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 risolverebbe 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 di fine.
$q_0 \in Q$ 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 si 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. $1 = 1, 2 = 11, 3 = 111, \dots$
$A = \{b, 1, 0, \$\}$
$b =$ ”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 TM sono imperative ma non nel senso di linguaggio di prog. imperativo.
- Il λ calcolo è alla base di ogni linguaggio di prog. funzionale.
- Le TM 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 TM 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 TM è difficile perché non vi è una definizione ricorsiva.
- Il λ calcolo ha il concetto di ricorsione.

Un λ calcolo è una controparte computazionale 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 $t, s, u, v, M, N, \cdots$.
- x è l'occorrenza di una variabile. Spesso si usano $x, y, z, w, \cdots$. 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 siano 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 $\lambda x.\, (\lambda y.\, xy)x \to_\beta \lambda x.\, xx$ 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 - 1)
```

che in λ calcolo diverrebbe

```
Y
  (λfact.
    λn.
```

```
    match n with
          | 0 -> 1
          | S m -> m * fact (n - 1))
```

si ha che

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

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

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

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

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

$$\text{S } 0$$

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

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

però così si avrebbe un'espansione all'infinito.

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

```
type b = true | false
(** 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 | Quadri | Picche | Fiori
```

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 *)
```

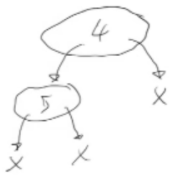
Oppure un parametrico, come una Lista di valori `T`. `List` è il tipo parametrico e `T` il parametro del tipo. La lista è vuota `[]` o una "cons" `(::)` in cui ha una testa e una coda. Una lista ha due elementi dunque: testa `T` e coda `List T`.

```
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 :: []))
*)
```

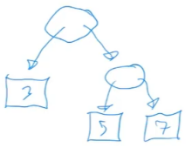
Si possono fare delle ottimizzazione in memoria: i valori piccoli numerici possono andare scritti come bit dato che occupano poco spazio; la lista vuota può semplicemente puntare a `nil`.

Un esempio di struttura albero può essere definito come

```
type Tree1 T = X : Tree1 T | O : Tree1 T -> T -> Tree1 T -> Tree1 T
(* in vero Ocaml sarebbe
type 't tree1 = X | O of 't tree1 * 't * 't tree1
*)
```




```
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:

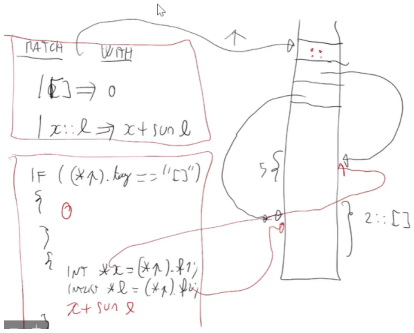
- `b1: b`

```
match b1 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.

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

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

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

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

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

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

$$\rightarrow_\beta \mathrm{M_e}$$

Invece si ha che

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

x è T, l è List T, e è X, c è $\mathrm{T} \rightarrow \mathrm{List\ T} \rightarrow \mathrm{X}$

$$\mathrm{match_{List}}\ (\mathrm{cons\ M_x\ M_c})\mathrm{M_e}\ \mathrm{M_c}$$

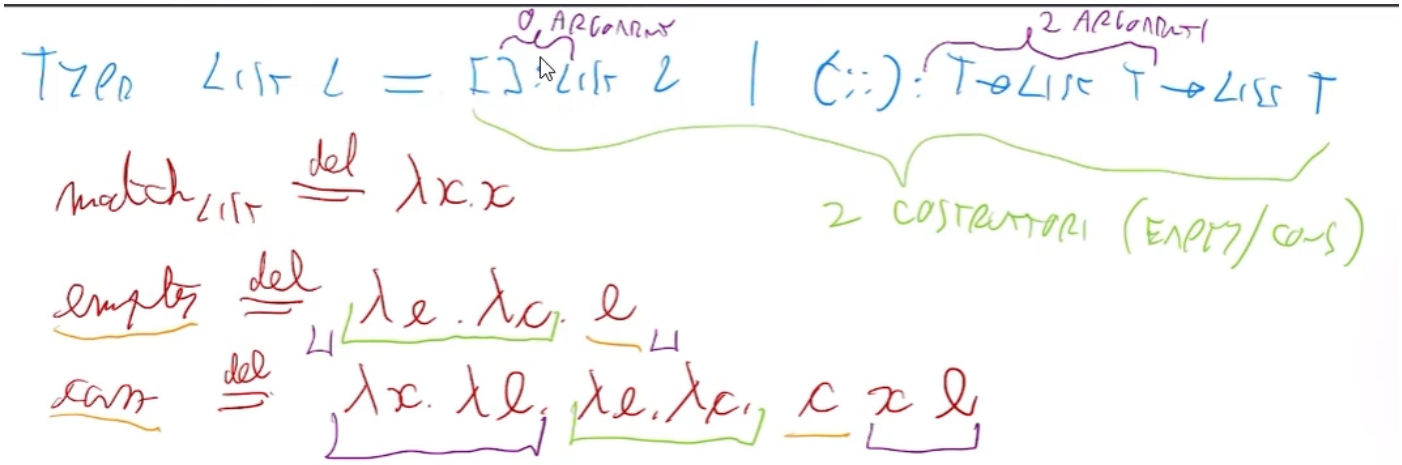
$$= (\lambda x.\, x)(\mathrm{cons\ M_x\ M_l})\mathrm{M_e}\ \mathrm{M_c}$$

$$\rightarrow_\beta \mathrm{cons\ M_x\ M_l\ M_e\ M_c}$$

$$= (\lambda x.\, \lambda l.\, \lambda e.\, \lambda c.\, cxl)\mathrm{M_x\ M_l\ M_e\ M_c}$$

$$\rightarrow_\beta^4 \mathrm{M_c\ M_x\ M_l}$$

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
```

$$\mathrm{match_B} \stackrel{\text{def}}{=} \lambda x.\, x$$

$$\mathrm{true} \stackrel{\text{def}}{=} \lambda t.\, \lambda e.\, t$$

$$\mathrm{false} \stackrel{\text{def}}{=} \lambda t.\, \lambda e.\, e$$

$$\mathrm{match_B\ true\ M_t\ M_e} = (\lambda x.\, x)\ \mathrm{true\ M_t\ M_e}$$

$$\rightarrow_\beta\ \mathrm{true\ M_t\ M_e} = (\lambda t.\, \lambda e.\, t)\mathrm{M_t\ M_e}$$

$$\rightarrow_\beta \mathrm{M_t}$$

Riprendendo l'esempio dei numeri naturali

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

$$\text{match}_{\text{N}} \overset{\text{def}}{=} \lambda x.\, x$$

$$0 = \lambda z.\, \lambda s.\, z$$

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

$$\text{match}_{\text{N}}\ (\text{S N})\ \text{M}_{\text{z}}\ \text{M}_{\text{s}}$$

$$\to_{\beta} \text{S N}\ \text{M}_{\text{z}}\ \text{M}_{\text{s}}$$

$$= (\lambda n.\, \lambda z.\, \lambda s.\, sn)\ \text{N}\ \text{M}_{\text{z}}\ \text{M}_{\text{s}}$$

$$\to_{\beta}^{3} \text{M}_{\text{s}}\ \text{N}$$

Riprendendo l'esempio dei semi

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

Tutti hanno 0 argomenti ma ci sono 4 costruttori.

$$\text{match}_{\text{Seme}} \overset{\text{def}}{=} \lambda x.\, x$$

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

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

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

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

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)`.

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

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

ma in λ calcolo si ha solo

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

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 ad avere 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 decide 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 restituisce 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. $\lambda x.\, xx$ non ha la proprietà "MISTERIOSA".

Isomorfismo

La corrispondenza che si ha con λ calcolo si chiama isomorfismo: si cambia da una forma 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.

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

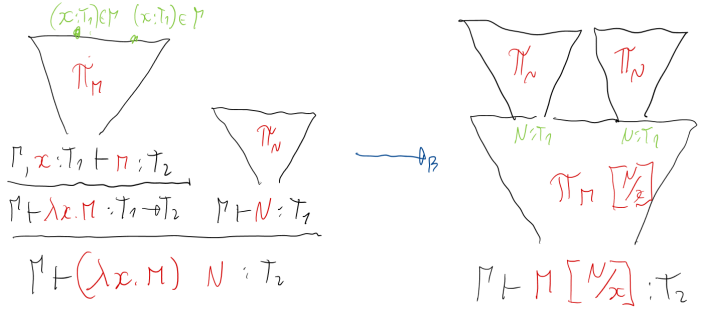
$$A \to A \to B \vdash A \to B$$

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

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

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

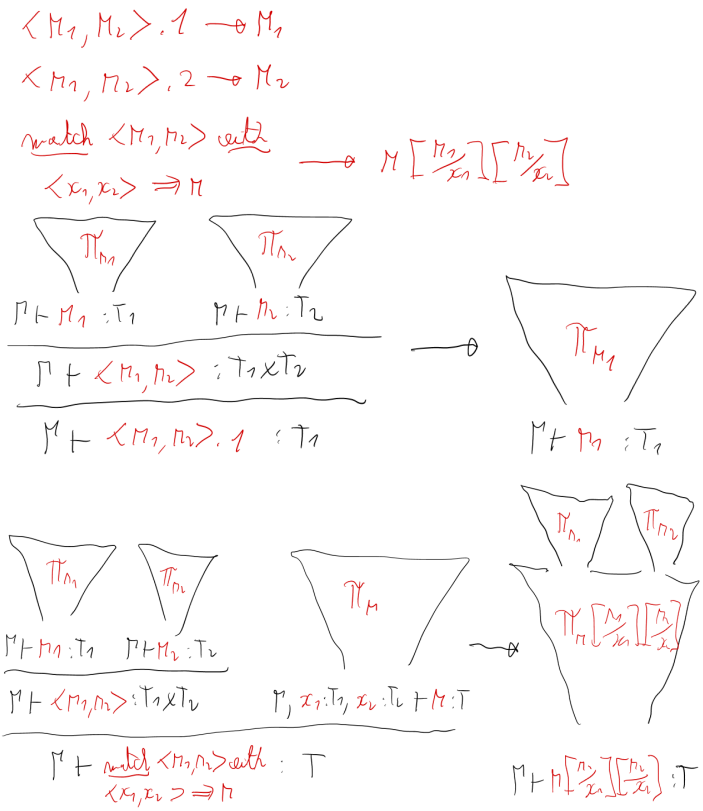
$$t ::= \ldots | \langle t, t \rangle | t.1 | t.2 | \text{match } t \text{ with } \langle x_1, x_2 \rangle \implies t$$

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

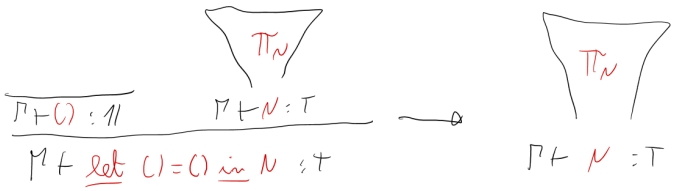
$$T ::= \ldots | \mathbb{1}$$

$\mathbb{1}$ 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 le ha; 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 $\nexists (t_i)_{i\in\mathbb{N}} : t = t_i \land \forall i.\, t_i \to_\beta t_{i+1}$
- 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

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

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

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

  due sotto alberi e dunque due ipotesi. Per ipotesi induttiva $M \in RED_{T_1 \to T_2}(I_1)$ e $N \in RED_{T_2}(I_2)$
  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 riducibilità

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

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 $\exists \nu(T) \in \mathbb{N}.$ 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), $MU' \in RED_{T_2}$

■

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 (non si sa quale).
  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

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

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:

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

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:

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

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.

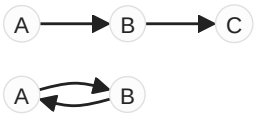
1. $A \neq \emptyset$ ed è chiamato *insieme degli stati* (o delle configurazioni).
2. $\to \subseteq A \times A$ ed è chiamata *relazione di transazione*.

- 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 $\mathbb{R}$ 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 $(\mathbb{Z}^{\mathbb{N}})$.

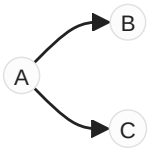
Un ARS $(A, \to)$ è deterministico quando $\forall q_1, q_2, q_2' \in A.\, q_1 \to q_2 \land q_1 \to q_2' \implies q_2 = q_2'$

Esempi deterministici:

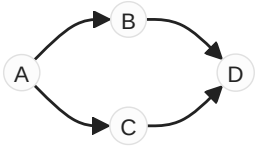


Esempi non deterministici. Un esempio sono i programmi concorrenti in cui, in base alla velocità, si avrà uno stato finale che può differire.

Caso 1.

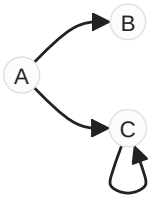


Caso 2.
Coppia critica, vi sono due cammini divergenti che però congiungono in un unico punto deterministico. Questa si chiama *confluenza*.

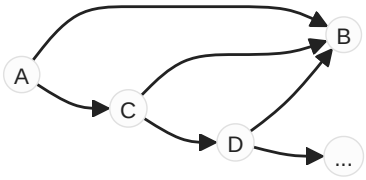


Caso 3.
Qui c'è anche il caso in cui si ha 1 solo stato possibile ma non è manco sempre raggiunto, quindi se il

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



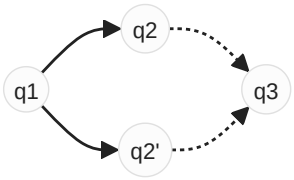
Caso 4.
Qui potrebbe non finire mai l'esecuzione però potrebbe anche finire nel caso di `C --> B` o `D --> B`.



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.

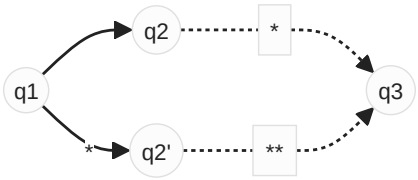
Tipi di confluenza

- Confluenza locale



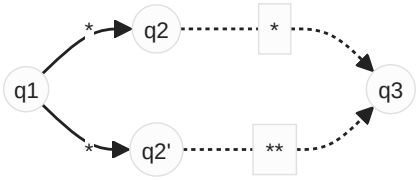
$\forall q_1, q_2, q_2'. q_1 \to q_2 \land q_1 \to q_2'$
$\exists q_3. q_2 \to q_3 \land q_2' \to q_3$
Che è uguale a dire
$\forall q_1, q_2, q_2'. q_2 \leftarrow q_1 \to q_2' \implies \exists q_3. q_2 \to q_3 \leftarrow q_2'$

- Semiconfluenza

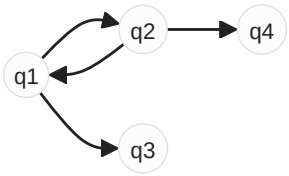


> `*` e `**` rappresentano la stessa cosa, però non me lo fa fare.

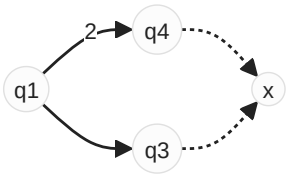
- Confluenza



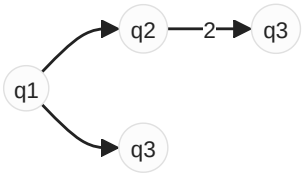
$$\text{Confluenza} \implies \text{Semiconfluenza} \implies \text{Confluenza locale}$$
$$\text{Confluenza locale} \not\implies \text{Semiconfluenza}$$



che può essere



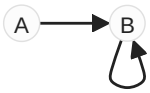
oppure



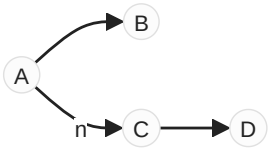
Il controesempio non è fortemente normalizzante.

- **Teorema** $\text{Fortemente normalizzate} \wedge \text{Confluente locale} \implies \text{Semiconfluente}$

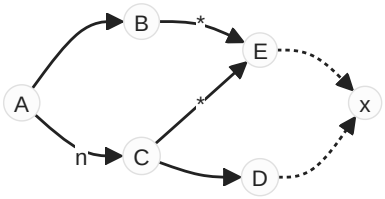
Dimostrazione (errata) di Confluenza locale $\implies$ Semiconfluenza
Caso 0 passi.



Caso $n+1$ passi.



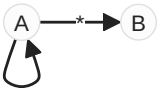
Per ipotesi induttiva vi è uno stato tra B e C in cui si va per un numero determinato di passi. La chiusura del grafo però è possibile? Cioè, esiste un modo per avere



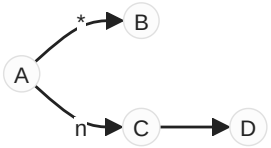
? No.

- **Teorema** $\text{Semiconfluenza} \implies \text{Confluenza}$

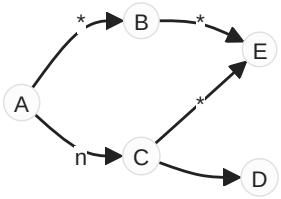
Caso 0.



Caso $n+1$.

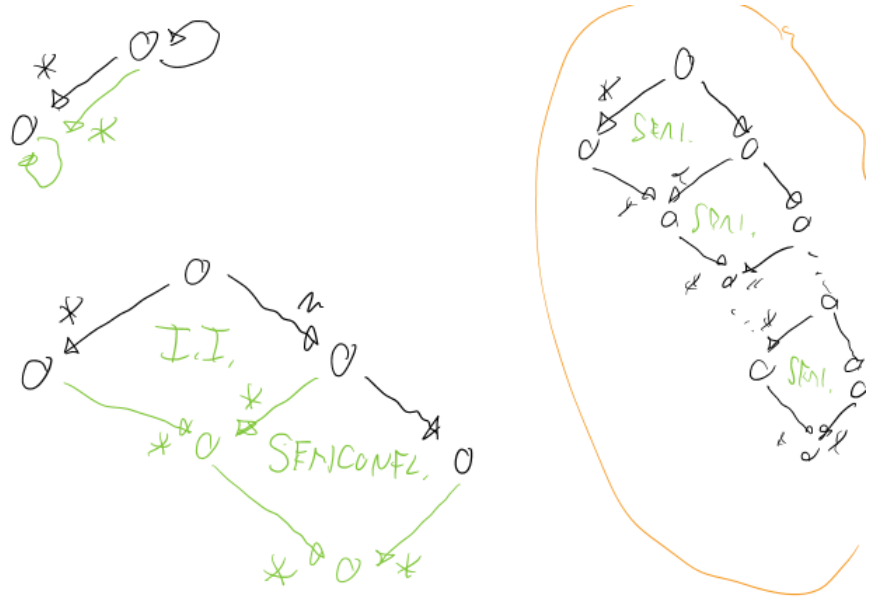


L'ipotesi induttiva mi dice che

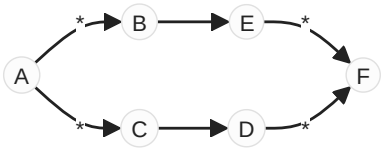


e che quindi si chiude.

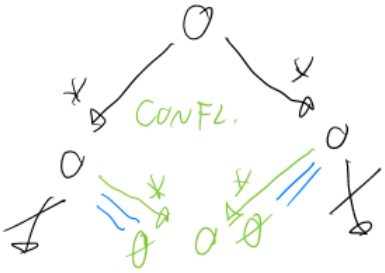
“Pasted image 20231230113024.png” could not be found.



- **Teorema** Confluenza $\implies$ Unicità delle forme normali

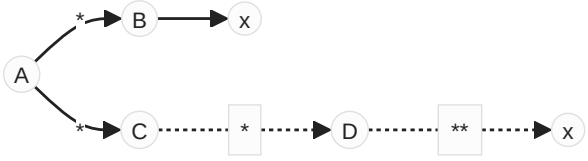


ma in realtà i passi da `E->F` e `D->F` sono 0 e dunque i nodi `E` e `D` sono uguali!
Se ci sono strade alternative arrivano alla stessa forma normale.



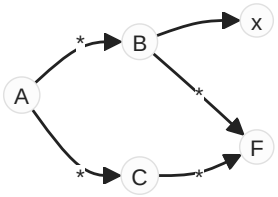
Teorema Confluenza $\implies$ Safety

Definizione di Safety

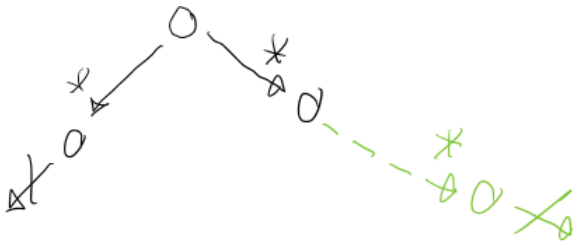


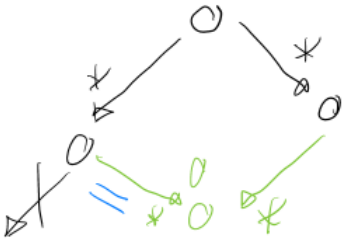
Ovvero si assume di aver fatto un certo numero di passi per arrivare alla forma normale.

Dimostrazione

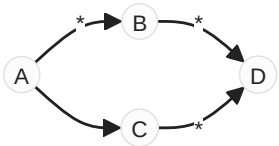


Ma in realtà `B->F` e `C->F` fanno 0 passi e dunque `B=C`.



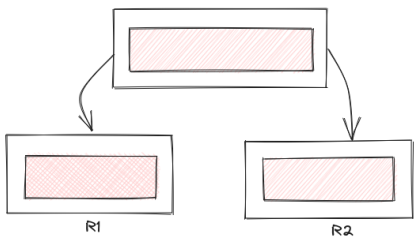


Teorema
Il λ calcolo è semiconfluente.

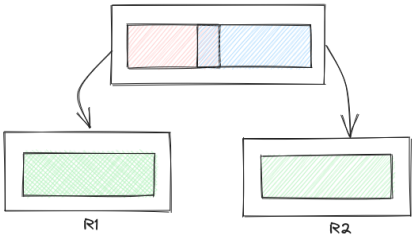


Fonti del non determinismo

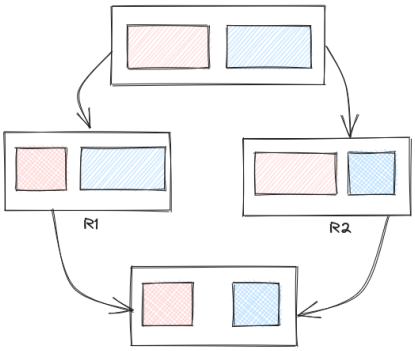
1. Un redex ha due ridotti: non avviene nel λ calcolo, ma nei linguaggi potrebbe. Ad esempio `flip()` può essere sostituita da `0` o `1`.



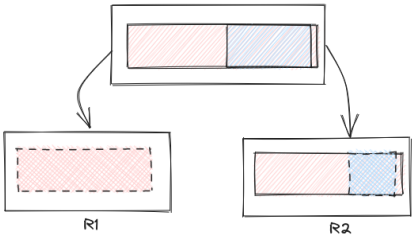
2. Due redex possono essere overlapping ma non uno strettamente incluso nell'altro: non avviene nel λ calcolo.



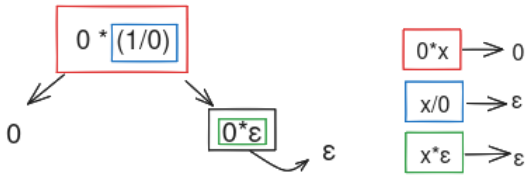
3. Redex non overlapping o paralleli: c'è nel λ calcolo.



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

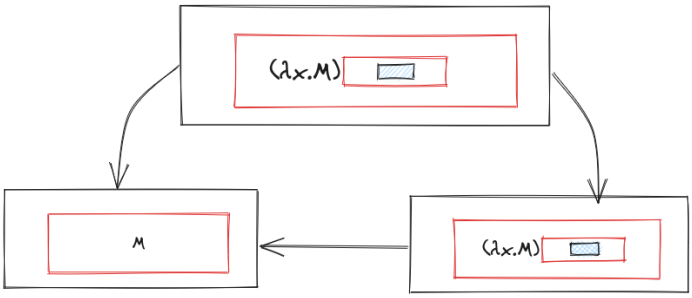


Ad esempio si può decidere cosa tornare in base a quale parte dell'espressione si fa prima il parsing.



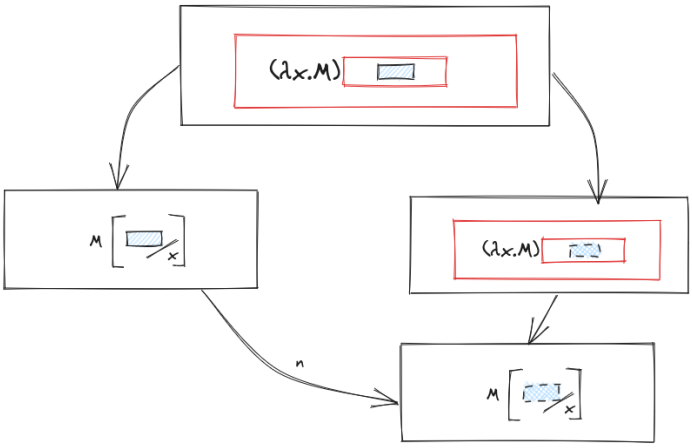
Caso 4 nel λ calcolo

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



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

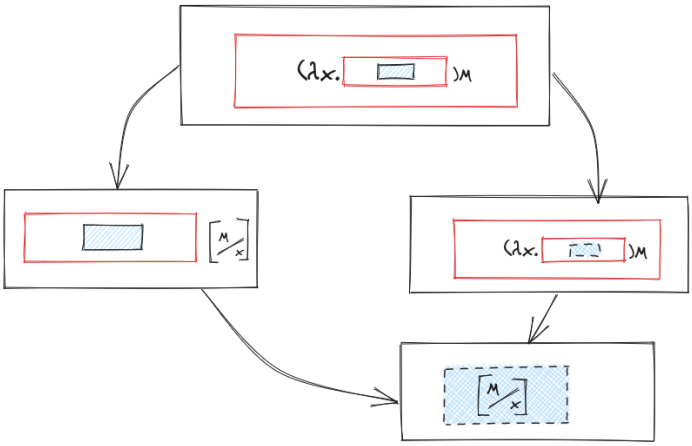
2. Con $x \in FV(M)$ si ha



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 possono 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, e.g. $0 \mapsto 00, 1 \mapsto 11, \# \mapsto 01$.
- Grafi
  Si crea una 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 o vero. Si può vedere questa funzione come sotto insieme delle stringhe per f per cui è uguale a 1, ed essa è chiamata *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 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 l'algoritmo $\mathcal{A}$ calcola f; la sua semantica è scritta come $[\![\mathcal{A}]\!] = f$.
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 stringa $x \in \mathbb{B}$ 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 sugli algoritmi che vengono usati per tali linguaggi. Presa una funzione $g : \mathbb{N} \to \mathbb{N}$ si definiscono delle classi concrete (sono insiemi di linguaggi):

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

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 ed usa lo spazio al posto del tempo, perché quest'ultimo sarebbe davvero troppo piccolo.

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 diciamo 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.

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

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

La congettura della tesi forte di Church-Turing parla di come sia possibile simulare efficientemente macchine se fisicamente realizzabili (no computer quantistici).

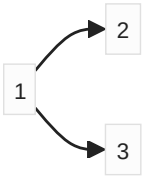
Complessità descrittiva

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.

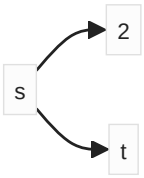
Esempio
(semantica) universo $\mathcal{A}_3 = \{1, 2, 3\}$
(sintassi) $\mathrm{vocabolario} = \{E(-,-), = (-,-), s, t\}$

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

$(\mathcal{A}_3, I) \models \exists x. \, E(s, x)$
Il vocabolario sta nella firma dell'universo.



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.
> $\mathrm{bin}^3(E) = \{(1,1), (1,2), (1,3), (2,1), (2,2), (2,3), (3,1), (3,2), (3,3)\}$

$\mathrm{bin}^3(E) = 011000000$
$\mathrm{bin}^3(s) = \lceil \log(3) \rceil = 2$

$\text{bin}^3(s) = 01$
$\text{bin}^3(t) = 11$

$\text{bin}^3(I) = 0110000000111$

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
$G = \forall X . \exists x . (P(x) \vee X(x))$

bisognerebbe capire come rendere vera tutta la formula, guardando tutte le possibili interpretazioni. Ci si appoggia comunque per 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 chiamato *logica al secondo ordine esistenziale*.
$\exists X^{n_1}. \cdots . \exists X^{n_m}. F$

$F \sim (\mathcal{A}, I) \models F$
dunque
$\text{struct}(F) = \{\text{bin}^n(I) | (\mathcal{A}, I) \models F\}$

Ricordando che bisogna essere polinomiale rispetto alla lunghezza dell'input.

Lemma 1 Supponiamo che il vocabolario abbia almeno un simbolo predicativo P_i con $ar(P_i) \geq 1$. Allora $|\text{bin}^n(I)| \geq n$ dove n è $|\mathcal{A}|$.

Dimostrazione di Lemma 1
$|\text{bin}^n(I)| = |\text{bin}^n(P_1) \cdots \text{bin}^n(P_m) \text{bin}^n(f_1) \cdots \text{bin}^n(f_f)| \geq |\text{bin}^n(P_i)| = n^{ar(P_i)} \geq n^1 = n$
■

Lemma 2 $FO \subseteq P$

Dimostrazione di Lemma 2
Si dimostra un risultato più forte: Per ogni formula F del primo ordine con variabili libere $x_1, \ldots, x_m$ esiste un algoritmo M_F polinomiale tale che su input $S_{i_1, \ldots, i_n}$ determina se $S = \text{bin}^n(I)$ e se $(\mathcal{A}_n, I), \xi \models F$ dove

$\xi(x_j) = i_j$.
Se F non ha variabili libere allora il lemma 2 è dimostrato. Quelle variabili vengono interpretate in base all'input, e in questo caso è la seconda parte dell'input. E questo si può dimostrare per induzione su F solo perché stiamo dicendo che ci stanno variabili libere.

- Caso base $F = P(t_1 \dots t_k)$
  Allora M_F procede così:
  1. Calcolo $[\![t_i]\!]_\xi$ dove ξ è l'interpretazione che assegna i_j a x_j.
  2. Verifico se $([\![t_1]\!]_\xi \dots [\![t_k]\!]_\xi) \in I(P)$ dove $I(P)$ si ricava dall'input S.
  3. Se è vero, accettiamo.
- Caso induttivo $F = F_1 \wedge F_2$, $F = F_1 \vee F_2$, $F = \neg F_1$
  Semplice dimostrazione di linguaggi decidibili (guardare note di Informatica teorica su Virtuale). Analoghi alle proprietà di chiusura dei linguaggi.
- Caso induttivo $F = \exists x.\, G$
  Per ipotesi induttiva abbiamo M_G che si aspetta un input i_q relativo alla variabile x. L'algoritmo M_F chiamerà M_G più volte: una per ogni possibile valore di i_q. Quindi l'algoritmo M_G verrà eseguito con $x = 1, x = 2, x = 3, \dots$. Se almeno una di loro accetta, allora accetto.
- Caso induttivo $F = \forall x.\, G$
  Medesimo di sopra ma si accetta solo se tutte accettano.

Per dimostrare $FO = P$ si usano i punti fissi.
■

Dimostrazione di Teorema di Fagin
Si inizia la dimostrazione da $\exists SO \subseteq NP$ e per farlo ci servirà il Lemma 2. La formula generica G di $\exists SO$ è

$$G \equiv \exists X_1. \cdots . \exists X_m.\, F$$

con F del primo ordine.
Osserviamo che, dato un modello e la sua interpretazione, la formula G è vera.

$$(\mathcal{A}_n, I) \models G$$

ed esso si verifica sse esiste un'interpretazione J che estende I t.c.

$$(\mathcal{A}_n, J) \models F\{R_1/X_1, \dots, R_m/X_m\}$$

dove $R_1, \dots, R_m$ sono simboli predicativi dentro I a cui J assegna un valore qualsiasi nel modello.
Data l'osservazione sopra, si costruisce un algoritmo polinomiale non deterministico per decidere

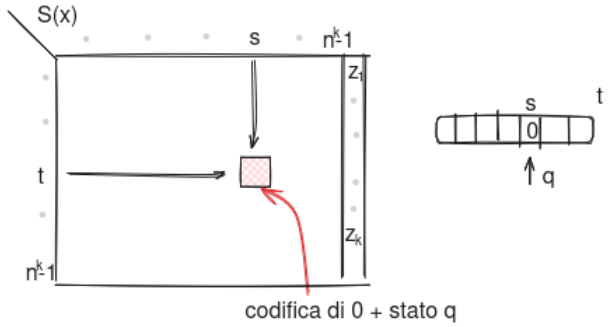
$$\text{struct}(G) = \{\text{bin}^n(I) | (\mathcal{A}_n, I) \models G\}$$

1. Estraiamo il parametro n (ovvero la cardinalità dell'universo).
2. Generiamo m stringhe binarie ciascuna corrispondente ad una interpretazione di $R_{i, 1 \leq i \leq m}$.
3. Modifichiamo la stringa di input (ricordiamo che codifica un'interpretazione I) usando le stringhe costruite al punto 2 in modo da far diventare la stringa di input una codifica di una interpretazione J per $F\{R_1/X_1, \dots, R_m/X_m\}$.
4. Chiamiamo l'algoritmo $M_{F'}$ per la decisione di $\text{struct}(F\{R_1/X_1, \dots, R_m/X_m\} = F')$. Questo è dato dal fatto di aver già dimostrato il Lemma 2.

L'algoritmo è polinomiale perché chiama una volta sola $M_{F'}$ che è polinomiale, ed inoltre è anche deterministico. Dunque $\text{struct}(G) \in NP$.
Adesso va dimostrato $NP \subseteq \exists SO$.
Prendendo come modello del linguaggio NP una macchina di Turing, dovremo vedere cosa succede con SO per questa macchina.

$$M \rightsquigarrow F_M \text{ t.c. } \text{struct}(F_M) = \{x \in \{0,1\}^\star | M \text{ accetta } x\}$$

Si cerca un predicato $S(x)$ per vedere dove l'elemento x contiene 1 o 0. Bisogna visualizzare la computazione di M su x. $S(3) = 1$ se il terzo bit della stringa è vero.
Riga (t) = tempo computazione
Colonna (s) = passo
Il numero di righe e colonne è finito perché l'algoritmo finisce, e lo fa in massimo $n^k - 1$ passi.
Gli elementi nelle celle sono in $\Sigma \uplus (\Sigma \times Q)$ dove Σ è l'alfabeto dei simboli e Q l'insieme di stati nella macchina.
Dunque si ha una visione della macchina ad ogni passo.
L'ultima colonna z_i è data dalla scelta non deterministica $0 = L, 1 = R$.



La formula F_N avrà la forma

$$F_M = \exists C_1^{2k}. \cdots . \exists C_g^{2k}. \exists \Delta^k. \phi_M$$

dove $g = |\Sigma \uplus (\Sigma \times Q)|$ ovvero numero di valori diversi che possiamo trovare nelle caselle.
Il C_i^{2k} è un predicato come $C_i^{2k}(t_1 \cdots t_k, s_1 \cdots s_k)$ che è vero se la casella indice del tempo e spazio $(t_1 \cdots t_k, s_1 \cdots s_k)$ contiene il valore i.
s e t sono termini e sono codifiche di indice riga/colonna $[\![t_R]\!] \in \{1, \ldots, n\}$. Allora possibili valori di $[\![t_1]\!] \ldots [\![t_k]\!]$ saranno n^k. L'arietà è $2k$ perché ci sono $k + k$ argomenti tra tempi e spazi.

dove Δ^k è il predicato speciale per i vari z delle scelte non deterministiche.
$\Delta^k(t_1 \ldots t_k)$ è vera se al tempo $[\![t_1]\!] \ldots [\![t_k]\!]$ ho operato la prima scelta non deterministica tra le due disponibili; falsa se ho operato l'altra scelta. $[\![t_1]\!] \ldots [\![t_k]\!]$ è un valore tra 0 e $n^k - 1$.

$$\phi_M = \alpha_M \wedge \beta_M \wedge \gamma_M \wedge \delta_M$$

$\alpha_M = C_i(\overline{0}, \overline{s})$ codifica l'input. Consideriamo il tempo 0 e un generico spazio. Quindi α descrive la configurazione iniziale della macchina.
β_M al più uno tra $C_i(\overline{t}, \overline{s})$ e $C_j(\overline{t}, \overline{s})$ è vera. Dunque in una specifica casella può esserci al più un unico simbolo (i o j).
γ_M codifica la funzione di transizione.

δ_M vede se M all'istante $n^k - 1$ è in uno stato accettante o meno.
I primi tre vedono se la codifica è valida, non ambigua e riflette un passo conforme alla transizione.

$$\begin{aligned}\alpha_M = \forall x. (\\ (S(x) \to C_{<1>}(0 \dots 0, 0 \dots 0, x)) \wedge \\ (\neg S(x) \to C_{<0>}(0 \dots 0, 0 \dots 0, x)) \wedge \\ (\forall y_1 \dots \forall y_k (C_{<\text{blank}>}(0 \dots 0, 1, y_2 \dots y_k)) \to \cdots) \\ )\end{aligned}$$

$C_{<1>}$ significa che in quella casella c'è il simbolo 1. Nell'ultima parte bisogna considerare tutte le possibili combinazioni.

$$\beta_M = \forall \overline{x} \forall \overline{y}. (\bigwedge_{i \neq j} C_i(\overline{x}, \overline{y}) \to \neg C_j(\overline{x}, \overline{y}))$$

Quindi dato un tempo e uno spazio, una tupla è solo in uno stato.

$$\gamma_M = \forall \overline{x} \forall \overline{y}. (C_{<a,q>}(\overline{x}, \overline{y}) \to (\Delta(\overline{x}) \to C_{<a',q'>}(\overline{x} + 1, \overline{y} - 1) \wedge \neg \Delta(\overline{x}) \to C_{<a'',q''>}(\overline{x} + 1, \overline{y} + 1)))$$

Considerando la transizione $(a, q) \mapsto \{(a', q', \leftarrow), (a'', q'', \to)\}$ però è non deterministica, per questo si ha più un opzione. Tutte le celle restano immutate, quindi $\bigvee_{i \notin \{<a,q>, <a',q'>, <a'',q''>\}} C_i(\overline{x}, \overline{y}) = C_i(\overline{x} + 1, \overline{y})$

$$\delta_M = \bigvee_{a \in \Sigma, q_{acc} \text{ accettante}} \exists y. \cdots. \exists y_k. C_{<a,q_{acc}>}(\overline{\max}, \overline{y})$$

$(\mathcal{A}_n, I) \models F_n$ se $\exists \xi$ t.c. $(\mathcal{A}_n, I), \xi \models \phi_M$. Possiamo dimostrare che se ϕ_M è vera, allora $\xi(C_i)$ avrà il suo valore corretto, ovvero $\xi(C_i(\overline{t}, \overline{s}))$ sarà vera se M su input x dopo $\overline{t}$ passi si trova in posizione $\overline{s}$ nella configurazione.

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, $y \leq z \implies F^I(y) \leq F^I(z)$. 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 $F(\mu F) \subseteq \bigcap \{Y | F(Y) \subseteq Y\}$ ma la seconda parte è μF dunque è come fare $F(\mu F) \subseteq \mu F$
   Allora, per monotonicità, $F(F(\mu F)) \subseteq F(\mu F)$
   quindi $F(\mu F) \in \{Y | F(Y) \subseteq Y\}$
   allora $\bigcup \{Y | F(Y) \subseteq Y\} \subseteq F(\mu F) = \mu F$
4. Se $Z = F(Z)$ allora $F(Z) \subseteq Z$
   quindi $Z \in \{Y | F(Y) \subseteq Y\}$
   allora $\bigcup \{Y | F(Y) \subseteq Y\} \subseteq Z = \mu F$
   ■

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.

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

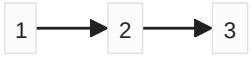
Esempio Reachability in $LFP(FO)$

$$F \equiv LFP(R, (x, y), (x = y) \vee \exists z.\, (E(x, z) \wedge R(z, y)))$$

R variabile del secondo ordine che vuole catturare l'insieme che testimonia la reachability.
(x, y) variabili libere in cui dipende questa formula.
$(x = y) \vee \exists z.\, (E(x, z) \wedge R(z, y))$ espressione del primo ordine.

R è la minima variabile che soddisfa la situazione della formula $R(x, y) \equiv (x = y) \vee \exists z.\, (E(x, z) \wedge R(z, y))$

Con un vocabolario fatto solo da E, dall'uguaglianza $=$ (che è la funzione identità) e dai nodi del grafo, si può generare un modello definito come un grafo $(\mathcal{A}_4, I)$



$$F^I : \mathcal{P}(\mathcal{A}_4)^2 \to \mathcal{P}(\mathcal{A}_4)^2$$

$\mathcal{P}(\mathcal{A}_4)^2$ sono insiemi di relazioni, ovvero del tipo $(1, 2), (3, 4), (2, 3), \cdots$. Si ha $\_^2$ perché sono due le variabili libere nella definizione, ovvero (x, y).
Preso un generico $S \in \mathcal{P}(\mathcal{A}_4)^2$ si ha il map definito

$$S \mapsto \{(n', m') | (\mathcal{A}_4, I), \xi \models F \text{ dove } \xi(R) = S \wedge \xi(x) = n' \wedge \xi(y) = m'\}$$

dopo la mappatura F non ha più variabili libere e quindi stabilire se è vera o meno nel modello.

Domanda, $(\mathcal{A}_4, I), \xi \models F$? con $\xi(x) = 1, \xi(y) = 3$ be sì
Domanda, $(\mathcal{A}_4, I), \xi \not\models F$? con $\xi(x) = 1, \xi(y) = 4$
$\mu F^I = \bigcup_n (F^I)^n(\emptyset)$ e questo arriva ad un tal punto in cui non si aggiunge più nulla, e quindi
$(F^I)^m(\emptyset) = (F^I)^{m+1}(\emptyset)$.

$$F^I(\emptyset) = \{(n', m') | (\mathcal{A}_4, I), \xi \models x = y \vee \exists z.\, (E(x, z) \wedge R(z, y)) \text{ dove } \xi(R) = \emptyset \wedge \xi(x) = n' \wedge \xi(y) = m'\}$$

ma $\nexists z.\,(E(x,z) \wedge R(z,y))$ perché $R(z,y) = \emptyset$ e dunque

$$F^I(\emptyset) = \{(n',m') | (\mathcal{A}_4, I), \xi \models x = y \text{ dove } \xi(x) = n' \wedge \xi(y) = m'\} =$$
$$= \{(n',m') | n' = m'\} = \{(1,1),(2,2),(3,3),(4,4)\}$$

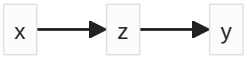
e questi sono proprio tutti i casi limiti, i casi di reachability triviale.

$$F^I(F^I(\emptyset)) = \{(n',m') | (\mathcal{A}_4, I), \xi \models x = y \vee \exists z.\,(E(x,z) \wedge z = y) \text{ dove } \xi(R) = \{(1,1),(2,2),(3,3),(4,4)\} \wedge \xi(x) = n$$
$$= \{(n',m') | (\mathcal{A}_4, I), \xi \models x = y \vee E(x,y) \text{ dove } \xi(x) = n' \wedge \xi(y) = m'\} =$$
$$= \{(1,1),(2,2),(3,3),(4,4),(1,2),(2,3)\}$$

Si ha $R(z,y)$ è diventata $z = y$ perché è mappato sui vari $(1,1), \cdots$. E inoltre $\exists z.\,(E(x,z) \wedge z = y)$ è lo stesso di dire $E(x,y)$.

$$F^I(F^I(F^I(\emptyset))) = \{(n',m') | (\mathcal{A}_4, I), \xi \models x = y \vee \exists z.\,(E(x,z) \wedge R(z,y))$$
$$\text{dove } \xi(R) = \{(1,1),(2,2),(3,3),(4,4),(1,2),(2,3)\}$$
$$\wedge\, \xi(x) = n' \wedge \xi(y) = m'\} =$$

Quindi sono i map



e dunque estende al caso di percorsi con al massimo 2 passi.

$$= \{(1,1),(2,2),(3,3),(4,4),(1,2),(2,3),(1,3)\}$$

$F^I(F^I(F^I(\emptyset))) = F^I(F^I(F^I(F^I(\emptyset))))$

10. Logica e database

Dati dei domini che possono essere $D_1 = \mathbb{N}$ e $D_2 = \mathbb{B}$ le relazioni sono $\mathcal{R} \subseteq D_1 \times D_2$ e sono finite. Esso vive all'interno di

$$\mathcal{P}_{fin}\left(\prod_{i \leq i \leq n} D_i\right) = \mathcal{P}_{fin}(D_1 \times \cdots \times D_n)$$

quindi possiamo vedere una relazione come un insieme finito di n-uple nella forma $(d_1, \ldots, d_n)$ dove $d_i \in D_i \forall 1 \leq i \leq n$.
Una relazione ordinata può essere trasformata in una non ordinata e viceversa.

Una query è una sequenza di relazioni.

$$[\![Q]\!] : \mathcal{P}_{fin}(D^{n_1}) \times \cdots \times \mathcal{P}_{fin}(D^{n_k}) \to \mathcal{P}_{fin}(D^{n_m})$$

Si fa un esempio per un linguaggio di database relazionali. L'algebra relazionale (ci interessa solo il fatto che possiamo scrivere interrogazioni mediante linguaggio formale) sarà:

$$Q ::= R_i | Q \cup P | P - Q | Q \times P | \pi_l(Q) | \sigma_c(Q)$$

in cui π_l è una proiezione e σ_c è una selezione.
La condizione nella selezione è definita come:

$$c ::= i \leq j | i = j | \neg c | c \wedge d | c \vee d$$

i è semplicemente un indice, ricordando che i simboli di relazione sono $\{R_i, \ldots, R_k\}$.
l è una sequenza di numeri naturali, usato per filtrare i valori dentro un certo valore nella proiezione.

Un'interrogazione Q deve soddisfare i vincoli di integrità, ovvero i numeri interi che occorrono sono coerenti con la relazione. Ad esempio, in $Q \cup P$ questi ultimi devono avere la stessa arietà: se P lavora in un tupla di lunghezza 5 non potrà prendere l'indice 6; idem per i join. Nel prodotto cartesiano non si ha questo vincolo.

Il dominio della funzione generata da $[\![Q]\!] : \mathcal{P}_{fin}(D^{n_1}) \times \cdots \times \mathcal{P}_{fin}(D^{n_k}) \to \mathcal{P}_{fin}(D^{n_m})$ è un prodotto cartesiano. Ogni insieme di tuple avrà arietà corrispondente al simbolo predicativo R che sta in quella posizione. L'arietà di una relazione R sono le colonne. La funzione Q è dunque definita su k elementi. D è il dominio in cui sono definite le relazioni. n_1 è l'arietà di R_1 e così via. Il codominio è dato da m che è l'arietà di Q e quindi $\mathcal{P}_{fin}(D^m)$.
La semantica è definita per induzione sulla struttura.
Se Q è R_i, si ha una funzione costante perché

$$[\![Q]\!](\mathcal{R}_1, \ldots, \mathcal{R}_k)$$

è semplicemente $\mathcal{R}_i$.
Gli operatori $\cup, -, \times$ hanno un'interpretazione naturale. Lavorando per induzione, per definire la semantica di $[\![Q \cup P]\!]$ possiamo assumere di sapere la semantica di $[\![Q]\!]$ e $[\![P]\!]$.

$$[\![Q \cup P]\!](\mathcal{R}_1, \ldots, \mathcal{R}_k) = [\![Q]\!](\mathcal{R}_1, \ldots, \mathcal{R}_k) \cup [\![P]\!](\mathcal{R}_1, \ldots, \mathcal{R}_k)$$

Nella definizione della **proiezione** c'è una lista di interi che considera i campi da considerare in essa.

$$[\![\pi_{i_1,\dots,i_s}Q]\!](\mathcal{R}_1,\dots,\mathcal{R}_k) = \{(d_{i_1},\dots,d_{i_s})|(d_1,\dots,d_n) \in [\![Q]\!](\mathcal{R}_1,\dots,\mathcal{R}_k)\}$$

n è l'arietà di $[\![Q]\!]$.
Un indice out-of-bound ($> n$) non sarebbe ben definito.

Nella **selezione** bisogna trovare una condizione c soddisfatta da valori $t = (d_1,\dots,d_n)$. Esempi possono essere

- $(d_1,\dots,d_n) \vdash i = j$ sse $d_i = d_j$
- $t \vdash c \wedge d$ sse $t \vdash c$ e $t \vdash d$
  Diciamo che $t \vdash c$ quando c è soddisfatta da t.

$$[\![\sigma_c(Q)]\!](\mathcal{R}_1,\dots,\mathcal{R}_k) = \{t|t \in [\![Q]\!](\mathcal{R}_1,\dots,\mathcal{R}_k) \wedge t \vdash c\}$$

Esempio
Prendo $\mathcal{R}_1$ per dei soci

1	**2**	**3**	**4**	**5**
0012	Rossi	Mario	1973	0
1492	Verdi	Carlo	1978	1
9834	Gialli	Luca	1980	1
7511	Bianchi	Andrea	1971	0

e $\mathcal{R}_2$ per i risultati

1 (6)	**2 (7)**	**3 (8)**	**4 (9)**
0012	1492	3	2
1492	7511	1	3
9834	7511	0	3

una query per prendere gli anni di nascita dei giocatori che hanno vinto almeno una partita:

$$\pi_4(\sigma_{(1=6)\wedge(8>9)}(R_1 \times R_2)) \cup \pi_4(\sigma_{(1=7)\wedge(9>8)}(R_1 \times R_2))$$

Il campo 1 si riferisce alla prima colonna di R_1. Nel prodotto cartesiano il campo 6 sarà la prima colonna di R_2.

L'insieme delle funzioni che sono espresse da qualche espressione dell'algebra relazionale ben formata è definita come:

$$\mathcal{AR} = \{[\![Q]\!]|Q \text{ è una query ben formata}\}$$

La complessità del calcolo relazionale, che è un formalismo logico, è facilmente lavorabile grazie alla logica predicativa.
Si vuole ottenere una relazione Q di arietà m date delle relazioni $\mathcal{R}_1,\dots,\mathcal{R}_k$ aventi arietà $n_1,\dots,n_k$.
Gli unici simboli funzionali di cui abbiamo bisogno sono delle costanti, come D; i simboli predicativi sono le relazioni vere e proprie $R_1,\dots,R_k$, e poi si hanno dei simboli che permettono di interpretare questi simboli,

come $\leq$ e $=$. Le variabili libere in F (che è la formula predicativa costruita) dovranno essere incluse in $\{f_1, \ldots, f_m\}$ in un campo di relazione Q. Il tipo di logica costruita sarà un frammento della logica predicativa. Fissato questo vocabolario e l'insieme delle variabili libere da cui attingere, si costruiscono tutte le logiche del primo ordine.
Un esempio di calcolo relazionale, prendendo gli stessi dati dell'esempio sull'algebra relazionale:

$$\exists p.\, \exists s.\, \exists c.\, \exists n.\, \exists o.\, \exists pp.\, \exists ps.\, R_1(p, c, n, f, o) \land R_2(p, s, pp, ps) \land (pp > ps)$$
$$\lor$$
$$\exists p.\, \exists s.\, \exists c.\, \exists n.\, \exists o.\, \exists pp.\, \exists ps.\, R_1(s, c, n, f, o) \land R_2(p, s, pp, ps) \land (ps > pp)$$

le variabili R_1 e R_2 sono sia legate che libere. Vi è una sola variabile libera, ed è f: la quarta colonna, gli anni di nascita in R_1. Questo avviene perché le variabili libere sono quelle dei dati che vogliamo estrarre. Facendo apparire le variabili abbiamo un modo per, implicitamente, associare le variabili ai campi. Però il $1 = 6$ e $1 = 7$ sono espresse mettendo la posizione di p e s dentro R_1 e R_2. L'universo delle interpretazioni è D, tutti gli elementi di tutti i campi. Gli unici simboli che vogliamo interpretare, dando i valori alle tabelle, sono $R_1, \ldots, R_k$. Dunque $\{\mathcal{R}_1, \ldots, \mathcal{R}_k\}$ è un'interpretazione. Il dominio di interpretazione è dato dal database stesso.

$$(D, \{\mathcal{R}_1, \ldots, \mathcal{R}_k\}), \xi \models F$$

se vale questo sopra allora vuol dire che $(\xi(f_1), \ldots, \xi(f_m))$ sta nella relazione $\mathcal{Q}$.
La formula F è vera, di conseguenza, ponendo

$$[\![F]\!](\mathcal{R}_1, \ldots, \mathcal{R}_k) = \{(\xi(f_1), \ldots, \xi(f_m)) | (D, \{\mathcal{R}_1, \ldots, \mathcal{R}_k\}), \xi \models F\}$$

F può essere definita come un'interpretazione di un insieme di relazioni su un altro insieme. La verità di F su quel modello si può definire come la funzione associato ad esso. ξ è la funzione che associa un qualche valore al parametro. F è vera quando ξ è interpretata sull'anno di nascita di un giocatore che ha vinto almeno una partita.
Vediamo quindi che $[\![F]\!] \subseteq D^m$ ma non vuol dire che è finita, perché basta vedere il caso in cui $F = (f_1 = f_1)$.

Ci interessa esprimere una condizione semantica che le formule identificate da query siano finite, attraverso un vincolo sintattico chiamato formule sicure. Sapendo che F è una formula di lunghezza finita e dunque ogni tupla in $[\![F]\!]$ occorrono valori tra quelli che occorrono in F e nelle tuple $\mathcal{R}_1, \ldots, \mathcal{R}_k$, dunque $[\![F]\!]$ è sempre finita. Tutte le variabili che appaiono libere sono scritte come $FV(F)$.

1. L'uso del quantificatore, del tipo "$\forall X$ t.c.", non è permesso perché avremmo una formula non sicura dato che si guarda su tutto il dominio.
2. Non si può allargare l'insieme delle variabili libere quando usiamo il quantificatore $\lor$. Dunque per $F \lor G$ sse $FV(F) = FV(G)$.
3. Tutte le sottoformule di una congiunzione $F_1 \land \cdots \land F_m$ dove $m \geq 1$ è massimale (non ci sono più congiunti; $F_1 \land \cdots \land F_m$ non sono congiunti), allora ogni $x \in \bigcup_{1 \leq i \leq m} FV(F_i)$ deve essere limitata, ovvero $\exists$ almeno una formula F_j t.c.:
    1. $x \in FV(F_j)$ con F_j non negativa e non predicato aritmetico (ovvero non è nella forma come, ad esempio, $x > y$).
    2. F_j può equiparare una costante, dunque nella forma $x = c$ con c costante.

3. F_j può equiparare una variabile, ma essa deve essere limitata. Dunque $x = y$ con y limitata. Un caso che non va bene $x > 0000$ perché è infinito; un caso che va bene $y < 1980$ e dunque, implicitamente, anche x è vincolato.
4. La formula è genericamente positiva. L'unica possibilità di negazione è in una delle formule $F_j = \neg G$ di una congiunzione $F_1 \wedge \cdots \wedge F_m$ in cui vi è almeno un congiunto positivo. Se ognuno di loro ha un vincolo e, poiché l'and è un intersezione, se vi è uno positivo, vi sarebbe un'intersezione positiva.

L'insieme delle funzioni per il calcolo relazionale sicuro.

$$\mathcal{CR} = \{[\![F]\!] | F \text{ è una formula sicura del calcolo relazionale}\}$$

Teorema di Codd

$$\mathcal{AR} = \mathcal{CR}$$

L'algebra relazionale identifica le stesse query del calcolo relazionale sicuro. Si descrivono query complessivamente buone, dato che l'algebra relazionale è strettamente inclusa in P. Il calcolo relazionale è solo un metodo di paragone per l'algebra relazionale.

Lemma Per ogni Q esiste P semplice t.c. $[\![Q]\!] = [\![P]\!]$.

Dimostrazione del lemma
(Nel lemma tutti gli operatori di selezione sono atomici o negazioni di essi; utile quando si trasforma nelle formule del calcolo relazionale)
Ricordando le uguaglianze di De Morgan:
$\neg(A \vee B) \sim \neg A \wedge \neg B$
$\neg(A \wedge B) \sim \neg A \vee \neg B$
ci permettono di assumere che ogni occorrenza di $\neg$ in c sia immediatamente vicina ad un operatore aritmetico.
Ad esempio, $\neg(1 = 6 \wedge 5 > 4) = \neg(1 = 6) \vee \neg(5 > 4)$.
La dimostrazione procede per induzione su Q.

- Tutti i casi diversi dalla selezione $Q = \sigma_c(S)$.
  Sono triviali perché la condizione di essere semplice ha senso solo se appaiono operatori di selezione. Ad esempio, $Q = S \times T$ allora per ipotesi induttiva sappiamo che esistono W e Z semplici t.c. $[\![S]\!] = [\![W]\!]$ e $[\![T]\!] = [\![Z]\!]$. Allora abbiamo che $W \cup Z$ è semplice e
  $[\![W \cup Z]\!] = [\![W]\!] \cup [\![Z]\!] = [\![S]\!] \cup [\![T]\!] = [\![S \cup T]\!] = [\![Q]\!]$.
- Caso $Q = \sigma_c(S)$
  Applichiamo l'ipotesi induttiva di S. Otteniamo T t.c. $[\![T]\!] = [\![S]\!]$. Dunque si avrà
  $[\![Q]\!] = [\![\sigma_c(S)]\!] = [\![\sigma_c(T)]\!]$ e vogliamo concludere che $[\![\sigma_c(T)]\!]$ è semplice. E se lo è, allora esiste un equivalente semplice $[\![R]\!]$.
  ■

Sotto-Lemma Se T è semplice, allora esiste R semplice t.c. $[\![R]\!] = [\![\sigma_c(T)]\!]$.

Dimostrazione del Sotto-Lemma
Per induzione su c.

- Caso base
  c è aritmetico o la sua negazione, che è proprio la definizione di semplice. In questo caso $R = \sigma_c(T)$.
- Caso induttivo $c = d \wedge e$
  Per ipotesi induttiva si ha che R_d è semplice t.c. $[\![R_d]\!] = [\![\sigma_d(T)]\!]$. A questo punto si applica di nuovo l'ipotesi induttiva a $\sigma_e(R_d)$. L'induzione su c vuol dire che in una condizione più semplice, come σ_e, si può sempre applicare l'ipotesi induttiva. Si ottiene R semplice t.c.
  $[\![R]\!] = [\![\sigma_e(R_d)]\!] = [\![\sigma_e(\sigma_d(T))]\!] = [\![\sigma_{d \wedge e}(T)]\!]$.
- Caso induttivo $c = d \vee e$
  Applichiamo l'ipotesi induttiva a $\sigma_d(T)$ ottenendo R_d semplice e a $\sigma_e(T)$ ottenendo R_e semplice.
  Definendo $R = R_d \cup R_e$ si ha che R è semplice ed inoltre
  $[\![R]\!] = [\![R_d \cup R_e]\!] = [\![R_d]\!] \cup [\![R_e]\!] = [\![\sigma_d(T)]\!] \cup [\![\sigma_e(T)]\!] = [\![\sigma_{d \vee e}(T)]\!]$
  Ad esempio, $\sigma_{(1=6 \vee 4>5)}(R) \sim \sigma_{1=6}(R) \cup \sigma_{4>5}(R)$ e $\sigma_{(1=6 \wedge 4>5)}(R) \sim \sigma_{1=6}(\sigma_{4>5}(R))$
  ■

Dimostrazione del teorema di Codd

Anch'essa viene fatta in due parti.

1. $\mathcal{AR} \subseteq \mathcal{CR}$
   È costruttiva: traduciamo una formula dell'algebra relazionale in formule del calcolo relazionale. Dunque si possono esprimere in ambedue modi.
   Per induzione sulla struttura di un'espressione dell'algebra relazionale. Bisogna fare un'assunzione sugli operatori di funzione. Una formula della formula è sempre equiparabile all'espressione dell'algebra.
   Definiamo **semplice** una query relazionale Q t.c. tutti gli operatori di selezione $\sigma_c(R)$ in Q sono t.c. c è un operatore aritmetico ($i = j$ o $i < j$ o $\neg(i < j)$ or $\neg(i = j)$) o la sua negazione. Quindi lo è, ad esempio, $\sigma_{1=6}(R)$ oppure $\sigma_{\neg(4<5)}(R)$. Non lo è, ad esempio, $\sigma_{1=6 \wedge 4<5}(R)$.
   Si prende una query Q di arietà m facente riferimento a $R_1, \ldots, R_k \implies$ formula F_Q sicura facente riferimento a $R_1, \ldots, R_k$ e con m variabili libere in $f_1, \ldots, f_m$. La dimostrazione procede per induzione su Q.

- Caso base $Q = R_i \in \{R_1, \ldots, R_k\}$
  $F_Q ::= R_i(f_1, \ldots, f_m)$
  Si verifica, dagli appunti sopra, come $[\![F_Q]\!] = [\![Q]\!]$ e inoltre F_Q è sicura perché tutte le variabili in esse sono limitate.
- Caso induttivo $Q = P^m \cup R^m$
  $F_Q ::= F_P \vee F_R$
  I due F_P e F_R sono date per ipotesi induttiva.
  Si verifica come $[\![F_Q]\!] = [\![F_P \vee F_R]\!] = [\![F_P]\!] \cup [\![F_R]\!] = [\![P]\!] \cup [\![R]\!] = [\![P \cup R]\!] = [\![Q]\!]$ e, per ipotesi induttiva, sappiamo che F_P e F_R sono sicure. Avendo medesima arietà si avrà che
  $FV(F_P) = \{f_1, \ldots, f_m\} = FV(F_R)$ ed entrambe saranno $= FV(F_Q)$ e quindi F_Q è sicura.
- Caso induttivo $Q = P - R$
  $F_Q ::= F_P \wedge \neg F_R$
  $[\![P - R]\!] = [\![F_P \cap \neg F_R]\!]$ e inoltre F_Q è sicura perché F_P è sicura e inoltre

$x \in FV(F_R) \implies x \in FV(F_P)$ e x è limitata. La sottoformula della negazione non garantisce che x è limitata; ma se appare in F_R allora appare anche in F_P, e dunque è limitata.

- Caso induttivo $Q = P \times R$
  $F_Q ::= F_P \wedge F'_R$
  con $F'_R = F_R$ dove le variabili libere $f_1, \dots, f_{ar(R)}$ vengono rinominate in $f_{ar(P)+1}, \dots, f_{ar(P)+ar(R)}$
  $ar(P) = 2$ allora $FV(F_P) = \{f_1, f_2\}$
  $ar(R) = 3$ allora $FV(F_R) = \{f_1, f_2, f_3\}$
  $FV(F'_R) = \{f_3, f_4, f_5\}$
  Questo viene fatto perché, ad esempio, con $R_1(f_1, f_2) \wedge R_2(f_1, f_2, f_3)$ si ha già il vincolo di uguaglianza per le coppie di f_1 e f_2.
- Caso induttivo $Q = \pi_{i_1,\dots,i_n}(P)$
  $F_Q ::= \exists f_{j_1}. \cdots . \exists f_{j_m}. F'_P$
  Si ha il vincolo di integrità usato per la proiezione $\{i_1, \dots, i_n\} \subseteq \{1, \dots, ar(P)\}$ e dunque ha senso, ad esempio, fare $\pi_{1,3}R^3_1$. Se il primo insieme lo si definisce come I e il secondo come $ar(P)$ allora possiamo usare $\{j_1, \dots, j_m\} ::= ar(P)/I$.
  $FV(F_Q) = \{f_{i_1}, \dots, f_{i_n}\}$
  Preso una tabella con due colonne R^2 si avrà $[\![\pi_1 R]\!] = [\![\exists f_2.\, R(f_1, f_2)]\!]$
- Caso induttivo $Q = \sigma_c(R)$
  Grazie al lemma possiamo supporre che Q sia semplice:
  $c ::= i = j | i < j | \neg(i = j) | \neg(i < j)$.
  In questa dimostrazione consideriamo $\$$ come una generica operazione e dunque
  $c ::= i\$j | \neg(i\$j)$.

$$F_Q ::= \begin{cases} F_R \wedge f_i \$ f_j & (\text{if } c = i\$j) \\ F_R \wedge \neg(f_i \$ f_j) & (\text{altrimenti } c = \neg(i\$j)) \end{cases}$$

  F_Q è sicura perché F_R sicura e $FV(F_Q) = FV(F_R)$.
  Per verificare che $[\![Q]\!] = [\![F_Q]\!]$ si può vedere che
  $(v_1, \dots, v_m) \in ([\![F_Q]\!](R_1, \dots, R_k)) \iff v_i \$ v_j \wedge (v_1, \dots, v_m) \in ([\![F_R]\!](R_1, \dots, R_k))$.

2. $\mathcal{CR} \subseteq \mathcal{AR}$
   Si può usare un sottolinguaggio logico in modo che $\mathcal{CR} \implies \text{DATALOG} \implies \mathcal{AR}$. Un programma di questo linguaggio è dato da un insieme finito di regole $M_1, \dots, M_n$.
   M_i avrà forma $H\text{:-}B_1 \& B_2 \& \cdots \& B_q$
   dove $H \rightsquigarrow P(A_1, \dots, A_n)$ in cui A_i è una variabile o una costante. Quindi, ad esempio, $P_1(X_1, X_2)$ e $P_2(\text{"Rossi"}, X_2, X_3)$
   dove

$$B \rightsquigarrow \begin{cases} \text{formula atomica o sua negazione} \\ \text{predicato } A = B, A < B \text{ o sua negazione} \\ \text{relazione } R_i \text{ o sua negazione} \end{cases}$$

   Un esempio di programma è dunque

$$P(x, y)\text{:-}R_1(x, z) \& R_2(x, y)$$

$$S(x, 20)\text{:-}R_4(x, 10) \& \neg(x < 7)$$

La traduzione $F \implies D_F$ deve fare in modo che entrambi abbiano la stessa semantica. Per fare ciò D_F deve avere un simbolo relazionale "principale" P_F.

- $F = G_1 \wedge \cdots \wedge G_n$
  dove le G_i non sono ulteriormente congiunzioni, e quindi non decomponibili in altre.
  $P_F(x_1, \ldots, x_n)$:-$G_1 \& G_2 \& \ldots \& G_n$
  dove $\{x_1, \ldots, x_n\} = FV(F)$
- $F = \exists X_i . G$
  $FV(G) = \{X_1, \ldots, X_n\}$
  per ipotesi induttiva abbiamo un programma D_G con simbolo principale P_G. Si definisce D_F come

$$P_F(X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_n) \text{:-} P_G(X_1, \ldots, X_n)$$

- $F = G \vee H$
  D_G con simbolo principale P_G e D_H con simbolo principale P_H.
  $P_F(X_1, \ldots, X_n)$:-$P_G(X_1, \ldots, X_n)$
  $P_F(X_1, \ldots, X_n)$:-$P_H(X_1, \ldots, X_n)$
  $FV(G) = FV(H) = \{X_1, \ldots, X_n\}$
- $F = G_1 \wedge \cdots \wedge G_m$
  dove $\exists G_i$ ulteriormente decomponibile.
  $P_F(X_1, \ldots, X_m)$:-$S_1 \& \ldots \& S_m$
  S_i è definito per casi:
  - Se G_i non è ulteriormente decomponibile, allora $S_i = G_i$ e $D_{G_i} = \emptyset$ perché magari è un predicato.
  - Se G_i è ulteriormente decomponibile, allora si applica la traduzione ottenendo D_{G_i} con simbolo principale P_{G_i} e poniamo $S_i = P_{G_i}.(X_1, \ldots, X_l)$ con $\{X_1, \ldots, X_l\} = FV(G_i)$. Quindi si applica finché non è più ulteriormente decomponibile.

Preso l'esempio per il club del tennis:

$$\begin{aligned} F = {} & [\exists p.\, \exists s.\, \exists c.\, \exists n.\, \exists o.\, \exists pp.\, \exists ps.\, R_1(p, c, n, f, o) \wedge R_2(p, s, pp, ps) \wedge (pp > ps)] \\ & \vee \\ & [\exists p.\, \exists s.\, \exists c.\, \exists n.\, \exists o.\, \exists pp.\, \exists ps.\, R_1(s, c, n, f, o) \wedge R_2(p, s, pp, ps) \wedge (ps > pp)] \end{aligned}$$

$$P_F(f) \text{:-} P_{first}(f)$$

$$P_F(f) \text{:-} P_{second}(f)$$

$$P_{first}(f) \text{:-} P_{body\ first}(p, s, c, n, o, pp, ps, f)$$

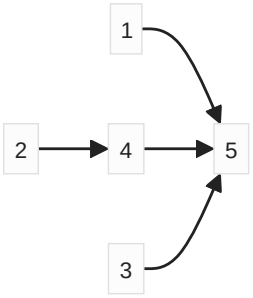
$$P_{second}(f) \text{:-} P_{body\ second}(p, s, c, n, o, pp, ps, f)$$

$$P_{body\ first}(p, s, c, n, o, pp, ps, f) \text{:-} R_1(p, c, n, f, o) \& R_2(p, s, pp, ps) \& (pp > ps)$$

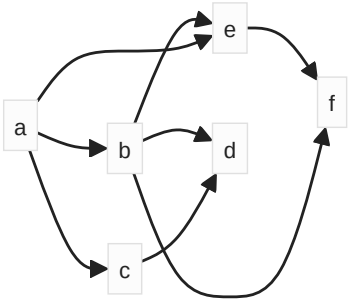
$$P_{second\ first}(p, s, c, n, o, pp, ps, f) \text{:-} R_1(s, c, n, f, o) \& R_2(p, s, pp, ps) \& (ps > pp)$$

Il grafo delle dipendenze è aciclico: utile quando si traduce in algebra relazionale.

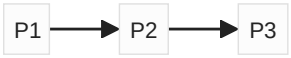
Per la conversione vi son 5 fasi definite in modo tale che servano come



1. Rettifica delle regole. Le regole vanno messe in una forma prestata meglio ad essere tradotta; chiaro che i due programmi sono equivalenti.
   Il programma consiste di n regole denominate $M_1, \ldots, M_n$. L'obbiettivo è che ciascuna regola la cui testa ha lo stesso simbolo ausiliario ha la stessa testa.
   $R_1(x, y, c, x)$:-$R_1(x, y) \longrightarrow R_1(x_1^R, x_2^R, x_3^R, x_4^R)$:-$R_1(x, y) \& x = x_1^R \& y = x_2^R \& c = x_3^R \& x = x_4^R$
   $R_1(x, x, z, d)$:-$x = d \& z = a \longrightarrow R_1(x_1^R, x_2^R, x_3^R, x_4^R)$:-$x = d \& z = a \& x = x_1^R \& x = x_2^R \& z = x_3^R \& d = x_4^R$
   $R_2(x, x)$:-$x = 1 \longrightarrow R_2(x_1^R, x_2^R)$:-$x = 1 \& x = x_1^R \& x = x_2^R$
   Si introducono variabili fresche $X_1^R, \ldots, X_m^R$ per ciascun simbolo ausiliario R di arietà m.
   Una regola $M_i \equiv R(A_1, \ldots, A_m)$:-$B_1 \& \ldots \& B_q$
   diventa $M_i' \equiv R(X_1^R, \ldots, X_m^R)$:-$B_1 \& \ldots \& B_q \& A_1 = X_1^R \& \ldots \& A_m = X_m^R$
2. Calcolo dell'espressione "DOM": predicato unario dentro la query, dominio di essa. In Datalog è implicita.
   L'obbiettivo è che DOM deve valutarsi nel predicato di arietà 1 che contiene i valori che occorrono nella base di dati e nel programma Datalog di partenza $\{M_1, \ldots, M_n\}$.
   Mettiamo insieme i valori che occorrono in una R_i di arietà m.
   $Q_{R_i} \equiv \pi_1(R_i) \cup \pi_2(R_i) \cup \cdots \cup \pi_m(R_i)$
   quindi avremo, in un unico enorme insieme, tutti i dati. Può sembrare inutile fare tutte le proiezioni di R, perché è come prendere tutta R. Il problema però è che R, essendo un insieme di tuple, è ordinato. Facendo le proiezioni togliamo l'ordine prendendo solo i valori, escludendo le colonne.
   Per i valori nel programma Datalog definiamo:
   $Q_{\{M_1, \ldots, M_n\}} \equiv \{(d_1)\} \cup \{(d_2)\} \cup \cdots \cup \{(d_k)\}$
   dove $d_1, \ldots, d_k$ sono le costanti che occorrono nel programma. Ad esempio $d_2 = 1980$ aggiunge il filtro dell'anno 1980 alla query.
   $\mathrm{DOM} \equiv Q_{R_1} \cup \cdots \cup Q_{R_j} \cup Q_{\{M_1, \ldots, M_n\}}$
   quello che fa DOM, definito in un modo coretto, è quello di prendere tutti i valori in cui occorrono nelle tabelle del database e come costanti nel programma Datalog.
3. Calcolo dell'ordine topologico del grado delle dipendenze.
   Dato, ad esempio, il seguente grafo aciclico



$a > b > c > d > e > f$
Se $m_j > m_i$ allora non esiste un cammino da m_j a m_i.
Il nodo di arrivo appare nel nodo di partenza che appare come testa. Quindi, se si ha qualcosa del tipo
P_1:-P_2
P_2:-P_3
allora il grafo sarà del tipo



4. Calcolo di un'espressione di $\mathcal{AR}$ per ciascuna regola del programma: ad ogni corpo.
   Presa le regola
   $\text{Anni}(A)$:- $\text{Soci}(idv, c, n, a, s)\&\text{Partite}(idv, idp, pv, pp)\&\neg(pv \leq pp)$
   si traduce creando la relazione
   $\pi_4(\sigma_{(1=6)\wedge\neg(8\leq 9)}(\text{Soci} \times \text{Partite}))$
   $ar(\text{Soci}) = 5, ar(\text{Partite}) = 4$

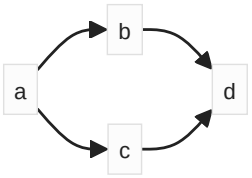
$P(X_1, X_2)$:-$\neg R(X_1, X_2, X_4)\&Q(X_3)\&X_2 = X_3\&X_4 = a\&X_1 = b$
Negare R vuol dire fare $\text{DOM} - R$ che, in questo caso, visto che R ha 3 variabili, sarà fatta come relazione $\text{DOM}^3 - R$.
$\pi_{1,5}(\sigma_c(\text{DOM}^3 - R \times Q \times \text{DOM} \times \{(a)\} \times \{(b)\}))$
Possiamo numerarli in base alle arietà, in cui sono tutte 1 ad eccezione della prima sottrazione che è di 3.
$(\star)$
Il corpo è espresso con l'espressione con la condizione $c = (1 = 2) \wedge (4 = 5) \wedge (3 = 6) \wedge (1 = 7)$

Preso, in modo generico, $P(X_1, \ldots, X_P)$:-$B_1\& \ldots \&B_n$ bisogna distinguere la parte relazionale (tutti i B nella forma $R(a_1, \ldots, a_n)$ oppure $\neg(R(a_1, \ldots, a_n))$ e sottoinsieme definito come $B_{i1}, \ldots, B_{il}$) e la parte aritmetica (tutti i B nella forma $a_i\$a_j$ e $\neg(a_i\$a_j)$ e sottoinsieme definito come $B_{j1}, \ldots, B_{jk}$).
$\{x_1, \ldots, x_p, x_{p+1}, \ldots, x_{p+c}\}$
dove $\{x_{p+1}, \ldots, x_{p+c}\}$ sono le variabili del corpo e $\{x_1 \ldots, x_p\}$ compaiono sia nel corpo che nella testa.
Nelle prime si può isolare un sotto insieme chiamato VARRED in cui compaiono solo nella parte relazionale.
$Q = Q_1 \times \cdots \times Q_l \times \text{DOM}^{c+p-|\text{VARRED}|} \times \{(a_1)\} \times \cdots \times \{(a_k)\}$
dove
$a_1, \ldots, a_k$ sono le costanti che occorrono nella regola
l è l'ultimo B della parte relazionale.
Q_h viene costante da B_{ih} (vi sono due indici perché la B può essere distribuita in punti arbitrari nel corpo; avremmo potuto mettere anche $C_i \in \{B_1, \ldots, B_l\}$ ed usare quello).
-> Se $B_{ih} = R(A_1, \ldots, A_n) \implies Q_h = R$
-> Se $B_{ih} = \neg(R(A_1, \ldots, A_n)) \implies Q_h = \text{DOM}^R - R$
-> $index(i)$ insieme degli indici in Q corrispondente a X_i
-> $pos(i)$ è uno degli elementi di $index(i)$
-> $and(i)$ è la congiunzione logica $(pos(i) = q_1) \wedge \cdots \wedge (pos(i) = q_l)$ dove $\{pos(i), q_1, \ldots, q_l\} = index(i)$
-> $pos(a)$ è l'indice in Q della relazione costante $\{(a)\}$.
È possibile sulla base di $pos(i)$ e $pos(a)$ definire una congiunzione $c_1 \wedge \cdots \wedge c_k$ dove c_i è ottenuto da B_{ji} sostituendo X_i con $pos(i)$ e a con $pos(a)$: modo più complesso di definire la stessa cosa di $(\star)$.

La query finale sarà

$$\pi_{pos(1)\ldots pos(p)}(\sigma_{c_1\wedge\cdots\wedge c_k\wedge\bigwedge_{i=1} and(i)}(Q))$$

5. Calcola un'espressione di $\mathcal{AR}$ per ciascuna relazione ausiliaria del programma: ad ogni testa. Costruzione per induzione sulla posizione del simbolo posizionale ausiliario. Nell'ordinamento topologico del grafo delle dipendente come $a > b > c > d$.



- Caso base
  Per i punti 1 e 4 sappiamo che le relative regole ausiliarie nella forma
  $R(X_1, \ldots, X_n)$:-B_1
  $\vdots$
  $R(X_1, \ldots, X_n)$:-B_p

Dal punto 1 sappiamo i $X_1, \ldots, X_n$ mentre per il punto 4 le query $Q_1, \ldots, Q_p$ per i corpi.
La query sarà:

$$\pi_{i_1^1}\ldots\pi_{i_n^1}(Q_i)\cup\cdots\cup\pi_{i_1^p}\ldots\pi_{i_n^p}(Q_p)$$

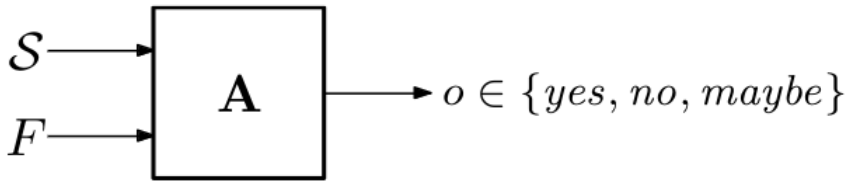
dove i_j^s è l'indice della variabile x_j nella query s.

- Caso induttivo
  Viene fatto per qualche simbolo che appare dentro il grafo delle dipendenze e supporre che abbiamo le query per un generico simbolo ausiliario che appare.
  $R'(X_1, \ldots, X_m)$:-B_1
  $\vdots$
  $R'(X_1, \ldots, X_m)$:-B_q
  Quindi, per punto 4 o per ipotesi induttiva, si hanno query corrispondenti Q_1 e Q_p. Nel caso induttivo appaiono solo simboli ausiliari, ma se lo fanno, allora abbiamo comunque la query corrispondente.

■

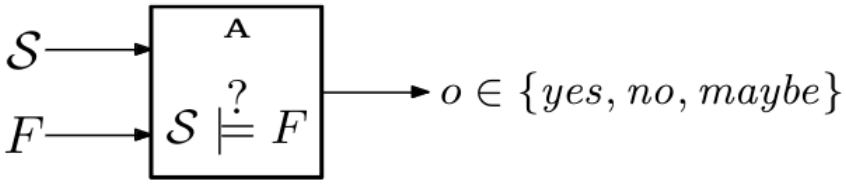
11. Logica e verifica

Un programma concorrente viene sotto posto a verifica (parziale o totale) in fase *mission critical*.
Data la descrizione di un sistema S e una proprietà generica P che descrive il comportamento atteso, come si verifica che S soddisfa effettivamente P?
In genere si può rappresentare in modo schematico come segue:



il maybe è dovuto alla possibile indecidibilità.

Nel model checking si fa verifica considerando la proprietà P come una formula di una logica e il sistema come interpretazione per essa.



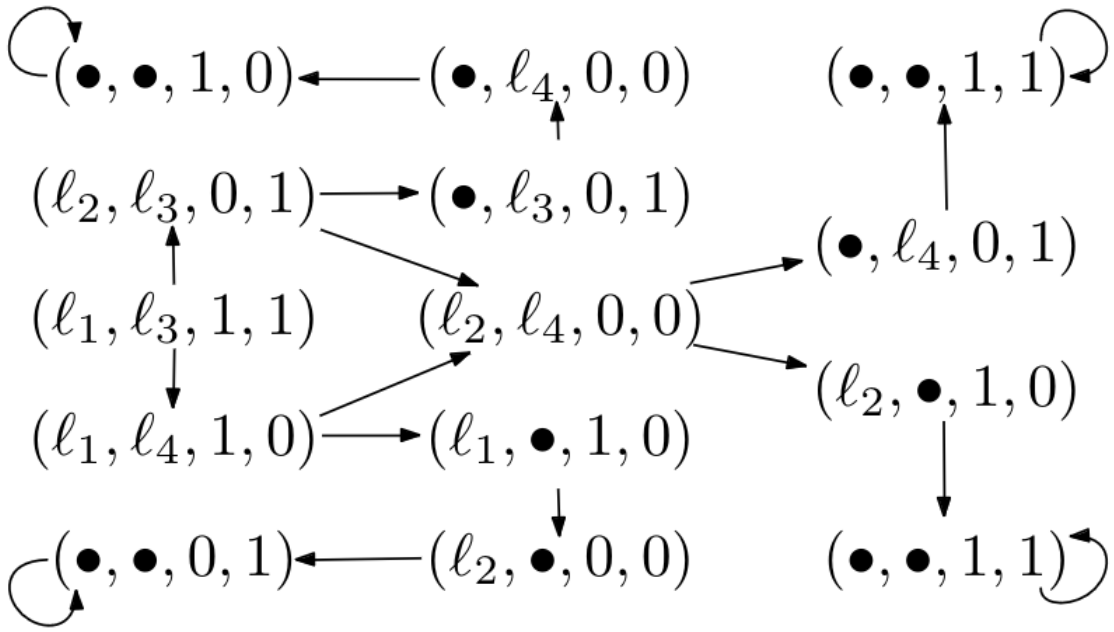
Strutture di Kripke

AP è un insieme di proposizioni atomiche che catturano la proprietà di interesse di uno stato.
Una struttura di Kripke su AP è $\mathcal{M} = (S, S_0, R, L)$ dove

- S è l'insieme di stati;
- $S_0 \subseteq S$ è l'insieme degli stati iniziali;
- $R \subseteq S \times S$ è la relazione di transizione. È supposta totale, ovvero $\forall s \in S \exists t \in S : (s, t) \in R$;
- $L : S \to \mathcal{P}(AP)$ è una funzione di etichettatura: dice quale proposizione atomica vale in ogni stato.

L'insieme degli stati è posto finito per garantire decidibilità.

Preso un programma concorrente $(l_1 : x \leftarrow 0; l_2 : y \leftarrow 1)||(l_3 : y \leftarrow 0; l_4 : x \leftarrow 1)$ l'insieme degli stati può essere $\{l_1, l_2, \circ\} \times \{l_3, l_4, \circ\} \times \{0, 1\} \times \{0, 1\}$, con stato iniziale $\{l_1, l_3, 1, 1\}$.



Etichettiamo con le proposizioni `null` lo stato in cui sia x che y sono $= 0$ e con `stop` la terminazione del programma.

Qui la verità è un'istanza di uno specifico stato. Se vogliamo generalizzarla su ogni stato iniziale si può dire che $\mathcal{M} \models F \iff \forall S \in S_0 : \mathcal{M}, S \models F$.

Ad esempio, si possono avere le proprietà di:

- Reachability, quando si raggiunge un determinato nodo del grafo.
- Safety, quando almeno uno delle due variabili è 1.

> Negli esempi e dimostrazione futura si useranno spesso le formule ϕ e ψ al posto di F e G per evitare dubbi con l'operatore *future*.

Un operatore modale permette di esprimere una formula che vale in futuro o in un determinato stato del futuro:

- $\mathrm{G}\ \phi$ (globally) rappresentante "in tutti gli stati globali è vera la formula ϕ";
- $\mathrm{X}\ \phi$ (oppure $\mathrm{n}\ \phi$) rappresentante come "nel prossimo passo vale ϕ";
- $\mathrm{F}\ \phi$ (future) rappresentante come "in un futuro varrà ϕ".
- $\phi\ \mathrm{U}\ \psi$ (until), in cui ϕ deve essere vero finché ψ è falso. Una volta che ψ diviene vero, non è più importante lo stato ϕ.
- $\phi\ \mathrm{R}\ \psi$ (releases), finché ϕ è falso, allora ψ deve essere vero.

I quantificatori sono considerati cammini:

- $\mathrm{A}\ \phi$ (in tutti i cammini vale ϕ)
- $\mathrm{E}\ \phi$ (esiste almeno un cammino tale che vale ϕ).

CTL*

È una logica temporale. Con pedice S si rappresenta lo stato; con pedice P il cammino.
Le formule di stato vengono valutate su uno specifico stato.

$$\phi_S, \psi_S, ::= P | \phi_S \wedge \psi_S | \phi_S \vee \psi_S | \neg \phi_S | \mathrm{E}\ \phi_P | \mathrm{A}\ \phi_P$$

A sta per $\forall$.
E sta per $\exists$.

Nelle formule di cammino si guarda in tutto il cammino.

$$\phi_P, \psi_P ::= \phi_S | \phi_P \wedge \psi_P | \phi_P \vee \psi_P | \neg \psi_P | \mathrm{X}\ \phi_P | \mathrm{F}\ \phi_P | \mathrm{G}\ \phi_P | \phi_P\ \mathrm{U}\ \psi_P | \phi_P\ \mathrm{R}\ \psi_P$$

Alcuni di questi operatori temporali si possono esprimere come punti fissi (minimi o massimi; massimi perché i cammini sono infiniti).

Un cammino π in una struttura $\mathcal{M} = (S, S_0, R, L)$ è una sequenza infinita di stati
$\forall n \in \mathbb{N} : s_0 s_1 \cdots \in S^\omega$ t.c. $(s_n, s_{n+1}) \in R$.
Un n-esimo suffisso di π è π^n, che è anch'esso un cammino dato che il percorso è comunque infinito. Un esempio di ciò si ha guardando le proprietà di un server in esecuzione.

Una formula di cammino ϕ_P su AP è vera in $\mathcal{M}$ su AP e in un π in $\mathcal{M}$.
$\mathcal{M}, \pi \models \phi_P$

I connettivi vengono interpretati in modo standard.
I quantificatori fanno riferimento alla semantica delle formule di cammino.

- Per almeno un π che inizia in s: $\mathcal{M}, s \models (\mathrm{E}\ \phi_P) \iff \mathcal{M}, \pi \models \phi_P$
- Per tutti i π che iniziano in s: $\mathcal{M}, s \models (\mathrm{A}\ \phi_P) \iff \mathcal{M}, \pi \models \phi_P$

Le formule di stato vengono valutate nel primo stato del cammino: $\mathcal{M}, \pi \models \phi_S \iff \mathcal{M}, s \models \phi_S$.
Nei cammini invece si ha:

- $$\mathcal{M}, \pi \models (\mathrm{X}\ \phi_P) \iff M, \pi^1 \models \phi_P$$
- $$\mathcal{M}, \pi \models (\mathrm{F}\ \phi_P) \iff M, \pi^i \models \phi_P$$

  per almeno un i.
- $$\mathcal{M}, \pi \models (\mathrm{G}\ \phi_P) \iff M, \pi^i \models \phi_P$$

  per tutti gli i;
- $$\mathcal{M}, \pi \models (\phi_P\ \mathrm{U}\ \psi_P) \iff \exists k \in \mathbb{N} : \mathcal{M}, \pi^k \models \psi_P \text{ e } \mathcal{M}, \pi^j \models \phi_P \forall 0 \leq j < k$$
- $$\mathcal{M}, \pi \models (\phi_P\ \mathrm{R}\ \psi_P) \iff \forall j \in \mathbb{N} \text{ if } \forall i < j, \mathcal{M}, \pi^i \not\models \phi_P \implies \mathcal{M}, \pi^j \models \psi_P$$

Logica CTL

Frammento di CTL in cui ogni operatore temporale è preceduto da un operatore sui cammini. Si hanno le medesime formule di stato, ma quelle di cammino sono semplificate; non si hanno formule di cammino complesse dato che vi è un semplice operatore modale davanti una formula di stato.

$$\phi_P, \psi_P ::= \mathrm{X}\ \phi_S | \mathrm{F}\ \phi_S | \mathrm{G}\ \phi_S | \phi_S\ U\ \psi_S | \phi_S\ \mathrm{R}\ \psi_S$$

Ad esempio, $\mathrm{A}\ (\phi_p \vee \phi_q)$ non è in CTL, perché dovrebbe essere $\mathrm{A}\ \phi_p$.
Ad esempio $\mathrm{X}\ \phi_p$ non è in CTL, perché dovrebbe essere $\mathrm{A}\ \mathrm{X}\ \mathrm{A}\ \phi_p$.

Logica LTL

Frammento di CTL in cui non si hanno formule di stato. Implicitamente vengono quantificate col quantificatore A.

$$\phi_P, \psi_P ::= P | \phi_P \wedge \psi_P | \phi_P \vee \psi_P | \neg \phi_P | \mathrm{X}\ \phi_P | \mathrm{F}\ \phi_P | \mathrm{G}\ \phi_P | \phi_P\ \mathrm{U}\ \psi_P | \phi_P\ \mathrm{R}\ \psi_P$$

Ad esempio, $\phi_p \vee \phi_q$ in CTL* sarebbe $\mathrm{A}\ (\phi_p \vee \phi_q)$.

In LTL si possono esprimere condizioni booleane non esprimibili in CTL.
La logica modale si trova in CTL; non è possibile esprimerla in LTL.
Il modal μ-calcolo è una logica del punto fisso con dentro anche CTL*.

Problema del model checking

Data una struttura di Kripke $\mathcal{M} = (S, S_0, R, L)$ e una formula di stato ϕ_S si determina:

- Universale: se $\forall s \in S_0 : \mathcal{M}, s \models \phi_S$
- Esistenziale: se $\exists s \in S_0 : \mathcal{M}, s \models \phi_S$

I problemi del model checking universale ed esistenziale sono PSPACE completi per CTL*.
In CTL sono in POLY.

Esempi

Esempio 1
Prese le richieste $\{1, \dots, n\}$. In AP si hanno due proposizioni generiche:

- $\mathrm{required}[i]$ con $i \in \{1, \dots, n\}$
- $\mathrm{acknowledged}[i]$ con $i \in \{1, \dots, n\}$

Ogni richiesta ricevuta sarà riconosciuta dal sistema prima o poi.

$$\mathrm{requested}[i] \rightarrow \mathrm{A}\ \mathrm{F}\ \mathrm{acknowledged}[i]$$

per avere tale cosa per un determinato stato s si avrebbe:

$$\mathcal{M}, s \models \mathrm{requested}[i] \rightarrow \mathrm{A}\ \mathrm{F}\ \mathrm{acknowledged}[i]$$

ma non ha senso nel caso si abbia

per avere una formula di cammino si avrebbe, nel caso globale,

$$\mathrm{G}(\mathrm{requested}[i] \rightarrow \mathrm{A\ F\ acknowledged}[i])$$

che in formula di stato diviene

$$\mathcal{M}, s_0 \models \bigwedge_{i=1}^{n} \mathrm{A\ G}(\mathrm{requested}[i] \rightarrow \mathrm{A\ F\ acknowledged}[i])$$

Esempio 2: sistema di controllo di un ascensore
In h piani, avremo AP:

- $\mathrm{floor}[i]$ con $i \in \{1, \ldots, h\}$
- $\mathrm{direction}[d]$ con $d \in \{up, down\}$
- $\mathrm{buttonPressed}[i]$ con $i \in \{1, \ldots, h\}$
  Se un ascensore sta scendendo e si trova ad un piano superiore al quinto, mentre il pulsante del quinto piano è premuto, deve continuare a scendere finché non raggiunge il quinto piano.

  $$\mathrm{A\ G}\ \left(\mathrm{direction}[down] \wedge \bigvee_{i=6}^{h} \mathrm{floor}[i] \wedge \mathrm{buttonPressed}[5] \rightarrow \mathrm{A}\ (\mathrm{direction}[down]\ \mathrm{U}\ \mathrm{floor}[5])\right)$$

  generalizzato per un determinato piano j si avrebbe

  $$\bigwedge_{j=1}^{n} \mathrm{A\ G}\ \left(\mathrm{direction}[down] \wedge \bigvee_{i=j+1}^{h} \mathrm{floor}[i] \wedge \mathrm{buttonPressed}[j] \rightarrow \mathrm{A}\ (\mathrm{direction}[down]\ \mathrm{U}\ \mathrm{floor}[j])\right)$$

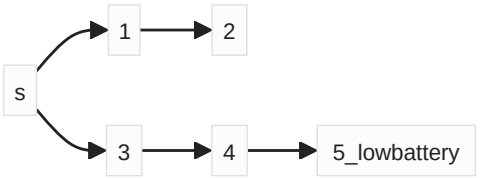
Esempio 3: sistema di allarme che segnala presenza di eventi anomali con garanzie temporali
In AP si hanno due sensori e due attuatori:

- $\mathrm{intrusion}$
- $\mathrm{lowbattery}$
- alarm
- signal

Si vuole che si scatti l'allarme entro 3 istanti (unità di tempo che occorre tra uno stato e il prossimo) se avviene un'intrusione.

$$\mathrm{A\ G}\ (\mathrm{intrusion} \rightarrow (\mathrm{A\ X\ alarm} \vee \mathrm{A\ X\ A\ X\ alarm} \vee \mathrm{A\ X\ A\ X\ A\ X\ alarm} \vee \mathrm{alarm}))$$

Se c'è la possibilità di rilevare che la batteria è scarica nei prossimi n istanti, occorre segnalarlo immediatamente e per almeno due istanti.



$$\mathrm{A\ G}\ (\mathrm{E\ X}^{n}\ \mathrm{lowbattery} \rightarrow \mathrm{signal} \wedge \mathrm{A\ X\ signal} \wedge \mathrm{A\ X\ A\ X\ signal})$$

con:
$\mathrm{E\ X}^0\phi \equiv \phi$
$\mathrm{E\ X}^{n+1}\phi \equiv \phi \vee \mathrm{E\ X\ (E\ X}^n\phi)$
però non è un arbitrario n, ma è fissato. Se così non fosse si potrebbe semplicemente usare $\mathrm{E\ F\ lowbattery}$.

CTL model checking

Presa $\mathcal{M}$ e ϕ si fa model checking in CTL con risultato P_ϕ. $\mathcal{M}, s \models \phi \iff s \in P_\phi$
Equivalenza logica tra formule in CTL
$\phi \equiv \psi$ se $\forall \mathcal{M} \forall s (\mathcal{M}, s \models \phi \iff \mathcal{M}, s \models \psi)$
Vogliamo costruire l'insieme di stati in cui ϕ è vera.

Lemma
$\mathrm{A\ X}\ \phi \equiv \neg(\mathrm{E\ X}\ \phi)$
$\mathrm{A\ G}\ \phi \equiv \neg(\mathrm{E\ F}\ \neg\phi)$
$\mathrm{A}(\phi\ \mathrm{R}\ \psi) \equiv \neg\mathrm{E}(\neg\phi\ \mathrm{U}\ \neg\psi)$
$\mathrm{A\ F}\ \phi \equiv \neg\mathrm{E\ G}\ \neg\phi$
$\mathrm{E\ F}\ \phi \equiv \mathrm{E}(\mathrm{true\ U}\ \phi)$
$\mathrm{E}(\phi\ \mathrm{R}\ \psi) \equiv \neg A(\neg\phi\ \mathrm{U}\ \psi)$
$\mathrm{A}(\phi\ \mathrm{U}\ \psi) \equiv \neg(\mathrm{E}\ (\neg\psi\ \mathrm{U}\ (\neg\phi \wedge \neg\psi))) \wedge \neg(\mathrm{E\ G}\ \neg\psi)$

È un algoritmo polinomiale visto che parliamo di esistenziali.

Il valore $|\phi|$ è il numero di sottoformule di ϕ. Ad esempio,
$|\mathrm{E\ G(A\ F(alarm} \vee \mathrm{signal))}| = 5$

$\mathrm{CTLModelChecking}((S, S_0, R, L), \phi)$:
    for $\psi \sqsubseteq \phi$:
        $\mathrm{Done}[\psi] = \mathrm{false}$
    while $\neg\mathrm{Done}[\phi]$ do :
        pick ψ such that $\neg\mathrm{Done}[\psi]$ and $\mathrm{Done}[\psi']\forall\psi' \sqsubseteq \psi$
        match ψ with :
            $P \in AP \mapsto \mathrm{States}[\psi] = \{s \in S | P \in L(s)\}$
            $\neg\chi \mapsto \mathrm{States}[\psi] = S - \mathrm{States}[\chi]$
            $\psi_1 \wedge \psi_2 \mapsto \mathrm{States}[\psi] = \mathrm{States}[\psi_1] \cap \mathrm{States}[\psi_2]$
            $\psi_1 \vee \psi_2 \mapsto \mathrm{States}[\psi] = \mathrm{States}[\psi_1] \cup \mathrm{States}[\psi_2]$
            $\mathrm{E\ X}\ \psi' \mapsto \mathrm{States}[\psi] = \{s \in S | \exists q.\ (s, q) \in R \wedge q \in \mathrm{States}[\psi']\}$
            $\mathrm{E}(\psi_1 \cup \psi_2) \mapsto \mathrm{States}[\psi] = \mathrm{CheckEU}(\psi_1, \psi_2, \mathrm{States})$
            $\mathrm{E\ G}\ \psi' \mapsto \mathrm{States}[\psi] = \mathrm{CheckEG}(\psi', \mathrm{States})$
            $\mathrm{Done}[\psi] = \mathrm{true}$
        return $\mathrm{States}[\psi]$

$\mathrm{CheckEU}(\psi_1, \psi_2, \mathrm{States})$:
    $\mathrm{StatesInc} = \mathrm{States}[\psi_2]$
    repeat :
        $\mathrm{Aux} = \mathrm{StatesInc}$

$\text{StatesInc} = \text{StatesInc} \cup \{s \in \text{States}[\psi_1] | \exists q.\, (s, q) \in R \wedge q \in \text{Aux}\}$

until $\text{Aux} = \text{StatesInc}$

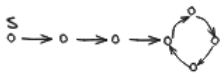
return StatesInc

Lemma

$\mathcal{M}, s \models \text{E G } \psi' \iff$ è possibile costruire un cammino che da s porti a uno SCC massimale e non triviale contenente solo stati che soddisfano ψ'.

Dimostrazione

$(\Longleftarrow)$



$(\Longrightarrow)$

C'è un cammino π_i dove ψ' vale $\pi = ss_0s_1s_2 \ldots \rho$. Tutti gli stati in ρ occorrono in π un numero infinito di volte. Gli stati prima di ρ occorrono in π un numero finito di volte.
La SCC sarà l'insieme di stati che occorrono in ρ. Un qualsiasi stato $s' \in \rho$ avrà sempre un'occorrenza $s'' \in \rho$ e viceversa.

■

$\text{CheckEG}(\psi', \text{States})$:

Determiniamo le SCC massimali e non triviali di $\mathcal{M}$ che contengono solo stati in $\text{States}[\psi']$

Verifica da quali stati in $\text{States}[\psi']$ tali SSC siano raggiungibili