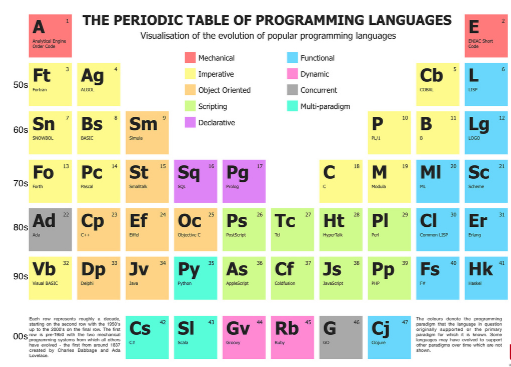
Riassunto di Fondamenti di linguaggi di programmazione

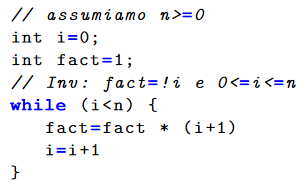
Dagli anni 50 ai 200 sono nati molti linguaggi aventi differenti paradigmi, da quelli imperativi, a quelli object oriented, a quelli di scripting, eccetera, la tendenza attuale è quella dei linguaggi multiparadigma come ad esempio C#.



Un paradigma di programmazione indica come viene descritta la computazione, esso può essere:

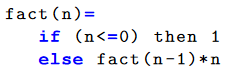
| Linguaggio | Computazione | Programma | Operazione Fondamentale |
| --- | --- | --- | --- |
| Imperativo | cambiamento di stato determinato dall’esecuzione dei comandi | insieme di istruzioni da eseguire | assegnazione di un valore a una variabile |
| Funzionale | valutazione di espressioni | insieme di espressioni che devono essere valutate | chiamata di una funzione con relativi argomenti |
| Object oriented | scambio di messaggi tra oggetti | insieme di oggetti aventi stato e metodi che lo modificano | invio di un messaggio |
| Logico | ricerca di dimostrazione di asserzioni | assiomi e regole che derivano asserzioni | applicazione di una regola |
| Event-driven | esecuzione asincrona di manipolatori di eventi | non esiste |  |

Esempio di paradigma imperativo:



Nel paradigma imperativo si utilizzano variabili per mantenere i valori intermedi di ogni iterazione, tutti gli assegnamenti devono essere effettuati nel modo consono al fine di ottenere il risultato corretto.

Esempio di paradigma funzionale:



Nel paradigma funzionale si utilizza la ricorsione per il calcolo dei risultato intermedi, in questo caso la funzione itera finchè n è diverso da 0.

Esempio di paradigma object oriented:



Facendo 1 fact, self riceve il messaggio fact, if True è un messaggio della classe Boolean avente un parametro, il blocco [^ 1], esso ritorna l’oggeto dopo ^ quando viene valutato.

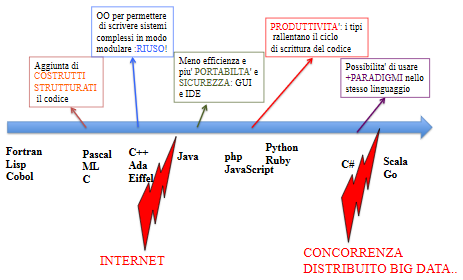
Se l’oggetto false riceve ifTrue, non fa nulla e passa alla riga successiva, se invece a riceverlo è true allora valuta il blocco e ritorna il suo valore.

Esempio di paradigma dichiarativo:







fact è un predicato binario in cui fact(x,y) indica se x è diverso da y, s indica il successore di un numero dato mentre prod e sum sono predicati in cui prod(x,y,z) e sum(x,y,z) indicano rispettivamente se x\*y=z e x+y=z

# Induzione

Dato che ogni linguaggio ha una grammatica , esse è una definizione induttiva di termini.

Esempio: Il principio di induzione ordinaria nei numeri naturali indica che, supponendo che P sia un predicato sui numeri naturali, se vale P(0) e per ogni i vale P(i), allora vale anche per P(i+1), quindi P vale per tutti i numeri naturali.

Esempio: la somma di tutte le potenze di 2 da 0 a n equivale a 2^(n+1)-1 per ogni n>=0

Caso base: n=0

2^(0+1) -1 =2 -1=1

Caso induttivo: n>=0

dato i in N tale che i<=n

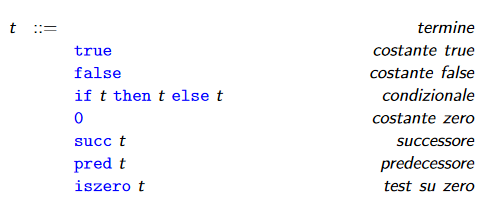
sum(j=0,i) [ 2^j ] = 2^ (i+1) -1

per ipotesi induttiva si sa che sum(j=0,i)[ 2^j ] = 2^(i+1) -1, quindi:

dato k=i+1

sum(j=0,k)[2^k]= sum(j=0,k)[2^i] + 2^k =2^(i+1)-1 + 2^(i+1) = 2^(i+2)-1

## Sintassi delle semplici espressioni aritmetiche



Data la grammatica per descrivere i termini aritmetici, con essa si definisce un insieme di stringhe che la rispettano ma allo stesso tempo vi è un insieme di alberi di sintassi, queste grammatiche sono infatti astratte dal momento che non serve effettuare il parsing, di conseguenza potrebbero esserci ambiguità ma non fa nulla.

### Definizione più esplicita

L’insieme dei termini è il più piccolo insieme T tale che:

* {true,false,0} è contenuto in T;
* se t è un elemento di T, allora {succ t, pred t, iszero t} è contenuto in T;
* se t1,t2,t3 appartengono a T, allora if t1 then t2 else t3 appartiene a T.

Entrambi i metodi sono utili dal momento che la grammatica rende la definizione più compatta e facile da leggere (definizione dall’alto verso il basso) mentre quella esplicita dà un principio di induzioni per provare proprietà di termini (definizione dal basso verso l’alto).

Da questo, è possible fare induzione sui termini, prima facendolo sui casi base true, false e 0 e dopo su succ e tutti gli altri costrutti.

La caratterizzazione induttiva dei termini di un linguaggio permette di provare le proprietà in modo induttivo e allo stesso modo definire delle funzione.

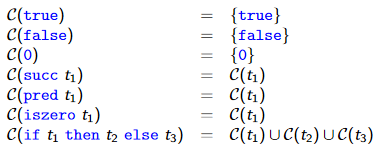
### Principio di induzione strutturale dei termini

Si suppone che P sia un predicato sui termini, se dall’ipotesi che per ogni termine t’, P valga per tutti i sottotermini immediati di t’ si può dimostrare che P(t’), allora P vale per tutti i termini tT.

#### Dimostrazione per le espressioni aritmetiche

1. Caso base: si dimostra l principio per P(0), P(true), P(false), essi non hanno sottotermini e quindi non si può utilizzare il passo induttivo;
2. Caso induttivo: si dimostra la proprietà per P(succ t), P(pred t) e P(iszero t) con l’ipotesi che valga P(t);
3. Caso induttivo: si dimostra la proprietà per P(if t1 then t2 else t3) con l’ipotesi che valgano P(t1), P(t2) e P(t3);
4. Da questo si deriva che P(t) vale per tutte le espressioni aritmetiche t.

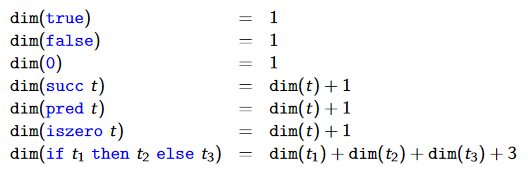
Si definisce ora la funzione C(t) come l’insieme delle costanti (true, false e/o 0) che compaiono in un termine t:



Questa è una funzione induttiva e la sua definizione è molto simile al principio scritto sopra, nel caso delle espressioni aritmetiche:

1. si dà un valore della funzione per true, false, 0;
2. si dà una valora delle funzione a succ t, pred t e iszero t utilizzando f(t1) nella definizione;
3. si dà un valore della funzione a if( t1 then t2 else t3 utilizizzando f(t1), f(t2) e f(t3) nella definizione.

Si definisce ora una funzione dim che indica la dimensione di un termine t:



#### Teorema

Il numero delle costanti distinte in un termine è al più la dimensione deltermine. Cioè per ogni tinT vale |C(t)|≤dim(t).

#### Dimostrazione

Casi base:

|C(true|=dim(true)=1

Stessa cosa vale per false e 0;

Casi induttivi:

dim(succ t)=dim(t)+1

C(succ)=C(t)

per ipotesi induttiva: |C(t)|<=dim(t)

quindi |C(succ t)|=|C(t)|<=dim(t)<dim(succ t)

Lo stesso discorso vale anche per pred e iszero.

Nel caso dell’if invece:

C(if…)=C(t1) U C(t2) U C(t3)

dim(if…)=dim(t1)+dim(t2)+dim(t3)+3

Per ipotesi induttiva: |C(if…)|<=|C(t1)|+|C(t2)|+|C(t3)|

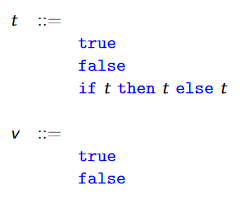
quindi: |C(if…)|<=||C(t1)|+|C(t2)|+|C(t3)|<=dim(t1)+dim(t2)+dim(t3)<dim(if…)

# Macchine astratte

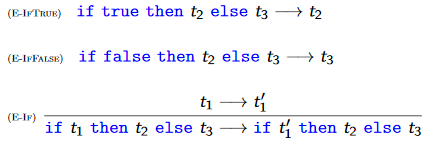
Una macchia astratta è un insieme di sati S aventi relazioni di transizione, ogni stato memorizza tutte le informazioni della macchina in un dato momento.

Nel caso della programmazione funzionale, essa è basata sull espressioni e sulla loro valutazione, quindi si può dire che lo stato è il termine da valutare e come questo viene modificato.

Esempio: Espressioni booleane



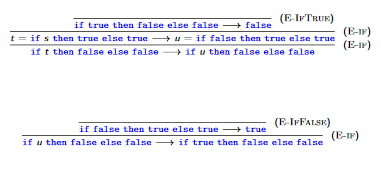
Data la grammatica, la regola di valutazione t→ t’ è la più piccola relazione chiusa per:



Nell’ultima regola si può notare che t1 viene ridotto in t1’ al fine di valutare il risultato, la regola che lo applica è detta regola di congruenza e indica dove applicare la computazione, tutte le altre che sono “su una riga” vengono dette regole di computazione, i passi computazionali in cui parte del termine viene consumato.

# Derivazioni

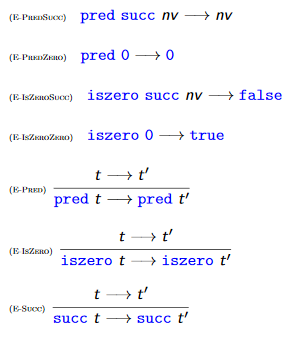
Si può giustificare il fatto che una coppia di termini sia nella relazione di valutazione come la derivazione delle formule logiche, per farlo si mette sopra la riga la sotto derivazione da cui dipendono:

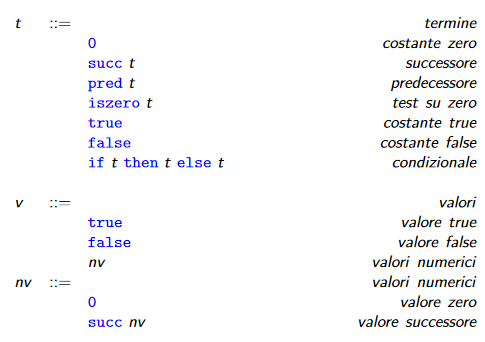


Supponendo di avere un albero di derivazione D avente t→ t’ come conclusione, vi sono tre possibili casi:

* E-true è la regola finale utilizzata e t=if true then t2 else t3 dove t2 è t’;
* E-false è la regola finale utilizzata e t=if true then t2 else t3 dove t3 è t’;
* E-If è la regola finale utilizzata e t= if t1 then t2 else t3, t’=if t1’ then t2 else t3 e t1-->A t1’ è l’immediata sottoderivazione di D.

# Semantica operazionale per le espressioni aritmentiche

Data la grammatica, si può osservare che essa è un’estensione di quella delle espressioni booleane.



## Forma normale

Un termine t è in forma normale quando non può essere più valutato ulteriormente, quindi non esiste un t’ tale che t→ t’.

Ogni valore è una forma normale, tuttavia non è detto il contrario in quanto esistono i termini bloccati, una forma normale che non è appunto un valore, essi permettono di modellare gli errori a runtime.

Nel caso delle espressioni booleane, ogni forma normale è un valore, in quelle aritmentiche invece no dato che termini come “succ true” non possono essere ridotti.

# Introduzione al lambda-calcolo

Il lambda-calcolo è un linguaggio di programmazione semplice avente le seguenti caratteristiche:

* é Turing-completo, ovvero ogni possibile algoritmo è codificabile in questo modo;
* è di ordine superiore, vi sono quindi funzioni che possono restituire funzioni;
* presenta legami tra variabili e scoping statico.

Si suppone di avere piu3 x=succ(succ(succ x))), quindi piu3 è una funzione che produce succ(succ(succ x))) quando applicata a x, quindi:

piu3= lambda(x).succ(succ(succ x))

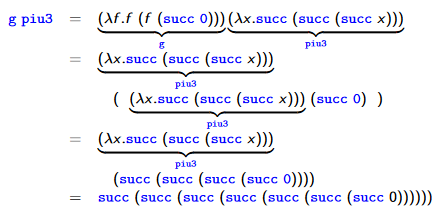
se x ad esempio vale succ 0:

piu3 (succ 0) = lambda(x).succ(succ(succ x)) (succ 0) = succ(succ(succ(succ 0)))

Lambda(x.t è una primitiva sintattica indicante un’astrazione di un termine t per un sottotermine x e indica che nel termine t ogni occorrenza di x viene sostituita con un valore.

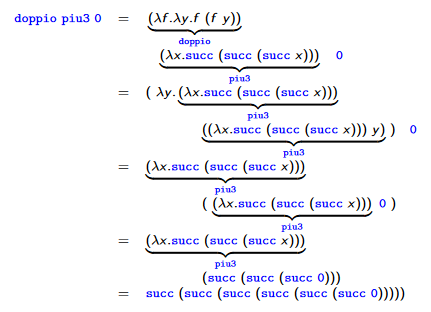
t1 t2 è invece una primitiva indicante l’applicazione di una funzione(t1) ad un argomento (t2)

Esempio: dato g=lambda(f).f (f (succ 0)), cosa si ottiene se si fa g piu3?



Si può notare che f è utilizzato come una funzione nel corpo di g, quindi g è una funzione di ordine superiore.

Si considera ora doppio=lambda(f).lambda(y).f (f y), quindi:



## Lambda-calcolo puro

Nel lambda-calcolo puro, tutto è una funzione, quindi:

* le variabili denotano funzioni;
* le funzioni prendono altre funzioni come parametri;
* il risultato di una funzione è sempre una funzione.

I termini del lambda-calcolo puro sono detti lambda-termini, quelli nella forma lambda(x).t sono invece detti lambda-astrazioni o semplicemente astrazioni.

Vi sono inoltre delle convenzioni sintattiche da rispettare in modo da evitare di scrivere le parentesi tonde:

* L’applicazione di una funzione è associativa a sinistra:

t1 t2 t3 = ((t1 t2) t3)

* il corpo di un’astrazione estende il più possibile a destra:
* lambda(x).lambda(y).y x = lambda(x).(lambda(y).(y x))

Inoltre nel lambda calcolo puro le funzioni sono tutte di un argomento, tutte le altre vengono scritte nella cosiddetta forma currificata.

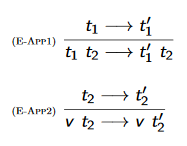
le astrazioni nella forma lambda(x).t sono legate alla variabile x in cui lo scope è t, quindi tutte le occorrenze di x in t sono dette legate dall’alstrazione.

Le variabili nel body t che non sono legate da scope sono dette libere.

La regola di computazione indica che ogni termine x in t viene sostituito con v.

(lambda(x).t) v → [x→ v]t

Regole di congruenza:



## Strategie di riduzione

Quando vi è un’astrazione applicata a un valore, essa prende il nome di redesso, ovvero un termine che si può ridurre in un altro termine più piccolo.

Un insieme di regole è deterministico se dato un termine t, se t è riducibile a t’, allora vi è una sola regola applicabile.

Un insieme di regole è confluente se, dati t, t1 e t2 con t che può ridurre a t1 o t2, allora esiste un termine t3 tale che sia raggiungibile sia da t1, sia da t2.

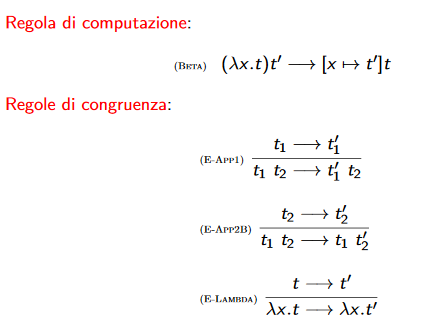
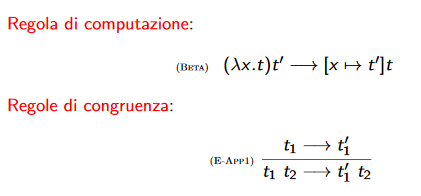
Le semantiche possono essere ragionevoli anche quando non deterministiche, non è così invece se non sono confluenti.

La strategia di riduzione utilizzata nei precedenti esempi viene detta chiamata per valore e riduce prima gli argomenti in valori e poi li applica alla funzione, ciò riflette la convenzione di molti linguaggi di programmazione.

Altre strategie sono:

* la beta-riduzione: si riduce ogni redesso riducibile;
* la chiamata per nome: si riduce prima il redesso più esterno e più a sinistra, ciò permette di non ridurre le espressioni in valori.

### Regole per beta-riduzione e chiamata per nome



### Currificazione

La currificazione è un metodo utile per convertire una funzione con più argomenti in una equivalente che prende un singolo argomento:

lambda(x,y).t =lambda(x).lambda(y).t

Quindi, dato un valore v per x, viene prodotta una funzione che, dato un valore u per y, produce t con v al posto di x e u al posto di y.

## Booleani nel lambda-calcolo

i booleani nel lambda-calcolo si rappresentano con due funzioni di due argomenti:

tr = lambda(x).lambda(y).x fls=lambda(x).lambda(y).y

Entrambe le funzioni hanno il compito di restituite il primo o il secondo argomento, quindi:

tr a b → a fls a b → b

#### Operazioni coi booleani:

* Negazione: è una funzione che, in base al booleano in input, deve restituire il suo inverso:

not=lambda(b) fls tr

* And logico: è una funzione che prende due booleani in input e restituisce true se sono entrambi veri, altrimenti è false, quindi:

and=lambda(b1).lambda(b2).b1 b2 fls

* Or logico: è una funzione simile all’and, la differenza è che viene restituito false solo se entrambi gli argomenti lo sono, quindi:

or=lambda(b1).lambda(b2).b1 tr b2

## Numeri naturali nel lambda-calcolo

Per quanto riguarda i numeri naturali, l’idea è quella di rappresentarli come una funzione che ripete tante volte un’azione su un valore iniziale, quindi:

c0= lambda(s).lambda(z).z

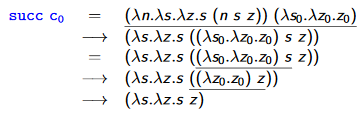
c1= lambda(s).lambda(z).s z

cn= lambda(s).lambda(z).s (s(... (s z)))

### Operazioni sui naturali:

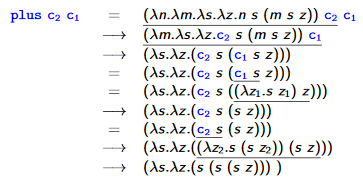
* il successore è una funzione che, dato un numero, restituisce un altro numero che applica l’azione s al risultato dell’applicazione (n s z):

succ= lambda(n).lambda(s).lambda(z).s (n s z)



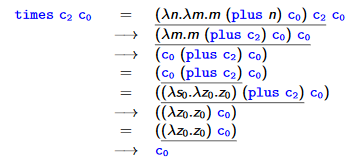
* La somma di due naturali n e m è molto simile al successore, quel che si fa infatti è applicare i numerali di Church uno in cascata all’altro:

plus= lambda(n).lambda(m).lambda(s).lambda(z).n s (m s z)



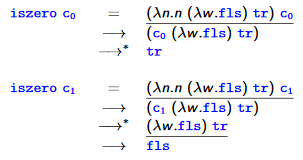
* La moltiplicazione di due naturali n e m applica n volte la somma su m partendo con un valore iniziale uguale a 0:

times= lambda(n).lambda(m).m (plus n) c0



* Il test su zero è una funzione che riestituisce false se l’argomento non è zero, altrimenti restituisce true:

iszero= lambda(n).n (lambda(w).fls) tr



## Variabili libere e lambda-termini chiusi

Dal momento che un programma per essere eseguito deve contenere tutta l’informazione, nella programmazione funzionale non devono esserci variabili libere.

### Teorema

Se un termine t è chiuso, allora è un valore oppure può essere ridotto in t’

### Dimostrazione

Casi base:

x è chiuso e non può essere ridotto, quindi è un valore stessa cosa vale per lambda(x).t

Caso Induttivo: DA COMPLETARE

## Forme normali

Come già detto un forma normale è un termine che non può più essere ridotto mentre un termine bloccato è una forma normale ma non è un valore.

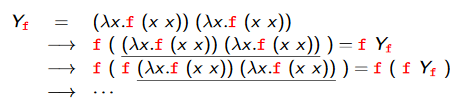
Nel lambda-calcolo puro non vi sono termini bloccati ma non tutto viene ridotto a una forma normale

### Divergenza

Si considera w=( (lambda(x).x x) (lambda(x).x x) ), si può notare che la valutazione di w non riduce mai a una forma normale, ciò significa che w diverge. Scrivere computazioni che divergono non è molto utile, ci sono però varianti di w che lo sono.

## Applicazione iterata

Dato il seguente termine, si suppone che f sia un’astrazione:



Si può notare che il pattern di divergenza, anche se non ancora realmente utile, diverge con la chiamata per valore cercando di valutare l’argomento.

Con la chiamata per nome, le cose cambiano:

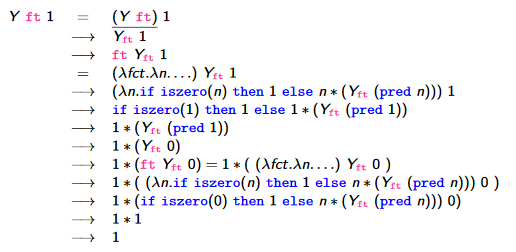
si considera il fattoriale di un numero n, si definisce ft che prende la funzione f da iterare e il parametro n:

ft= lambda(ftc).lambda(n).if iszero(b) then 1 else n\* (ftc (pred n))

ft è il fattore ordinario, la chiamata ricorsiva invece avviene su fct passando il predecessore di n come parametro.

ft ha come input una funzione da int a int e ritorna un’altra funzione fatta allo stesso modo, quindi è di ordine superiore

Esempio: data Y=lambda(f).Yf, per otterere una funzione fattoriale ricorsiva, si procede come segue:



## Associazioni tra nomi e valori

L’associazione tra nomi e valori è data dal costrutto let, let x = 15\*10 assegna il valore 150 alla variabile x, quindi l’espressione alla destra dell’uguale viene valutata come il valore mentre a sinistra come il nome.

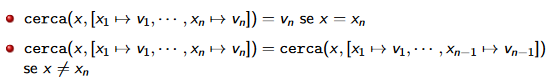
Per la definizione di funzioni ricorsive vi è il costrutto letrec:

letrec f = espressione in cui compare f

traducibile come let f = Y (lambda(f).lambda(x).espressione in cui compare f)

# Lambda-calcolo con stack

Nei linguaggi imperativi i record di attivazione sono gestiti da uno stack, esso è definito come un’associazione tra variabili e valori, la ricerca è definita nel seguente modo:



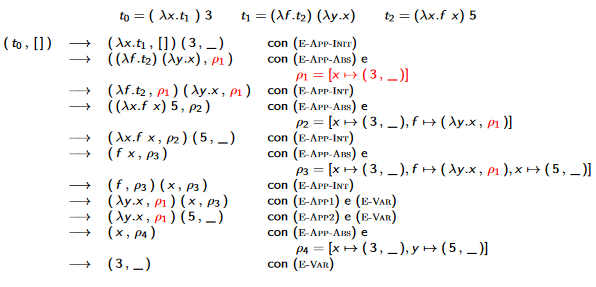
L’inserimento di un’associazione x→ v avviene mettendolo in cima allo stack.

Esempio: ( lambda(x).(lambda(f).(lambda(x).f x) 5) (lambda(y).x)) 3

Risolvendo tutto con la sostituzione, si ottiene 3 come risultato, tuttavia con lo stack si ottiene un risultato differente a causa dello stack condiviso ( permettendo il binding dinamico).

Per risolvere questo problema si utilizza il binding statico attraverso lo stack esplicito: ogni termine ha un proprio stack che viene ereditato dai suoi precedenti mentre le astrazioni, invece di associare solamente il valore, associa anche lo stack.

L’esempio sopra viene quindi svolto come segue:



Si può quindi dimostrare che la semantica per sostituzione è equivalente a quella con stack esplicito e binding statico: dato t → \* vp per sostituzione:

* (t,[]) → \* vp’,p con stack esplicito e binding statico implica che vp=vp’;
* (t,[]) → \* vp’,p con stack esplicito e binding dinamico non implica che vp=vp’;

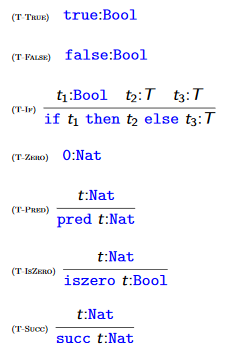
# Sistema dei tipi per espressioni aritmetiche e booleane

Definire un sistema di tipi significa definire una classificazione dei valori in base alla loro forma, precisamente si definisce una relazione di tipaggio t:T dove t è un termine e T il suo tipo, essa classifica i termini in base alla forma dei valori in cui vengono ridotti.

Per dimostrare che la relazione di tipaggio è corretta, essa deve rispettare la semantica operazionale:

* se t:T e t→ \* v, allora v:T;
* se t:T allora la valutazione non è bloccata.

## Regole



I tipi, come tutte le analisi statiche, non sono precisi dal momento che non predicono esattamente il tipo del valore restituito ma ne danno solo una stima.

Infatti se si considera la regola T-IF, non è possibile assegnare 0 a t2 e false a t3 anche se il risultato sarà un numero.

## Proprietà

La correttezza del sistema di tipi è determinata dalle seguenti proprietà:

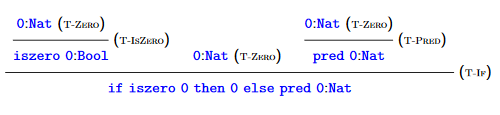
* Conservazione del tipo: i tipi vengono conservati nelle relazioni di riduzione, quindi:

se t:T e t→ t’, allora t’:T.

* Progresso: un termine ben tipato non riduce a un termine bloccato, quindi:

se t:T, allora t è un valore oppure può essere ridotto a t’.

Ogni coppia t:T nella relazione può essere giustificata da un albero di derivazione costruito da istanze delle regole di inferenza, ad esempio:



## Forme canoniche

Una forma è detta canonica quando per ogni tipo vi possono essere solo valori specifici, se un valore è booleano può essere true oppure false mentre se è un naturale è un valore numerico.

## Dimostrazione di conservazione del tipo

## Dimostrazione del progresso

Dal momento che le espressioni ben tipate non possono essere termini bloccati, con queste espressioni non possono esservi errori a runtime.

## Tipi nel lambda-calcolo

Per introdurre una sistema di tipi semplice nel lambda-calcolo, si può dire che i valori possono essere:

* Booleani (Bool);
* Naturali (Nat);
* Funzioni che prendono un argomento di tipo T1 e restituiscono un valore di tipo T2 (non è necessario che t1 e t2 siano uguali);

Dal momento che è necessario sapere il tipo di una variabile quando ne viene incontrata una, si introducono i contesti (Γ), ovvero sequenze di coppie x:T indicanti il tipo di ogni valore incontrato.

Si assume che nel contesto vi sia per ogni valore x al massimo un’occorrenza x:T e, quando viene scritto Γ,x:T, viene rimossa da Γ un evnetuale x:T’.

## Inferenza e ricostruzione di Tipo

A volte non è necessario specificare tutte le informazioni sul tipo, queste infatti possono essere dedotte in base a come vengono usati i termini, quindi il tipo può essere inferito.

L’inferenza di tipi è definibile come una funzione da lambda-termini tipati a lambda-termini senza tipi:

* inference(x)=x;
* inference(lambda(x):T.t)= lambda(x).t;
* inference(t1 t2)=inference(t1) inference(t2).

Un lambda-termine t senza tipi è tipabile quando esiste un lambda termine t’ tipato e un contesto Γ tale che inference(t’)=t e Γ⊦t’:T per qualche T.

Il passaggio che va da t a t’ è un processo detto inferenza di tipo ed è un procedimento effettuato da vari interpreti e compilatori almeno in parte.

## Tipi nei linguaggi di programmazione

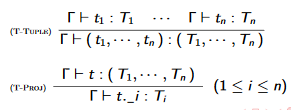
### Tipo Unit

Unit è il tipo restituito da funzioni di cui interessa solamente il suo effetto laterale e non cosa viene restituito, come ad esempio l’istruzione di stampa. il tipo Unit ha un solo valore che è unit o ().

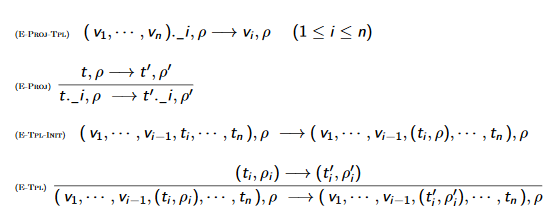
### Tipo Tuple

Il tipo Tuple indica insiemi di valori i quali possono avere differenti tipi, esse sono immutabili una volta definite.

#### Regole di tipo



#### Regole di valutazione



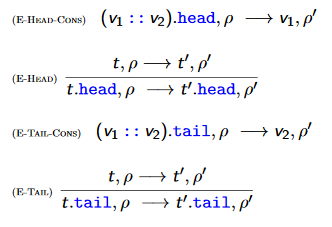
### Tipo List

Il tipo List è molto simile al tipo Tuple con la differenza che questo è mutabile e i valori all’interno hanno tutti lo stesso tipo. La lista vuota si indica con Nil, altrimenti si indica con t::t.

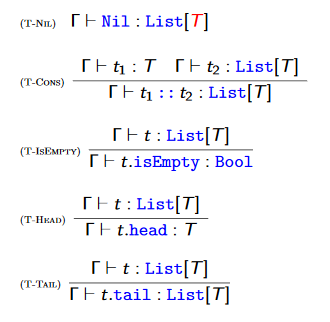
Sulle liste si possono fare le seguenti operazioni:

* isEmpty: indica se la lista è vuota o meno restituendo un booleano;
* head: restituisce l’elemento in testa alla lista;
* tail: restituisce la coda della lista.

#### Regole di valutazione



#### Regole di tipo



### Dichiarazione di variabili

La dichiarazione di variabili immutabili avviene utilizzando la keyword val seguita dal nome della variabile, dal tipo e infine dal termine da assegnare:

val x:T=t;ts

Per quanto riguarda le variabili mutabili, la dichiarazioni si fa allo stesso modo con la differenza che c’è var al posto di val.

#### Regole di tipo e di valutazione





### Blocchi

I blocchi permettono di considerare una sequenza di istruzioni come una sola, quindi confinando il binding di eventuali dichiarazioni in una sola porzione di codice.

La sequenza di istruzioni in un blocco viene indicata tra le parentesi graffe.

#### Regole di valutazione e di tipo



### Sequenzializzazione e blocco come forme derivate

Per quanto riguarda la sequenzializzazione t;ts, per scriverla a partire da un’astrazione e un applicazione bisogna definire una traduzione:

translate(t;ts)= ( lambda(x):T.ts) t dove x non è libera in ts

Il tipo T deve essere il tipo di t, in Scala è comunque possibile assegnare il tipo Any a ogni termine.

Per rendere le due rappresentazioni equivalenti, si definiscono:

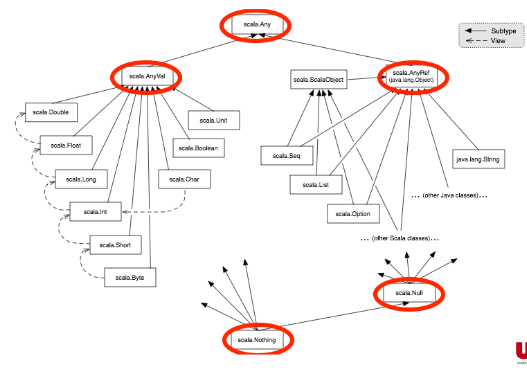
* un linguaggio esterno, ovvero il lambda-calcolo tipato semplice, per la sequenzializzazione aventi regole E-APP1, E-APP2, E-APPABS, E-SEQ-INIT, E-SEQ, E-SEQ-NEXT, T-VAR, T-APP, T-AUS, T-SEQ;
* un linguaggio interno, ovvero un lambda-calcolo tipato semplice aventi regole E-APP1, E-APP2, E-APPABS, T-VAR, T-APP,T-ABS;
* una funzione translate che converte da linguaggio esterno a interno.

Anche il blocco può essere considerato come una forma derivata, in tal caso la funzione translate è la seguente:

translate({ts})) (lambda(x):Unit.ts) unit

## Sistema di sottotipi

Dal momento che le regole di tipo sono troppo rigide nell’applicazione di una funzione (l’argomento deve avere lo stesso tipo del parametro) o nell’associazione di un valore a una variabile, è opportuno definire la nozione di sottotipo: se un tipo T è un sottotipo di T’ (T<=T’), allora si può utilizzare T in ogni contesto in cui apparte T’.



La relazione è:

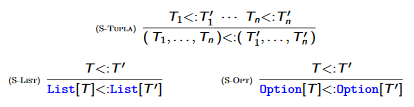
* riflessiva: T<:T;
* transitiva: se T1<T2 e T2<T3, allora T1<T3;
* antisimmetrica: se T1<T2 e T2<T1, allora T1=T2.

Se T e T’ sono tipi base, allora T<T’ se T è discendente di T’ nella gerarchia di classi Scala, in quest’ultima si può dire che:

* Nothing<T;
* T<Any.

Per quanto riguarda i tipi parametrici:

* Una tupla è minore di un altra se tutti i tipi della prima sono minori di quelli della seconda;
* List[T]<List[T’] se T<T’;
* una funzione f:T1->T2 è minore di g:T1’->T2’ se T1’<T1 (controvarianza sull’argomento) e T2<T2’ (covarianza sul risultato);



# Programmazione imperativa in Scala

Nei linguaggi imperativi le variabili possono cambiare, quindi per esse è necessario:

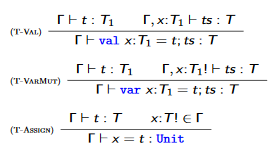
* un nome per un valore calcolato in precedenza;
* la possibile riscrittura di questo valore con un altro utilizzando lo stesso nome come riferimento.

Nei linguaggi funzionali queste caratteristiche sono separato:

* il legame tra variabile e valore è infatti immutabile;
* esistono però certi tipi di valori, detti riferimenti, i quali permettono la modifica del valore che essi mantengono.

La dichiarazione di una variabile modificabile nel linguaggio Scala viene fatta con la keyword var seguita dal nome e dal valore da assegnare.

L’assegnazione restituisce un valore Unit e non presenta un deferenziamento esplicito: ogni vaariabile in un contesto restituisce il valore che esso contiene.

Per differenziare le variabili mutabili da quella immutabili, si modificano le regole di valutazione nel seguente modo:

* il tipo delle variabili mutabili sarà seguito da !;
* nel caso delle variabili immutabili, invece, il tipo non è seguito da niente;
* il carattere ? indica che il costrutto vale sia per mutabili, sia per immutabili.

Questi costrutti sono utilizzabili nel modello con stack esplicito, tuttavia nelle variabili immutabili vi è un problema:

var y=3;

y=y+1;

non può essere equivalente a:

var y=3;

3=3+1;

Per permettere la modifica di y, si utilizza la memoria, un’insieme di associazioni rappresentate come un mapping tra locazioni e valore:

u=[l1:=v1,...,ln:=vn]

Nello stack le variabili avranno associata la locazione anzichè direttamente il valore, ciò permette quindi la modifica di un valore su una locazione per effetto dell'assegnazione.

L’associazione tra variabile e locazione invece rimarrà invariata.

La memoria è comune a tutti e non va distribuita come nel caso dello stack.

La valutazione di un termine ora dipende da stack e memoria, per farlo bisogna mantenere traccia di tutti i cambiamenti fatti su quest’ultima.

La relazione di valutazione sarà quindi la seguente:

(t,p) | u → (t’,p’) | u’ dove p è lo stack e u la memoria.

## Valutazione delle dichiarazioni con var

Il termine a destra dell’uguale deve essere valutato con lo stesso stack con cui si valuta il termine:





Una volta che il termine t a destra dell’uguale è stato ridotto a un valore v, viene creata una nuova locazione di memoria l, l’associazione x→ l viene inserita nello stack mentre l:=v viene inserito nella memoria:





L’assegnamente non ha regole tanto differenti rispetto a quelle della dichiarazione:



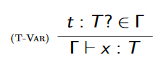




Per consentire la modifica dei valori assegnati alle variabili, bisogna avere una regola che tenga conto del fatto che le variabili non sono nello stack ma in memoria:



Per quanto riguarda il tipo invece, non è necessario sapere se la variabile è mutabile o meno:



## Correttezza del tipo

Per verificare la correttezza di un termine, si può dire che se t è ben tipato per un tipo T, la sua valutazione non è bloccata, essa avviene in un contesto in cui stack e memoria non possono essere arbitrari.

# Valutazione lazy

La valutazione lazy è un tipo di valutazione che utilizza la chiamata per nome, riducendo quindi il redesso più a sinistra prima degli argomenti.

L’astrazione di un parametro per nome è un termine e si indica con (x:=>T)=>t, un valore funzione invece si indica con ((x:=>T)=>t,p).

Per quanto riguarda il tipo, si utilizza la seguente regola:



Le regole per la semantica operazionale non sono tanto differenti da quelle già utilizzate, la differenza è che è possibile inserire un intero termine nello stack anzichè ridurlo a un valore:





Data una funzione f con un solo parametro:

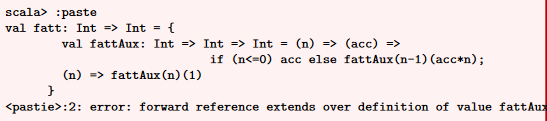
* è stretta se il parametro p è per valore e, se la valutazione di p diverge, allora diverge anche f(p);
* è lazy se il parametro p è per nome e, se la valutazione di p diverge, f(p) non necessariamente diverge.

Operatori come && e || sono lazy in quanto non necessariamente valutano tutti gli argomenti.

La chiamata per nome può essere simulata da una chiamata per valore utilizzando le funzioni, si considera la funzione f:=>T=>T’:

* si definisce f’:(()=>T)=>T’ sostituendo nel body di f ogni occorrenza del primo parametro x con x();
* ogni chiamata f(p) viene sostituita da f’(()=>p), in questo modo si ottiene un termine equivalente.

La valutazione lazy permette inoltre i cosiddetti riferimenti forward, ovvero quando una variabile/funzione viene utilizzata in un momento antecedente alla sua dichiarazione.



## Dichiarazione lazy

La dichiarazione lazy viene fatta con lazy val, questa definizione indica che il controllo di tipo include nel contesto di valutazione anche il nome della variabile.

E’ anche possibile dichiarare in sequenza più valori lazy, in tal caso si inseriscono tutte nello stack e nella memoria così come sono, la valutazione della parte destra avverrà quando le variabili verranno effettivamente utilizzate.

### Regole:





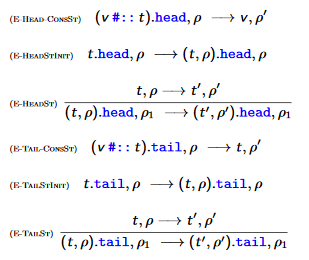
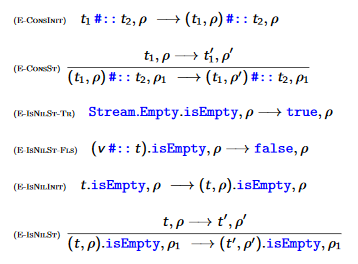
## Strutture dati lazy

### Stream

Una stream è una struttura dati lazy costituita da:

* un head, indicante un valore;
* una tail, ovvero in generatore di valori.

#### Regole di valutazione e di tipo:



Questa struttura dati è deprecata dato che non completamente lazt, esse sono state sostituite dalle lazylist.

### LazyList

LazyList p una struttura dati che sostituisce Stream, rispetto a quest’ultima è completamente lazy, quindi il primo elemento viene computato solamente quando necessario.