f,g(e[n])|e[n]≤x} =
U{{n∈f|0∈h}u{m∈g(e[n])|∃k.k+1∈h}|e[n]≤x} =
{n∈f|0∈h}u{m∈U{g(e[n])|e[n]≤x}|∃k.k+1∈h} =
{n∈f|o∈h}u{m∈g(x)|∃k.k+1∈h} =
h:>f,g(x)

f) se M=P(Q) e P,Q definiscono funzioni continue f,g

f.1) continuita' rispetto alle variabili libere di f
U{f(e[n])(g)|e[n]≤x} =
U{{m|∃e[k]≤g.(k,m)∈f(e[n])}|e[n]≤x} =
{m|∃e[k]≤g.(k,m)∈U{f(e[n])|e[n]≤x}} =
{m|∃e[k]≤g.(k,m)∈f(x)} =
f(x)(g)

f.2) continuita' rispetto alle variabili libere di g
U{f(g(e[n]))|e[n]≤x} =
U{{m|∃e[k]≤g(e[n]).(k,m)∈f}|e[n]≤x} =
{m|∃e[k]≤U{g(e[n])|e[n]≤x}.(k,m)∈f} =

$\{m \mid \exists e[k] \leq g(x).(k,m) \in f\} =$
$f(g(x))$

g) se $M=\lambda a.N$ e N definisce una funzione f allora M definisce una funzione con una variabile libera in meno. E' sufficiente considerare f(x,y) come funzione di due variabili e dimostrare che $\lambda x \in Pw.f(x,y)$ e' una funzione continua nella variabile y.

$U\{\lambda x \in Pw.f(x,e[k]) \mid e[k] \leq y\} =$
$U\{\{(n,m) \mid m \in f(e[n],e[k])\} \mid e[k] \leq y\} =$
$\{(n,m) \mid m \in U\{f(e[n],e[k]) \mid e[k] \leq y\}\} =$
$\{(n,m) \mid m \in f(e[n],y)\} =$
$\lambda x \leq Pw.f(x,y)$

Prova della Prop 8

@) $\lambda b.M[b/a] = \{(n,m) \mid m \in M[b/a][e[n]/b]\}$
$= \{(n,m) \mid m \in M[e]/a]\}$
$= \lambda a.M$
dove il passaggio $M[b/a][e[n]/b]=M[e[n]/a]$ e' valido solo se $b \notin FV(M)$

ß) $(\lambda a.M)(N) = \{m \mid \exists e[n] \leq N.(n,m) \in \{(n,m) \mid m \in M[e[n]/a]\}\}$
$= \{m \mid \exists e[n] \leq N.m \in M[e[n]/a]\}\}$
$= \{m \mid m \in U\{M[e[n]/a] \mid e[n] \leq N\}\}$
[Lemma alla Prop 7]
$= U\{M[e[n]/a] \mid e[n] \leq N\}$ [Prop 7]
$= M[N/a]$

n°) Nel senso '=>' (n°) deriva da (ß).
Nell'altro senso se $\forall x.M[x/a]=N[x/a]$ allora M e N definiscono la stessa funzione. Basta quindi dimostrare che $\lambda a.M$ definisce la stessa funzione definita dalle variabili libere di M.
$fun(\lambda a.M)(x) =$
$\{m \mid \exists e[n] \leq x.(n,m) \in \{(n,m) \mid m \in M[e[n]/a]\}\} =$
$\{m \mid \exists e[n] \leq x.m \in Me[n]/a]\} =$
$\{m \mid m \in U\{M[e[n]/a] \mid e[n] \leq x\}\} =$
[Lemma alla Prop 7]
$U\{M[e[n]/a] \mid e[n] \leq x\} =$
$M[x/a]$

Prova della Prop 9

Si trova nel testo.

Prova della Prop 10

$Y(a) = (\lambda x.a(x(x)))(\lambda x.a(x(x))) = a(Y(a))$
e questo dimostra che Y(a) e' un punto fisso di a.
Sia adesso $d=\lambda x.a(x(x))$ e sia a° un punto fisso di a; dimostriamo che $d(d) \leq a^\circ$.
Supponiamo che per tutti gli $i<1$ valga:
$e[i] \leq d \Rightarrow e[i](e[i]) \leq a^\circ$ (essa vale per e°)
Sia allora $e[1] \leq d$ e $m \in e[1](e[1])$.
$m \in e[1](e[1]) \Rightarrow \exists n.(n,m) \in e[1] \ \& \ e[n] \leq e[1]$
Per le ipotesi di ben fondatezza sulle codifiche:
$n \leq (n,m) < 1$ con $e[n] \leq d$ per costruzione.
Dall'ipotesi di induzione si ricava che $e[n](e[n]) \leq a^\circ$.
Inoltre $(n,m) \in d$; essendo d una λ-astrazione si ha per

definizione che $m \in d(e[n])$. Per monotonicita'
$a(e[n](e[n])) \leq a(a^\circ) = a^\circ$, quindi:
$d(e[n]) = a(e[n](e[n])) \leq a^\circ$.
Abbiamo mostrato che per ogni l:
$e[l] \leq d \Rightarrow e[l](e[l]) \leq a$
Ma $d(d) = \bigcup\{e[l](e[l]) \mid e[l] \leq d\}$ e quindi $d(d) \leq a^\circ$.

~~Prova della Prop 11~~

Prova della Prop 12

Sia a una funzione continua e $A = \{x \mid x = a(x)\}$ l'insieme dei suoi punti fissi. A e' naturalmente ordinato da '$\leq$'. Mostriamo che ogni sottinsieme S di A ha un lub (least upper bound) in A.
Sia y la minima soluzione dell'equazione:

$$y = \bigcup\{x \mid x \in S\} \sqcup a(y) \qquad (1)$$

la cui esistenza e' garantita dalla Prop 6. Per costruzione $x \leq y$ per ogni $x \in S$ (cioe' y e' un upper bound di S0 e $x = a(x) \leq a(y)$.
Ma se $\forall x \in S. x \leq a(y)$ anche $\bigcup\{x \mid x \in S\} \leq a(y)$ e $\bigcup\{x \mid x \in S\} \sqcup a(y) = a(y)$. Cioe' $y = a(y)$ e $y \in A$.
Sia $z \in A$ un altro upper bound di S, cioe':
$\forall x \in S. x \leq z$. Allora z soddisfa l'eguazione (1).
Ma y e' per ipotesi il minimo valore che soddisfa (1) e quindi $y \leq z$.
Un insieme parzialmente ordinato con tutti i lubs ha anche tutti i glbs ed e' un lattice completo.

Prova della Prop 13

a) bool e' una iniezione.
bool$^\circ$bool=bool per le proprieta' del condizionale.
$\forall x \in Pw.\ x \leq bool(x)$
Infatti $\perp \leq bool(\perp) = \perp$
$0 \leq bool(0) = 0$
$n+1 \leq bool(n+1) = \tau - 1$
altrimenti $x \leq bool(x) = \tau$

b) num e' una iniezione.
num$^\circ$num=num per le proprieta' del condizionale.
$\forall x \in Pw.\ x \leq num(x)$
Infatti $x \leq num(\perp) = \perp$
$0 \leq num(0) = 0$
$n+1 \leq num(n+1) = n+1$
altrimenti $x \leq num(x) = \tau$

c) fun e' una iniezione.
$fun^\circ fun = \lambda w.(\lambda f\ x.f(x))((\lambda g\ y.g(y))(w))$
$= \lambda w.(\lambda f\ x.f(x))(\lambda y.w(y))$
$= \lambda w\ x.(\lambda y.w(y))(x)$
$= \lambda w\ x.w(x)$
$= fun$
$\forall a \in Pw.\ a \leq fun(a) = \lambda x.a(x)$
$\lambda x.a(x) = \{(n,m) \mid m \in a(e[n])\}$
$= \{(n,m) \mid m \in \{m \mid \exists e[k] \leq e[n].(k,m) \in a\}\}$
$= \{(n,m) \mid \exists e[k] \leq e[n].(k,m) \in a\}$
$\geq \{(n,m) \mid (n,m) \in a\}$
$= a$

Prova della Prop 14

$$
\begin{aligned}
&(a\&b)\circ(a\&b) = \\
&\quad \lambda w.\,(\lambda z.\langle a(left(z)),b(right(z))\rangle) \\
&\qquad (\langle a(left(w)),b(right(w))\rangle) = \\
&\quad \lambda w.\langle a(left(\langle a(left(w)),b(right(w))\rangle)), \\
&\qquad b(right(\langle a(left(w)),b(right(w))\rangle))\rangle = \\
&\quad \lambda w.\langle a(a(left(w))),b(b(right(w)))\rangle = \\
&\quad \lambda w.\langle a(left(w)),b(right(w))\rangle = \\
&\quad a\&b \\
&(a\&b)(x) = \langle a(left(x)),b(right(x))\rangle \\
&\qquad \geq \langle left(x),right(x)\rangle \\
&\qquad = x
\end{aligned}
$$

dove viene utilizzata la proprieta' delle coppie per cui $a' \leq a, b' \leq b \Rightarrow \langle a',b'\rangle \leq \langle a,b\rangle$ e perche' per ipotesi: $\forall v.\ v \leq a(v), v \leq b(v)$

Prova della Prop 15

$$
\begin{aligned}
&(a \oplus b)\circ(a \oplus b) = \\
&\quad (\lambda x.b\circ x\circ a)\circ(\lambda y.b\circ y\circ a) = \\
&\quad \lambda w.(\lambda x.b\circ x\circ a)((\lambda y.b\circ y\circ a)(w)) = \\
&\quad \lambda w.(\lambda x.b\circ x\circ a)(b\circ w\circ a) = \\
&\quad \lambda w.b\circ b\circ w\circ a\circ a = \\
&\quad \lambda w.b\circ w\circ a = \\
&\quad a \oplus b
\end{aligned}
$$

$x \leq a(x)$ per ipotesi

$v(x) \leq v(a(x))$

$v(a(x)) \leq b(v(a(x)))$ per ipotesi

$v(x) \leq b(v(a(x)))$

$v(x) \leq (a \oplus b)(u)(x)$

$\lambda x.v(x) \leq \lambda x.(a \oplus b)(v)(x)$

$\lambda x.v(x) \leq (a \oplus b)(v)$

$v \leq (a \oplus b)(v)$ per la Prop 13c $(v \leq \lambda x.v(x))$

Prova della Prop 16

Sia $c = a \oplus b$

$$
\begin{aligned}
&(c\circ c)(v) = \underline{let}\ x \leftarrow (\langle left(v):>0,0\rangle\ u\ \langle right(v):>1,1\rangle \rightarrow \\
&\qquad \langle a'(left(v)),\perp\rangle,\langle\perp,b'(right(v))\rangle \\
&\qquad \underline{in}\ (\langle left(x):>0,0\rangle\ u\ \langle right(x):>1,1\rangle \rightarrow \\
&\qquad \langle a'(left(x)),\perp\rangle,\langle\perp,b'(right(x))\rangle
\end{aligned}
$$

Per casi su v:

se $v = \langle\perp,\perp\rangle = \perp$; $(c\circ c)(v) = \perp = c(v)$

se $v = \langle p,\perp\rangle$ con $p \neq \perp$;
$(c\circ c)(v) = \langle a'(a'(p)),\perp\rangle = \langle a'(p),\perp\rangle, c(v)$

se $v = \langle\perp,q\rangle$ con $q \neq \perp$;
$(c\circ c)(v) = \langle\perp,b'(b'(q))\rangle = \langle\perp,b'(q)\rangle = c(v)$

se $v = \langle p,q\rangle$ con $p,q \neq \perp$; $(c\circ c)(v) = \top = c(v)$

Quindi $(c\circ c) = c$

$$
\begin{aligned}
&(a \oplus b)(v) = (\langle left(v):>0,0\rangle\ u\ \langle right(v):>1,1\rangle) \rightarrow \\
&\qquad \langle a'(left(v)),\perp\rangle,\langle\perp,b'(right(v))\rangle \\
&\qquad \geq (\langle left(v):>0,0\rangle\ u\ \langle right(v):>1,1\rangle) \rightarrow \\
&\qquad \langle left(v),\perp\rangle,\langle\perp,right(v)\rangle \\
&\qquad \geq (\langle left(v):>0,0\rangle\ u\ \langle right(v):>1,1\rangle) \rightarrow \\
&\qquad \langle left(v),\perp\rangle,\langle\perp,right(v)\rangle \\
&\qquad = \langle left(v),\perp\rangle\ u\ \langle\perp,right(v)\rangle \\
&\qquad = \langle left(v),right(v)\rangle
\end{aligned}
$$

= v per una proprieta' delle coppie

Prova della Prop 17

Sia $y^0=D(a)(x)$. y^0 e' il minimo y tale che $x \cup a(y) \leq y$.
Il minimo z tale che $y^0 \cup a(z) \leq z$ e' ancora y^0.
Questo prova che $D(a) \circ D(a) = D(a)$, cioe'
$\forall a \in Pw.\ D(a)$ e' una retrazione.
Incltre dalla definizione $D(a)(x)=N\{y \mid x \leq y \ \& \ a(y) \leq y\}$
si vede che $\forall a,x \in Pw.\ x \leq D(a)(x)$ e quindi
$\forall a \in Pw.\ D(a)$ e' una iniezione.
Allora se $a=D(a)$, a e' una iniezione e inversamente se a e' una iniezione, $D(a)=a$. I punti fissi di D formano l'insieme di tutte e sole le iniezioni.

$D(a)$ e' sempre una iniezione e $D(x)=x$ se x e' una iniezione. Ne discende che $D(D(a))=D(a)$, cioe':
$D \circ D = D$ D e' una retrazione.
Da $x \leq D(a)(x)$ per monotonicita' $a(x) \leq a(D(a)(x))$.
$D(a)(x) = \mu y.x \cup a(y) = x \cup a(\mu y.x \cup a(y)) \geq a(D(a)(x))$
da cui $a(x) \leq D(a)(x)$
$\lambda x.a(x) \leq \lambda x.D(a)(x)$
$\lambda x.a(x) \leq D(a)$
$a \leq D(a)$ per la Prop 13c $(a \leq \lambda x.a(x))$
Quindi $\forall a \in Pw.\ a \leq D(a)$ e D e' una iniezione.

Prova della Prop 18

$Yf = \cup\{f[n](\perp) \mid n \in w\}$
$Yf \circ Yf = \cup\{f[n](\perp) \mid n \in w\} \circ \cup\{f[n](\perp) \mid n \in w\}$
$= \cup\{f[n](\perp) \circ f[n](\perp) \mid n \in w\}$
$= \cup\{f[n](\perp) \mid n \in w\}$ perche' $\perp \circ \perp = \perp$ e $f: D \otimes D$
$= Yf$
Yf e' una retrazione.

$f: D \otimes D \Rightarrow f(\perp) = D(f(D(\perp))) = D(f(i))$
$f: D \otimes D \Rightarrow f: D \Rightarrow f(i) \geq i \Rightarrow D(f(i)) \geq D(i) = i$
$f(\perp) \geq i$
$f(\perp) \geq \perp \Rightarrow f[n](\perp) \geq f[n+1](\perp)$
$\cup\{f[n](\perp) \mid n \in w\} \geq i$
$Yf \geq i$
Yf e' una iniezione.

Prova della Prop 19

Discende dalla Def 14 e dalla Prop 14.

Prova della Prop 20

Discende dalla Def 15 e dalla Prop 16

Prova della Prop 21

$a^*(x) = \langle a(left(x)), a^*(right(x)) \rangle$
$(a^* \circ a^*)(x) = \langle a(a(left(x)), a^*(a^*(right(x))) \rangle$
$= \langle a(left(x)), a^*(a^*(right(x))) \rangle$
$= \langle a(left(x)), a^*(right(x)) \rangle$ per induzione

$= a^*(x)$

Cioe' $a^* = a^* \circ a^*$ e a^* e' una retrazione.

$a^*(x) = \langle a(left(x)), a^*(right(x))\rangle$
$\geq \langle left(x), a^*(right(x))\rangle$
$\geq \langle left(x), right(x)\rangle$ per induzione
$= x$

Cioe' $x \leq a^*(x)$ e a^* e' una iniezione.

Prova della Prop 22

Si dimostra per induzione che $set(a) \circ set(a) = set(a)$
Per dimostrare $x \leq set(a)(x)$ basta mostrare che
$set(a)(x) = x$ se x e' nella lista a, e τ altrimenti
(grazie alla definizione di lista).