0. Introduzione

Un **formalismo** è una descrizione matematica rigorosa su un ragionamento, dunque può essere in diversi modi: espressioni algebriche, grafici, o altro.
Un **formalismo di calcolo** risponde alla domanda su cosa voglia dire "computare".

Alcuni esempi di questi ultimi sono la macchina di Turing, il λ calcolo, un linguaggio di programmazione. Vi sono diversi causa la **tesi di Church-Turing** la quale è una congettura: *Ogni funzione calcolabile da un formalismo di calcolo sufficientemente espressivo è calcolabile da una macchina di Turing e viceversa*, dunque ogni formalismo può essere calcolato usando la macchina di Turing. Tutti sono equivalenti. Tutti i formalismi non possono calcolare *tutti i problemi matematici*. Si sceglie il formalismo in base ai pro/cons: ad esempio la scelta di un linguaggio è dato da quanto è veloce a compilare o dalla semplicità nella sintassi.

Macchina di Turing

Come si calcola dal punto di vista meccanico? Si usa un supporto fisico per mantenere i dati, fatto da celle discrete (eg: cella della lavagna) contenenti finite informazioni. Il supporto fisico è infinito perché, sennò, avremmo un numero finito di combinazioni di dati e non risolvere ogni problema possibile. In probabilità usiamo il concetto di input che può essere infinito, anche se realmente non è nel calcolatore.
Nel nastro vi è una testina che legge la cella muovendosi in essa verso dx o sx. In base allo stato della macchina decide se scrivere o leggere l'informazione. Meccanicamente non potrebbe implementare infiniti stati.

Definita con una tupla (A, Q, q_0, q_f, δ) dove
$A \neq \emptyset$ chiamato alfabeto composto da un numero finito di simboli.
$Q \neq \emptyset$ è l'insieme degli stati finiti.
$q_0 \in Q$ è lo stato in cui si trova inizialmente la macchina.
$q_f \in Q$ è lo stato in cui si trova la macchina quando finisce.
δ è una funzione con $dom(\delta) = A \times Q$ e $cod(\delta) = A \times Q \times \{L, R\}$ che definisce cosa fa la macchina quando arriva in una cella. L ed R sono, rispettivamente, *left* e *right*, ovvero il movimento della testina che è in 1 dimensione.

Uno stato è definito come (α, i, q) dove
$\alpha : \mathbb{Z} \to A$ è una funzione che definisce il nastro infinito. $\alpha(k) = a \iff k$-esima cella del nastro contiene il simbolo a.
$i \in \mathbb{Z}$ è la posizione della testina sul nastro. Testina posizionata sulla i-esima cella di contenuto $\alpha(i)$.
$q \in Q$ è lo stato corrente.

La macchina finisce in uno stato non finale (α', i', q') se:

- $\delta(\alpha(i), q) = (a, q', x)$. La testina legge il contenuto $\alpha(i)$ della corrente i. Lo stato corrente di aggiorna da q a q'.
- $\alpha'(i) = a$ e $\alpha'(n) = \alpha(n)$ per $n \neq i$ la testina sovrascrive il valore della cella con a

- $i' = i + 1$ se $x = R$
  $i' = i - 1$ se $x = L$
  si muove a dx o sx a seconda del valore di x.

Esempio: confrontare due numeri espressi in base 1: si scrive 1 n-volte in base a che numero voler scrivere. $1 = 1, 2 = 11, 3 = 111, \ldots$
$A = \{b, 1, 0, \$\}$
$b =$ ”blank” significa che è vuoto.
$\$$ usato per separare i numeri.
l'alfabeto è spesso allargato.
...
Lo spazio occupato dall'output è 1 perché non si calcola lo spazio dell'input e si scrive una sola cella per l'output.

Differenze tra macchine di Turing e λ calcolo

A livello di complessità

- nelle macchine di Turing ogni passo (tempo) e cella (spazio) sono costanti $O(1)$: questo perché anche lo stato della macchina è finito. Dunque tempo costante + scrittura/lettura; ogni cella ha un'unità di spazio fissa. Ottimo per studio di complessità.
- nel λ calcolo ogni passo implementato *naive* ha un costo di $O(n^2)$ che dipende dalla dimensione dell'espressione che si vuole semplificare. Un'implementazione di questo tipo diviene più complessa con un tempo non ben definito.
- Le m. T. sono imperative ma non nel senso di linguaggio di prog. imperativo.
- Il λ calcolo è alla base di ogni linguaggio di prog. funzionale.
- Le m. T. sono non composizionali: bisogna farne una ad-hoc ogni volta.
- Il λ calcolo è composizionale: ogni funzione può essere decomposta per risolvere problemi simili.
- Le m. T. sono a basso livello. Diviene difficile implementare costrutti e meccanismi.
- Il λ calcolo ad alto, non ci sono strutture dati vincolanti. Molto facile implementare costrutti e meccanismi di diversi linguaggi di programmazione.
- Testare m. T. è difficile perché non vi è una definizione ricorsiva.
- Il λ calcolo ha il concetto di ricorsione.

Un λ calcolo è una controparte computazione della logica. La corrispondenza tra λ calcolo e logica permette di fare logica $\leftrightarrow$ matematica $\leftrightarrow$ ling. programmazione.

1. λ-calcolo

Definito da Church, calcolare significa semplificare delle espressioni. Una forma primitiva di calcolo è l'esecuzione di espressioni di somma. Il risultato del calcolo è la massima semplificazione dell'espressione. Si parte dall'idea che tutte le espressioni sono funzioni unarie anonime (1 input e 1 output). Ad ogni modo è un linguaggio Turing completo, dunque permette di definire funzione n-arie anche senza usare le chiamate di funzioni (perché sono anonime!), cicli, condizioni o qualsiasi altro costrutto presente in un qualsiasi linguaggio di programmazione.

$$t ::= x \mid tt \mid \lambda x.\, t$$

- t è il termine. Spesso si usano $t, s, u, v, M, N, \ldots$.
- x è l'occorrenza di una variabile. Spesso si usano $x, y, z, w, \ldots$. Dunque si restituisce un valore.
- $t_1 t_2$ è la chiamata di funzione, chiamata applicazione. t_1 è una funzione unaria con parametro t_2. La notazione matematica potrebbe essere $t_1(t_2) \equiv t(t) \equiv tt$.
- $\lambda x.\, t$ è una funzione anonima, chiamata astrazione. Il parametro formale è x e il corpo è t. La notazione matematica è $x \mapsto t \equiv \lambda x.\, t$ Se usassi un nome di funzione avrei qualcosa come $f(x) = t$ (il nome della funzione in questo caso è f). Si usano le parentesi per disambiguare.

La riscrittura di un termine viene detta **riduzione** e non *semplificazione* perché non defluisce la complessità ad ogni passaggio. La riduzione potrebbe anche complicare l'espressione.

Le variabili sono termini. I termini non sono tutti variabili.

Esempi

- $\lambda x.\, x$ è identità
- $(\lambda x.\, x)(\lambda y.\, y)$ risulta $\lambda y.\, y$
- $\lambda x.\, y$ risulta sempre y

A livello di sintassi si ha la precedenza sull'astrazione

- $\lambda x.\, xx$ si legge come $\lambda x.\, (xx)$
  e non si ha l'associatività sennò che a sinistra
- xyz si legge $(xy)z$

Una funzione binaria del tipo $f(x, y) = g(x, y)$ può essere vista come una funziona unaria che ritorna una funzione unaria:

$$\lambda x.\, \lambda y.\, gxy$$

Questo perché usa l'associatività a sx $(gx)y$.

In questo modo si può passare un solo input, come ad esempio

$$(\lambda x.\, \lambda y.\, x + y)2$$

si riduce a

$$\lambda y.2 + y$$

e dunque è come avere una nuova funzione che incrementa di 2 il valore dell'input. Ad esempio con parametro $y = 3$ si riduce come:

$$(\lambda y.2 + y)3 \qquad \rightarrow \qquad 2 + 3$$

Riduzione

Un λ termine t si può ridurre ad un altro t' rimpiazzando una chiamata di funzione $(\lambda x.\, M)N$ con il corpo M dove sostituisco x con N.

Ad esempio $(\lambda x.\, yx)(zz)$ si riduce a yzz, o meglio, $y(zz)$.

Sostituzione

I nomi dei parametri non sono importanti ma invece quelli delle variabili globali sì. $\lambda x.\, y$ e $\lambda x.\, z$ sono programmi diversi.

Il $\lambda x.\, t$ si ha che λ è chiamato **binder**: lega la variabile x al corpo t.
Una variabile non legata è **libera**.

$FV(t)$ è l'insieme delle variabili libere di t.

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

Una funzione ricorsiva strutturale esegue ricorsione solo su parti più piccole dell'input, attuabile solo quando si hanno forme finite possibili, come nel lambda calcolo.

Esempio

$$FV(\lambda x.\, xy(\lambda y.\, yz))$$

si vede come il termine più interno $\lambda y.\, yz$ ha una y legata. In quello più esterno si ha $\lambda x.\, xy(..)$ con x legata.
Più in dettaglio si avrà

$$\begin{aligned}
FV(\lambda x.\, xy(\lambda y.\, yz)) &= FV(xy(\lambda y.\, yz)) - \{x\} \\
&= \Big(FV(xy) \cup FV(\lambda y.\, yz)\Big) - \{x\} \\
&= \Big(FV(xy) \cup \Big(FV(yz) - \{y\}\Big)\Big) - \{x\} \\
&= \Big(FV(x) \cup FV(y) \cup \Big(FV(y) \cup FV(z) - \{y\}\Big)\Big) - \{x\} \\
&= \Big(\{x\} \cup \{y\} \cup \Big(\{y\} \cup \{z\} - \{y\}\Big)\Big) - \{x\} \\
&= \{y\} \cup \{z\} \\
&= \{y, z\}
\end{aligned}$$

Si può, più semplicemente, guardare il legame nel λ ed evitare tutta l'espressione sopra. Essere legato fa riferimento all'occorrenza, non al nome della variabile in sé.

α conversione

Due λ termini t_1 e t_2 sono α convertibili se si può ottenere l'uno dall'altro ridenominando le sole variabili legate in modo che le occorrenze legate di una variabile in una corrispondenza lo sia anche nell'altra. Idem per le variabili libere.
α equivalenza è una relazione simmetrica, riflessiva e transitiva.

Ad esempio $\lambda x.\, \lambda y.\, xyz \equiv_\alpha \lambda x.\, \lambda w.\, xwz$
perché i legami λy e y sono nella medesima posizione di λw e w.
Invece $\not\equiv_\alpha \lambda x.\, \lambda z.\, xzz$
Oppure $\not\equiv_\alpha \lambda x.\, \lambda y.\, yxz$
Oppure $\not\equiv_\alpha \lambda x.\, \lambda y.\, xyw$

La sostituzione "classica" avviene sostituendo in M un termine N al posto della variabile x, scritto come $M\{N/x\}$.

Ad esempio
$(\lambda x.\, xy)\{zz/y\} = \lambda x.\, x(zz)$
però in
$(\lambda x.\, xy)\{xx/y\} \neq \lambda x.\, x(xx)$
ci sono alterazioni nel senso di valori legati. Però con α-conversione si potrebbe avere:
$(\lambda x.\, xy)\{z/x\} \equiv_\alpha (\lambda z.\, zy)\{xx/y\} = \lambda z.\, z(xx)$

Vi sono diversi casi, formalmente:

- $x\{N/x\} = N$
- $y\{N/x\} = y$
- $(t_1 t_2)\{N/x\} = t_1\{N/x\}t_2\{N/x\}$
- $(\lambda x.\, M)\{N/x\} = \lambda x.\, M$
- $(\lambda y.\, M)\{N/x\} = \lambda z.\, M\{z/y\}\{N/x\}$ per $z \notin FV(M) \cup FV(N)$

z è **fresca** se non è mai stata utilizzata. È detta **sufficientemente fresca** se $\notin FV(M) \cup FV(N)$. Se è fresca, lo è anche sufficientemente.
Chiaramente è più semplice prenderne una fresca.

β riduzione

$t_1 \to_\beta t_2 \iff$ ottengo t_2 da t_1 rimpiazzando da qualche parte in t_1 il **redex** $(\lambda x.\, M)N$ col ridotto $M\{N/x\}$.

Ad esempio $\lambda x.\, (\lambda y.\, yx)x \to_\beta \lambda x.\, xx$ dove è stato ridotto il redex $(\lambda y.\, xy)x$

La relazione binaria $\to_\beta$ è definita mediante un **sistema di inferenza**, un sistema stile deduzione naturale che si possono comporre fra di loro.

$$\frac{}{(\lambda x.M)N \to_\beta M\{N/x\}}$$

$$\frac{M \to_\beta M'}{MN \to_\beta M'N}$$

$$\frac{M \to_\beta M'}{NM \to_\beta NM'}$$

$$\frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}$$

Il primo è un assioma: non ha ipotesi (premesse) ma solo la conclusione.

Ad esempio si ha

$$\frac{\frac{}{(\lambda x.yx)y \to_\beta yy}}{\frac{y((\lambda x.yx)y) \to_\beta y(yy)}{\lambda y.y((\lambda x.yx)y) \to_\beta \lambda y.y(yy)}}$$

col primo assioma e le altre varie successioni.

$t_1 \to_\beta^n t_{n+1}$ (t_1 si riduce in n passi a t_{n+1}) $\iff t_1 \to_\beta t_2 \to_\beta \cdots \to_\beta t_{n+1}$
Formalmente:

- $t \to_\beta^0 t$
- $t \to_\beta^{n+1} t'' \iff t \to_\beta t'$ e $t' \to_\beta^n t''$

$t \to_\beta^\star t'$ (t riduce in $0+$ passi a t') $\iff \exists n : t \to_\beta^n t'$

Ad esempio.

$$(\lambda x.\, \lambda y.\, xy)(\lambda z.\, z)(\lambda z.\, z) \to_\beta^3 \lambda z.\, z$$

$(\lambda x.\, \lambda y.\, xy)(\lambda z.\, z)(\lambda z.\, z) \to_\beta (\lambda y.\, (\lambda z.\, z)y)(\lambda z.\, z)$ e qui si vede come ci siano due altri redex da fare. In totale 3.
$\to_\beta (\lambda y.\, y)(\lambda z.\, z) \to_\beta \lambda z.\, z$

Forme normali

Una forma canonica è quando si ha un insieme di rappresentazioni e se ne battezza una come rappresentazione di riferimento. Quella normale è quando, dato un insieme di rappresentazioni, si prende una canonica riscritta all'ennesimo.

t è una forma normale $t \nrightarrow_\beta \iff \nexists t' : t \to_\beta t'$
t ha forma normale $t' \iff t \to_\beta^\star t' \wedge t' \nrightarrow_\beta$
t ha una forma normale (o può convergere) $\iff \exists t' : t$ ha forma normale t'

Non determinismo

La relazione $\to_\beta$ è non deterministica quando da uno stato si può transitare in un altro differente. Dunque $t : \exists t_1, t_2, t_1 \neq t_2$ con $t \to_\beta t_1$ e $t \to_\beta t_2$

Ad esempio
$(\lambda x.\, y)((\lambda z.\, z)w) \to_\beta (\lambda x.\, y)w$

$(\lambda x.\, y)((\lambda z.\, z)w) \to_\beta y$

Non si specifica l'ordine in cui applicare i redex. Potenzialmente, nei sistemi non deterministici, si potrebbero avere risultati diversi in base alla strada presa. In questo caso in ambedue casi si arriva al redex
$(\lambda x.\, y)w \to_\beta y$

Strade diverse non portano, per forza, a forme normali. Ma se arrivo ad una forma normale, sono sicuro che sarà uguale a quella in cui arriveranno tutti.

Un linguaggio Turing completo dovrebbe avere tipi di dato, scelta (if-else) e ripetizione (while, ricorsione). Il linguaggio SQL non è Turing-completo, ma poi gli altri più famosi sì. Basta avere i numeri naturali e codificare tutto il resto con essi.

I linguaggi funzionali non modificano memoria e usano funzioni ricorsive.

Il λ calcolo non ha dati, scelta (if-else) e non può avere ricorsione dato che le funzioni sono anonime.

Paradosso di Russell

L'assioma di comprensione (inconsistente) definisce, data una proprietà P, l'esistenza $\{X : P(X)\}$ e si ha $\forall y : y \in \{X | P(X)\} \iff P(Y)$.

Russell però definisce

$$X \stackrel{\text{def}}{=} \{Y | Y \notin Y\}$$

quindi l'insieme non appartiene a se stesso. Ma $X \in X$? Sì, ma $\iff X \notin X$.

O meglio,

$$X \in X \iff \neg(X \in X) \iff \neg(\neg(X \in X)) \iff \ldots \iff \neg(\ldots \neg(X \in X))) \iff \ldots$$

Questo lo si ottiene senza ricorsione.

In λ calcolo è tutta una funzione, dunque si può passare una funzione a se stessa (nel medesimo modo in cui nella teoria degli insiemi lo si fa con gli insiemi).

Tutto è un insieme	Tutto è una funzione
$X \in X$	xx
$\neg$	f
$X \notin X$	$f(xx)$

Nel primo caso:

- Assioma di comprensione: da $P(Y)$ ricava $\{Y : P(Y)\}$
- $\{Y : Y \notin Y\} \in \{Y : Y \notin Y\} \iff \neg(\{Y : Y \notin Y\} \in \{Y : Y \notin Y\})$

Nel secondo caso:

- λ astrazione: da M ricava $\lambda y.\, M$
- $(\lambda y.\, f(yy))(\lambda y.\, f(yy)) \to_\beta f((\lambda y.\, f(yy))(\lambda y.\, f(yy)))$

Per essere Turing-completi si deve ripetere lo stesso codice più volte. Ma secondo quanto sopra, non si ripete il codice bensì lo copia e lo esegue. Dunque, per essere Turing-completi, è *sufficiente* che lo ripeta più volte. Mi basta eseguire una nuova copia del codice $\lambda x.\, f(xx)$

Sia l'insieme A e una funzione f con $\mathrm{dom} f = A$ e $\mathrm{cod} f = A$. x è un **punto fisso** di $f \iff x = f(x)$.
Un esempio il valore assoluto $|.|$ ha infiniti punti fissi su $\mathbb{Z}$.
$x \mapsto x + 1$ non ha punti fissi.
In λ calcolo t è punto fisso di $f \iff f(t) =_\beta t$, dove l'uguaglianza è una chiusura riflessiva, simmetrica e transitiva di $\to_\beta$.

Teorema
In λ calcolo ogni termine M ha almeno un punto fisso.

Dimostrazione
$(\lambda x.\, M(xx))(\lambda x.\, M(xx))$ è un punto fisso di M. Infatti

$$(\lambda x.\, M(xx))(\lambda x.\, M(xx)) \to M((\lambda x.\, M(xx))(\lambda x.\, M(xx)))$$

A differenza delle funzioni matematiche, potrebbe non terminare.

Definizione
Y è un operatore di punto di fisso $\iff \forall M : YM$ è un punto fisso di M.

Teorema

$$Y \overset{\text{def}}{=} \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))$$

è un operatore di punto fisso. In matematica il calcolo è più complesso e diverso per ogni funzione; qui basta mettere in prefisso λf.

Dimostrazione

$$YM = (\lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx)))M \to_\beta (\lambda x.\, M(xx))(\lambda x.\, M(xx))$$

che è un punto fisso di M.

Si può ottenere il più piccolo programma divergente, che non finisce mai di ridurre:

$$(\lambda x.\, xx)(\lambda x.\, xx) \to_\beta (\lambda x.\, xx)(\lambda x.\, xx) \to_\beta \dots$$

Preso un codice in OCaml come

```
let rec f n =
        match n with
```

```
                | 0 => 0
                | S m => if even(S m) then S m + f m else f m
```

che ha pattern-matching, ricorsione e condizione if-else. Come si scrive in λ calcolo?

Scrivendo la funzione con un funtore non ricorsivo. Nel λ calcolo un funtore è una funzione che restituisce una funzione: tecnicamente lo è tutto.

```
let F : (nat -> nat) -> (nat -> nat) =
  λf.
        λn.
          match n with
                | 0 => 0
                | S m => if even(S m) then S m + f m else f m
```

in λ calcolo sarebbe
$f = YF$ dove Y è un punto fisso.

$$f = YF \rightarrow_\beta F(YF) = Ff$$

Quindi una qualsiasi funzione si trasforma mettendo tutto il corpo dopo Y.

```
Y
  (λf.
        λn.
          match n with
                | 0 => 0
                | S m => if even(S m) then S m + f m else f m)
```

le modifiche sono locali: stesso codice di prima ma con piccole cose. Dunque il λ calcolo permette di codificare le cose con poche modifiche. Con le macchine di Turing ci sarebbe un bel po' di lavoro dietro per farne il porting.

Esempio

```
let rec fact =
  λn.
  match n with
    | 0 => 1
    | S m => m * fact n
```

che in λ calcolo diverrebbe

```
Y
  (λfact.
    λn.
    match n with
```

```
          | 0 => 1
          | S m => m * fact m)
```

si ha che

$$\text{fact}(\text{S } 0) = Y(\lambda \text{fact}.\, \lambda n. \cdots)(\text{S } 0) \to_\beta$$

$$(\lambda \text{fact}. \cdots)(Y(\lambda \text{fact}. \cdots))(\text{S } 0) \to_\beta$$

$$\star^1 \qquad (\lambda n.\, \text{match n with } 0 => 1 \mid \text{S n} => \text{S n} * Y(\lambda \text{fact}. \cdots)n)(\text{S } 0) \to^\star_\beta$$

$$\text{S } 0 * Y(\lambda \text{fact}. \cdots)0 \to^\star_\beta$$

$$\text{S } 0 * 1 \to^\star_\beta$$

$$\text{S } 0$$

In $\star^1$ si potrebbe però espandere la parte $Y(\lambda \text{fact}. \cdots)$ e dunque avere

$$(\lambda n.\, \text{match n with } 0 => 1 \mid \text{S n} => \text{S n} * (\lambda n.\, \text{match n with } 0 => 1 \mid \text{S n} => \text{S n} * Y(\lambda \text{fact}. \cdots)n)(\text{S } 0) \to^\star_\beta$$

però così si avrebbe un'espansione all'infinito.

Un tipo di dato algebrico, sempre in OCaml, può essere definito come:

```
type B = true : B | false : B
(** Es: true: B *)
```

Il valore booleano, definito come B, può avere solo due valori ed essi sono del tipo booleano.
B è il nome del tipo, true/false sono costruttori.

Un esempio, simile alle `enum` in Rust, i semi delle carte:

```
type Seme = Cuori : Seme | Quadri : Seme | Picche : Seme | Fiori: Seme
```

Oppure i numeri naturali, in cui `0` è una possibile forma dei numeri naturali. Il simbolo `S` non è un successore diretto, bensì è come se fosse una funzione che prende un numero naturale e ritorna un altro numero naturale. Dunque è ricorsivo in questo caso.

```
type N = 0 : N | S : N -> N
(** Es: S (S 0) : N *)
```

O una coppia di numeri naturali

```
type N2 = Pair : N -> N -> N2
(** Es: Pair 3 5 : N2 *)
```

Oppure un parametrico, come una Lista di valori `T. List` è il tipo parametrico e `T` il parametro del tipo. La lista è vuota `[]` o una "cons" `(::)` in cui ha una testa e una coda. Una lista ha due elementi dunque:

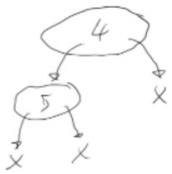
testa `T` e coda `List T`.

```
type List T = [] : List T | (::) : T -> List T -> List T
(** Es: (S 0) :: 0 :: [] : List N
La testa è  (S 0)
La coda è   0 :: []
la coda a sua volta ha
testa       0
coda        []
---
Altro esempio è
1 :: (2 :: (3 :: []))
*)
```

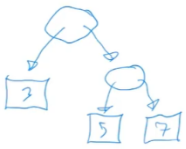
Si possono fare delle ottimizzazione in memoria: i valori piccoli numerici possono andare scritti come bit dato che occupano poco spazio; la lista vuota può semplicemente puntare a `nil`.

Un esempio di struttura albero può essere definito come

```
type Tree1 T = X : Tree1 T | O : Tree1 T -> T -> Tree1 T -> Tree1 T
```




```
type Tree2 T = □ : T -> Tree2 T | O : Tree2 T -> Tree2 T -> Tree2 T
```



un tipo di dato algebrico può essere confrontato col pattern-matching.

```
match x with
  | ki xi ... xn => Mi
```

dove k_i è il nome dell'i-esimo costruttore; $x_i \dots x_n$ sono variabili, una per ogni argomento del costruttore, che funzionano come variabili legate; M_i è il codice da eseguire che può usare le variabili $x_i \dots x_n$.
Esempi dai tipi definiti prima:

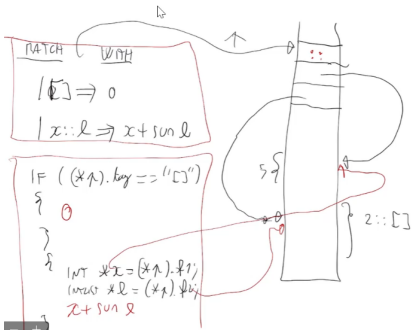
- `b: B`

```
match b with
  | true => M1
  | false => M2
```

- calcolare la somma di una lista di `N`

```
let rec sum l =
  match l with
    | [] => 0
    | x :: l => x + sum l
```

qui `x` è `N`, mentre `l` è `List N`. Con un esempio di `l` con valore iniziale `5 :: (2 :: [])` si avrà un heap tipo:



si hanno un numero finito di `if-and-else` e di defer.

Il λ calcolo è un linguaggio di programmazione unitipato. Non ha tipi, però si può fare un ponte con la logica.

Presa come esempio un tipo lista

```
type List T = [] : List T | (::) T -> List T -> List T
```

```
let rec sum =
  λl.
    match l with
      | [] => 0
      | x::l => x + sum l
```

Introduciamo il concetto di **interfaccia**: un insieme di funzioni. Una nuova lista significa definire tre nuovi costrutti: empty, cons e pattern-matching.

$$\text{empty} : \text{List T}$$

$$\text{cons} : \text{T} \to \text{List T} \to \text{List T}$$

$$\text{match}_{\text{List}} : \forall \text{X}.\text{List T} \to \text{X} \to (\text{T} \to \text{List T} \to \text{X}) \to \text{X}$$

si def. come $\text{match}_{\text{List}}$ perché ogni match è diverso per ogni tipo.

```
let rec sum =
  λl.
    match_List
    l
```

```
    0
    (λx.λl. x + sum l)
```

Il costrutto di pattern-matching è considerato una volta sola. I linguaggi che hanno la possibilità di essere più flessibili nel fare questo matching lo fanno in fase di compilazione.

$$\mathrm{match}_{\mathrm{List}}\ \mathrm{empty}\ \mathrm{M_e}\ \mathrm{M_c} \to^{\star}_{\beta} \mathrm{M_e}$$

$\mathrm{M_e}$ corrisponde a cosa restituire quando la lista è vuota (0 in questo caso)
$\mathrm{M_c}$ corrisponde a $(\mathrm{T} \to \mathrm{List\ T} \to \mathrm{X})$

Passo una funzione che in un qualche modo codifica il dato e un'altra funzione estrae questo dato codificato.

$$\mathrm{match}_{\mathrm{List}}(\mathrm{cons\ M_x\ M_l})\ \mathrm{M_e}\ \mathrm{M_c} \to^{\star}_{\beta} \mathrm{M_c}\ \mathrm{M_x}\ \mathrm{M_l}$$

Con $\to^{\star}_{\beta}$ diciamo che c'è una strada che porta a quel risultato in maniera deterministica, ma non c'è solo quella.

Prendere un termine astratto su delle variabili e poi metterlo in altre funzioni sono teoremi di riduzione semantica: "dato un insieme di ipotesi Γ e un'ipotesi X e da lì si dimostra una formula f" è equivalente a dire che "a partire da Γ si dimostra $X \to f$".

Da un punto di vista logico si può riscrivere

$$\mathrm{match}_{\mathrm{List}} : \mathrm{List\ T} \to \forall \mathrm{X}.\ \mathrm{X} \to (\mathrm{T}\ \to \mathrm{List\ T} \to\ \mathrm{X}) \to \mathrm{X}$$

$$\mathrm{List\ T} \overset{\mathrm{def}}{=} \forall \mathrm{X}.\mathrm{X}(\mathrm{T}\ \to\ \mathrm{List\ T} \to \mathrm{X}) \to \mathrm{X}$$

Il tipo è ricorsivo.

Il pattern-matching in questo caso (in realtà sempre) è la funzione identità.

$$\mathrm{match}_{\mathrm{List}} \overset{\mathrm{def}}{=} \lambda x.\ x$$

$$\mathrm{empty}\colon \mathrm{List\ T} = \forall \mathrm{X}\colon \mathrm{X} \to (\mathrm{T} \to \mathrm{List\ T} \to\ \mathrm{X}) \to \mathrm{X} \overset{\mathrm{def}}{=} \lambda e.\ \lambda c.\ e$$

e è X, c è $\mathrm{T} \to \mathrm{List\ T} \to \mathrm{X}$

È una funzione con due input, quindi sono due λ.

Poi si ha

$$\mathrm{empty} \overset{\mathrm{def}}{=} \lambda e.\ \lambda c.\ e$$

$$\mathrm{match}_{\mathrm{List}}\ \mathrm{empty}\ \mathrm{M_e}\ \mathrm{M_c}$$

$$= (\lambda x.\ x)(\lambda e.\ \lambda c.\ e)\mathrm{M_e}\ \mathrm{M_c}$$

$$\to_{\beta} (\lambda e.\ \lambda c.\ e)\mathrm{M_e}\ \mathrm{M_c}$$

$$\to_{\beta} \mathrm{M_e}$$

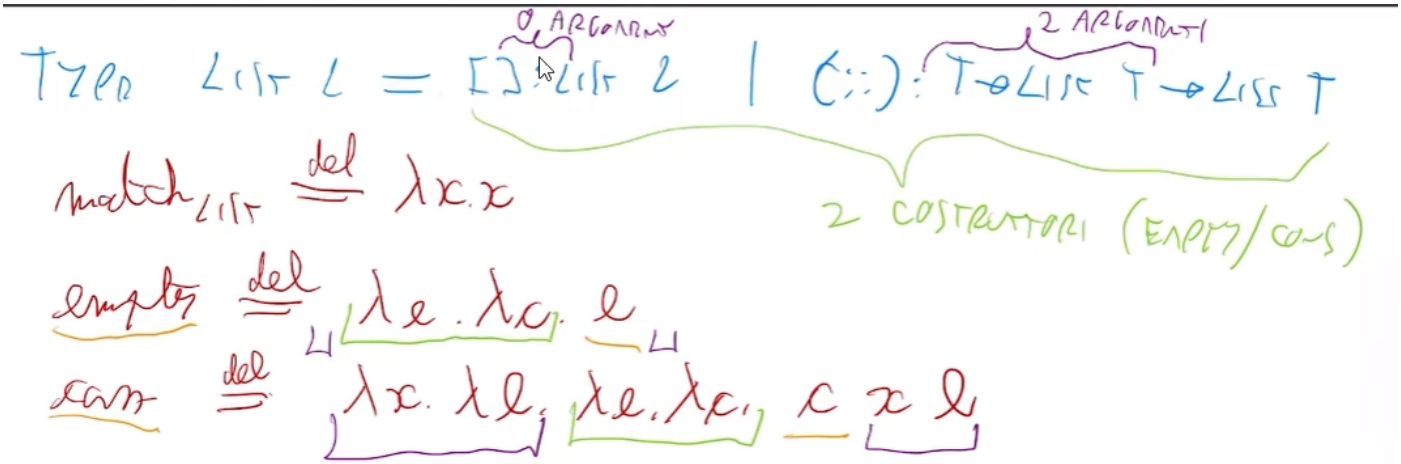
Invece si ha che

$$\text{cons: T} \to \text{List T} \to \text{List T} = \text{T} \to \text{ List T} \to \forall X.\, X \to \text{ (T} \to \text{List T} \to \text{ X)} \to \text{X} \stackrel{\text{def}}{=} \lambda x.\, \lambda l.\, \lambda e.\, \lambda c.\, cxl$$

x è T, l è List T, e è X, c è T $\to$ List T $\to$ X

$$\text{match}_{\text{List}}\ (\text{cons M}_\text{x}\ \text{M}_\text{c})\text{M}_\text{e}\ \text{M}_\text{c}$$
$$= (\lambda x.\, x)(\text{cons M}_\text{x}\ \text{M}_\text{l})\text{M}_\text{e}\ \text{M}_\text{c}$$
$$\to_\beta \text{cons M}_\text{x}\ \text{M}_\text{l}\ \text{M}_\text{e}\ \text{M}_\text{c}$$
$$= (\lambda x.\, \lambda l.\, \lambda e.\, \lambda c.\, cxl)\text{M}_\text{x}\ \text{M}_\text{l}\ \text{M}_\text{e}\ \text{M}_\text{c}$$
$$\to_\beta^4 \text{M}_\text{c}\ \text{M}_\text{x}\ \text{M}_\text{l}$$

Dunque, ignorando i tipi che tanto non servono ad eccezione di guida per la dimostrazione, si hanno che i 3 sono formati come segue



Riprendendo l'esempio del tipo booleano

```
type B = true : B | false : B
```

$$\text{match}_\text{B} \stackrel{\text{def}}{=} \lambda x.\, x$$
$$\text{true} \stackrel{\text{def}}{=} \lambda t.\, \lambda e.\, t$$
$$\text{false} \stackrel{\text{def}}{=} \lambda t.\, \lambda e.\, e$$
$$\text{match}_\text{B}\ \text{true}\ \text{M}_\text{t}\ \text{M}_\text{e} = (\lambda x.\, x)\ \text{true}\ \text{M}_\text{t}\ \text{M}_\text{e}$$
$$\to_\beta\ \text{true}\ \text{M}_\text{t}\ \text{M}_\text{e} = (\lambda t.\, \lambda e.\, t)\text{M}_\text{t}\ \text{M}_\text{e}$$
$$\to_\beta \text{M}_\text{t}$$

Riprendendo l'esempio dei numeri naturali

```
type N = 0 : N | S : N -> N
```

$$\text{match}_\text{N} \stackrel{\text{def}}{=} \lambda x.\, x$$
$$0 = \lambda z.\, \lambda s.\, z$$

$$\mathrm{S} = \lambda n.\, \lambda z.\, \lambda s.\, sn$$

$$\mathrm{match_N}\ (\mathrm{S\ N})\ \mathrm{M_z}\ \mathrm{M_s}$$

$$\to_\beta \mathrm{S\ N}\ \mathrm{M_z}\ \mathrm{M_s}$$

$$= (\lambda n.\, \lambda z.\, \lambda s.\, sn)\ \mathrm{N}\ \mathrm{M_z}\ \mathrm{M_s}$$

$$\to_\beta^3 \mathrm{M_s\ N}$$

Riprendendo l'esempio dei semi

```
type Seme = Cuori : Seme | Quadri : Seme | Picche : Seme | Fiori : Seme
```

Tutti hanno 0 argomenti ma ci sono 4 costruttori.

$$\mathrm{match_{Seme}} \overset{\text{def}}{=} \lambda x.\, x$$

$$\mathrm{cuori} \overset{\text{def}}{=} \lambda c.\, \lambda q.\, \lambda p.\, \lambda f.\, c$$

$$\mathrm{quadri} \overset{\text{def}}{=} \lambda c.\, \lambda q.\, \lambda p.\, \lambda f.\, q$$

$$\mathrm{picche} \overset{\text{def}}{=} \lambda c.\, \lambda q.\, \lambda p.\, \lambda f.\, p$$

$$\mathrm{fiori} \overset{\text{def}}{=} \lambda c.\, \lambda q.\, \lambda p.\, \lambda f.\, f$$

Assegnazione

```
var a = 2;
f(x, y) {
        var z = 2;
        x = z * y;
        z = y + a;
        a = 3;
        x = x + g();
        return x + z;

        g() {
                z = z+1;
                return 3;
        }
}
```

nel λ calcolo e programmazione funzionale non si può mutare nulla: se voglio comunicare un cambiamento, si da un output in più.
Trasformare in un linguaggio funzionale uno snippet scritto in linguaggio imperativo vuol dire esplicitare tutto ciò che dipende e cambia una funzione.

```
f(x, y, a) {
        // x viene usata, dunque deve restituire il valore nuovo
        // y no, quindi non ritorna
        // a sì, quindi ritorna
        // z sì ma è locale, quindi non ritorna

        var z = 2;
        var x' = z * y;
        var z' = y + a;
        var a' = 3;
        var (z'', res) = g(z');
        var x'' = x' + res;
        return (a', x'', x''+z'')

        g(z) {
                var z' = z + 1;
                return (z', 3);
        }
}
```

visto che non esiste l'assegnazione di variabile, una soluzione è quella di espandere le variabili.

```
var x = 4;
...
g(x);
```

diviene

```
// g(x){4/x}
...
g(4);
```

oppure si può scrivere un redex
$(\lambda x. \cdots . g(x))4$

in programmazione funzionale può esser fatto come

```
let x = 4 in g(x)
```

Cicli

```
var x = 10;
var res = 0;
while_ x > 0 do
```

```
        res = res + x;
        x = x - 1;
done
```

diviene

```
let rec while_ x res =
        if x > 0 then
                while_ (x-1) (res+x)
        else
                (x, res)
```

Record

```
struct person {
        name = "Claudio"
        age = 47
        city = "Bologna"
}

if person.age > 20 then
        return person.name
```

come sono messi in memoria potrebbe essere semplice zucchero sintattico per n-uple, in questo caso triple perché ha 3 campi.

```
type Person = mk : string -> int -> string -> Person
```

la funzione `age` diviene solo un modo per estrarre valore. Idem per le altre due.

```
age p =
        match p with
                mk name age city => age

if age person > 20 then
        person.name
```

OOP

```
object person {
        age = 41
        name = "Claudio"
        grow(n) {
                if self.age + n > 100 then
                        self.die()
                else
```

```
                        self.age = self.age + n
        }
        die() {
                self.name = "RIP"
        }
}
```

In questo linguaggio si hanno gli oggetti, non classi.

```
struct person {
        age = 41
        name = "Claudio"
        grow =
                λself.λn
                        if self.age + n > 100 then
                                // passa come parametro l'oggetto stesso
                                self.die self
                        else
                                person {
                                        age = self.age + n
                                        name = self.name
                                        grow = self.grow
                                        die = self.die
                                }
}
```

Eccezioni

Il controllo del programma può passare all'istruzione successiva o può saltare ad una computazione completamente diversa.

```
type e = E1 : N -> e | E2 : string -> e | E3 : N -> N -> e
```

si definiscono due costrutti nuovi

`throw e`

```
try M with
        | E1 x => M1n
        | E2 x => M2
        | E3 x y => M3
```

```
try
        ( if even(4) then
```

```
                true
        else
                throw (E1 7) ) or false
with
        | E1 x => even(x)
        | E2 x => false
        | E3 x y => true
```

prima viene eseguito il codice dentro `if`.
$\rightarrow^*$

```
try true
        with .. | .. | ..
                => true
```

ma invece con

```
try
        ( if even(5) then
                true
        else
                throw (E1 7) ) or false
with
        | E1 x => even(x)
        | E2 x => false
        | E3 x y => true
```

$\rightarrow$

```
try throw(E1 7) or false
with
        | E1 x => even(x)
        | E2 x => false
        | E3 x y => true
```

non si passa il controllo al `false` bensì ad un eccezione che poi viene gestita grazie al `try ... with` e dunque viene eseguito `E1 x => even(x)`.

$$M : B \vee N \vee \text{string} \vee (N \times N)$$

$$M : I \rightarrow O_1 \vee O_2 \vee \cdots \vee O_n$$

ma in λ calcolo si ha solo

$$M : I_1 \rightarrow I_2 \rightarrow \cdots \rightarrow I_k \rightarrow O \cong I_1 \wedge I_2 \wedge \cdots \wedge I_k \rightarrow O$$

Ricordando che

$\neg F \equiv F \to \bot$

$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$

$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$

$\neg\neg F \equiv F$

$F_1 \wedge F_2 \to G \equiv F_1 \to F_2 \to G$

si ha che

$$\begin{aligned} I \to O_1 \vee O_2 &\equiv I \to \neg\neg(O_1 \vee O_2) \\ &\equiv I \to \neg(\neg O_1 \wedge \neg O_2) \\ &\equiv I \to ((O_1 \to \bot) \wedge (O_2 \to \bot) \to \bot) \\ &\equiv I \to (O_1 \to \bot) \to (O_2 \to \bot) \to \bot \end{aligned}$$

si deve invocare una delle due funzioni $(O_1 \to \bot)$ oppure $(O_2 \to \bot)$ dove, nel codice, ci sono eccezioni.

```
f : Z -> Z                 or Z             or string
      'term. con succ.'   'throw neg.'     'throw toobig'
=
        λx.
                ( if x < 0 then
                        throw (negative x)
                else if x > 10 then
                        throw (toobig "reduce")
                else
                        x ) * 10
```

```
f : Z -> (Z -> ⊥) -> (Z -> ⊥) -> (string -> ⊥) -> ⊥
=
        λn.λk_{return}.λk_{e1}.λk_{e2}.
                if x < 0 then
                        k_{e1} x
                else if x > 10 then
                        k_{e2} "reduce"
                else
                        k_{return} x * 10
```

è una trasformazione globale del codice. Dunque

```
try
        f 2
with
        | E1 x => x+2
        | E2 x => 0
```

che è

$f(\lambda x.\, x)(\lambda x.\, x + 2)(\lambda x.0)$

2. Logica proposizionale minimale

Si ha solo il connettivo di implicazione ($\to$).

```
F ::= A | F -> F
```

dove A è una variabile proposizionale (rappresenta un valore vero o falso).
dove $F \to F$ è un'implicazione materiale "se ... allora ...".

$\to$ è associativo a destra. Ad esempio $A \to B \to A \equiv A \to (B \to A)$.

I contesti di ipotesi

```
Γ ::= | Γ, F
```

in cui si suppone che F valga.

Un judgement di derivazione logica è definita come $\Gamma \vdash F$, ovvero "dall'ipotesi Γ riesco a dimostrare F". Un modo per definire le regole di derivazione sono chiamate **deduzione naturale**.

$$\frac{F \in \Gamma}{\Gamma \vdash F}$$

$$\frac{\Gamma \vdash F_1 \to F_2 \qquad \Gamma \vdash F_1}{\Gamma \vdash F_2}$$

quest'ultimo è chiamato *modus ponens* o $\to_e$

$$\frac{\Gamma, F_1 \vdash F_2}{\Gamma \vdash F_1 \to F_2}$$

che è anche scritto come $\to_i$

Queste 3 regole sono le medesime usate per definire le regole del λ calcolo tipato.

Ad esempio

$$\dfrac{\dfrac{\dfrac{\dfrac{A \to B \to C \in \Gamma}{A \to B \to C, B, A \vdash A \to B \to C} \quad \dfrac{A \in \Gamma}{A \to B \to C, B, A \vdash A}}{A \to B \to C, B, A \vdash B \to C} \quad \dfrac{B \in \Gamma}{A \to B \to C, B, A \to B}}{A \to B \to C, B, A \vdash C}}{A \to B \to C, B \vdash A \to C}}{A \to B \to C \vdash B \to A \to C}$$

3. Rapporto tra λ-calcolo e logica

Fissato un linguaggio di programmazione si ha un insieme di tutti i programmi P (λ termini). Una **proprietà** è un qualunque sotto insieme di P. x ha la proprietà Q sse $x \in Q$. Una proprietà è banale sse
$Q = \emptyset \vee Q = P$.
Un esempio è $Q = \{p \in P : p \text{ usa 2 variabili}\}$ nel caso di programma che parla di com'è scritto.
Un esempio è $Q = \{p \in P : p(0) = 1\}$ nel caso di programma che parla di cosa fa sotto forma di funzione.

Una proprietà Q è **decidibile** sse $\exists p \in P. \forall q \in P. (q \in Q \iff p(q) = \text{true} \wedge q \notin Q \iff p(q) = \text{false})$.
Ad esempio, con Q è decidibile se $Q = \{p \in P : |p| < 100 \text{ caratteri}\}$

Una proprietà Q è **estensionale** sse $\forall p, q, Q : (\forall i. p(i) = 0 \iff q(i) = 0) \implies p \in Q \iff q \in Q$
Dunque non parla dei programmi come sono scritti ma di quello che calcolano. Ad esempio bubble sort e quick sort.
Una proprietà Q è **intenzionale** se non è estensionale.

Teorema di Rice

$\forall Q$. se Q non è banale e Q è estensionale, allora Q non è decidibile.
Ad esempio divergere lo è: dato un input il programma non termina. Non è banale perché ci sono programmi che terminano ed è estensionale perché non interessa com'è scritto tale programma.

Approssimazione

R è un'**approssimazione da dentro** di Q sse $R \subseteq Q$.
S è un'**approssimazione da fuori** di Q sse $Q \subseteq S$.
Supponiamo che R o S siano decidibili e decide da un programma r o s.
$\forall p. (r(p) = \text{true} \implies p \in Q) \wedge (s(p) = \text{false} \implies p \notin Q)$
Nel caso di approssimazione che sta in un'area "grigia": ovvero un po' dentro e un po' fuori, posso allargare l'approssimazione da dentro o stringerla da fuori ma non riuscirò mai una zona di certezza.
Un'approssimazione in area grigia non serve a nulla, quindi meglio allargare o stringere.
Dunque dobbiamo cercare sempre approssimazione da dentro oppure da fuori.
Le approssimazioni modulari sono un po' meno precise, ma un buon compromesso.

Sistema di tipi

È l'implementazione di un programma che decide un'approssimazione da dentro o da fuori in **maniera modulare**.
Molti sistemi di tipo si inseriscono nei linguaggi non tipati, andando a cercare errori gravi nel codice. In C e Java, se si dice che è mal tipato, vuol dire che potrebbe non funzionare a run time.
Il fatto che sia modulare vuol dire che il programma p può essere diviso in $p_1, \ldots, p_n$ moduli. Un sistema di tipi è modulare se, per decidere la modalità, analizza un modulo per volta e poi decidere in base a quelli analizzati se è ben tipato o no.
Ad ogni modulo viene associata un'informazione T_i chiamata **tipo**.
$r(p) = \text{true} \iff p \in R \iff \overline{r}(T_1, \ldots, T_n)$

Con questo vuol dire che non c'è bisogno di avere a disposizione tutto il codice sorgente.

Se si ha una situazione gerarchica tale che un modulo è diviso in altri moduli si vede come p è decomposto in $p_1, \dots, p_n$, p_1 è decomposto in $p_{11}, \dots, p_{1m}$ e così via. Da questo si potrà trovare, a partire dalle foglie, i vari tipi a salire verso l'alto. Tutto ciò ad arrivare a capire che il tipo T del programma p è ben tipato da dentro o da fuori.

λ calcolo tipato semplice

Per semplice si intende che si sceglie quello più semplice tra quelli possibili. I tipi sono quelli che si associano dal basso verso l'alto (dal T_{11m} al T per dire).

```
T ::= A | T -> T
```

$A, B, \cdots$ sono variabili di tipo. Intuizione bool, int, string, $\cdots$.
$T \to T$ sono tipo delle funzioni con un certo input/output.

Ad esempio, il tipo $A \to B$ è il tipo delle funzioni che dato un input A restituisco un output di tipo B.

"$\to$" è associativo a destra. Dunque $A \to B \to C \equiv A \to (B \to C)$.

Un termine può avere più tipi. $\lambda x.\, y$ è un esempio del perché abbiamo un linguaggio modulare, visto che il codice y è esterno.

Contesto

```
Γ ::= | Γ, x : T
```

quindi Γ è vuota oppure gli si associa la variabile x al tipo. Non si hanno termini.

Ad esempio `x: A, y: B, z: A -> B`.
Come ipotesi si ha che in Γ nessuna variabile è ripetuta.
$(x : T) \in \Gamma$ vuol dire che `Γ = ..., x : T, ...`.

Judgement di tipaggio $\Gamma \vdash t : T$

È una relazione ternania. Si legge "in Γ, t ha tipo T", quindi t ha tipo T sotto l'ipotesi Γ.
Definisco $\Gamma \vdash t : T$ attraverso un sistema di inferenza.

1. Termine

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}$$

2. Applicazione

$$\frac{\Gamma \vdash M : T_1 \to T_2 \qquad \Gamma \vdash N : T_1}{\Gamma \vdash MN : T_2}$$

3. Astrazione

$$\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x.\, M : T_1 \to T_2}$$

Ad esempio

$$\dfrac{\dfrac{(f : A \to A) \to B) \in \Gamma}{f : (A \to A) \to B, x : A \to A \vdash f(A \to A) \to B} \quad \dfrac{(x : A \to A) \in \Gamma}{f : (A \to A) \to B, x : A \to A \vdash x, A \to B}}{\dfrac{f : (A \to A) \to B, x : A \to A \vdash fx : B}{f : (A \to A) \to B \vdash \lambda x.\, fx : (A \to A) \to B}}$$

Ad esempio, uno non ben tipato

$$\dfrac{\dfrac{(x : T_3? \to T_4?) \in \Gamma}{x : T_3? \to T_4? \vdash x : T_3? \to T_4?} \quad \dfrac{(x : T_3?) \in x : T_3? \to T_4?}{x : T_3? \to T_4? \vdash x : T_3?}}{\dfrac{x : T_3? \to T_4? \vdash xx : T_2?}{\vdash \lambda x.\, xx}}$$

quello più a destra $(x : T_3?) \in x : T_3? \to T_4?$ sarebbe vero solo se $T_3? = T_3? \to T_4?$ ma sintatticamente non possono esserlo. $\lambda x.\, xx$ on ha la proprietà "MISTERIOSA".

Isomorfismo

La corrispondenza che si ha con λ calcolo si chiama isomorfismo: si cambia da una form all'altra senza perdere informazioni. Il fatto che abbiamo lo stesso risultato, non vuol dire che le facciano nel medesimo modo.

Isomorfismo di Curry-Howard-Kolmogorov

λ **calcolo**	**Logica**
Tipo	Formula
Termini	Prove
Costruttore di tipo	Connettivo
Costruttore di termini	Passi di prova
Variabili libere/legate	Ipotesi globali/locali (quelle locali sono denotate anche come "scaricate")
Type checking	Proof checking
Type inaditation (dato Γ, T cerco un t tc. $\Gamma \vdash t : T$)	Ricerca di prove
Riduzione	Normalizzazione di prove

Escludendo i tipi si può cambiare forma.

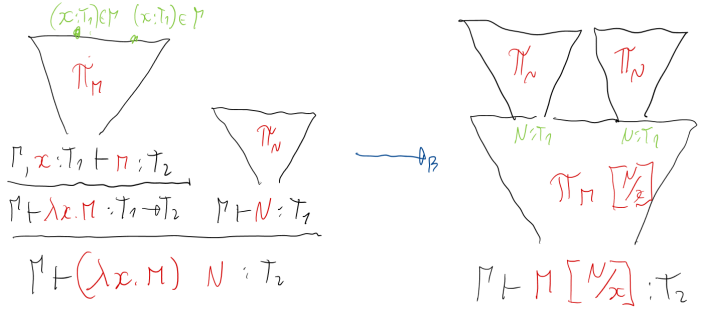
$$\dfrac{\dfrac{\dfrac{(A \to A \to B) \in \Gamma}{\Gamma \vdash A \to A \to B} \dfrac{A \in \Gamma}{\Gamma \vdash A}}{A \to A \to B, A \vdash A \to B} \dfrac{A \in \Gamma}{A \to A \to B, A \vdash A}}{A \to A \to B, A \vdash B}}{A \to A \to B \vdash A \to B} \implies_i$$

$$\dfrac{\dfrac{\dfrac{(f : A \to A \to B) \in \Gamma}{\Gamma \vdash f : A \to A \to B} \quad \dfrac{(x : A) \in \Gamma}{\Gamma \vdash x : A}}{f : A \to A \to B, x : A \vdash fxA \to B} \quad \dfrac{(x : A) \in \Gamma}{f : A \to A \to B, x : A \vdash x : A}}{f : A \to A \to B, x : A \vdash fxx : B}}{f : A \to A \to B \vdash \lambda x.\, fxxA \to B} \implies_i$$

il quale diviene, semplicemente

$$\lambda x.\, fxx$$

(Π sono alberi di prova)



Questo isomorfismo è capace di scalare. Se si prova ad aggiungere altri connettivi, come ad esempio, l'AND.

$$F ::= \ldots | F_1 \wedge F_2$$

$$\dfrac{\Gamma \vdash F_1 \qquad \Gamma \vdash F_2}{\Gamma \vdash F_1 \wedge F_2} \wedge_i$$

$$\dfrac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_1} \wedge_{e_1}$$

$$\dfrac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_2} \wedge_{e_2}$$

$$\dfrac{\Gamma \vdash F_1 \wedge F_2 \qquad \Gamma, F_1, F_2 \vdash F}{\Gamma \vdash F} \wedge_e$$

ma lato λ calcolo si avrà

$$T ::= \ldots | T \times T$$

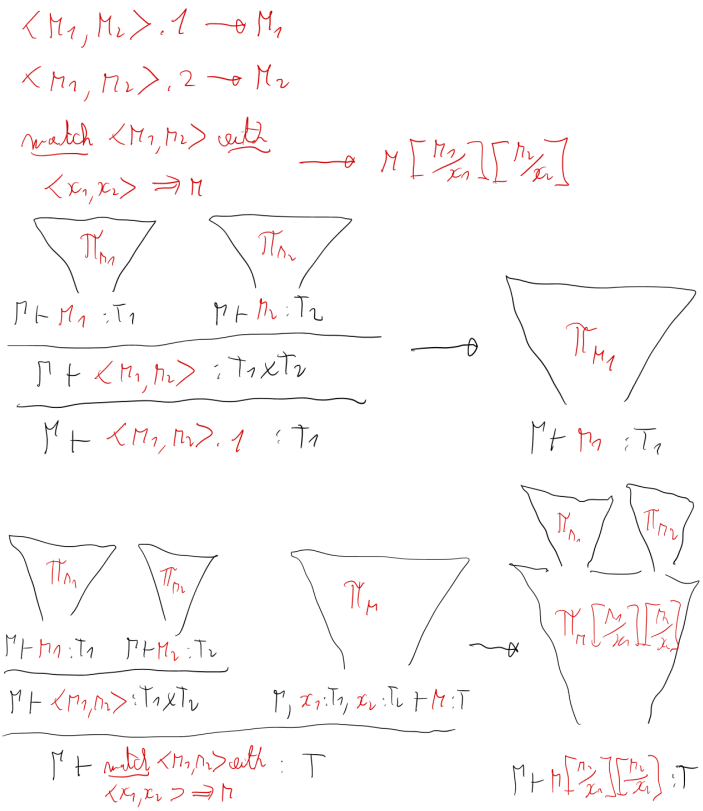
$$t ::= \ldots | \langle t, t \rangle | t.1 | t.2 | \text{match } t \text{ with } \langle x_1, x_2 \rangle \implies t$$

quindi si aggiungono le tuple nel linguaggio di programmazione.

$$\dfrac{\Gamma \vdash M_1 : T_1 \qquad \Gamma \vdash M_2 : T_2}{\Gamma \vdash \langle M_1, M_2 \rangle : T_1 \times T_2}$$

$$\dfrac{\Gamma \vdash C : T_1 \times T_2}{\Gamma \vdash C.1 : T_1} \qquad \dfrac{\Gamma \vdash C : T_1 \times T_2}{\Gamma \vdash C.2 : T_2}$$

$$\dfrac{\Gamma \vdash C : T_1 \times T_2 \qquad \Gamma, x_1 : T_1, x_2 : T_2 \vdash M : T}{\Gamma \vdash \text{match } c \text{ with } \langle x_1, x_2 \rangle \implies M : T}$$



Estendendo la logica aggiungendo T, ovvero il *TOP*, in cui è sempre vero.

$$F ::= \ldots | T$$

$$\frac{}{\Gamma \vdash T} T_i$$

$$\frac{\Gamma \vdash T \qquad \Gamma \vdash F}{\Gamma \vdash F} T_e$$

e lato λ calcolo si avrà

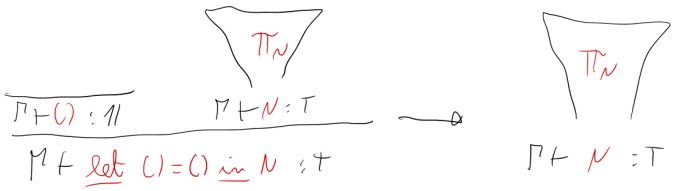
$$T ::= \ldots | \mathbb{1}$$

$\mathbb{1}$ è il costrutto *unit* (in Haskell è `()`, in C è `void`, in Python è `None`, ...).

$$t ::= \ldots | () | \text{let } () = t \text{ in } t$$

$$\frac{}{\Gamma \vdash () : \mathbb{1}}$$

$$\frac{M \vdash M : \mathbb{1} \qquad \Gamma \vdash N : T}{\Gamma \vdash \text{match } M \text{ with } () \implies N : T}$$



Estendendo la logica aggiungendo B, ovvero il *BOTTOM*, in cui è sempre falso. Si dimostra con *ex falso sequitur quodlibet* (dal falso sempre qualunque cosa vera).

$$F ::= \ldots \mid \bot$$

$$\frac{\Gamma \vdash \bot}{\Gamma \vdash F}$$

e lato λ calcolo si avrà

$$T ::= \ldots \mid \mathbb{0}$$

$$t ::= \ldots \mid \text{abort}(t)$$

esempio del codice "mai eseguito". Non si hanno regole di riscrittura perché un programma che esegue un abort, termina lì.

$$\frac{\Gamma \vdash M : \bot}{\Gamma \vdash \text{abort}(M) : T}$$

Estendendo la logica aggiungendo l'or.

$$F ::= \ldots \mid F_1 \vee F_2$$

$$\frac{\Gamma \vdash F_1}{\Gamma \vdash F_1 \vee F_2} \vee_{i_1}$$

$$\frac{\Gamma \vdash F_2}{\Gamma \vdash F_1 \vee F_2} \vee_{i_2}$$

$$\frac{\Gamma \vdash F_1 \vee F_2 \qquad \Gamma, F_1 \vdash F \qquad \Gamma, F_2 \vdash F}{\Gamma \vdash F} \vee_e$$

e lato λ calcolo si avrà un unione disgiunta, dunque sempre dato algebrico.

$$T ::= \ldots \mid T_1 + T_2$$

$$t ::= \ldots \mid \underline{L}(t) \mid \underline{R}(t) \mid \text{match}$$

$$\frac{\Gamma \vdash M_1 : T_1}{\Gamma \vdash \underline{L}(M_1) : T_1 + T_2}$$

$$\frac{\Gamma \vdash M_2 : T_2}{\Gamma \vdash \underline{R}(M_2) : T_1 + T_2}$$

$$\frac{\Gamma \vdash M : T_1 + T_2 \qquad \Gamma, x_1 : T_1 \vdash N_1 : T \qquad \Gamma, x_2 : T_2 \vdash N_2 : T}{\Gamma \vdash \text{match } M \text{ with } \underline{L}(x_1) \implies N_1 \mid \underline{R}(x_2) \implies N_2 : T}$$

Teorema di consistenza della logica proposizionale

$$\nvdash \bot$$

Dimostrazione
Si dimostra per assurdo. Assumo che $\vdash \bot$
Per Curry-Howard $\exists M. \vdash M : \bot$
Sia M t.c. $\vdash M : \bot$
Sia N la forma normale di M che esiste per il teorema di normalizzazione forte, si ha $\vdash N : \bot$
Il $\bot$ non ha regole di introduzione, dunque neanche N non lo è; non è una variabile perché non è un'ipotesi; dunque dovrebbe essere una regola di eliminazione (cioè N è un pattern matching, come

$N = \text{match } M \text{ with } \cdots$).
M non è una variabile per lo stesso motivo di N; non è una regola di introduzione perché se lo fosse ci sarebbe una match su una regola di introduzione e dunque sarebbe un redex, ma una forma normale non ha una riduzione (N è normale); dunque è anch'essa una regola di eliminazione, ma così all'infinito dunque, perché sarebbe un match di un'altra regola di eliminazione e così via. ASSURDO perché manca il caso base di fine di questo loop. Quindi $\not\vdash \bot$.

4. Teorema della normalizzazione forte

Facendo prima delle definizioni:

- Un termine t si dice **debolmente normalizzato** quando ha una forma normale.
- Un termine t si dice **fortemente normalizzante** se $\nexists (t_i)_{i\in\mathbb{N}} : t = t_i \wedge \forall i.\, t_i \to_\beta t_{i+1}$
- Un λ calcolo si dice avere una proprietà Q sse ogni termine ce l'ha.
  Si enuncia il teorema come:

  $$\forall \Gamma, M, T.\, \Gamma \vdash M : T \implies M \text{ è fortemente normalizzante}$$

  Quindi non esistono sequenze divergenti.
  Dunque si ha la proprietà Q indicibile che è "M fortemente normalizzato"; l'approssimazione da dentro è $\Gamma \vdash M : T$. Però vi sono termini fortemente normalizzanti non tipabili, come $\lambda x.\, xx$

Dimostrazione (per induzione ma che poi però non arriva a fine)

> Visto che l'induzione è logica, per il teorema di 3. Rapporto tra λ-calcolo e logica > Isomorfismo di Curry-Howard-Kolmogorov si vede come qualsiasi cosa dimostrata per induzione la si può fare anche per ricorsione (λ calcolo quest'ultimo).

Si procede per induzione sull'albero di prova $\Gamma \vdash M : T$

Caso base: dimostrato mediante caso della variabile.

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : F}$$

Bisogna dimostrare che x è fortemente normalizzante, che è ovvio perché è una variabile e quindi non ha redex.

Caso

$$\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x.\, M : T_1 \to T_2}$$

quello sopra è un albero di prova. Per ipotesi induttiva M è fortemente normalizzante. Bisogna dimostrare come $\lambda x.\, M$ sia fortemente normalizzante.
Per assurdo si suppone che $\lambda x.\, M$ non sia fortemente normalizzante, ovvero che $\lambda x.\, M$ riduci all'infinito.
L'unico modo per fare ciò è mediante:

$$\frac{M \to_\beta M_1 \to_\beta M_2 \to_\beta \cdots}{\lambda x.\, M \to_\beta \lambda x.\, M_1 \to_\beta \lambda x.\, M_2 \to_\beta \cdots}$$

ma per ipotesi induttiva non possiamo fare ciò perché M è fortemente normalizzante.

Caso

$$\frac{\Gamma \vdash M : T_1 \to T_2 \qquad \Gamma \vdash N : T_2}{\Gamma \vdash MN : T_2}$$

Per ipotesi induttiva M è fortemente normalizzante (I_1), N è fortemente normalizzante (I_2). Bisogna dimostrare che MN è fortemente normalizzante.
Farlo per assurdo come prima farebbe un po' di confusione perché si dovrebbe estendere MN all'infinito con diversi casi perché vi è la combinazione di M e N, e non posso farlo singolarmente perché sono comunque entrambi fortemente normalizzanti.
Visto che MN è un applicazione vuol dire che potrebbe essere un redex. Se M è nella forma $\lambda x.\, t$ (con t fortemente normalizzante ofc) ma $MN = (\lambda x.\, t)N \to_\beta t[N/x]$ non c'è garanzia che $t[N/x]$ sia fortemente normalizzante. C'è un controesempio nel ciò: $(\lambda x.\, xx)(\lambda x.\, xx)$ diverge ma $\lambda x.\, xx$ è fortemente normalizzante.

Questo contro esempio fa vedere come la proprietà di essere fortemente normalizzante non è modulare, mentre il tipaggio lo è. Visto che tipaggio implica normalizzazione forte, vuol dire che deve esserci una proprietà intermedia modulare che approssimi meglio la normalizzazione forte, ovvero ci deve essere l'insieme di tutti i programmi P, la proprietà di normalizzazione forte
$SN =^{\text{def}} \{t | t \in P \land t \text{ fortemente normalizzante}\}$ e una proprietà di approssimazione per i tipati
$WT =^{\text{def}} \{t \in P | \exists \Gamma, T.\, \Gamma \vdash t : T\}$. Bisogna trovare una proprietà che stia in mezzo a SN e WT e questa la si chiama $RED_T =$ "insieme dei termini riducibili di tipo T".

Piano di lavoro 1 (che FALLISCE)

Si definisce RED_T e poi bisogna dimostrare che $WT \subseteq RED_T \subseteq SN$. Si mette una proprietà in mezzo che però non può essere dimostrata la sua intuitività.

I tipi hanno strutture ricorsive quindi, per definire qualcosa sul tipo, si può procedere a definirli ricorsivamente. Quindi la definizione di RED_T avviene con due casi:

$$RED_A \overset{\text{def}}{=} \{M | \exists \Gamma.\, \Gamma \vdash M : A \land M \in SN\}$$

(A è un tipo di cui non sappiamo nulla)
$(\star)$

$$RED_{T_1 \to T_2} \overset{\text{def}}{=} \{M | \exists \Gamma.\, \Gamma \vdash M : T_1 \to T_2 \land (\forall N \in RED_{T_1}.\, MN \in RED_{T_2})\}$$

(In realtà vi è anche una proprietà ridondante che è $M \in SN$)

Teorema (tentativo): $WT \subseteq RED$ ovvero $\forall \Gamma, M, T.\, \Gamma \vdash M : T \implies M \in RED_T$
Per induzione sull'albero $\Gamma \vdash M : T$

- Caso

  $$\frac{\Gamma \vdash M : T_1 \to T_2 \qquad \Gamma \vdash N : T_2}{\Gamma \vdash MN : T_2}$$

  due sotto alberi e dunque due ipotesi. Per ipotesi induttiva $M \in RED_{T_1 \to T_2}(I_1)$ e $N \in RED_{T_1}(I_2)$
  Bisogna dimostrare che $MN \in RED_{T_2}$ ma essa è ovvia per T_1, T_2 e la definizione di riducibile $T_1 \to T_2 (\star)$.
- Caso

  $$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T}$$

Bisogna dimostrare che $x \in RED_T$. Bisogna dimostrare dunque che se passo in input qualcosa, il valore resti riducibile. Ovvio per il Lemma $CR\,4$ (una qualunque variabile riducibile per qualsiasi tipo). La sigla CR sta per "candidato di riducibilità", ovvero un insieme di elementi che potrebbe essere riducibile ma che lì in mezzo, effettivamente, ci sono elementi che lo sono.

- Caso

$$\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x.\, M : T_1 \to T_2}$$

per ipotesi induttiva $M \in RED_{T_2}$. Bisogna dimostrare che $\lambda x.\, M \in RED_{T_1 \to T_2}$ ovvero $\exists \Gamma.\, \lambda x.\, M : T_1 \to T_2$ (ovvio perché è nell'ipotesi) $\wedge \forall N \in RED_{T_1}.\, (\lambda x.\, M)N \in RED_{T_2}$.
Per dimostrare ciò serve:
- (OK) Caso particolare $CR\,3$: $(\lambda x.\, M)N \to t \in RED_{T_2} \implies (\lambda x.\, M)N \in RED_{T_2}$
- (FALLISCE) $M \in RED_{T_2} \implies M[N/x] \in RED_{T_2}$

Piano di lavoro 2

1. Definito RED_T.
2. si identificano le proprietà dei candidati $CR\,1$ - $CR\,3$.
3. si dimostrano $CR\,1$ - $CR\,3$ per RED_T.
4. si generalizza l'enunciato che i ben tipati sono sotto insieme dei riducibili $WT \subseteq RED_T$ usando $CR\,1$ - $CR\,3$.
5. si dimostra che i riducibili sono fortemente normalizzanti $RED_T \subseteq SN$ usando $CR\,1$.

Definizione (termine neutrale)
Un termine M è neutrale quando i redex di $N[M/x]$ sono redex di M o di N (non ne ho creati di nuovi).

È molto generale perché questa definizione vale anche per altre tipologie, come ad esempio, le coppie. Un termine neutrale è una λ applicazione. Ad esempio, (MN) è neutrale perché se $R[(MN)/x]$ contiene un redex della forma $(\lambda z.\, U)W$ allora non può essere stato creato perché vuol dire che prima c'era qualcosa del tipo $(\lambda z.\, U)x$ ma quindi vuol dire che non è stato creato perché prima era un redex. Oppure anche qualcosa del tipo xR non avrebbe creato un redex nuovo.

Teorema
M è neutrale sse M non è una λ astrazione.

Un termine non neutrale è $\lambda x.\, M$, poiché $(zy)[(\lambda x.\, M)/z] = (\lambda x.\, M)y$ che è un redex ma $(\lambda x.\, M)y \notin \lambda x.\, M$ e $(\lambda x.\, M)y \notin (zy)$ quindi ho creato un nuovo redex!

Candidati di reducibilità

- $CR\,1 : RED_T \subseteq SN$
- $CR\,2 : \forall M, N.\, M \in RED_T \wedge M \to_\beta^\star N \implies N \in RED_T$
- $CR\,3 : \forall M.\, (M \text{ neutrale} \wedge (\forall N.\, M \to_\beta N \implies N \in RED_T) \implies M \in RED_T)$

Il fatto che sto a dire che $M \to_\beta N \wedge N \in RED_T$ sto dicendo che se ho M che è ben tipato, una volta ridotto a N, anch'esso è ben tipato. Questa proprietà non vale nei sistemi di tipo, dunque non è una proprietà generale.

- $CR\,4$ (Corollario di $CR\,3$): $\forall T.\, x \in RED_T$ e si ha una dimostrazione ovvia per $CR\,3$: x è neutrale perché non è una λ astrazione e poi vedo che in qualunque modo lo muovo, ho comunque qualcosa riducibile, ma x essendo una variabile normale, non posso muoverla in alcun modo! Banalmente x è riducibile su qualunque termine.

Teorema
$\forall T.\, CR\,1(T) \land CR\,2(T) \land CR\,3(T)$

Dimostrazione
Si dimostra per induzione mutua sui 3 candidati di riducibilità su T.

Caso A:
devo dimostrare
- $CR\,1(A) : RED_A \subseteq SN$ ovvero $\{M | \exists \Gamma.\, \Gamma \vdash M : A \land M \in SN\} \subseteq SN$ (ovvio)
- $CR\,2(A) : \forall M, N.\, M \in RED_A \land M \to_\beta^\star N \implies N \in RED_A$
Proprietà ovvia per la proprietà di fortemente normalizzante. Se ci fosse un cammino infinito da N allora ci sarebbe anche da M.
- $CR\,3(A) : \forall M.\, (M \text{ neutrale} \land (\forall N.\, M \to_\beta N \implies N \in RED_A) \implies M \in RED_A)$ (ovvio)
Considero tutti i modi di poter fare un passo da M. Se tutti i passi sono riducibili (e dunque fortemente normalizzanti dato che $RED_A \subseteq SN$) vuol dire che da qualunque passo io vada, M non avrà comunque un cammino infinito.
Caso $T_1 \to T_2$:
per ipotesi induttiva, $CR\,1(T_1) \land CR\,2(T_1) \land CR\,3(T_1)$ e $CR\,1(T_2) \land CR\,2(T_2) \land CR\,3(T_2)$.
bisogna dimostrare
- $CR\,1(T_1 \to T_2) : RED_{T_1 \to T_2} \subseteq SN$ ovvero $\{M | \exists \Gamma.\, \Gamma \vdash M : T_1 \to T_2 \land \forall N \in RED_{T_1}.\, MN \in RED_{T_2}\} \subseteq SN$
Poiché per ipotesi induttiva vale $CR\,3(T_1)$ allora vale anche $CR\,4(T_1)$, ovvero $x \in RED_{T_1}$.
Fisso M t.c. $\exists \Gamma, \Gamma \vdash M : T_1 \to T_2$ e $H = (\forall N \in RED_{T_1}.\, MN \in RED_{T_2})$ e dimostro $M \in SN$.
Da H e da $x \in RED_{T_1}$ ho $Mx \in RED_{T_2}$.
Per ipotesi induttiva $CR\,1(T_2)$ si ha $Mx \in SN$ quindi $M \in SN$ ed è ovvio perché se $M \to \cdots$ allora anche $Mx \to \cdots$, ed è impossibile.
- $CR\,2(T_1 \to T_2) : \forall M, N.\, M \in RED_{T_1 \to T_2} \land M \to_\beta^\star N \implies N \in RED_{T_1 \to T_2}$.
Fisso M, N t.c. $M \in RED_{T_1 \to T_2}$ (H_1) e $M \to_\beta^\star N$ (H_2). Dimostro che
$N \in RED_{T_1 \to T_2} = \{U | \exists \Gamma.\, \Gamma \vdash U : T_1 \to T_2 \land \forall W \in RED_{T_1}.\, UW \in RED_{T_2}\}$ ovvero
1. dimostro che $\exists \Gamma.\, \Gamma \vdash N : T_1 \to T_2$.
Preso H_1 e quindi da H_2 e *subject reduction* (se è ben tipato, rimane ben tipato, una proprietà dei sistemi di tipo) è ovvio.
2. dimostro che $\forall W \in RED_{T_1}.\, NW \in RED_{T_2}$.
Fisso $W \in RED_{T_1}$ e dimostro che $NW \in RED_{T_2}$.
Per ipotesi induttiva vale $CR\,2(T_2)$ ovvero $\forall V, V'.\, V \in RED_{T_1}.\, V \to V' \implies V' \in RED_{T_2}$ e per H_1, $MW \in RED_{T_2}$ poiché $M \to_\beta^\star N$ per H_2 si ha $MW \to_\beta^\star NW$, quindi $NW \in RED_{T_2}$
- $CR\,3(T_1 \to T_2) : \forall M.\, (M \text{ neutrale} \land (\forall N.\, M \to_\beta N \implies N \in RED_{T_1 \to T_2}) \implies M \in RED_{T_1 \to T_2})$
Fisso M t.c. M è neutrale (H_1) e $\forall N.\, M \to_\beta N \implies N \in RED_{T_1 \to T_2}$ (H_2). Dimostro che $M \in RED_{T_1 \to T_2}$ ovvero
1. dimostro che $\exists \Gamma.\, \Gamma \vdash N : T_1 \to T_2$ (OMESSA, non facile, usa l'ipotesi di neutralità)
Esempio $(\lambda x.\, y)(\lambda z.\, zz)$ non è neutrale e non è tipato, ma $(\lambda x.\, y)(\lambda z.\, zz) \to_\beta y$ è ben tipato.

2. dimostro $\forall U.\, U \in RED_{T_1} \implies MU \in RED_{T_2}$.
Fisso U t.c. $U \in RED_{T_1}$ (H_3) e dimostro che $MU \in RED_{T_2}$.
> Considerando un albero finitely branching T (= un nodo ha un numero finito di figli) e senza rami infiniti.
Usando l'**assioma della scelta** vale che $\exists \nu(T) \in \mathbb{N}.\ \text{nessun ramo è più lungo di } \nu(T)$
Si procede su $\nu(T)$ per dimostrare che $MU \in RED_{T_2}$
- caso base: $\nu(U) = 0$ ovvero $U \nrightarrow$
usando l'ipotesi induttiva su $CR\,3(T_2)$ mi riduco a dimostrare che $MU \to V \implies V \in RED_{T_2}$.
Sia V t.c. $MU \to_\beta V$ ci sono due possibilità:
1.

$$\frac{M \to_\beta M'}{MU \to_\beta M'U = V}$$

per H_2, $M' \in RED_{T_1 \to T_2}$ quindi $M'U \in RED_{T_2}$ quindi $V \in RED_{T_2}$
2. $MU \to_\beta V$ in quanto M è una λ astrazione, ma impossibile per H_1, che dice come M sia neutrale, pertanto non può essere una λ astrazione.
- caso induttivo: $\nu(U) = n + 1$ e sia $U \to_\beta U'$ t.c. $\nu(U') = n$
Per ipotesi induttiva, se $MU' \in RED_{T_2}$ (II). Bisogna dimostrare che $MU \in RED_{T_2}$
Per $CR\,3(T_2)$ mi riduco a dimostrare che $MU \to V \implies V \in RED_{T_2}$. Ci son 3 casi, con i primi due uguali a quelli di prima.
3.

$$\frac{U \to_\beta U'}{MU \to_\beta MU'}$$

per (II), $MU' \in RED_{T_2}$

q.e.d.

Ricapitolando:
Il teorema si può enunciare come

$$\forall \Gamma, M, T.\, \text{dato } \{N_i | (x_i : T_i) \in \Gamma, N_i \in RED_{T_i}\} \text{ si ha } \Gamma \vdash M : T \implies M[\vec{N_i}/\vec{x_i}] \in RED_T$$

Il corollario è

$$\forall \Gamma, M, T.\, \Gamma \vdash M : T \implies M \in RED_T$$

Dimostrazione del corollario
Per avere M in $M[N_i/x_i]$ bisogna scegliere $N_i = x_i$. Ma esso dev'essere RED_{T_i} ed è vero per $CR\,4$.

Dimostrazione del teorema per induzione

- Caso

$$\frac{(x_j : T_j) \in \Gamma}{\Gamma \vdash x_j : T_j}$$

Bisogna dimostrare che $x_j[\vec{N}_i/\vec{x}_i] \in RED_{T_j} = N_j \in RED_{T_j}$ ed è ovvio perché sono stati scelti gli N_i come riducibili di RED_{T_i}.

- Caso

$$\frac{\Gamma \vdash M : T_1 \to T_2 \qquad \Gamma \vdash N : T_2}{\Gamma \vdash MN : T_2}$$

Per ipotesi induttive $M[\vec{N}_i/\vec{x}_i] \in RED_{T_1 \to T_2}$ (II_1) e $N[\vec{N}_i/\vec{x}_i] \in RED_{T_1}$ (II_2).
Bisogna dimostrare che $(MN)[\vec{N}i/\vec{x}_i] \in RED_{T_2} = M[\vec{N}i/\vec{x}_i]N[\vec{N}i/\vec{x}_i]$ che è ovvio per $(II_1), (II_2)$ e definizione di $RED_{T_1 \to T_2}$.

- Caso

$$\frac{\Gamma, x_{n+1} : T_{n+1} \vdash M : T}{\Gamma \vdash \lambda x_{n+1}.\, M : T_{n+1} \to T}$$

dove $n = |\Gamma|$.
Per ipotesi induttiva $\forall N_{n+1} \in RED_{T_{n+1}} : M[\vec{N}_i/\vec{x}_i] \in RED_T$.
Bisogna dimostrare che $(\lambda x_{n+1}.\, M)[\vec{N}_i/\vec{x}_i] \in T_{n+1} \to T$ ovvero

1. $\exists \Gamma.\, \Gamma \vdash (\lambda x_{n+1}.\, M)[\vec{N}_i/\vec{x}_i] : T_{n+1} \to T$ che è ovvio da ipotesi $\Gamma \vdash \lambda x_{n+1}.\, M : T_{n+1} \to T$, per $N_i : T_i$ e per un lemma
2. $\forall N_{n+1} \in RED_{T_{n+1}}.\, (\lambda x_{n+1}.\, M)[\vec{N}_i/\vec{x}_i] \in RED_{T_{n+1}}$.
   Fissato $N_{n+1} \in RED_{T_{n+1}}$ (H) si può sfruttare $CR\ 3$ poiché si ha neutrale $(\lambda x_{n+1}.\, M)[\vec{N}_i/\vec{x}_i]N_{n+1}$.
   (Sfruttare $CR\ 3$ vuol dire considerare tutti i possibili casi per la riduzione: se si riducesse il redex si avrebbe esattamente l'ipotesi induttiva perché si otterrebbe $M[\vec{N}_i/\vec{x}_i] \in RED_T$).
   Poiché $N_{n+1} \in RED_{T_{n+1}}$ per (H) e poiché
   $M[\vec{N}_i/\vec{x}_i] = M[\vec{N}_i/\vec{x}_i; x_{n+1}/x_{n+1}] \in RED_{T_{n+1}} \implies \exists \nu(N_{n+1}), \nu(M[\vec{N}_i/\vec{x}_i])$.
   Per dimostrare che $\forall M, \vec{N}_i, N_{n+1}$ vale $\lambda x_{n+1}.\, M[\vec{N}_i/\vec{x}_i]N_{n+1} \to^\star U \in RED_T$ si procede per induzione su $\nu(N_{n+1}) + \nu(M[\vec{N}_i/\vec{x}_i])$.

- Caso 2.1

$$\frac{M[\vec{N}_i/\vec{x}_i] \to W}{(\lambda x_{n+1}.\, M)[\vec{N}_i/\vec{x}_i]N_{n+1} \to_\beta (\lambda x_{n+1}.\, W)N_{n+1}}$$

Poiché $M[\vec{N}_i/\vec{x}_i] \to_\beta W$ si ha $\nu(M[\vec{N}_i/\vec{x}_i]) = \nu(W) + 1$ e si conclude per l'ipotesi induttiva.

- Caso 2.2

$$\frac{N_{n+1} \to W}{(\lambda x_{n+1}.\, M)[\vec{N}_i/\vec{x}_i]N_{n+1} \to_\beta (\lambda x_{n+1}.\, M)[\vec{N}_i/\vec{x}_i]W}$$

Analogo a quello sopra perché $\nu(N_{n+1}) = \nu(W) + 1$ e si conclude per l'ipotesi induttiva.

- Caso 2.3

$$\frac{}{(\lambda x_{n+1}.\, M)[\vec{N}_i/\vec{x}_i]N_{n+1} \to_\beta M[\vec{N}_i/\vec{x}_i; N_{n+1}/x_{n+1}]}$$

ed è valida per II.

5. Note sul tipaggio

Nei linguaggi di programmazione vi è costrutto esplicito di ricorsione.

$$f := T : M \qquad \text{Presa una funzione dichiarata al top-level avere tipo } T \text{ e corpo } M$$

è zucchero sintattico per il costrutto

$$f := (\nu f : T.\, M) \qquad \text{termine che dichiara una funzione ricorsiva di tipo } T \text{ che, nel corpo, può richiamarsi usar}$$

- f è il nome top-level.
- ν è il binder di punto fisso.
- $(\nu f : T.\, M)$ è il corpo della funzione.

Questo che segue è usato per tipare funzioni divergenti, non presente nel λ calcolo.

$$\frac{\Gamma, f : T \vdash M : T}{\Gamma \vdash (\nu f : T.\, M) : T}$$

Ad esempio:

$$\dfrac{\dfrac{\dfrac{f : \mathbb{N} \to \mathbb{N}, x : \mathbb{N} \vdash f : \mathbb{N} \to N \qquad f : \mathbb{N} \to \mathbb{N}, x : \mathbb{N} \vdash x : \mathbb{N}}{f : \mathbb{N} \to \mathbb{N}, x : \mathbb{N} \vdash fx : \mathbb{N}}}{f : \mathbb{N} \to \mathbb{N} \vdash \lambda x : \mathbb{N}.\, fx : \mathbb{N} \to \mathbb{N}}}{\vdash (\nu f : \mathbb{N} \to \mathbb{N}.\, \lambda x : \mathbb{N}.\, fx) : \mathbb{N} \to \mathbb{N}}$$

tale regola implica la non consistenza del sistema logico.

$$\frac{\vdash \nu f : T \to \bot \,.\, \lambda x : T.\, fx : T \to \bot}{\vdash (\nu f : T \to \bot \,.\, \lambda x : T.\, fx) I : \bot}$$

Un corollario sulle osservazioni precedenti:

$$Y = \lambda f.\, (\lambda x.\, f(xx))(\lambda x.\, f(xx))$$

non è tipabile.

6. Logica proposizionale del secondo ordine

$$F ::= \cdots | \forall A.\, F$$

A è una variabile proposizionale, qualcosa che può essere vero/falso.
(nella logica del primo ordine si ha $F ::= \cdots | \forall x.\, F | P^n(x_1, \ldots, x_n)$ dove x è una variabile di termine, elemento del dominio del discorso e $P^n(x_1, \ldots, x_n)$ è un predicato, come l'essere pari o dispari).

Un esempio di logica proposizionale del primo ordine è

$$\forall x.\, x \leq x$$

Un esempio di logica proposizionale del secondo ordine è

$$\forall A, B, C.\, (A \wedge B \to C) \to \neg C \to \neg (A \wedge B)$$

Le regole qui sono

$$\frac{\Gamma \vdash F[B/A]}{\Gamma \vdash \forall A.\, F} \forall_i \qquad \text{dove } B \text{ è una variabile fresca non usata in } \Gamma$$

$$\frac{\Gamma \vdash \forall A.\, F}{\Gamma \vdash F[G/A]} \forall_e$$

$$\frac{\dfrac{\dfrac{(\forall A.\, (A \to B)) \in \forall A.\, (A \to B)}{\forall A.\, (A \to B) \vdash \forall A.\, (A \to B)} \forall_e}{\forall A.\, (A \to B) \vdash (D \to D) \to B} \forall_e}{\forall A.\, (A \to B) \vdash \forall C.\, (C \to C) \to B} \forall_i$$

Polimorfismo uniforme o generico o template

$$T ::= \cdots | \forall A.\, T$$

o anche, come usata in molti linguaggi di programmazione

$$T ::= \cdots | \langle A \rangle T$$

$$\frac{\Gamma \vdash M : T[B/A]}{\Gamma \vdash M : \forall A.\, T} \forall_i$$

$$\frac{\Gamma \vdash M : \forall A.\, T}{\Gamma \vdash M : T[T'/A]} \forall_e$$

che poi, quest'ultimo, $M : T[T'/A]$ è $M\langle T' \rangle$

Questo, alla fine di tutto, è il $\forall$ della logica proposizionale del secondo ordine.

Esistenza

$$F ::= \cdots | \exists A.\, F$$

essendo al secondo ordine, vuol dire che A è una variabile proposizionale.

$$\frac{\Gamma \vdash F[G/A]}{\Gamma \vdash \exists A.\, F} \exists_i$$

G è completamente variabile.

$$\frac{\Gamma \vdash \exists A.\, F \qquad \Gamma, F[B/A] \vdash G}{\Gamma \vdash G}\exists_e$$

Nel medesimo modo fatto per $\forall$ si prende una variabile nuova di cui non si sa assolutamente nulla (fresca: non usata in Γ e G).

La medesima cosa ma per Curry-Howard sarà

$$T ::= \cdots \mid \exists A.\, T$$

$$t ::= \cdots \mid \text{open } t \text{ as } x \text{ in } t$$

che nei linguaggi di programmazione lo si può trovare come "interfaccia" o "tipo di dato astratto" o "classe" o "mixin" o "modulo" o "trait".

Tipo di dato astratto

Un tipo di dato astratto è un tipo per il quale non viene data l'implementazione ma solo la sua interfaccia come insieme di segnature di funzioni.
Esempio di stack di interi con tipo di dato astratto

```
module stack
        type stack
        fn empty : stack
        fn push : stack × Z -> stack
        fn pop : stack -> 1 + Z × stack
end

---

open stack

(λx.λs.
        push ⟨x, s⟩
        ) 2 empty

---

module instance stack
        type stack = array⟨Z⟩ × N
        fn empty = ⟨[], 0⟩
        fn push ⟨x, s⟩ = ⟨s.1[s.2<-x], x.2+1⟩
end

---
```

```
// Curry-Howard ottenuto
∃ stack.stack × (stack × Z -> stack) × (stack -> 1 + Z × stack)
open M as f in ... f.1 ... f.2.1 ... f.2.2 ...
                                    empty     push      pop
```

$$\frac{\Gamma \vdash M : F[G/A]}{\Gamma \vdash M : \exists A.\, F}$$

nei linguaggi di programmazione questo sarebbe la "module instance" sopra.

```
module instance
        type A = F
        M:F[G/A]
end
```

$$\frac{\Gamma \vdash M : \exists A.\, F \qquad \Gamma, f : F[B/A] \vdash N : G}{\Gamma \vdash \text{open } M \text{ as } f \text{ in } N : G}$$

in questo caso si ha che il primo implementa M senza conoscere N, mentre il secondo implementa N senza conoscere M. La parte sotto fa il lavoro del linker.

Questo si dimostra con varie regole:

$$\frac{\dfrac{\Gamma \vdash M : F[T/A]}{\Gamma \vdash M}\exists_i \qquad \Gamma, f : F[B/A] \vdash N : G}{\Gamma \vdash \text{open } M \text{ as } f \text{ in } N : G}\exists_e \to \frac{}{\Gamma \vdash N[M/f] : G}$$

da una parte abbiamo implementazione del modulo + linker e dall'altra il codice senza usare moduli.

7. λx.xx

$\lambda x.\, xx$ è non tipato nel λ calcolo tipato semplice.
$\lambda x.\, xx$ è tipato nel λ calcolo con polimorfismo uniforme.

Esempio d'uso con funzione identità:

$$\dfrac{\dfrac{\dfrac{(x : \forall A.\, A \to A) \in \Gamma}{x : \forall A.\, A \to A \vdash x : \forall A.\, A \to A}}{x : \forall A.\, A \to A \vdash x : (\forall A.\, A \to A) \to (\forall A.\, A \to A)}\forall_e \qquad \dfrac{(x : \forall A.\, A \to A) \in \Gamma}{x : \forall A : A \to A \vdash x : \forall A.\, A \to A}}{x : \forall A.\, A \to A \vdash xx : \forall A.\, A \to A}$$
$$\vdash \lambda x.\, xx : (\forall A.\, A \to A) \to (\forall A.\, A \to A)$$

1. Il λ calcolo con polimorfismo uniforme è un'approssimazione migliore della proprietà della normalizzazione forte.
2. $\lambda x.\, xx$ mostra che non è sempre possibile monomorfizzare programmi che usano il polimorfismo $\implies$ abbiamo incrementato la potenza espressiva.

8. Sistemi di scrittura astratti

Un *Abstract Rewriting System* (ARS) è una coppia $(A, \to)$ t.c.

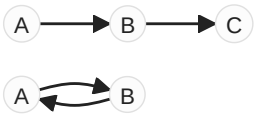
1. $A \neq \emptyset$ ed è chiamato *insieme degli stati* (o delle configurazioni).
2. $\to \subseteq A \times A$ ed è chiamata *relazione di transazione*.

- Nel caso λ calcolo come ARS si ha $(\Pi, \to_\beta)$ con $\Pi =$ insieme dei λ termini.
- Nel caso delle macchine di Turing (A, Q, q_0, q_f, δ) si può come vedere come ARS $(A^{\mathbb{Z}} \times \mathbb{Z} \times Q, \to)$.
- Nel caso di un generico linguaggio di programmazione funzionale si ha la medesima cosa del λ calcolo.
- Nel caso di un generico linguaggio di programmazione imperativo si sceglie la configurazione in cui ha il linguaggio, che in questo caso, in base al suo livello (alto o basso), potrebbero essere le celle dei registri $(\mathbb{R} \times \mathbb{Z}^{\mathbb{N}}, \to)$ dove $\mathbb{Z}$ sono i registri, $\mathbb{Z}$ è la memoria, $\to$ è fetch-decode-execute; in quelli ad alto livello diviene più complesso perché bisogna considerare stack, heap, IP, IR, etc.

> Dominio sopra e codominio sotto $(\mathbb{Z}^{\mathbb{N}})$.

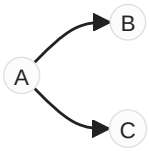
Un ARS $(A, \to)$ è deterministico quando $\forall q_1, q_2, q_2' \in A.\, q_1 \to q_2 \land q_1 \to q_2' \implies q_2 = q_2'$

Esempi deterministici:

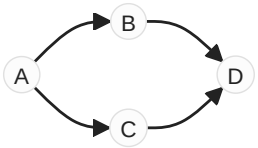


Esempi non deterministici. Un esempio sono i programmi concorrenti in cui, in base alla velocità, si avrà uno stato finale che può differire.

Caso 1.

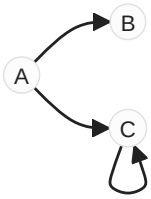


Caso 2.
Coppia critica, vi sono due cammini divergenti che però congiungono in un unico punto deterministico. Questa si chiama *confluenza*.

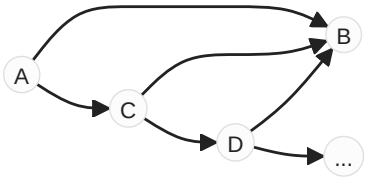


Caso 3.
Qui c'è anche il caso in cui si ha 1 solo stato possibile ma non è manco sempre raggiunto, quindi se il

compilatore sbaglia a scegliere la biforcazione la prima volta, allora potrebbe non raggiungere mai la fine.



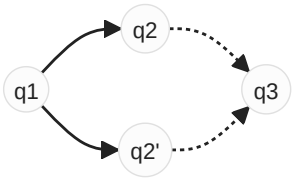
Caso 4.
Qui potrebbe non finire mai l'esecuzione però potrebbe anche finire nel caso di `C --> B` o `D --> B`.



La scelta di lasciare al compilatore la scelta della semantica in modo dunque non deterministico è perché il compilatore si può adattare alle varie architetture.

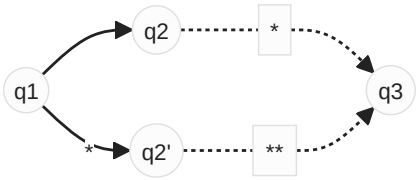
Tipi di confluenza

- Confluenza locale



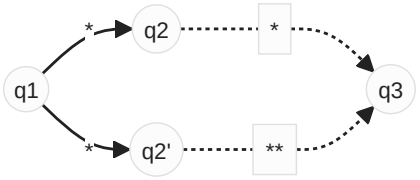
$\forall q_1, q_2, q_2'. q_1 \to q_2 \land q_1 \to q_2'$
$\exists q_3. q_2 \to q_3 \land q_2' \to q_3$
Che è uguale a dire
$\forall q_1, q_2, q_2'. q_2 \leftarrow q_1 \to q_2' \implies \exists q_3. q_2 \to q_3 \leftarrow q_2'$

- Semiconfluenza



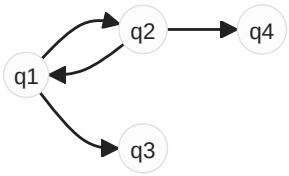
> `*` e `**` rappresentano la stessa cosa, però non me lo fa fare.

- Confluenza

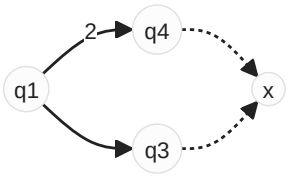


$$\text{Confluenza} \implies \text{Semiconfluenza} \implies \text{Confluenza locale}$$

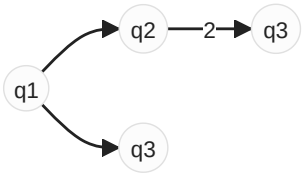
$$\text{Confluenza locale} \not\implies \text{Semiconfluenza}$$



che può essere



oppure

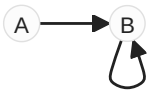


Il controesempio non è fortemente normalizzante.

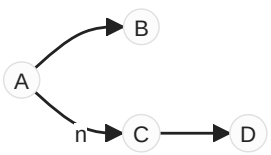
Teorema

$$\text{Fortemente normalizzate} \land \text{Confluente locale} \implies \text{Semiconfluente}$$

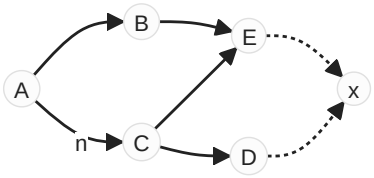
Dimostrazione (errata) di Confluenza locale => Semiconfluenza
Caso 0 passi.



Caso $n + 1$ passi.



Per ipotesi induttiva vi è uno stato tra B e C in cui si va per un numero determinato di passi. La chiusura del grafo però è possibile? Cioè, esiste un modo per avere

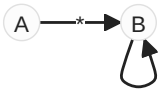


? No.

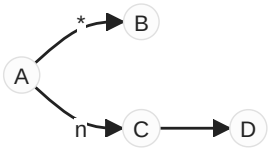
Teorema

$$\text{Semiconfluenza} \implies \text{Confluenza}$$

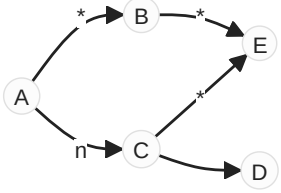
Caso 0.



Caso $n+1$.



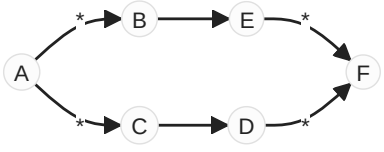
L'ipotesi induttiva mi dice che



e che quindi si chiude.
`aggiornare da slide`

Teorema

$$\text{Confluenza} \implies \text{Unicità delle forme normali}$$

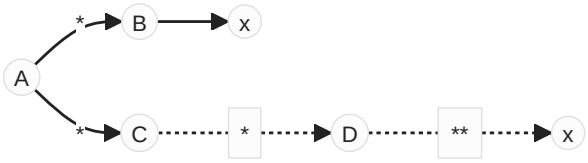


ma in realtà i passi da `E->F` e `D->F` sono 0 e dunque i nodi `E` e `D` sono uguali!
Se ci sono strade alternative arrivano alla stessa forma normale.

Teorema

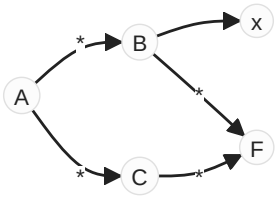
$$\text{Confluenza} \implies \text{Safety}$$

Definizione di Safety



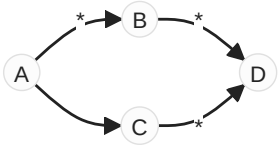
Ovvero si assume di aver fatto un certo numero di passi per arrivare alla forma normale.

Dimostrazione



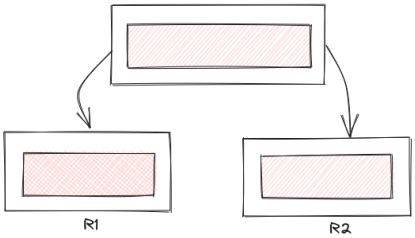
Ma in realtà `B->F` e `C->F` fanno 0 passi e dunque `B=C`.

Teorema
Il λ calcolo è semiconfluente.

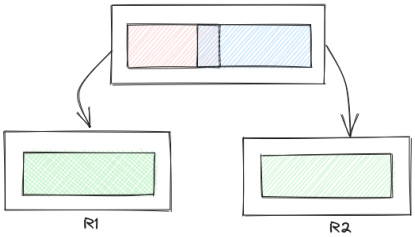


Fonti del non determinismo

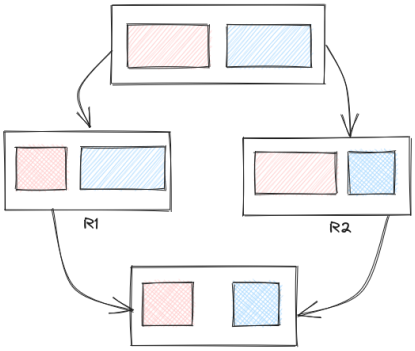
1. Un redex ha due ridotti: non avviene nel λ calcolo, ma nei linguaggi potrebbe. Ad esempio `flip()` può essere sostituita da `0` o `1`.



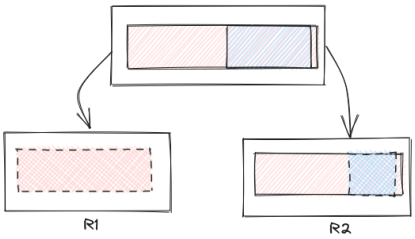
2. Due redex possono essere overlapping ma non uno strettamente incluso nell'altro: non avviene nel λ calcolo.



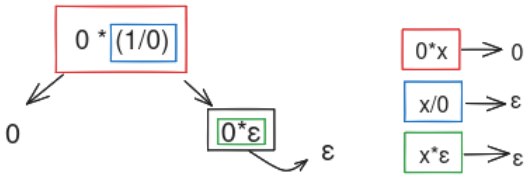
3. Redex non overlapping o paralleli: c'è nel λ calcolo.



4. Un redex interamente contenuto nell'altro: c'è nel λ calcolo. (Di solito si perde confluenza)

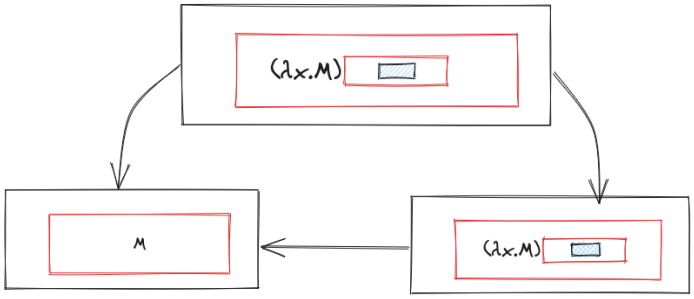


Ad esempio si può decidere cosa tornare in base a quale parte dell'espressione si fa prima il parsing.



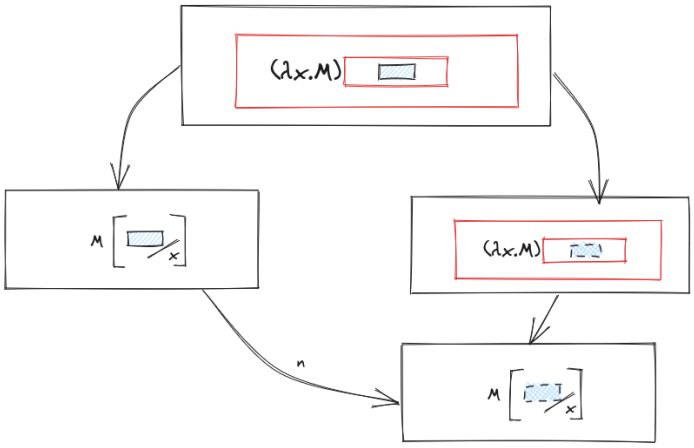
Caso 4 nel λ calcolo

1. Con $x \notin FV(M)$ si ha



Ma con la call-by-value si rischia di divergere quando non necessario.

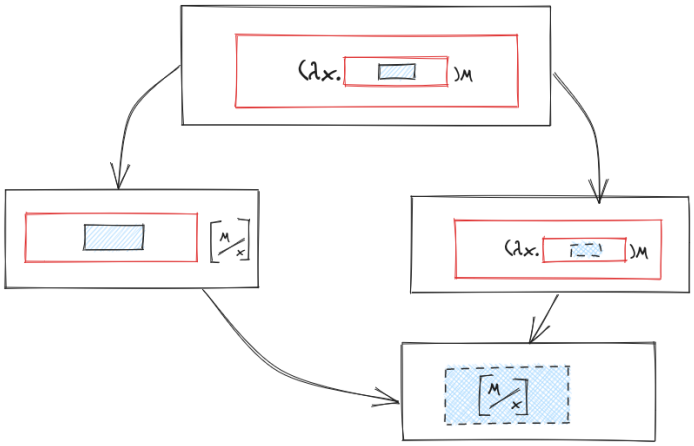
2. Con $x \in FV(M)$ si ha



La call-by-value è la strada più corta.

3. Usando il lemma

Se $M \to_\beta N$ allora $M[R/x] \to_\beta N[R/x]$



9. Logica e complessità computazionale

Il concetto di problema computazionale è correlato alla soluzione di una funzione. Dato come assunzione che dominio e codominio siano stringhe binarie perché qualsiasi altra struttura è codificata in binario, si pososno avere, ad esempio:

- Numeri $\mathbb{N}$

$$34 \mapsto 10010 =: \lfloor 34 \rfloor$$

- Tuple

$$\langle x, y \rangle \mapsto \lfloor x \rfloor \# \lfloor y \rfloor \mapsto$$

  questo si ha facendo "parsing" disambiguando dove e quando inizia prima/seconda stringa, eg $0 \mapsto 00, 1 \mapsto 11, \# \mapsto 01$.
- Grafi
  Si fanno matrice di adiacenze, la quale è una tupla di tupla.

Dunque queste funzioni esprimono un determinato predicato perché definite come $f : \mathbb{B} \to \{0, 1\}$ in cui è falso / vero. Si può vedere questa funzione come sotto insieme delle stringhe per f per cui è uguale a 1, ed esso è chiamato *linguaggio*.

$$\mathscr{L} \in \mathbb{B}$$
$$\mathscr{L}_f = \{b \in \mathbb{B} | f(b) = 1\}$$

Il problema è che non si ha idea di come venga ispezionato il grafo o il perché venga ad un valore della funzione venga assegnato 1. Questa è detta visione estensionale (o dichiarativa) perché si fa riferimento al fattore insiemistico. Si descrive un problema, non un algoritmo per la risoluzione di tale problema.

Bisogna pensare f dividendola in determinati passi usando trasformazioni $\to^{\mathcal{A}}$. Diciamo che $\mathcal{A}$ calcola f e la sua semantica $[\![\mathcal{A}]\!] = f$.
Si usa il tempo per misurare i passi elementari. Si astrae il tempo di calcolo perché bisogna creare una teoria, cancellando dunque dettagli, eliminando i secondi in 'sto caso. Il tempo impiegato su tale algoritmo con un dato input x è $\mathrm{TIME}_{\mathcal{A}}(x)$.
Nello spazio si contano il numero di passi che occorrono in memoria: su un dato input x si calcola come $\mathrm{SPACE}_{\mathcal{A}}(x)$. Non si fa una somma di tutte le celle ma solo quelle necessarie da un passo all'altro, cancellando e/o scrivendoci sopra. Non si conta lo spazio per l'input e l'output della funzione.

Le classi concrete che ne derivano si basano in base agli algoritmi che vengono usati per tali linguaggi. Presa una funzione $g : \mathbb{N} \to \mathbb{N}$ si definisco delle classi concrete (sono insiemi di linguaggi):

$$\mathrm{DTIME}(g) = \{L \subseteq \mathbb{B} | \exists A. [\![\mathcal{A}]\!] = L \wedge \forall x. \mathrm{TIME}_{\mathcal{A}}(x) \leq g(|x|)\}$$
$$\mathrm{DSPACE}(g) = \{L \subseteq \mathbb{B} | \exists A. [\![\mathcal{A}]\!] = L \wedge \forall x. \mathrm{SPACE}_{\mathcal{A}}(x) \leq g(|x|)\}$$

Mentre le classi di complessità:

$$P = \bigcup_{g \in \mathrm{POLY}} \mathrm{DTIME}(g)$$

Se è in P vuol dire che si risolve efficientemente. Se un algoritmo in P va più veloce in un altro computer, resta comunque in P.
$\mathrm{DTIME}(g) \neq O(g)$ però ci si può arrivare considerando che g può essere sufficientemente grande.

$$\mathrm{PSPACE} = \bigcup_{g \in \mathrm{POLY}} \mathrm{DSPACE}(g)$$

$$L = \bigcup_{g \in \mathrm{LOGA}} \mathrm{DSPACE}(g)$$

Ahimè sono ristretti ad una funzione g ma è una descrizione troppo precisa che restringe e dunque si estende un po' usando la classe polinomiale POLY (quindi in maniera efficiente con buone proprietà).
PSPACE non è ottimale, però rimane buona in alcuni contesti. L è efficiente.

La "D" sta a significare che è deterministica.

Se una funzione non è deterministica si definisce $f(x) = 1$ se esiste almeno un nodo nel cammino tale che abbia questo valore. Se si ha che l'algoritmo porta a tale risultato dciamo che $\mathcal{A}$ decide f.

$$\mathrm{NDTIME}(g) = \{L \subseteq \mathbb{B} | \exists \mathcal{A} \text{ non deterministico. } [\![\mathcal{A}]\!] = L \wedge \forall x. \, \mathrm{TIME}_{\mathcal{A}}(x) \leq g(|x|)\}$$

$$NP = \bigcup_{g \in \mathrm{POLY}} \mathrm{NDTIME}(g)$$

La classe NP dei linguaggi L per cui si può creare una data computazione che dice se è accettabile o meno, tutto in tempo polinomiale. Verificare è più semplice di creare.

$$L \subseteq P \subseteq NP \subseteq \mathrm{PSPACE}$$

$$L \subset \mathrm{PSPACE}$$

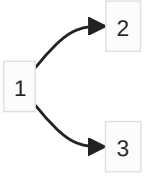
Complessità descrittiva

Applicato al caso d'esempio sui grafi si può creare un predicato $E(x, y)$ nel vocabolario della logica descrittiva che ritorna 1 nel caso esista un arco che va da x a y.

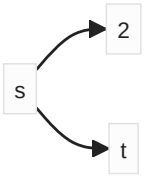
Esempio
$\mathcal{A}_3 = \{1, 2, 3\}$
$v = \{E(-,-), =(-,-), s, t\}$

$$I : \begin{cases} E \mapsto \{(1,2),(1,3)\} \\ = \mapsto \{(1,1),(2,2),(3,3)\} \\ s \mapsto 1 \\ t \mapsto 3 \end{cases}$$

$(\mathcal{A}_3, I) \models \exists x. \, E(s, x)$



che si può vedere anche come



questa struttura è data dalla teoria dei modelli.

$(\mathcal{A}_n, I) \models \phi$
$\mathscr{L}_\phi = \{(\mathcal{A}_n, I) | (\mathcal{A}_n, I) \models \phi\}$
Tupla di insieme + interpretazione non è proprio semplice, dunque si possono esprimere come stringhe binarie.

Interpretazioni di stringhe

Prendiamo un generico vocabolario fatto da m simboli di predicati P e k simboli di funzione f . Una qualunque interpretazione è data da stringa $\mathrm{bin}^n(I) \in \mathbb{B}$ dove

$$\mathrm{bin}^n(I) = \mathrm{bin}^n(P_1) \cdots \mathrm{bin}^n(P_m) \mathrm{bin}^n(f_1) \cdots \mathrm{bin}^n(f_k)$$

la stringa binaria è composta in modo tale che il carattere è 1 se la tupla i-esima fa parte dell'interpretazione.
La stringa associata a P_i ha lunghezza $n^{ar(P_i)}$ e $ar(P_i)$ è l'arietà (numero di argomenti). Specifica se la tupla fa parte di $(P_i)_I$.
La stringa associata a f_i sarà la sua interpretazione espressa come stringa binaria e dunque con lunghezza a $\lceil \log_2(n) \rceil$ (qual è l'elemento di $\mathcal{A}_n$).
Si chiama bin^n perché è parametrica. Serve per fare il parsing della stringa.

Prendiamo ad esempio
$\mathcal{A}_3 = \{1, 2, 3\}$

$$I : \begin{cases} E \mapsto \{(1,2)(1,3)\} \\ s \mapsto 1 \\ t \mapsto 3 \end{cases}$$

Tutte le tuple di $\mathcal{A}_3$ lunghe 2 è data da $n = 3, ar(E) = 2$
$|\mathrm{bin}^3(E)| = 3^2 = 9$

> Con ordine lessico-grafico: ordine in cui posso sempre ordinare le tuple.
> $\mathrm{bin}^3(E) = \{(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3)\}$

$\mathrm{bin}^3(E) = 011000000$

$\mathrm{bin}^3(s) = \lceil \log(3) \rceil = 2$
$\mathrm{bin}^3(s) = 01$
$\mathrm{bin}^3(t) = 11$
$\mathrm{bin}^3(I) = 0110000000111$

Formule

Si usano formule chiuse. Nel momento in cui si prendono variabili bisogna avere sue interpretazioni. Le formule sono chiuse perché non vi sono variabili libere. Ad ogni formula chiusa si può prendere un linguaggio. Con la formula F si ha

$$\text{struct}(F) = \{\text{bin}^n(I) | (\mathcal{A}_n, I) \models F\} \subseteq \mathbb{B}$$

La logica può essere vista come insieme di linguaggi. Bisogna capire se esiste una logica a tali spazi.

Logica predicativa

FO = logica predicativa chiusa.

$$FO = \{\text{struct}(F) | F \text{ è formula predicativa chiusa}\}$$

I problemi usati nella logica del primo ordine sono molto efficienti.

$$FO \subseteq L$$

Anche se da teorema si ha che

$$FO \subset L$$

Si ha un'estensione del primo + secondo ordine.

$$F ::= \ldots | X^n(t_1, \ldots, t_n) | \exists X^n . F | \forall X^n . F$$

Stiamo considerando solo formule aperte, dunque bisogna interpretare le formule usando un appoggio, con ξ che si occupa di queste relazioni.

$$(\mathcal{A}, I), \xi \models X^n(t_1, \ldots, t_n) \text{ sse } (\llbracket t_1 \rrbracket_\xi^{(\mathcal{A},I)}, \ldots, \llbracket t_n \rrbracket_\xi^{(\mathcal{A},I)}) \in \xi(X^n)$$

$$(\mathcal{A}, I), \xi \models \exists X^n . F \text{ sse } (\mathcal{A}, I), \xi[X^n := \mathcal{R}] \models F \qquad \text{per qualche } \mathcal{R} \subseteq \mathcal{A}^n$$

$$(\mathcal{A}, I), \xi \models \forall X^n . F \text{ sse } (\mathcal{A}, I), \xi[X^n := \mathcal{R}] \models F \qquad \text{per tutte } \mathcal{R} \subseteq \mathcal{A}^n$$

Preso ad esempio per il primo ordine
$F = \exists x . (P(x) \vee P(y))$
Prendo ξ perché mi serve sapere su cos'è mappato y. Variabili del primo ordine -> elementi su $\mathcal{A}$.
$(\mathcal{A}, I), \xi^? \models F$

Preso ad esempio per il secondo ordine
$G = \exists x . (P(x) \vee X(x))$
Prendo ξ perché mi serve sapere su cos'è mappato $X(x)$. Variabili del secondo ordine -> relazioni su $\mathcal{A}$.
$(\mathcal{A}, I), \xi^? \models G$

La variabile è libera però, perché se avessi
$G = \forall X . \exists x . (P(x) \vee X(x))$
bisogna capire come rendere vero tutta la formula, guardando tutte le possibili interpretazioni. Ci si appoggia comunque per quando si guarda la semantica.

Raggiungibilità di un grafo

È una proprietà che risponde se un grafo è collegato dai nodi s e t. Preso universo e interpretazione $(\mathcal{A}, I)$ con I fatto da $E(-,-), s, t$.

$$\text{struct}(\psi_{s,t}) = \{\text{bin}^n(I) | (\mathcal{A}_n, I) \text{ è un grafo dove } t \text{ è raggiungibile da } s\}$$

Non si può esprimere con la logica del primo ordine perché manca un livello di espressività per vedere insiemi di nodi. Dunque si può fare

$$\psi_{s,t} = \exists R^\star (s = t \vee (\phi_L \wedge \phi_E \wedge \phi_F))$$

$$\phi_L = \forall u. \neg R^\star(u,u) \wedge \forall v \forall w. R^\star(u,v) \wedge R^\star(v,w) \implies R^\star(u,w)$$

$$\phi_E = \forall u \forall v. (R^\star(u,v) \wedge \forall w. (\neg R^\star(u,w) \wedge \neg R^\star(w,v))) \implies E(u,v)$$

$$\phi_F = \forall u. \neg R^\star(u,s) \wedge \neg R^\star(t,u) \wedge R^\star(s,t)$$

Se c'è un cappio tra s e t la raggiungibilità è triviale. ϕ_F viene usato perché s e t devono essere, rispettivamente, primo e ultimo per forza nella catena: così viene caratterizzata la catena più piccola.

Teorema di Fagin e logica del secondo ordine esistenziale

Un problema è in NP sse quel problema è definito mediante formula del secondo ordine esistenziale.

$$\exists SO = \{\text{struct}(F) | F \text{ è una formula al second'ordine esistenziale}\}$$

$$\exists SO = NP$$

Le variabili appaiono in F dove è nel primo ordine, e questo è chiamata logica al secondo ordine esistenziale.
$\exists X^{n_1} ..... \exists X^{n_m} . F$

Esempio (non sono del secondo ordine esistenziale)
$\forall x. \forall X. (X(x))$
$\forall x. \exists X. (X(x))$

Esempio (buono)
$\exists X. \forall x. X(x)$

Questo teorema non dà soluzioni per quanto riguarda P. Si può scrivere un nuovo predicato per l'esempio del grafo sopra come

$$E^\star(x,y) \equiv x = y \vee \exists z. (E(x,z) \wedge E^\star(z,y))$$

ed è il "più piccolo" perché $E^\star$ appare sia a destra che a sinistra e dunque questa definizione è accettabile sotto alcune condizioni, in cui $E^\star$ a destra dev'essere più piccolo.
I punti fissi vengono usati per avere la logica più espressiva su P. Il minimo punto fisso della funzione F è definito come μF.
Se esiste $F(y) = y$ allora $\mu F \leq y$. Dato un insieme X si considera il $\mathcal{P}(X)$.

X^m è positiva se ogni X^m in F è in un ambito pari di negazioni.

Esempio
$F = \forall y. (X^m(x_1, \dots, x_m) \vee P(y))$ lo è perché le negazioni sono $= 0$
$F = \forall y. (\neg X^m(x_1, \dots, x_m) \vee P(y))$ non lo è perché le negazioni sono $= 1$

Per ogni X^m positiva si può associare un funzionale

$$F^I = \mathcal{P}(\mathcal{A}_n^m) \to \mathcal{P}(A_n^m)$$

e quindi una funzione che associa insiemi di tuple ad insiemi di tuple.

$$D \longmapsto \{(a_1, \dots, a_m) \in \mathcal{A}_n^m | (\mathcal{A}_n, I), \xi \models F \text{ t.c. } \xi(X^m) = D \wedge \xi(x_i) = a_i\}$$

Se la formula è positiva allora F^I è monotono, ovvero, dati y e z, $y \leq z \implies F(y) \leq F(z)$. La monotonia costituisce sicurezza nel trovare μF.

Teorema di Knaster-Tarski

Se $F : A \to A$ è monotono, allora ha un punto fisso.

$$\mu F ::= \bigcap \{Y | F(Y) \subseteq Y\}$$

che è uguale a

$$\mu F ::= \bigcup \{Y | Y \subseteq F(Y)\}$$

dato che stiamo considerando l'insieme delle parti, allora si può considerare in modo più specifico.

Se $F : \mathcal{P}(\mathcal{A}) \to \mathcal{P}(\mathcal{A})$ è monotono, allora ha un punto fisso.

$$\mu F ::= \bigcup_n F^n(\emptyset) = \emptyset \cup F(\emptyset) \cup F(F(\emptyset)) \cup F(F(F(\emptyset))) \cup F^n(\emptyset) = F^{n+1}(\emptyset)$$

Dimostrazione $\mu F ::= \bigcap \{Y | F(Y) \subseteq Y\}$

1. $\mu F = F(\mu F)$
2. Se Y è tale che $Y = F(Y)$ allora $\mu F \subseteq Y$
   si dimostra che sono inclusi fra di loro e in quel caso allora avremmo eguaglianza.
3. $\forall Y \in \{Y | F(Y) \subseteq Y\}, \mu F \subseteq Y \implies F(\mu F) \subseteq F(Y)$ per monotonicità.
   Allora $F(\mu F) \subseteq F(Y) \subseteq Y$
   Allora $F(\mu F) \subseteq \cap \{Y | F(Y) \subseteq Y\}$ ma la seconda parte è μF dunque è come fare $F(\mu F) \subseteq \mu F$
   Allora, per monotonicità, $FF(\mu F) \subseteq F(\mu F)$
   quindi $F(\mu F) \in \{Y | F(Y) \subseteq Y\}$
   allora $\cup \{Y | F(Y) \subseteq Y\} \subseteq F(\mu F) = \mu F$
4. Se $Z = F(Z)$ allora $F(Z) \subseteq Z$
   quindi $Z \in \{Y | F(Y) \subseteq Y\}$
   allora $\cup \{Y | F(Y) \subseteq Y\} \subseteq Z = \mu F$

La forma $LFP(X^m, x_1, \dots, x_m, F)$ dove si ha una formula F di tipo X^m positiva e un numero di variabili libere del primo ordine, rappresenta una logica chiamata del punto fisso molto espressiva.

Esempio del primo ordine

$$F = \forall y. (X(y) \vee X(z))$$

Logica del punto fisso

$$G = LFP(X, z, F)$$

non si prende y perché è legata. Minimo punto fisso della variabile X in F della variabile libera z.

$$FO(LFP) = \{\text{struct}(F) | F \text{ è formula predicativa con minimi punti fissi}\}$$

Teorema di Immerman-Vardi

$$FO(LFP) = P$$

La logica del primo ordine è troppo semplice per la complessità computazionale.
La logica del secondo ordine è troppo espressiva per la complessità computazionale.

$$P = NP \text{ sse } FO(LFP) = \exists SO$$