Appunti Fondamenti Logici per l'informatica

Gabriele Genovese

February 14, 2024

Contents

1 Primo modulo: λ-Calcolo

1.1 Formalismo di calcolo

Un formalismo di calcolo è un formalismo che risponde alla domanda "cosa vuol dire calcolare".

1.1.1 Formalismo

Un formalismo è una descrizione matematicamente rigorosa di un fenomeno, in genere ottenuta tramite manipolazione di espressioni simboliche.

1.1.2 Esempi formalismi di calcolo

λ-calcolo, macchine di Turing, sistemi di Post, funzioni primitive ricorsive con operatore di minimizzazione, Random Access Machines, linguaggi di programmazione rigorosamente specificati, etc...

1.2 Tesi di Church-Turing

Tesi intesa come congettura, perché è impossibile da dimostrare.

> Ogni funzione calcolabile da un formalismo di calcolo sufficientemente espressivo è calcolabile da una macchina di Turing e viceversa.

Tutti i formalismi sono equivalenti dal punto di vista di cosa calcolano. Formalismi diversi hanno punti di forza/debolezza diversi (come i linguaggi di programmazione).

1.3 Macchine di Turing contro λ-calcolo

Nelle macchine di Turing "calcolare" significa modificare un supporto fisico di celle discrete, ognuna contenente una quantità finita di informazione. Il Calcolo è ottenuto tramite operazioni locali (una testina r/w si muove sul supporto fisico). Oltre al supporto fisico, la macchina è in uno stato scelto da un insieme finito.

Nel λ-calcolo "calcolare" significa semplificare delle espressioni. Le espressioni sono funzioni. In particolare, le funzioni prendono in input funzioni e danno in output funzioni.

Macchine di Turing	**λ-calcolo**
Ogni passo $O(1)$ (tempo)	Implementazione naif di un passo: $O(n^2)$
Ogni cella $O(1)$ (spazio)	Implementazione efficiente: $O(???)$
Ottimo per lo studio della complessità	Pessimo per lo studio della complessità

Table 1: Differenze riguardanti la complessità

Macchine di Turing	**λ-calcolo**
Imperativo, ma $\neq$ dai linguaggi imperativi	Cuore di tutti i linguaggi funzionali
Non composizionale	Composizionale
Basso livello	Alto livello
Difficile implementare costrutti/-dati/meccanismi	Facile implementare costrutti/-dati/meccanismi
Definizione non ricorsiva, quindi fare prove formali è complesso	Definizioni ricorsiva, quindi fare prove formali è facile per induzione
Pessimo per lo studio dei linguaggi di programmazione	Ottimo per lo studio dei linguaggi di programmazione
Ad-hoc	Controparte computazionale della logica

Table 2: Differenze riguardanti le caratteristiche

1.4 Macchine di Turing

Una macchina di Turing è definita da una tupla (A, Q, q_0, q_f, δ) dove:

- L'alfabeto A è un insieme non vuoto, finito di simboli
- L'insieme di stati Q è un insieme non vuoto, finito di stati
- $q_0 \in Q$ è lo stato iniziale
- $q_f \in Q$ è lo stato finale
- La funzione di transizione δ ha dominio $A \to Q$ e codominio $A \to Q \to \{L, R\}$

1.4.1 Stato di una macchina di Turing

Lo stato di una macchina di Turing (A, Q, q_0, q_f, δ) è una tripla (α, i, q) dove:

- Il nastro infinito α è una funzione da $\mathbb{Z}$ a A
  Intuizione: $\alpha(k) = a$ sse la k-esima cella del nastro contiene il simbolo a
- $i \in \mathbb{Z}$ è la posizione della testina sul nastro
  Intuizione: la testina è posizionata sulla i-esima cella di contenuto $\alpha(i)$

- $q \in Q$ è lo stato corrente

Uno stato (α, i, q) è iniziale di input α sse $i = 0$ e $q = q_0$ e finale di output α sse $q = q_f$.

1.4.2 Esecuzione di una macchina di Turing

Una macchina di Turing (A, Q, q_0, q_f, δ) in uno stato (α, i, q) non finale transisce in un nuovo stato (α', i', q') se:

- $\delta(\alpha(i), q) = a, q', x$
  Intuizione: la testina legge il contenuto $\alpha(i)$ della cella corrente i, lo stato corrente si aggiorna da q a q' e ...
- $\alpha'(i) = a$ e $\alpha'(n) = \alpha(n)$ per $n \neq i$
  ... la testina sovrascrive il valore della cella con a e ...
- $i' = i + 1$ se $x = R; i' = i - 1$ se $x = L$
  ... si muove a destra o a sinistra a seconda del valore di x.

1.4.3 Programmare una macchina di Turing

Programmare una macchina di Turing significa definire per ogni stato e per ogni simbolo una tripla della mossa che corrisponde a uno stato.

1.5 λ-calcolo

1.5.1 Sintassi

Tutto è una funzione unaria (cioè, con un solo input) anonima: $t ::= x \mid tt \mid \lambda x.t$ dove:

- t viene chiamato termine (useremo $t, s, u, v, M, N...$ per indicare un termine; $x, y, z, w...$ indica occorrenza di una variabile)
- $t_1 t_2$ (chiamata applicazione) è la chiamata di funzione: passo t_2 in input alla funzione t_1 (notazione matematica standard: $t_1(t_2)$)
- $\lambda x.t$ (chiamata astrazione) è la una funzione anonima il cui parametro formale è x e il cui corpo è t (notazione matematica standard: $x \to t$ o anche $f(x) = t$ se la funzione avesse un nome f)

1.5.2 Esempi

- $\lambda x.x$ è la funzione identità: prende in input x e lo restituisce in output
- $\lambda(x.x)(y.y)$ applica la funzione identità a un'altra copia della funzione identità. Il risultato atteso è la funzione identità $\lambda y.y$
- $\lambda x.y$ è la funzione costante che ignora l'input x e restituisce sempre y

- $\lambda(x.y)(z.z)$ applica la funzione costante alla funzione identità. Il risultato atteso è y
- $\lambda x.xx$ prende in input una funzione x e la applica a se stessa
- $\lambda x.y.xy$ prende in input una funzione x e restituisce una funzione che prende in input una y e applica x a y
- $\lambda(x.y.xy)(z.z)$ ridurrà a $\lambda y.(z.z)y$ che ridurrà alla funzione identità $\lambda y.y$

1.5.3 Regole di precedenza e associatività

- L'applicazione ha la precedenza sull'astrazione: $\lambda x.xx$ si legge come $\lambda x.(xx)$ e non come $(\lambda x.x)x$
- L'applicazione è associativa a sinistra: xyz si legge come $(xy)z$ e non come $x(yz)$

1.5.4 Funzioni n-arie e applicazione parziale

Una funzione binaria $f(x, y) = g(x, y)$ può essere vista come una funzione unaria che restituisce una funzione unaria $\lambda x.\lambda y.gxy$. Da notare che il passaggio simultaneo di una coppia di input $g(x, y)$ viene codificato con il passaggio sequenziale di un input alla volta: gxy, ovvero $(gx)y$.

Il vantaggio principale è che le funzioni possono essere *applicate parzialmente* passando solamente il primo parametro: $((\lambda x.\lambda y.x + y)2$ riduce alla funzione $\lambda y.2 + y$ che incrementa un numero di 2.

1.5.5 Riduzione

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

Esempio: $(\lambda x.yx)(zz)$ riduce a $y(zz)$. Tuttavia devo fare attenzione a definire correttamente la nozione di sostituzione.

1.5.6 Variabili libere e legate

I nomi dati ai parametri formali non sono importanti: $\lambda x.x$ e $\lambda y.y$ sono la stessa funzione. Tuttavia i nomi delle variabili globali lo sono eccome: $\lambda x.y$ e $\lambda x.z$ sono due programmi diversi.

Intuizione: non posso rimpiazzare uno con l'altro in un contesto dove $y = 0$ e $z = 1$ senza ottenere risultati diversi. Introduciamo una terminologia.

Il λ nell'astrazione $\lambda x.t$ è un *binder*: esso lega la variabile x nel corpo t. Una variabile che non è legata si dice *libera*.

Nel λ-calcolo le variabili legate sono tutte parametri formali (c'è un solo binder) e quelle libere sono tutte variabili globali.

1.5.7 Definizione dell'Insieme delle Variabili Libere

L'insieme $FV(t)$ delle variabili libere di t si calcola come segue:

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

Esempio: $FV(\lambda x.xy(\lambda y.yz)) = \{y, z\}$. Nota: nell'esempio la prima occorrenza di y è libera, mentre la seconda è legata.

1.5.8 α-conversione $\equiv_\alpha$

Due λ-termini t_1 e t_2 sono α-convertibili (i.e. $t_1 \equiv_\alpha t_2$) se posso ottenere l'uno dall'altro ridenominando le sole variabili legate in modo tale che le occorrenze legate di una variabile in posizione corrispondente nei due termini siano legate dai binder in posizione corrispondente e che le occorrenze di variabili libere abbiano in posizione corrispondente una variabile libera con lo stesso nome.

Esempi:
$\lambda x.\lambda y.xyz \equiv_\alpha \lambda y.\lambda w.ywz$
$\lambda x.\lambda y.xyz \not\equiv_\alpha \lambda x.\lambda z.xzz$
$\lambda x.\lambda y.xyz \not\equiv_\alpha \lambda x.\lambda y.yxz$
$\lambda x.\lambda y.xyz \not\equiv_\alpha \lambda x.\lambda y.xyw$

Da questo momento in avanti considereremo (quasi) sempre i termini α-equivalenti come uguali. Più formalmente: l'α-equivalenza è una relazione di equivalenza (simmetrica, riflessiva e transitiva) e noi lavoreremo con le classi di equivalenza di λ-termini modulo l'α-equivalenza.

1.5.9 Sostituzione

Formalmente, $M\{N/x\}$ (M dove sostituisco alle occorrenze libere di x il termine N) è definito come:

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

Terminologia: z è sufficientemente fresca se $z \notin FV(M) \cup FV(N)$ e fresca se non è mai stata utilizzata prima. Ogni variabile fresca è automaticamente sufficientemente fresca.

1.5.10 β-riduzione

t_1 β-riduce a t_2 in un passo (indicato $t_1 \to_\beta t_2$) sse ottengo t_2 da t_1 rimpiazzando da qualche parte in t_1 il redex $(\lambda x.M)N$ con il suo ridotto $M\{N/x\}$.

Esempio: $\lambda x.((\lambda y.xy)x) \to_\beta \lambda x.xx$ dove è stato ridotto il redex $(\lambda y.xy)x$.

Formalmente definiamo la relazione binaria $\to_\beta$ tramite un sistema di inferenza (cfr. deduzione naturale).

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

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

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

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

Osservazione: l'ultima regola dice che posso ridurre il corpo di una funzione prima che questa sia invocata! (come quando si ottimizza il codice). Esempio:

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

Si possono considerare riduzioni in più passi: $t_1 \to_\beta^n t_{n+1}$, cioè t_1 riduce in n passi a t_{n+1} sse $t_1 \to_\beta t_2 \to_\beta ... \to_\beta t_{n+1}$. Formalmente:

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

$t \to_\beta^* t'$ (t riduce in 0 o più passi a t') sse $\exists n.t \to_\beta^n t'$.

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

1.5.11 Forme Normali

t è una forma normale (anche scritto $t \not\to_\beta$) sse $\nexists t'.t \to_\beta t'$.

t ha forma normale t' sse $t \to_\beta^* t' \wedge t' \not\to_\beta$.

t ha una forma normale o può convergere sse esiste un t' tale che t ha una forma normale t'.

1.5.12 Non determinismo

La relazione $\rightarrow_\beta$ è non deterministica, ovvero ci sono dei t tali per cui esistono t_1, t_2 distinti tali che $t_1 \leftarrow_\beta t \rightarrow_\beta t_2$. Esempio: $y_\beta \leftarrow (\lambda x.y)((\lambda z.z)w) \rightarrow_\beta (\lambda x.y)w$.

Intuizione: non viene specificato in quale ordine vanno ridotti i redessi e un'implementazione può scegliere liberamente. Questo potenzialmente può portare a forme normali distinte, ma vedremo che non sarà così. (Continuando l'esempio: $(\lambda x.y)w \rightarrow_\beta y$).

1.6 Quanto è espressivo il λ-calcolo

Un linguaggio Turing completo necessita di:

- Tipi di dato (almeno i numeri naturali con cui codificare tutti gli altri)
- il costrutto if-then-else (scelta)
- ripetizione (ciclo o ricorsione)

Il λ-calcolo sembra non avere nessuno di questi meccanismi! Mostreremo come esprimere in λ-calcolo ricorsione, scelta e tipi di dato assumendo di volta in volta di avere già a disposizione gli altri meccanismi.

1.6.1 Ricorsione in λ-calcolo

La ricorsione è necessaria per la Turing completezza perché, senza, non sarebbe possibile gestire input di lunghezza arbitraria: poniamo di avere un programma che ripete l'algoritmo n volte per gestire un input di lunghezza n, se però abbiamo un input di lunghezza $n+1$ quel programma non sarà più risolvibile; invece, grazie ai cicli, è possibile risolverlo.

Per essere Turing-Completi è sufficiente ripetere lo stesso codice più volte (cicli o ricorsione) oppure eseguire ogni volta una nuova copia del codice (vedremo la funzione $\lambda x.f(xx)$). Le funzioni anonime non possono richiamare sé stesse esplicitamente e non ci sono altri meccanismi per ciclare. Prendiamo qualche idea dal paradosso di Russel.

Assioma di comprensione (inconsistente): Data una proprietà P, esiste $\{x|P(x)\}$ e si ha $\forall y.(y \in \{x|P(x)\} \iff P(y))$.

Paradosso di Russel: $X = \{Y|Y \notin Y\}$, cioè l'insieme di tutti gli insiemi che non appartengono a se stessi.

Quindi se $X \in X \iff \neg(X \in X) \iff \neg(\neg(X \in X) \iff \neg...\neg(X \in X) \iff$ .... Sto rieseguendo molte volte la funzione $\neg$. Abbiamo ricavato un modo per eseguire la ricorsione. Vediamo se è possibile in λ-calcolo. Per ottenere questo risultato c'è bisogno di quattro "ingredienti":

1. un predicato binario $\in$ per il quale gli insiemi siano sia oggetto (il contenuto) che il soggetto (il contenitore)
2. l'auto-applicazione di una negazione intorno all'autoapplicazione
3. l'introduzione di una negazione intorno all'autoapplicazione
4. la trasformazione del predicato $\not(Y \in Y)$ in insieme autoapplicabile (via assioma inconsistente di comprensione)

Mettiamo a confronto con il λ-calcolo.

Russel	**λ-calcolo**
Tutto è un insieme	Tutto è una funzione
$x \in x$	xx
$\neg$	f
$x \notin x$	$f(xx)$
Assioma di comprensione: da $P(y)$ ricava un'insieme $\{y\|P(y)\}$	λ-astrazione: da M (in cui occorre y) ricava $\lambda y.M$
$\{y\|y \notin y\} \in \{y\|y \notin y\} \iff \neg(...)$	$(\lambda y.f(yy))(\lambda y.f(yy)) \to_\beta f((\lambda y.f(yy))(\lambda y.f(yy))) \to_\beta ...$

Table 3: Similitudini tra la teoria di Russel e il λ-calcolo

Per il λ-calcolo quindi, possiamo definire una funzione $\lambda y.f(yy)$, dove f è la funzione che vogliamo ripetere.

Divergenza: Un termine t_0 può divergere sse $\forall i.\exists t_{i+1}.t_i \to_\beta t_{i+1}$. Esempio: prendiamo la funzione identità $(\lambda x.xx)(\lambda x.xx) \to_\beta (\lambda x.xx)(\lambda x.xx) \to_\beta ....$ Un termine può divergere e può convergere allo stesso tempo. Esempio: $y_\beta \leftarrow (\lambda x.y)((\lambda x.xx)(\lambda x.xx)) \to_\beta (\lambda x.y)((\lambda x.xx)(\lambda x.xx)) \to_\beta ....$

Punti fissi

Definizione di punto fisso in matematica: sia A un insieme e $f : A \to A$. Un $x \in A$ è un punto fisso di f sse $x = f(x)$. Ad esempio, la funzione $|\cdot|$ (valore assoluto) ha dei punti fissi in $\mathbb{Z}$, invece la funzione $x \to x + 1$ non ha punti fissi.

Definizione di punto fisso in λ-calcolo: un λ-termine t è punto fisso di f sse $f(t) =_\beta t$, dove $=_\beta$ (β-conversione) è la chiusura riflessiva, simmetrica e transitiva di $\to_\beta$. I punti fissi esistono sempre: $(\lambda x.f(xx))(\lambda x.f(xx))$ è punto fisso di f.

Teorema: in λ-calcolo ogni termine M ha almeno un punto fisso.
Dimostrazione: $(\lambda y.M(yy))(\lambda y.M(yy))$ è un punto fisso di M; infatti, $(\lambda y.M(yy))(\lambda y.M(yy)) \to M((\lambda y.M(yy))(\lambda y.M(yy)))$. Qed.

Operatore di punto fisso

Def: Y è un operatore di punto fisso sse $\forall M.YM$ è un punto fisso di M.

Teorema: $Y = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$ è un operatore di punto fisso.
Dimostrazione: ovvia. $YM = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))M \rightarrow_\beta$ $(\lambda x.M(xx))(\lambda x.M(xx))$ che è un punto fisso di M. Qed.

1.6.2 Codifica delle funzioni ricorsive

Vediamo un esempio con la funzione ricorsiva (scritta in un linguaggio simile ad OCaml) che somma tutti in numeri pari minori di un input n:

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

La **funzione non ricorsiva associata** è la seguente. Da notare il dominio e il codominio.

```
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
```

Per creare la codifica di f in λ-calcolo e renderla ricorsiva si compongo le funzioni nel modo seguente: $f = YF$ dove Y è un operatore di punto fisso. Infatti $f = YF \rightarrow_\beta F(YF) = Ff$. Di seguito, ecco la **codifica in λ-calcolo** della funzione:

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

Di seguito, ecco un altro esempio con la funzione fattoriale:

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

Di seguito, la codifica in λ-calcolo:

```
Y
 (λfact.
   λn.
     match n with
       | 0 => 1
       | S m => S m * fact m)
```

L'esecuzione avverrà nel modo seguente:

```
Sia fact = Y(λfact...)
Si ha
 fact(S 0) =
  Y (λfact...)(S 0) →β
   (λfact...)(Y(λfact...))(S O) →β
    (λn. match n with 0 => 1 | S m => S m * Y(λfact...) m
        ) (S O) →β*
     S 0 * Y (λfact...) 0 →β*
      S 0 * 1 →β*
       S 0
```

1.6.3 Tipi algebrici

I Tipi di Dato Algebrici sono collezioni di valori (anche strutturati), detti "termini", definiti per iniezione a partire da un insieme finito di funzioni iniettive, dette "costruttori".

type $\mathbb{B}$ = true : $\mathbb{B}$ | false : $\mathbb{B}$

`true` e `false` sono **costruttori**, mentre $\mathbb{B}$ è il nome del tipo. Abbiamo definito il tipo dei dati booleani. ES: `true` : $\mathbb{B}$. Facciamo un altro esempio definendo i tipo del seme di una carta da poker:

```
type seme = cuori : seme
          | quadri : seme
          | picche : seme
          | fiori : seme
```

Si possono definire dei tipi "ricorsivi" (definendo una funzione):

type $\mathbb{N}$ = 0 : $\mathbb{N}$ | S : $\mathbb{N} \to \mathbb{N}$

Abbiamo definito il tipo dei dati dei numeri naturali (dove S sta per *successivo*). ES: `S 0` : $\mathbb{N}$ corrisponde al numero 1.

Si possono definire funzioni che prendono in input più dati e possono prendere in input loro stessi dei tipi:

type List T = [] : List T | (::) : T → List T → List T

`List T` è il tipo del dato *lista*, dove T è un parametro (variabile di tipo), quindi la lista è tipo generico. `(::)` si legge "cons". ES: `(S 0) :: 0 :: [] : List` $\mathbb{N}$ è una lista di interi naturali con tre elementi (1, 0 e `[]`).

Scriviamo ora due tipi di alberi:

```
type Tree1 T = x : Tree1 T | [] : Tree1 T → T → Tree1 T
    → Tree1 T
type Tree2 T = [] : T → Tree2 T | () : Tree2 T → Tree2 T
     → Tree2 T
```

Entrambi sono alberi binari (dato dal fatto che nella seconda parte compaiono due volte i tipi `TreeN T`). Il primo albero può contenere un elemento per ogni nodo e avrà come foglie sempre il simbolo `x`. Invece, il secondo conterrà solo nelle foglie gli elementi.

I tipi algebrici sono quindi un metodo molto efficace e potente per rappresentare i dati in un linguaggio. Sono implementati in tutti i principali linguaggi di programmazione.

1.6.4 Scelta

Nel λ-calcolo non esiste un operatore di scelta. Tuttavia è possibile implementare il costrutto *if-then-else* attraverso il pattern matching del tipo booleano. Infatti, il codice seguente equivale a `if b then` M_1 `else` M_2.

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

Quindi, trovando un implementazione per il pattern matching, avremo anche un meccanismo di scelta.

Come funziona il pattern matching

```
match b with
    ...
    | Ki  x1...xn => Mi
    ...
```

Il pattern matching è formato dalle seguenti parti

- K_i: nome dell'i-esimo costruttore;
- $x_1...x_n$: variabili, una per ogni argomento del costruttore (vengono legate in M_i);
- M_i: codice da eseguire (può usare le variabili definite prima).

L'esempio del booleano segue questa definizione. Vediamo un esempio fatto sul tipo `List` $\mathbb{N}$:

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

Questa funzione calcola la somma di tutti gli elementi di una lista. Esegue prima il pattern matching sul caso base e poi sul caso composto, dove definisce due nuove variabili: `x` è la testa della lista e sarà di tipo $\mathbb{N}$, `f` è la coda e sarà di tipo `List` $\mathbb{N}$. In particolare, quello che avviene è una sostituzione nel corpo da eseguire (vedi esempio seguente).

```
x :: f = 5 :: (2 :: []) →
    (x + sum f){5/x}{(2 :: f)/f} =
        5 + sum (2 :: [])
```

Come comporre il pattern matching

Il pattern matching è un insieme di funzioni che vengono definite per ogni tipo e per ogni costruttore di tipo. Prendiamo l'esempio di prima delle liste, convertendolo parzialmente in λ-calcolo:

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

Listing 1: Funzione convertita in λ-calcolo

`List` è composto da due possibili costruttori, quello di base e quello complesso. Questi costruttori non sono altro che delle funzioni. Andiamo a definirle il tipo delle funzioni, in seguito verrà scritto il corpo:

```
empty : List T
cons : T → List T → List T
```

`empty` restituisce il costruttore di base e `cons` restituisce il costruttore complesso.

Andiamo a definire la funzione che esegue il pattern matching sul tipo `List`.

$\textbf{match}_{List\,T}$: List T → ∀X. X → (T → List T → X) → X

$\texttt{match}_{List\,T}$ è quindi una funzione che prende in input il dato su cui eseguire il pattern matching e un corpo di codice da eseguire per ogni costruttore presente nel tipo. Nel caso di `List` sono due: `empty` e `cons`.

Quindi, si vuole ottenere i seguenti risultati quando viene fatto il pattern matching delle liste:

$\mathbf{match}_{List\,T}$ empty M_e M_c $\rightarrow^*_\beta$ M_e
$\mathbf{match}_{List\,T}$ (cons M_x M_f) M_e M_c $\rightarrow^*_\beta$ M_c M_x M_f

dove M_e ed M_c sono i corpi delle funzioni da eseguire. M_x e M_f sono le nuove variabili definite. Nel primo esempio vogliamo eseguire la prima funzione, essendo il tipo `empty`. Nel secondo esempio vogliamo eseguire la seconda funzione, ricevendo in input un dato di tipo `cons`.

Definiamo quindi il corpo delle funzioni rispetto ai costruttori (e rispettando le firme delle funzioni):

empty = λe. λc. e
cons = λx. λf. λe. λc. c x f

Di conseguenza tutta la logica della scelta è nei dati algebrici e l'ultima funzione da definire (il `match`) è semplicemente la funzione identità:

$\mathbf{match}_{List\,T}$ = λx. x

Eseguiamo la prova del primo esempio:

$\mathbf{match}_{List\,T}$ empty M_e M_c =
(λx. x) (λe. λc. e) M_e M_c $\rightarrow_\beta$
(λe. λc. e) M_e M_c $\rightarrow_\beta$ M_e

Eseguiamo la prova del secondo esempio:

$\mathbf{match}_{List\,T}$ (cons M_x M_f) M_e M_c=
(λx. x) (cons M_x M_f) M_e M_c $\rightarrow_\beta$
cons M_x M_f M_e M_c $\rightarrow_\beta$
(λx. λf. λe. λc. c x f) M_x M_f M_e M_c $\rightarrow_\beta$ M_c M_x M_f

Di seguito verranno mostrati esempi con i tipi booleani e interi naturali.

type $\mathbb{B}$ = true : $\mathbb{B}$ | false : $\mathbb{B}$
$\mathbf{match}_{\mathbb{B}}$ = λx. x
true = λt. λe. t
false = λt. λe. e

Test:

$\mathbf{match}_{\mathbb{B}}$ true M_T M_E =
(λx. x) true M_T M_E =
true M_T M_E =
(λt. λe. t) M_T M_E = M_T

type $\mathbb{N} = 0 : \mathbb{N} \mid S : \mathbb{N} \to \mathbb{N}$
$\textbf{match}_{\mathbb{N}} = \lambda x.\ x$
$0 = \lambda z.\ \lambda s.\ z$
$S = \lambda n.\ \lambda z.\ \lambda s.\ s\ n$

Test:

$\textbf{match}_{\mathbb{N}}\ (S\ N)\ M_Z\ M_S =$
$(\lambda x.\ x)\ (S\ N)\ M_Z\ M_S =$
$S\ N\ M_Z\ M_S =$
$(\lambda n.\ \lambda z.\ \lambda s.\ s\ n)\ N\ M_Z\ M_S = M_S\ N$

Viene quindi riscritto il codice 1 in λ-calcolo con la notazione nuova del pattern matching:

```
let rec sum =
  λl.
    match_{List ℕ}
      l
      0
      (λx. λf. x + sum f)
```

1.7 Conversioni di codice in λ-calcolo

1.7.1 Funzione con variabili globali

Di seguito, viene mostrato a destra un pezzo di codice che esegue operazioni con variabili globali. A sinistra, viene mostrato il corrispettivo in λ-calcolo. In λ-calcolo non possono venire cambiate le variabili e quindi ne viene creata una copia. Verrà infine ritornata la copia e usata al posto della variabile globale.

```
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;
  }
}
```

```
f(x, y, a) {
  var z = 2;
  var x' = z * y;
  var z' = y + a;
  var a' = 3;
  var <z'', res> = g(x');
  var x'' = x' + res;
  return <a', x'' + z''>
  g(z) {
    var z' = z + 1;
    return <z', 3>;
  }
}
```

1.7.2 Ciclo while

In λ-calcolo, i cicli `while` e `for` non esistono, ma possono essere implementati in modo equivalente tramite una funzione ricorsiva.

```
var x = 10;
var res = 0;
while x > 0 do
  res = res + x;
  x = x - 1;
done
```

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

1.7.3 Struct

I tipi di dato strutturati si implementano con i tipi algebrici, come visto in precedenza. Le funzioni per estrarre un particolare dato da una struct sono zucchero sintattico e si implementano in λ-calcolo attraverso delle semplici funzioni di pattern matching.

```
struct person {
  string name;
  int age;
  string city;
}

persona p =
    {"Gabriele",
     22,
     "Bologna"}

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

```
type person =
  mk : string
       -> int
       -> string
       -> person;

person mypersona =
  mk "Gabriele" 22 "Bologna"

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

if age mypersona > 20
then name mypersona
```

1.7.4 Oggetti

Gli oggetti in λ-calcolo non differiscono dai tipi strutturati (struct).

```
object person {
  name = "Gabriele";
  age = 22;
  grow(n) {
    if self.age + n > 100
    then self.die()
    else self.age += n
  }
  die() {
    self.name = "rip";
  }
}
```

```
struct person {
  name = "Gabriele";
  age = 22;
  grow =>
    λself. λn.
      if self.age + n > 100
      then self.die self
      else self.age += n
  die =>
    λself.
      self.name = "rip"
}
```

1.7.5 Eccezioni

TODO

1.8 Proprietà del λ-calcolo

Aggiungendo i tipi al λ-calcolo si ottiene una proprietà interessante. Viene quindi dato contesto su cosa sia proprietà, decidibilità e modularità.

1.8.1 Proprietà

Prendiamo l'insieme di tutti i possibili programmi P. Una proprietà è un sottoinsieme di P.

x ha la proprietà Q sse $x \in Q$

Una proprietà Q è banale sse $Q = \emptyset \vee Q = P$ (cioè se non esistono programmi con quella proprietà o se è una proprietà comune a tutti i possibili programmi.

Esempi: $Q = \{p \in P | p \text{ usa 2 variabili}\}$; $Q = \{p \in P | p(0) = 1\}$.

Una proprietà Q è estensionale sse $\forall p, q \in P.\, (\forall i. p(i)$ restituisce $g \iff q(i)$ restituisce $g)$ (cioè i due programmi calcolano la stessa funzione). Una proprietà è intensionale se non è estensionale.

La proprietà che ci interessa quindi è una proprietà estensionale, ma non sarà facile trovarla in quanto non sarà banale e quindi indecidibile.

1.8.2 Decidibile

Una proprietà Q è decidibile sse $\exists p \in P. \forall q \in P.\, (q \in Q \iff p(q) = \text{true} \vee q \notin Q \iff p(q) = \text{false})$, cioè la proprietà è decidibile se esiste un programma che ritorna *vero* o *falso* per ogni programma con quella proprietà. Esempio: $Q = \{p \in P \mid |p| < 100 \text{ caratteri}\}$. Q è decidibile.

1.8.3 Teorema di Rice

$\forall Q$. se Q non banale ed estensionale allora Q **NON** è decidibile.

Vogliamo trovare due proprietà che facciano da approssimazione per la nostra proprietà del λ-calcolo tipato. Siano S, Q, R delle proprietà. 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 e S siano decidibili e decide da un programma r e s.

Teorema: $\forall p.\, (r(p) = \text{true} \Rightarrow p \in Q) \wedge (s(p) = \text{false}\ p \notin Q)$.

Il nostro scopo sarà quindi trovare un'approssimazione di due proprietà decidibili (una approssimata da dentro e una da fuori) per trovare la proprietà Q. Un sistema di tipi è l'implementazione di un programma che decide un'approssimazione da dentro o da fuori in maniera modulare. Per esempio, se il compilatore stabilisce che il programma è ben tipato, sta facendo un'approssimazione da dentro. I sistemi di tipo approssimati da fuori di solito sono creati per linguaggi non tipati che vengono controllati da un programma esterno.

Modulare vuol dire che posso spezzare un mio programma p in più parti $p_1, ..., p_n$ e analizzo le parti in maniera indipendente. Associo dei tipo a $p_1, ..., p_n$ con nome $T_1, ..., T_n$ in modo che $r(p)) = \text{true} \iff p \in R \iff d(T_1, ..., , T_n)$. Quindi posso guardare il programma in modo modulare. In un contesto reale, è come quando il programma usa diverse librerie.

Esiste anche la modularità gerarchica dove l'analisi si basa su un albero di gerarchie. Prima calcolo il tipo delle foglie e se tutti i figli di un nodo sono tipati allora anche il nodo è tipato. Il tipaggio quindi viene propagato.

1.9 Il λ-calcolo Tipato Semplice

Sintassi: $T ::= A \mid T \to T$, dove A sono variabili di tipo ($A|B|C|...$ come $|\mathbb{B}|\mathbb{N}|\text{string}|...$) e $T \to T$ è il tipo delle funzioni con un certo input (dominio) e output (codominio) (è associativo a destra, es: $A \to B \to C = A \to (B \to C)$).

1.9.1 Contesto

Un programma può aggiungere libreria che possono esporre funzioni (API) già tipate, questo viene rappresentato dal contesto. Il contesto è definito come $\Gamma ::= \emptyset \mid \Gamma, x : T$, dove $\emptyset$ è l'assenza di contesto e $\Gamma, x : T$ è l'aggiunta al contesto della **variabile** x con tipo T (non termini perché non posso applicarli tra di loro quindi sono semplicemente variabili). In Γ nessuna variabile è ripetuta. Verrà usato $(x : T) \in \Gamma$ per dire che $\Gamma = ..., x : T, ...$ per comodità.

1.9.2 Judgement di Tipaggio - Sistemi di inferenza

Per valutare il tipo di un programma, introduciamo $\Gamma \vdash t : T$ che è una relazione ternaria e si legge "in Γ, t ha tipo T". Lo definisco attraverso un sistema di inferenza (cerco di derivare il tipo da quello che so).

Introduciamo le regole per creare un albero deduttivo. La parte sotto sono le "conclusioni" (quello che vogliamo provare) e sopra scriviamo le ipotesi. Se riusciamo a concludere una albero di dimostrazione, vuol dire che il programma è ben tipato.

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

La seguente regola stabilisce come tipare l'applicazione di una chiamata di funzione MN. In particolare, si sfrutta la modularità, tipando M e N separatamente.

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

La seguente regola stabilisce come tipare il l'astrazione di una funzione. Si passa **aggiungendo** (quindi *assumendo*) la nuova variabile x in Γ.

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

Esercizio di esempio: dimostriamo che l'espressione $\lambda x.f\,x$ ha tipo $(A \to A) \to B$ avendo nel contesto la funzione $f : (A \to A) \to B$.

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

Guardando dal basso verso l'alto abbiamo applicato prima la regola per l'astrazione, poi la regola per l'applicazione e poi nei due rami generati abbiamo applicato la regola del termine. Quindi il sistema è ben tipato.

Adesso dimostriamo che $\lambda x.xx$ **NON** è ben tipato. Quindi, $\lambda x.xx$ non ha la proprietà che stiamo cercando.

$$\dfrac{\dfrac{(x : T_3? \to T_4?) \in \Gamma}{x : T_1? \vdash x : T_3? \to T_4??} \quad \dfrac{(x : T_3?) \in x : T_3? \to T_4?}{x : T_3? \to T_4? \vdash x : T_3?}}{\dfrac{x : T_1? \vdash xx : T_2?}{\vdash \lambda x.xx : ?}}$$

Applicando le regole, si conclude che $T_3 = T_3 \to T_4$ che è impossibile, quindi $\lambda x.xx$ **NON** è ben tipato.

Quindi in pratica, provare se un programma è corretto si effettua tramite il controllo del tipo.

1.9.3 Ripasso Logica Proposizionale Minimale

Sintassi formule: $F ::= A|F \to F$, dove A è una qualsiasi variabile proposizionale e $F \to F$ è un implicazione se ... allora ... ($\to$ è associativo a destra). Il contesto di ipotesi è $\Gamma ::= |\,\Gamma, F$ (suppongo che F valga).

Il judgement di derivazione di questa logica è $\Gamma \vdash F$ che vuol dire "a partire dalle ipotesi Γ dimostro F".

Definiamo le regole di introduzione ed eliminazione per la sintassi.

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

Introduciamo prima la regola di eliminazione dell'implica (detta *modus pones*) e poi la regola di introduzione dell'implica.

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

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

Nota bene: se queste regole vengono messe in contrapposizione alle regole del λ-calcolo tipato semplice, si può notare che sono praticamente identiche. L'unica differenza è che nel λ-calcolo tipato semplice i termini hanno un tipo.

1.10 Isomorfismo di Curry-Howard-Kolmogorov

É uno dei risultati più importanti per la logica e l'informatica perché trova un isomorfismo tra il λ-calcolo e la logica (e la matematica con le categorie).

Estendiamo il concetto di isomorfismo trovando altre regole di eliminazione e introduzione e i loro corrispettivi. In questo caso vogliamo poter esprimere la combinazione di due tipi (tuple). Il corrispettivo in logica è l'and.

$T :: ...|T \times T$
$F ::= ...|F \wedge F$
$t :: ... \mid \langle t, t\rangle \mid t.1 \mid t.2 \mid$ match t with $\langle x_1, x_2\rangle \Rightarrow t \mid$ `let` $\langle x_1, x_2\rangle = C$ `in` t

Regole di introduzione:

λ-calcolo	**Logica**
Tipo	Formula
Termini	Prove
Costruttore di tipo	Connettivo
Costruttore di termini	Passi di prova
Variabili libere e legate	Ipotesi globali e locali (scaricate)
Type checking	Proof checking
Type Inabitation	Ricerca di prove
Riduzione	Normalizzazione

Table 4: Corrispondenze tra λ-calcolo e logica

$$\frac{\Gamma \vdash M_1 : T_1 \qquad \Gamma \vdash M_2 : T_2}{\Gamma \vdash \langle M_1, M_2 \rangle : T_1 \times T_2} \qquad \frac{\Gamma \vdash F_1 \qquad \Gamma \vdash F_2}{\Gamma \vdash F_1 \wedge F_2}$$

Regole di eliminazione:

$$\frac{\Gamma \vdash C : T_1 \times T_2}{\Gamma \vdash C.1 : T_1} \qquad \frac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_1}$$

$$\frac{\Gamma \vdash C : T_1 \times T_2}{\Gamma \vdash C.2 : T_2} \qquad \frac{\Gamma \vdash F_1 \wedge F_2}{\Gamma \vdash F_2}$$

Regola di eliminazione (C è inizializzato con `let...`)

$$\frac{\Gamma \vdash C : T_1 \times T_2 \qquad \Gamma, x_1 : T_1, x_2 : T_2 \vdash M : T}{\Gamma \vdash \text{match } C \text{ with } \langle x_1, x_2 \rangle \Rightarrow M : T} \qquad \frac{\Gamma \vdash F_1 \wedge F_2 \qquad \Gamma, F_1, F_2 \vdash F}{\Gamma \vdash F}$$

Introduciamo in λ-calcolo il tipo *unit* che corrisponde al Top (T) della logica. Non è da confondere con il tipo `void`.

$T ::= ...|\mathbb{1}$
$F ::= ...|T$
$t ::= ...|()$
`let () = t` in t

Regola di introduzione:

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

Regola di eliminazione (dove M è inizializzato con `let...`:

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

Estendiamo ulteriormente con il tipo *empty*, che corrisponde al *bottom* della logica. In codice questo concetto si traduce in un ramo dell'esecuzione non possibile e quindi `abort(t)`.

$T ::= ...|0$
$F ::= ...|\bot$
$t ::= ...|$`abort(t)`

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

L'or della logica invece corrisponde al *sum type* del λ-calcolo (le *union* nei linguaggi di programmazione).

1.10.1 Consistenza della logica proposizionale

Teorema: $\not\vdash \bot$.

Dimostrazione: Assumo $\vdash \bot$. Per Curry-Haward $\exists M. \vdash M : \bot$. Sia M tale che $\vdash M : \bot$. Sia N la forma normale di M che esiste per il teorema di normalizzazione forte. Si ha $\vdash N : \bot$. Che forma ha N?

- N **NON** è una regola di introduzione (il $\bot$ non ne ha)
- N non è una variabile
- che N sia una regola di eliminazione? Si esegue con un `match`, ma questo approccio genera un loop

Quindi è un assurdo perché manca il caso base. Quindi $\not\vdash \bot$. Qed.

Definizione: Un termine t si dice (debolmente) normalizzante quando ha una forma normale.
Definizione: Un termine t si dice (fortemente) normalizzante se $\nexists (t_i)_{i\in\mathbb{N}}. t = t_0 \wedge \forall i. t_i \to_\beta t_{i+1}$.
Definizione: Un λ-calcolo si dice avere una proprietà Q sse ogni termine ce l'ha.

1.10.2 Teorema di Normalizzazione Forte

$\forall \Gamma, M, T. \Gamma \vdash M : T \Rightarrow M$ è fortemente normalizzante, dove $\Gamma \vdash M : T$ è un'approssimazione da dentro e M è fortemente normalizzante è una proprietà Q indecidibile.

Osservazione: $\lambda x.xx$ non è tipabile, ma $\lambda x.xx$ è fortemente normalizzante.

Dimostrazione: TODO.

1.10.3 Proprietà del λ-calcolo tipato semplice e dimostrazioni

Possiamo concludere i seguenti fatti:

- Il tipaggio è modulare
- La normalizzazione forte non è modulare
- Deve quindi esserci una proprietà intermedia modulare che approssimi meglio la normalizzazione forte.

Quindi, il λ-calcolo tipato semplice gode della proprietà di normalizzazione forte, cioè ogni termine è fortemente normalizzante. Vuol dire che qualsiasi strategia di riduzione adottata, porta sempre a una forma normale in un numero finito di passi.

Possiamo riassumere quindi che dati i seguenti insiemi: P (l'insieme di tutti i programmi), $SN = \{t | t \text{ è fortemente normalizzante}\}$ (l'insieme dei programmi fort. norm.) e $WT = \{T | \exists \Gamma, T.\, \Gamma \vdash t : T\}$ (l'insieme dei programmi ben tipati). Questi insiemi sono in questa relazione $WR \subseteq SN \subseteq P$. $\lambda x.\, xx$ è un espressione che sta tra SN e WT, noi vogliamo catturare una proprietà più fine che sta tra $\lambda x.\, xx$ e WT, che chiameremo RED_T (l'insieme dei termini RIDUCIBILI di tipo T).

Piano dei lavori:

1. Trovare una buona definizione di RED_T
2. Dimostrare che $WT \subseteq RED_T$
3. Dimostrare che $RED_T \subseteq SN$

Definiamo RED_T per ricorsione sul tipo T:

$$RED_A = \{M \mid \exists \Gamma.\, \Gamma \vdash M : A \wedge M \in SN\}$$

$$RED_{T_1 \to T_2} = \{M \mid \exists \Gamma.\, \Gamma \vdash M : T_1 \to T_2 \wedge \forall N \in RED_{T_1}.\, MN \in RED_{T_2}\}$$

Definendo in questo modo RED_T vedremo che il primo tentativo di prova (semplice) di $WT \subseteq RED_T$ (ovvero che $\forall \Gamma, M, T.\, \Gamma \vdash M : T \Rightarrow M \in RED_T$) andrà male. Avremo bisogno di ulteriori lemmi per dimostrarlo. La dimostrazione avviene per induzione sui vari casi, ma non viene sul caso $\Gamma \vdash (\lambda x.\, M)N : T_1 \to T_2$ perché non riesco a dimostrare che $\lambda x.M \in RED_{T_1 \to T_2}$ (cioè che l'espressione appartiene ai candidati di riducibilità di RED, in particolare non possiamo dimostrare $\forall N \in RED_{T_1}.\, (\lambda x.M)N \in RED_{T_2}$). Per dimostrarlo servono i lemmi $CR1 - CR3$ e dovremmo cambiare l'ipotesi del teorema.

Nuovo piano di lavoro:

1. Definiamo RED_T
2. Identifichiamo $CR1 - CR3$ le proprietà dei *candidati di riducibilità*
3. Dimostriamo $CR1 - CR3$ per RED_T
4. Generalizziamo l'enunciato $WT \subseteq RED_T$ e lo dimostriamo usando $CR1 - CR3$ (la difficoltà si sposterà sul caso delle variabili, prima era sulla λ-astrazione). Teorema da dimostrare generalizzato: $\forall \Gamma, M, T.$ dato $\{N_i \mid (x_i : T_i) \in \Gamma,\ N_i \in RED_{T_i}\}$
5. Dimostriamo $RED_T \subseteq SN$ usando $CR1$

Enunciamo i 3 lemmi da dimostrare (CR sta per Candidati di Riducibilità, cioè insiemi che sono potenzialmente riducibili; sono utili perché lì in mezzo ci sono quelli che ci interessano cioè i riducibili):

- $CR1(T) : RED_T \subseteq SN$
- $CR2(T) : \forall M, N.\ M \in RED_T \wedge M \to_\beta^* N \Rightarrow N \in RED_T$
- $CR3(T)$: $\forall M (M$ neutrale $\wedge (\forall N.\ M \to_\beta N \Rightarrow N \in RED_T) \Rightarrow M \in RED_T)$
- Corollario di $CR3$: $CR4 = \forall T. x \in RED_T$

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

Teorema: M è neutrale sse M non è una λ-astrazione. Es: $\lambda x.M$ NON è neutrale perché $(zy)[\lambda x.M/z] = (\lambda x.M)y$ e $(\lambda x.M)y \notin \lambda x.M$ e $(\lambda x.M)y \notin (zy)$. (MN) è neutrale perché se $R[MN/x)]$ contiene un redesso della forma $\lambda z.UW$ non può essere stato creato.
μ = operazione TODO.

1.11 Ricorsione nei linguaggi di programmazione

Nei linguaggi di programmazione c'è sempre un costrutto esplicito (inteso così perché il λ-calcolo ce lo siamo costruito, ma nei linguaggi la cosa avviene in automatico) di ricorsione.

Il problema non è banale dal punto di vista della logica: in una funzione si sta chiamando la stessa funzione che è ancora in costruzione e quindi di cui non si sa il tipo.

Prendiamo la funzione $f : T := M$ (funzione dichiarata al toplevel; avrà tipo T e corpo M). Essa è zucchero sintattico per il costrutto: $f : (\nu f : T.M)$ dove la f è il nome esterno che si usa all'interno del programma; $(\nu f : T.M)$ è una

funzione ricorsiva anonima dove ν è un binder di punto fisso e la f è la funzione che appare dentro al corpo. La seguente regola viene introdotta:

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

Grazie al ν, che non esiste nel λ-calcolo, causa la possibilità di divergere! Possiamo fare vedere che il seguente programma è ben tipato grazie alla nuova regola:

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

In questa maniera possiamo controllare la divergenza andando a controllare il tipaggio delle funzioni ricorsive: se voglio essere Turing-Completo, tipo le funzioni ricorsive ma perdo l'isomorfismo perché dimostri l'assurdo (e quindi non sono fort. norm.). Infatti, tale regola implica la **NON** consistenza del sistema logico. Prendendo l'esempio di prima se tipiamo la funzione con da Top a Bottom e passando in input I, possiamo dimostrare Bottom, quindi è inconsistente.

Ho quindi un trade-off tra Turing-completezza e coerenza del sistema logico. Corollario della osservazione precedente: $y = \lambda f.(\lambda x f(xx))(\lambda x.f(xx))$ non è tipabile per nessun sistema di tipi.

1.12 λ-calcolo e logica proposizionale del secondo ordine

1.12.1 Quantificatore $\forall$ al secondo ordine

Sintassi $F ::= ...|\forall A.F$ (cioè stiamo quantificando su variabili proposizionali, nella logica del prim'ordine si quantifica sui termini dove il termine è un elemento del dominio del discorso, quindi un insieme, un numero, una persona). Quindi è molto più potente perché quantifichiamo anche sulle proposizioni.

Mostriamo la regola di introduzione ed eliminazione del $\forall$:

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

Nella regola di introduzione, vado a sostituire in F una variabile fresca non usata in Γ al posto di A. Nella regola di eliminazione, posso scegliere di sostituire una qualunque G al posto di A. Andiamo a vedere un esempio di dimostrazione (viene applicata prima la regola di introduzione e poi quella di eliminazione):

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

Nel contesto del λ-calcolo corrisponde al *polimorfismo uniforme* (o generico o template) perché è come tipare un programma su tutti i tipi.

$T ::= ...| \forall A.T$, una sintassi usata spesso nei vari linguaggi di programmazione è $\langle A \rangle T$.

Vediamo le regole di introduzione ed eliminazione.

$$\dfrac{\Gamma \vdash M : T[B/A]}{\Gamma \vdash M : \forall A.T} \qquad \dfrac{\Gamma \vdash M : \forall A.T}{\Gamma \vdash M : T[G/A]}$$

Un grande cambiamento è che $\lambda x.xx$ è tipato nel λ-calcolo con *polimorfismo uniforme*. Prova: TODO.

Possiamo concludere quindi:

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

Ora abbiamo la gerarchia come $ST \subseteq PT \subseteq SN \subseteq \lambda T \subseteq P$ (ST è il tipato semplice, PT è tipato coi polimorfi, SN è fort. norm., λT è il λ-calcolo senza tipo) dove $\lambda x.xx$ sta tra ST e PT.

1.12.2 Quantificatore $\exists$ al secondo ordine

Sintassi: $F ::= ...| \exists A F$, dove A è una variabile proposizionale. Mostriamo le regole di introduzione ed eliminazione (dove B rappresenta una variabile fresca, grazie a questa regola abbiamo modularizzato la nostra dimostrazione).

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

Vediamo un esempio, dove prima uso la regola di eliminazione e poi introduzione dell'esiste, poi eliminazione dell'implica e poi eliminazione del and:

$$\dfrac{\dfrac{\dfrac{}{\exists A.(A \to B) \vdash \exists A.(A \to B)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{..., E \to B, E \wedge D \vdash E \to B} \qquad \dfrac{}{..., E \to B, E \wedge D \vdash E \wedge D}}{..., E \to B, E \wedge D \vdash E}}{..., E \to B, E \wedge D \vdash B}}{..., E \to B \vdash E \wedge D \to B}}{\exists A.(A \to B), E \to B \vdash \exists C.(C \wedge D \to B)}}{\exists A.(A \to B) \vdash \exists C.(C \wedge D \to B)}$$

Introduciamo il corrispettivo dell'isomorfismo di Curry-Haward: $T ::= ... | \exists A.T$ che possiamo vedere come un interfaccia/tipo dato astratto/classe/mixin/modulo etc..., dove $t ::= ... |$ `open t as x in t`.

Il tipo di dato astratto è un tipo di dato del quale non viene data l'implementazione, ma solo l'interfaccia come insieme di segnature di funzioni. Scriviamo le regole di introduzione ed eliminazione (la regola di eliminazione può essere comparata al lavoro del linker).

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

Nella regola di eliminazione, la parte destra corrisponde all'implementazione del modulo e la parte sinistra corrisponde all'uso del modulo.

Dalla regola di eliminazione, posso derivare la regola di riduzione che dice che per dimostrare $N : G$ basta sostituire in tutti i punti in cui è usata f il modulo M (preso della regola di introduzione).

$$\frac{}{\Gamma \vdash N[M/f] : G}$$

Questo corrisponde al codice dopo il linking, cioè al codice senza usare i moduli.

1.13 Abstract Rewriting System

Un sistema di riscrittura astratto è una coppia $(A, \to)$ tale che:

1. $A \neq \emptyset$ (insieme degli stati)
2. $\to \subseteq A \times A$ (relazione di transizione)

Esempi:

- λ-calcolo come ARS: $(\pi, \to_\beta)$, dove π è l'insieme dei λ-termini
- Macchina di Turing (A, Q, q_0, q_f, δ) come ARS: $(A^{\mathbb{Z}} \times \mathbb{Z} \times Q, \to)$
- Linguaggio di programmazione: ...

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

Un ARS deterministico quindi parte da uno stato e arriva sempre in un unico stato. Quando da uno stato posso raggiungere altri due stati differenti si ha del non-determinismo. Se da quei due stari raggiungo uno stato comune, si parla di confluenza.

Possono esserci diversi tipi di confluenza:

- Confluenza locale (succede quando vado a confluire in un unico stato a partire da due stati differenti)
- Semiconfluenza (succede quando vado a confluire in un unico stato a partire da uno stato a cui sono arrivato ad un passo e da uno stato a cui sono arrivato in n passi)
- Confluenza (succede quando vado a confluire in un unico stato a partire da due stati dai quali sono arrivato in n passi)

Posso sempre passare da confluenza a semiconfluenza e viceversa. Posso passare da semiconfluenza a confluenza locale ma NON viceversa.

Teorema: fortemente normalizzante $\wedge$ confluenza locale $\Rightarrow$ semiconfluenza

Teorema: confluenza $\Rightarrow$ safety

Teorema: confluenza $\Rightarrow$ unicità delle forme normali

Teorema: il λ-calcolo è semiconfluente

Fonti del non-determinismo:

1. Un redesso ha due ridotti diversi (non nel λ-calcolo); es: `flip_a_coin()` può dare 0 o 1
2. due redessi possono essere overlapping (cioè hanno una intersezione non vuota e non sono uguali), ma non uno strettamente incluso nell'altro (non nel λ-calcolo)
3. redessi non overlapping o paralleli (c'è nel λ-calcolo), es: $x((\lambda z.z)y)((\lambda w.w)z)$ che riduce in due passi a xyz in due modi diversi, si potrebbe perdere confluenza
4. un redesso è interamente contenuto nell'altro (c'è nel λ-calcolo) e si perde confluenza. Altri sottotipi sono:
   (a) $(\lambda x.M)(...(\lambda...)...)$ dove $x \notin FV(M)$, posso eseguire il redesso interno o subito quello esterno (che può però risultare anche in un loop infinito $\rightarrow$ call by value non conveniente)
   (b) $(\lambda x.M)(...(\lambda...)...)$ dove $x \in FV(M)$ e x occorre n volte in M, si può confluire ma in n passi (call by value vantaggiosa)
   (c) $(\lambda x.(..(\lambda...)..)M \rightarrow$ lemma

Lemma: se $M \rightarrow_\beta N$ allora $M[R/x] \rightarrow_\beta N[R/x]$

2 Secondo modulo: Logica in contesti diversi

2.1 Complessità descrittiva

Tradizionalmente, i problemi di cui si occupa la complessità computazionale sono il calcolo di funzioni nella forma $f : \mathbb{B} \to \mathbb{B}$ dove $\mathbb{B} = \{0,1\}^*$ è l'insieme di tutte le *stringhe binarie*. Ci si concentra sui problemi **decisionali**, cioè quelli in cui $f(\mathbb{B}) \subseteq \{0,1\}$. Tali problemi sono in corrispondenza biunivoca con i sottoinsiemi di $\mathbb{B}$.

2.1.1 Calcolo e risorse

A calcola f viene scritto $[A] = F$. Dal punto di vista ingegneristico, si è sempre guardati a due risorse principali:

- Tempo: numero di passi/transizioni per concludere il calcolo di una funzione; il tempo che l'algoritmo A impiega sull'input x è indicato con $\text{TIME}_A(x)$
- Spazio: massima lunghezza dei risultati intermedi necessari per calcolare una funzione; non si tiene conto né dello spazio necessario a tenere traccia di x, né di quello necessario a tener traccia di $d(x)$; lo spazio che l'algoritmo A impiega sull'input x è indicato con $\text{SPACE}_A(x)$

La lunghezza di una stringa x è indicata con $|x|$.

2.1.2 Classi di complessità

Definiamo delle **classi concrete**, parametriche su una funzione $g : \mathbb{N} \to \mathbb{N}$.

$DTIME(G) = \{L \subseteq \mathbb{B} \mid \exists A.[A] = L \wedge \forall x.TIME_A(x) \leq g(|x|)\}$
$DSPACE(G) = \{L \subseteq \mathbb{B} \mid \exists A.[A] = L \wedge \forall x.SPACE_A(x) \leq g(|x|)\}$

Definiamo le seguenti classi di complessità:

- $P = \bigcup_{g \in POLY} DTIME(g)$
- $PSPACE = \bigcup_{g \in POLY} DSPACE(g)$
- $L = \bigcup_{g \in LOGA} DSPACE(g)$

dove POLY è la classe dei polinomi e LOGA è la classe delle funzioni logaritmiche.

2.1.3 Non-determinismo

Se il problema computazionale è decisionale, ha senso pensare ad algoritmi che non siano deterministici. In tal caso, $f(x) = 1$ sse esiste i dove $F_i(x)$ è finale. A decide f e scriviamo $[A] = f$.

Definiamo $NDTIME(g)$ nel modo seguente:
$NDTIME(G) = \{L \subseteq \mathbb{B} \mid \exists A \text{ non-det } .[A] = L \wedge \forall x.TIME_A(x) \leq g(|x|)\}$
$NP = \bigcup_{g \in POLY} NDTIME(g)$

2.1.4 Tesi di Church-Turing forte

Qualsiasi modello di calcolo fisicamente realizzabile può essere simulato efficientemente (con slow-down al più polinomiale) da una macchina di Turing.

Quindi il modello di calcolo **non conta**.

Sappiamo inoltre che:

1. $L \subseteq P \subseteq NP \subseteq PSPACE$
2. $L \subsetneq PSPACE$
3. una delle relazioni in mezzo al punto 1 è stretta, ma non si sa quale

Soprattutto non si sa se $P = NP$ o $P \neq NP$. Si è quindi cercato un modo per descrivere la complessità in maniera diversa, attraverso la complessità descrittiva. Dopo un po', è nata l'esigenza di caratterizzare le classi di complessità in termini di **linguaggi logici**.

2.1.5 Logica e grafi

Supponiamo di utilizzare un vocabolario costituito dal simbolo binario di uguaglianza e da un simbolo binario E. Consideriamo universi finiti nella forma $A_n = \{1, ..., n\}$. Un'interpretazione I per tale universo consta di una relazione binaria E_I su A_n. (A_n, I) è un grafo.

Ogni formula nel vocabolario costituito da E e dall'uguaglianza può quindi essere vista come un **riconoscitore di grafi**. Ad esempio possiamo esprimere con la seguente formula una proprietà dei grafi:

$$BIN \equiv \forall x.\exists y.\exists z.\forall w.(y \neq z \wedge E(x,y) \wedge E(x,z)) \wedge (E(x,w) \rightarrow w = y \vee w = z).$$

$(A_n, I) \vDash BIN$ sse (A_n, I) è un grafo binario completo.

2.1.6 Interpretazioni come stringhe

Dobbiamo trovare un modo per descrivere le interpretazioni in modo compatto. Supponiamo di lavorare con l'universo finito $A_n = \{1, ..., n\}$. Supponiamo che un vocabolario sia costituito da m simboli predicativi $P_1, ..., P_m$ e da k simboli di funzione $f_1, ..., f_k$ tutti di arietà 0.

Una qualunque interpretazione I per tale vocabolario può essere quindi descritta da una stringa $\mathbf{bin}^n(I) \in \mathbb{B}$ dove

$$\mathbf{bin}^n(I) = \mathbf{bin}^n(P_1)...\mathbf{bin}^n(P_m)\mathbf{bin}^n(f_1)...\mathbf{bin}^n(f_k)$$

con

- per ogni $1 \leq i \leq m$, la stringa $\mathbf{bin}^n(P_i)$ ha lunghezza pari a $n^{\mathbf{ar}(P_i)}$, dove $\mathbf{ar}(P_i)$ è l'arietà di P_i. Tale stringa specifica quando ogni possibile tupla fa parte di $(P_i)_I$ oppure no
- per ogni $1 \leq i \leq k$, la stringa $\mathbf{bin}^n(f_i)$ ha lunghezza pari a $[\log_2(n)]$ e specifica quale elemento di A_n interpreta f_i

2.1.7 Formule come funzioni

Una formula predicativa F si dice *chiusa* quando nessuna variabile occorre libera in F. Ad ogni formula predicativa chiusa di F si può far corrispondere un sottoinsieme $\mathbf{struct}(F)$ come segue:

$$\mathbf{struct}(F) = \{\mathbf{bin}^n(I) \mid (A_n, I) \vDash F\} \subseteq \mathbb{B}$$

Ci possiamo chiedere quale sia la classi di linguaggi che una certa logica caratteristica. Prendiamo per esempio la logica predicativa:
$FO = \{\mathbf{struct}(F) \mid F$ è una forma predicativa chiusa$\}$.

FO coincide con una classe interessante? Per fare in modo che la logica predicativa abbia un minimo di potere espressivo, occorre assumere che il vocabolario includa tre simboli funzionali di arietà nulla chiamati $0, 1$, max e due simboli predicativi binari $\leq$ e BIT.

Lemma: $FO \subseteq L$
Lemma: $PARITY \notin FO$
Teorema: $FO \subsetneq L$

La logica predicativa del primo ordine quindi è troppo poco espressiva. Passiamo quindi alla logica predicativa del secondo ordine. Vengono quindi aggiunti i seguenti termini:

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

La semantica di queste formule segue quella della logica predicativa, ma occorre che ξ assegni una relazione n-aria ad ogni variabile X^n. ξ è quindi una funzione che mappa variabili libere del secondo ordine in F ad elementi di A. In questo modo:
$(A, I), \xi \vDash X^n(t_1, ..., t_n)$ sse $([t_1]_\xi^{(A,I)}, ..., [t_n]_\xi^{A,I}) \in \xi(X^n)$
$(A, I), \xi \vDash \exists X^n.F$ sse $(A, I), \xi[X^n := R] \vDash F$ per qualche $R \subseteq A^n$
$(A, I), \xi \vDash \forall X^n.F$ sse $(A, I), \xi[X^n := R] \vDash F$ per tutte le $R \subseteq A^n$

Facciamo un esempio di proprietà dei grafi usando la logica del secondo ordine: la *reachability*. Lo definiamo come l'insieme $\mathbf{struct}(\phi_{s,t}) = \{\mathbf{bin}^n(I) \mid (A_n, I)$ è

un grafo e t è raggiungibile da s}. Non è definibile al prim'ordine. Costruzione: TODO.

La logica del secondo ordine però è troppo potente per i nostri scopi. Cerchiamone un frammento più interessante, cioè la **logica del second'ordine esistenziale**, le cui formule sono le formule che possono essere scritte nella forma $\exists X^{n_1}...\exists x^{n_m}.F$ dove F è una formula predicativa al *prim'ordine*.

Definiamo quindi

$$\exists SO = \{\mathbf{struct}(F) \mid F \text{ è una formula a second'ordine esistenziale}\}.$$

2.1.8 Il teorema di Fagin

Fagin dimostra che $\exists SO = NP$.

Dimostrazione: Passi principali:

1. Prima dimostriamo $\exists SO \subseteq NP$
2. Per farlo abbiamo bisogno di dimostrare alcuni lemmi.
3. Lemma 1: ogniqualvolta esista almeno un simbolo predicativo di arietà almeno pari ad 1, vale che $|\mathtt{bin}^n(I)| \geq n$
4. Lemma 2: $FO \subseteq P$. Questo lemma però è di difficile dimostrazione perché dovremmo andare per induzione su delle formule F che però potrebbero avere sottoformule aperte alle quali non si possono applicare le ipotesi induttive. Quindi bisogna dimostrare un lemma più generico
5. Dimostriamo il seguente: per ogni F con variabili libere $x_1, ..., x_n$ esiste un algoritmo A_F polytime tale che su input $S, i_1, ..., i_n$ determina se $S = \mathtt{bin}^n(I)$ dove $(A_n, I), \xi \vDash F$ dove $\xi(x_j) = i_j$. Questo lo possiamo dimostrare per induzione su F
6. Per dimostrare $\exists SO \subseteq NP$ dobbiamo quindi costruire un algoritmo di decisione nondeterministico e polytime
7. Dimostriamo $NP \subseteq \exists SO$, dobbiamo quindi codificare ogni problema in NP in una formula di $\exists SO$, per farlo dobbiamo creare una MdT M che accetti un input x.
8. per descrivere l'esecuzione di M costruiamo una matrice $n^k - 1 \times n^k - 1$ dove gli elementi sono dell'insieme $\Sigma \cup (\Sigma \times Q)$ cioè dal simbolo della quando la testina non è presente sulla cella e il simbolo quando la testina c'è

2.1.9 Formule Positive e Minimi Punti Fissi

Per catturare la classe di problemi P, consideriamo il vocabolario relativo ai grafi, ossia in cui l'unico simbolo relazionale E è binario e rappresenta l'adiacenza tra nodi. Il fatto che un nodo y sia *raggiungibile* in un numero non specificato di passi da x non è esprimibile nella logica del prim'ordine, ma è esprimibile con la logica del secondo ordine come mostrato prima. Per poterla esprimere, serve quindi arricchire la logica. In questo caso, ci servirebbe trovare il predicato "più piccolo" tra tutti quelli che soddisfano la seguente "equazione":

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

Questa equazione, in modo intuitivo, esprime correttamente la raggiungibilità. Da notare però che uso il predicato che stiamo definendo nel predicato stesso. Intuitivamente, la formula è corretta perché l'operazione di E^* che c'è a sinistra è più piccola di E^* a destra, perché abbiamo tolto un arco. Quello che stiamo definendo è quindi un minimo punto fisso. Viene detto minimo perché qualsiasi punto fisso lo contiene. Andiamo quindi ha definire come aggiungere il minimo punto fisso alla logica del prim'ordine.

Consideriamo una formula predicativa al prim'ordine F in cui le variabili libere sono una variabile del second'ordine X^m e m variabili al prim'ordine $x_1, ..., x_m$. Tale formula predicativa si dice X^m-positiva se ogni occorrenza di X^m in F è nello scope di un numero **pari** di negazioni. Data un'interpretazione I per A_n, possiamo pensare a F come ad un funzionale (una funzione che ha stesso dominio e codominio) F^I su $P(A_n^m)$, ossia il seguente:
$D \to \{(a_1, ..., a_m) \in A_n^m \mid (A_n, I), \xi \vDash F$, dove $\xi(X^m) = D$ e $\xi(x_i) = a_i\}$

Def. di monotono: $x \leq y \iff f(x) \leq f(y)$

Teorema di Knaster–Tarski: per ogni F che sia X^m-positiva, il funzionale F^I è monotono, e ammette quindi un minimo punto fisso $\mu^I X^m(x_1, ..., x_m).F$.

Un modo per calcolare il punto fisso è

$\mu F = \cap\{Y \mid Y \supseteq F(Y)\}$.

Dimostrazione: TODO.

Le formule della *logica predicativa con minimi punti fissi* sono le stesse della logica predicativa al prim'ordine, ma tra i simboli predicativi ve ne sono anche nella forma $LFP(X^m, x_1, ..., x_m, F)$ dove F è una formula X^m-positiva. Alle nuove formule si può dare semantica nel modo seguente:
$(A, I), \xi \vDash LFP(X^m, x_1, ..., x_m, F)(t_1, ..., t_m)$
sse $([t_1]_\xi^{(A,I)}, ..., [t_m]_\xi^{(A,I)}) \in \mu^i X^m(x_1, ..., x_m).F$.

Possiamo definire quindi la classe:
$FO(LFP) = \{\mathbf{struct}(F) \mid F$ è una formula predicativa con minimi punti fissi$\}$.

Teorema (Immerman, Vardi): $FO(LFP) = P$.
Corollario: $P = NP$ sse $FO(LFP) = \exists SO$.

2.2 Calcolo relazionale

L'introduzione del modello relazionale viene introdotto da Codd nel 1970. Il concetto centrale è quello di *query*.

2.2.1 Relazioni Ordinate

Supponiamo di lavorare con dei **domini** come i seguenti: l'insieme dei numeri naturali $\mathbb{N}$; l'insieme delle stringhe in un alfabeto Σ, ovvero Σ^*; l'insieme dei valori booleani $\mathbb{T} = \{0, 1\}$. Indichiamo un generico dominio con D.

Una **relazione** può essere vista come un sottoinsieme *finito* R di $D_1 \times ... \times D_n$ dove i D_i sono *domini*. R vive nell'insieme delle parti finite di $D_1 \times ... \times D_n$: $P_{fin}(\prod_{1 \leq i \leq n} D_i) = P_{fin}(D_1 \times ... \times D_n)$.

Una relazione è un insieme finito di n-uple nella forma $(d_1, ..., d_n)$ dove $d_i \in D_i$ per ogni $1 \leq i \leq n$. Per quello che ci serve possiamo dire che esiste un unico E tale che $E = D_i$ per ogni $1 \leq i \leq n$.

E	E	E	E
Rossi	Mario	1973	0
Verdi	Carlo	1978	1
Gialli	Luca	1980	1
Bianchi	Andrea	1971	0

$$E = ASCII^* \uplus \mathbb{N} \uplus \mathbb{T}.$$

Figure 1: Esempio di relazione ordinata

2.2.2 Relazioni Non Ordinate

Spesso conviene dare alle colonne di relazione un *nome* e non solo una *posizione*. Una relazione diventerebbe quindi: un insieme finito $\{f_1, ..., f_n\}$ dove $f_i : C \rightarrow$

D e C è un insieme finito di campi.

La nostra relazione d'esempio diventerebbe $\{f_1, f_2, f_3, f_4\}$ dove l'insieme C è, per esempio {COGNOME, NOME, ANNO, SOCIO}.

2.2.3 Equivalenza tra le due nozioni

Ogni relazione **ordinata** può essere trasformata in una relazione **non ordinata** (è sufficiente *dare un nome* a ciascun intero compreso tra 1 e n) e viceversa (basta fissare un *ordine totale* su C).

2.2.4 Query come funzioni

Che funzione calcola una query? La base di dati è vista come una sequenza di relazioni, quindi il risultato deve essere anch'esso una relazione. In altre parole, la funzione $[Q]$ calcolata da una query Q ha la forma seguente:
$[Q] : P_{fin}(D^{n_1}) \times ... \times P_{fin}(D^{n_k}) \to P_{fin}(D^{n_m})$

2.2.5 Algebra relazionale

La sintassi dell'algebra relazionale sull'insieme dei simboli relazionali $\{R_1, ..., R_k\}$ è la seguente:
$Q, P ::= R_i \mid Q \cup P \mid Q - P \mid Q \times P \mid \pi_l(Q) \mid \sigma_c(Q)$
$c ::= i \leq j \mid i = j \mid \neg c \mid c \wedge d \mid c \vee d$ dove:

- i e j sono numeri naturali positivi
- l è la sequenza di numeri naturali positivi, ossia un elemento di $\mathbb{N}_+^*$

Un'interrogazione Q soddisfa i **vincoli di integrità** se i numeri che occorrono in essa sono coerenti con la relazione a cui si riferiscono.

Data una query Q su $\{R_1, ..., R_k\}$, essa ha semantica:

$$[Q] : P_{fin}(D^{n_1}) \times ... \times P_{fin}(D^{n_k}) \to P_{fin}(D^{n_m})$$

dove $n_1, ..., n_k$ sono le arietà di $R_1, ..., R_k$ e m è l'arietà di Q. La funzione $[Q]$ è definita per induzione sulla struttura di Q. Se Q è R_i, allora $[Q](R_1, ..., R_k)$ è semplicemente R_i. Gli operatori $\cup, -, \times$ hanno un'interpretazione insiemistica.

Nella proiezione, interviene una lista di interi l, che indica semplicemente quali campi considerare nella proiezione. I vincoli d'integrità sono cruciali. Formalmente, $[\pi_{i_1,...,i_s}(Q)](R_1, ..., R_k)$ sarà l'insieme
$\{(d_{i_1}, ..., d_{i_s}) \mid (d_1, ..., d_n) \in [Q](R_1, ..., R_k)\}$

Nella selezione, interviene invece una condizione c, che indica quali tuple considerare nella selezione. I vincoli d'integrità sono cruciali. Data una tupla di valori $t = (d_1, ..., d_n)$ e una condizione c, possiamo definire quando quest'ultima è soddisfatta in $(d_1, ..., d_n)$, per induzione. La relazione $[\sigma_c(Q)](R_1, ..., R_k)$ sarà
$\{t \mid t \in [Q](R_1, ..., R_k) \wedge t \vdash c\}$

2.2.6 Algebra relazionale - potere espressivo

L'insieme delle funzioni che l'algebra relazionale ci permette di catturare sono quella della query ben formate:
$AR = \{[Q] \mid Q \text{ è una query ben formata}\}$

2.2.7 Calcolo relazionale

Un modo naturale per lavorare con le relazione è la logica predicativa. Supponiamo di voler interrogare una base di dati che consti delle relazioni $R_1, ..., R_k$ aventi arietà $n_1, ..., n_k$ ottenendo come **risultato** una relazione Q di arietà m.

Basterà costruire una formula predicativa F con:

- Gli unici simboli funzionali sono delle *costanti* che indicano gli elementi di D, mentre i simboli predicativi saranno $R_1, ..., R_k$, più i simboli $\leq$ e $=$, questi ultimi preinterpretati
- Le variabili che occorrono libere in F dovranno essere incluse nell'insieme $\{f_1, ..., f_m\}$ e corrispondere ad un campo della relazione Q

Esempio:

0012	Rossi	Mario	1973	0
6783	Verdi	Carlo	1978	1
0987	Gialli	Luca	1980	1
4562	Bianchi	Andrea	1971	0

Table 5: Tabella R_1

0012	6783	3	2
6783	4562	1	3
0987	4562	0	3

Table 6: Tabella R_2

Vogliamo trovare una formula per il calcolo relazione che catturi gli anni di nascita dei vincitori. Un esempio potrebbe essere:

$$\exists p.\exists s.\exists c.\exists n.\exists o.\exists pp.\exists ps.R_1(p, c, n, f, 0) \wedge R_2(p, s, pp, ps) \wedge (pp > ps)$$

$$\vee$$

$$\exists p.\exists s.\exists c.\exists n.\exists o.\exists pp.\exists ps.R_1(s, c, n, f, 0) \wedge R_2(p, s, pp, ps) \wedge (ps > pp)$$

2.2.8 Calcolo relazione - semantica

L'universo in cui *interpretare* una formula F del calcolo relazionale è D. Gli unici simboli che occorre *interpretare* per dare una semantica a F sono $R_1, ..., R_k$. Quindi, è naturale vedere la base di dati $\{R_1, ..., R_k\}$ come una tale interpretazione.

Se che vale che $(D, \{R_1, ..., R_k\}), \xi \vDash F$, cioè significa che $(\xi(f_1), ..., \xi(f_m))$ deve stare nella relazione Q.

Di conseguenza, possiamo porre $[F](R_1, ..., R_k)$ pari a
$\{(\xi(f_1), ..., \xi(f_m)) \mid (D, \{R_1, ..., R_k\}), \xi \vDash F\}$

In questo modo, $[F] \subseteq D^m$, ma *non è detto che* $[F]$ *sia finita!* Basti considerare la formula $F = (f_1 = f_1)$.

2.2.9 Calcolo relazione sicuro

Occorre quindi isolare un sottoinsieme delle formule del calcolo relazionale: le **formule sicure**. Se una formula F è sicura, allora ogni tupla in $[F]$ contiene valori tra quelli che occorrono in F e quelli che troviamo nelle tuple in $R_1, ..., R_k$. Quindi, $[F]$ è sempre finita. Useremo l'insieme delle variabili che occorrono libere in una formula F, detto $FV(F)$.

1. L'uso del quantificatore *universale* non è permesso.
2. Ogni volta che si utilizza l'operatore $\vee$ per formare $F \vee G$, deve valere che $FV(F) = FV(G)$.
3. Se una sotto-formula della formula data di può scrivere come $F_1 \wedge ... \wedge F_m$, (dove $m \geq 1$ è massimale), allora ogni $x \in \cup_{1 \leq i \leq m} FV(F_i)$ deve essere *limitata*, ossia deve esistere *almeno una* formula F_j tale che:
   (a) $x \in FV(F_j)$ e F_j non è un predicato aritmetico e non è nella forma $\neg G$.
   (b) F_j è nella forma $x = c$ oppure $c = x$, dove c è una costante.
   (c) F_j è nella forma $x = y$ dove y è anch'essa limitata.
4. L'unico uso permesso dell'operatore di negazione è in una delle formule $F_i = \neg G$ di una congiunzione $F_1 \wedge ... \wedge F_m$ in cui vi sia almeno una delle F_p che non sia essa stessa negata.

2.2.10 Calcolo Relazionale Sicuro - Potere Espressivo

Definiamo l'insieme delle funzioni che il calcolo relazionale sicuro cattura:

$CR = \{[F] \mid F$ è una formula sicura del calcolo relazionale$\}$

Teorema: $AR = CR$

Dimostrazione: Passaggi principali:

1. Dimostriamo prima $AR \subseteq CR$, la prova procede per induzione sulla struttura algebrica A che vogliamo tradurre in CR

2. occorre dire qualcosa sulle query di selezioni, definiamo **semplice** una query relazionale Q tale che tutti gli operatori di selezione $\sigma_c(R)$ in Q sono tali che c è un operatore aritmetico o la sua negazione

3. Lemma: per ogni Q esiste P semplice tale che $[Q] = [P]$. Per dimostrarlo, dimostriamo prima il seguente sottolemma per induzione su c: se Q è semplice, allora esiste una query semplice R equivalente a $\sigma_c(Q)$. Dato questo sottolemma, si dimostra il lemma per induzione su Q

4. tornando a dimostrare $AR \subseteq CR$, procedendo per induzione su una query Q. Nel caso di $Q = \sigma_c(R)$ usiamo il lemma

5. Ora dimostriamo $CR \subseteq AR$, per farlo passiamo per un linguaggio intermedio chiamato DATALOG. Un programma datalog ha forma $H :- B_1 \& ... \& B_q$ dove H è la testa della regola mentre B è il corpo. La testa ha forma $P(A_1, ..., A_n)$ dove A sono variabili oppure costanti. Il corpo può avere predicati, relazioni, formule atomiche e operazioni logiche

6. Per la prova ci interessa i programmi datalog non ricorsivi (bisogna controllare il grafo delle dipendenze) e sicuri (simile al CR sicuro)

7. Siamo interessati a dimostrare il seguente lemma: ogni programma datalog non ricorsivo e sicuro D ha semantica ben definita

8. Dobbiamo quindi trovare una funzione `c2d` che traduca il CR sicuro in datalog non ricorsivo e sicuro e una funzione `d2a` che traduca a sua volta in AR: $F \rightarrow$`c2a(c2d(`F`))`

9. la traduzione in datalog sarà definita per ricorsione sulla struttura F (l'unica difficoltà è sul caso $F = G_1 \wedge ... \wedge G_m$)

10. costruzione della funzione `d2a`: si procederà attraverso 5 fasi

    (a) rettificazione delle regole

    (b) calcolo dell'espressione dom

    (c) calcolo dell'ordine topologico del grafo delle dipendenze

    (d) calcolo di un'espressione dell'algebra relazionale per ciascuna regola del programma datalog

    (e) calcolo di un'espressione dell'algebra relazionale per ciascuna relazione ausiliaria del programma datalog

11. osserviamo coem sia stata fatta l'assuzione che la parte relazionale **NON** contenga costanti; tale assunzione non fa perdere generalità

2.3 Verifica dei sistemi e logica modulare

Cosa significa verificare la correttezza di un sistema? Data la descrizione di un sistema S e una proprietà P che descriva il comportamento atteso, occorre **verificare** che S effettivamente soddisfi P.

É possibile fare la verifica in modo automatico? Quasi sempre, lo scenario è il seguente: sia S il programma da verificare e P la proprietà da verificare. A è il verificatore che darà un risposta $o \in \{yes, no, maybe\}$. Tra le possibili risposte c'è anche *maybe* perché spesso verificare che S soddisfi P è spesso indecidibile (si potrà risolvere solo un'approssimazione del problema).

2.3.1 Model Checking

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

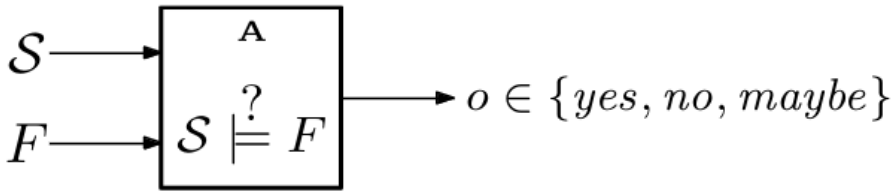

Figure 2: Concetto grafico di model checking

Bisogna però capire in che senso un sistema o programma possa essere visto come un'interpretazione e quale sia la logica adatta a specificare le proprietà d'interesse.

2.3.2 Strutture di Kripke

Supponiamo che AP sia un insieme di proposizioni atomiche, che catturino le proprietà di interesse di uno stato, magari astraendo sullo stato stesso. Una **struttura di Kripke** su AP è una quadrupla $M = (S, S_0, R, L)$ dove:

- S è un insieme di *stati*.
- $S_0 \subseteq S$ è l'insieme degli *stati iniziali*
- $R \subseteq S \times S$ è la *relazione di transizione*, che supponiamo totale: per ogni $s \in S$ esiste $t \in S$ con $(s, t) \in R$
- $L : S \to P(AP)$ è una *funzione di etichettatura*

L'insieme degli strati si suppone spesso essere finito, questo per garantire la decidibilità. La funzione di etichettatura ha il ruolo di dire quale proposizione atomiche valgono in ogni stato. Vediamo un esempio.

Consideriamo il seguente 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* di questo programma può essere visto come l'insieme

$\{l_1, l_2, \cdot\} \times \{l_3, l_4, \cdot\} \times \{0, 1\} \times \{0, 1\}$
mentre l'unico stato iniziale potrebbe essere $(l_1, l_3, 1, 1)$.

La relazione di transizione corrisponde a quella intuitiva, ed è costruita usando il principio dell'*interleaving*.

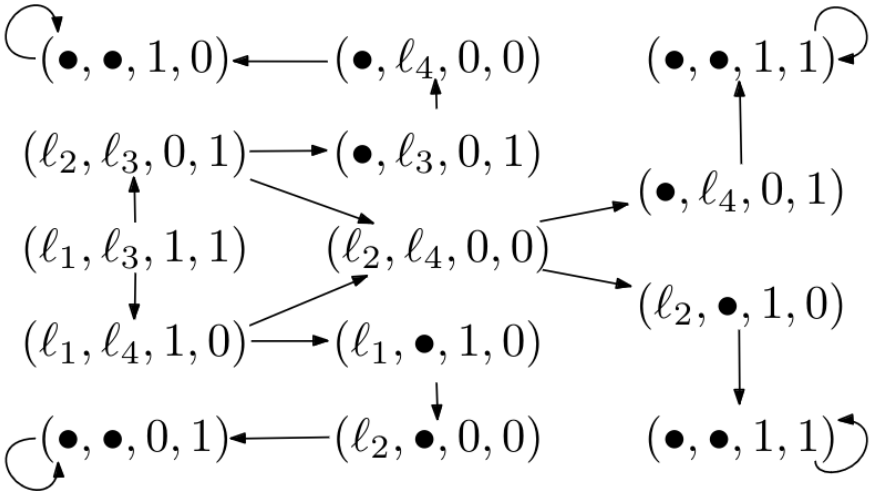

Figure 3: Sistema di transizione d'esempio

L'insieme AP potrebbe contenere queste proposizioni: una chiamata `null`, che modella il fatto che entrambe le variabili x e y valgono 0, e una chiamata `stop`, che modella la terminazione del programma. Formalmente, $\texttt{null} \in L(S)$ sse le ultime due componenti di S sono entrambe 0; $\texttt{stop} \in L(S)$ sse le prime due componenti di S sono entrambe $\cdot$. Ciò non toglie che AP possa contenere anche tante altre proposizioni atomiche, come per esempio $x = 0, y = 1, l_1$, il cui significato è intuitivo.

Come specificare proprietà di sistemi? Proviamo a sfruttare la **logica proposizionale** in cui le proposizioni atomiche sono gli atomi. In questo modo possiamo parlare del sistema in senso statico, ossia della struttura di Kripke e *di un suo stato*.

Ad esempio, possiamo concludere che:

$M, (l_1, l_3, 0, 0) \vDash$ `null`$\wedge\neg$`stop`

$M, (\cdot, \cdot, 1, 1) \vDash$ `null`$\rightarrow$ `stop`

Si potrebbe scrivere $M \vDash F$ sse $M, S \vDash F$ per ogni $S \in S_0$.

Manca l'aspetto dinamico. Come esprimere proprietà relative all'**evoluzione** del sistema? Ad esempio: **reachability, safety**, etc...

2.3.3 Logiche temporali

Le logiche temporali possono essere viste come estensioni della logica proposizionale ottenute dotando quest'ultima di **operatori modali** che permettono di esprimere in che senso una formula vale *nel futuro* e il fatto che certe formule valgano *in alcune* esecuzioni nondeterministiche, oppure *in tutte*.

Operatori temporali:

- se una formula F *vale ora e rimane valida nel futuro*, allora scriviamo $(G\ F)$ dove G sta per `global`
- se una formula F vale *dopo la prossima* transizione di stato, allora scriviamo $(X\ F)$, dove X sta per `next`
- se una formula D vale *in un certo istante indefinito del futuro*, allora scriviamo $(F\ D)$ dove F sta per `future`

Quantificatori sui cammini: se una formula F vale *indipendentemente* dal nondeterminismo, allora scriveremo $(A\ F)$. Invece, se una formula F vale per *almeno una* scelta nondeterministica, allora scriveremo $(E\ F)$. AP è un insieme di etichette (come `null` e `stop`).

2.3.4 CTL*

La logica temporale CTL* descrive una logica che, grazie agli operatori temporali, produce formule da valutare su *cammini* di esecuzione, mentre i quantificatori sui cammini vengono valutati su *stati*.

Formule di stato su AP:

$$F_S, G_S ::= P \mid F_S \wedge G_S \mid F_S \vee G_S \mid \neg F_S \mid E\ F_P \mid A\ F_P$$

dove $P \in AP$.

Formule di cammino su AP:

$$F_P, G_P ::= F_S \mid F_P \wedge G_P \mid F_P \vee G_P \mid \neg G_P \mid X\ F_P \mid F\ F_P \mid G\ F_P \mid F_P\ U\ G_P \mid F_P\ R\ G_P$$

Definiamo la *semantica* della logica temporale. Un **cammino** π in una struttura di Kripke $M = (S, S_0, R, L)$ è una sequenza infinita di stati $s_0 s_1 s_2 ... \in S^\omega$ tale che $(s_n, s_{n+1}) \in R$ per ogni naturale n.

Dato un cammino π e un naturale n, indicheremo con π^n l'n-esimo suffisso di π, anch'esso cammino. Una formula di stato F_S su AP è vera in una struttura di Kripke M su AP e in uno stato s di M. In tal caso scriveremo $M, s \vDash F_s$.

Una formula di cammino F_P su AP è vera in una struttura di Kripke M su AP e in un cammino π in M. In tal caso scriveremo $M, \pi \vDash F_P$.

Per quanto riguarda le formule di stato, i connettivi $\neg, \wedge$ e $\vee$ sono interpretati in modo standard. Le proposizioni atomiche si interpretano facendo riferimento alla struttura di Kripke: $(S, S_0, R, L), s \vDash P$ sse $P \in L(s)$. I quantificatori sui cammini fanno riferimento alla semantica delle formule di cammino:

- $M, s \vDash (E\ F_P)$ sse $M, \pi \vDash F_P$ per almeno un cammino π che inizi in s
- $M, s \vDash (A\ F_P)$ sse $M, \pi \vDash F_P$ per tutti i cammini π che inizi in s

Per quanto riguarda le formule di cammino, i connettivi $\neg, \wedge$ e $\vee$ sono interpretati in modo standard. Le formule di stato si valutano nel primo stato di cammino $F, \pi \vDash F_S$ sse $F, s \vDash F_S$. I quantificatori sui cammini fanno riferimento alla semantica delle formule di cammino:

- $M, \pi \vDash (X\ F_P)$ sse $M, \pi^1 \vDash F_P$
- $M, \pi \vDash (F\ F_P)$ sse $M, \pi^i \vDash F_P$ per almeno un i
- $M, \pi \vDash (G\ F_P)$ sse $M, \pi^i \vDash F_P$ per tutti gli i
- $M, \pi \vDash (F_P\ U\ G_P)$ sse esiste k naturale con $M, \pi^k \vDash G_P$ e $M, \pi^j \vDash F_P$ per ogni $0 \leq j \leq k$
- $M, \pi \vDash (F_P\ R\ G_P)$ sse per ogni j naturale, se per ogni $i < j, M, \pi^i \nvDash F_P$, allora $M, \pi^j \vDash G_P$

2.3.5 Frammenti di CTL*

Logica CTL:

- ogni operatore temporale deve essere immediatamente preceduto da un quantificatore sui cammini
- la grammatica per le formule di cammino diventa molto più semplice

$$F_P, G_P ::= X\ F_S \mid F\ F_S \mid G\ F_S \mid F_S\ U\ G_S \mid F_S\ R\ G_S$$

Logica LTL:

- le uniche formule considerate sono le formule di cammino, che però vengono implicitamente quantificate con il quantificatore $A\cdot$
- le formule diventano

  $$F_P, G_P ::= P \mid F_P \wedge G_P \mid F_P \vee G_P \mid \neg G_P \mid X\ F_P \mid F\ F_P \mid G\ F_P \mid F_P\ U\ G_P \mid F_P\ R\ G_P$$

2.3.6 Il problema del model checking

Model Checking Universale: data un struttura di Kripke M e una formula di F_S, determinare se $M, s \vDash F_S$ per ogni $s \in S_0$.

Model Checking Esistenziale: data un struttura di Kripke M e una formula di F_S, determinare se esiste $s \in S_0$ con $M, s \vDash F_S$.

> **Teorema:** i problemi del model checking universale e esistenziale sono $PSPACE$-completi per CTL*

Dimostrazione: non data.

> **Teorema:** i problemi del model checking universale e esistenziale per CTL sono risolvibili in tempo polinomiale.

Dimostrazione: Passaggi principali:

1. Grazie a dei lemmi possiamo assumere che la formula su ciò si voglia fare model checking contenga solo EX, EG, EU oltre agli operatori booleani
2. costruiamo un algoritmo che risolve il model checking in CTL
3. per l'algoritmo ci sarà bisogno di due procedure ausiliarie chiamate `CheckEU` e `CheckEG`
4. l'algoritmo di `CheckEU` procederà costruendo l'insieme degli stati che soddisfano $E(H\ U\ K)$ in un determinato modo
5. l'algoritmo `CheckEG` ha come idea cruciale, quella di utilizzare il concetto di componente fortemente connesso (SCC) di un grafo, ossia un sottoinsieme P dei nodi del grafo tale per cui ogni elemento di P sia raggiungibile da qualunque altro elemento di P
6. bisogna dimostrare il seguente lemma: $M, S \vDash EG\ F$ sse è possibile costruire un cammino che da s porti, tramite R ad una SCC (massimale e non triviale) contenente stati che soddisfino F
7. dimostrando il lemma è quello che serve per costruire `CheckEG`, il quale procederà determinando le SCC massimali e nontriviali di M che contengono solo stati in `States[F]`, per poi controllare da quali stati in `States[F]` tali SCC siano raggiungibili

2.3.7 Esempi di logica modale

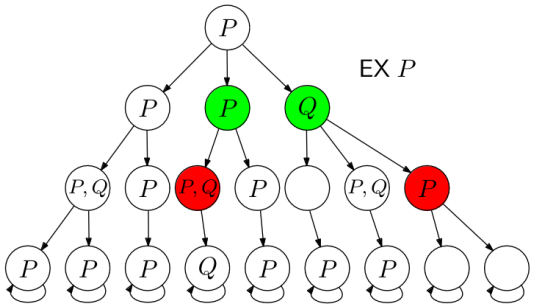

Figure 4: Esempio di logica modale per $EX\ P$

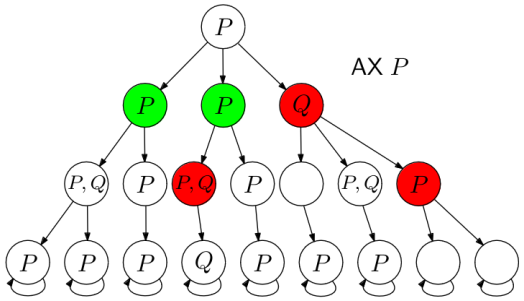

Figure 5: Esempio di logica modale per $AX\ P$

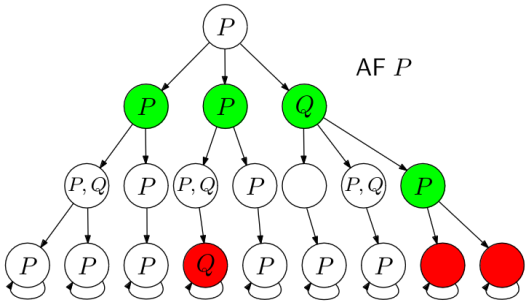

Figure 6: Esempio di logica modale per $AF\ P$

2.3.8 Esempi di logica modale con formule

Primo esempio. Abbiamo un sistema che riceve richieste che deve poi soddisfare. Definiamo le variabili proposizionali:

> `requested[i]` con $i \in \{1, ..., n\}$ vuol dire che nello stato corrente ho ricevuto la richiesta i

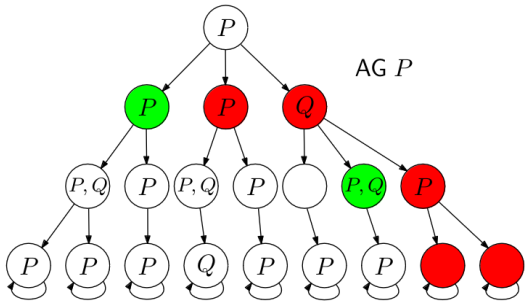

Figure 7: Esempio di logica modale per $AG\ P$

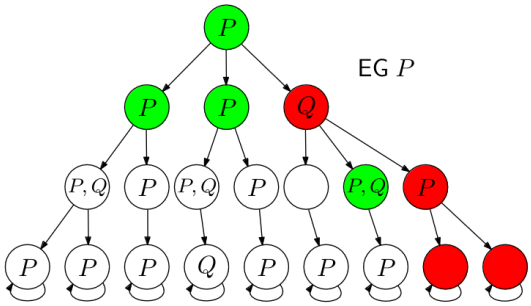

Figure 8: Esempio di logica modale per $EG\ P$

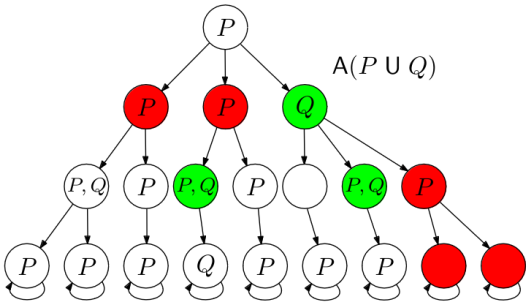

Figure 9: Esempio di logica modale per $A(P\ U\ Q)$

`acknowledged[i]` con $i \in \{1, ..., n\}$ vuol dire che nello stato corrente la i-esima richiesta è stata soddisfatta

Se vogliamo dire che ogni richiesta, se ricevuta, verrà soddisfatta prima o poi, possiamo scrivere il seguente statement:

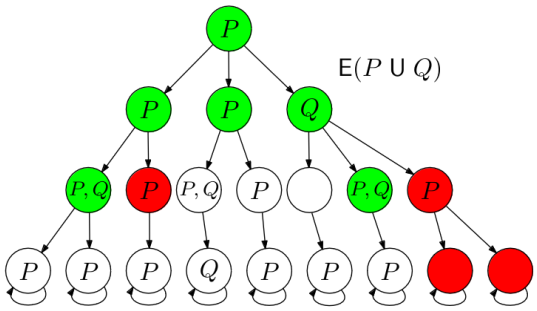

Figure 10: Esempio di logica modale per $E(P\ U\ Q)$

$\wedge_{i=1}^{n} A\,G\ (\texttt{requested[i]} \rightarrow A\,F\ \texttt{acknowledged[i]}))$

Secondo esempio. Stiamo cercando di verificare il sistema di controllo di un ascensore. Definiamo le variabili proposizionali:

`floor[i]` con $i \in \{1, ..., n\}$ vuol dire che l'ascensore è al piano i

`direction[d]` con $d \in \{up, down\}$ indica la direzione dell'ascensore

`buttonpressed[i]` con $i \in \{1, ..., n\}$ vuol dire che il bottone del piano i-esimo è premuto

Se un ascensore sta salendo e si trova in un piano inferiore al quinto, mentre il pulsante del quinto piano è premuto, l'ascensore deve continuare a salire finché non raggiunge il quinto piano.

$\wedge_{i=1}^{4}((\texttt{floor[i]} \wedge \texttt{direction[up]} \wedge \texttt{buttonpressed[5]}) \rightarrow$
$A(\texttt{direction[up]}\ \cup\ \texttt{floor[5]}))$

Terzo esempio. Abbiamo a che fare con un sistema di allarme che segnala la presenza di eventi anomali con certe garanzie temporali. Definiamo 4 atomi proposizionali: 2 per i sensori (`intrusion` e `lowbattery`) e 2 per gli attuatori (`alarm` e `signal`).

Vorremmo essere sicuri che, se viene rilevata un'intrusione, scatti l'allarme entro 3 istanti:

$A\,G\ (\texttt{intrusion} \rightarrow [AX(AX(AX(\texttt{alarm})))\vee$
$AX(AX(\texttt{alarm}))\vee$
$AX(\texttt{alarm})\vee$
$\texttt{alarm}])$

Inoltre, se c'è il rischio di rilevare un basso livello di batteria nei prossimi n istanti, occorre segnalarlo immediatamente e per almeno 2 istanti. Definiamo la formula $EX^n F$ per induzione su n:
$EX^0 F \equiv F$
$EX^{n+1} F \equiv F \vee EX(EX^n F)$

In questo modo abbiamo che la formula EX^n`lowbattery` cattura proprio il rischio che in n passi il sistema si possa trovare in una situazione di batteria scarica. Quindi possiamo scrivere la specifica:

$$AG\,[EX^n\ \texttt{lowbattery} \rightarrow \texttt{signal} \vee AX(\texttt{signal}) \vee AX(AX(\texttt{signal}))]$$