Appunti di Calcolabilità

A cura di *Roberta Prendin*

robertaprendin@gmail.com

Appunti delle lezioni di Calcolabilità e Linguaggi Formali *tenute dal professor* Antonino Salibra*, anno accademico 2013/2014. Trattandosi di appunti saranno presumibilmente presenti giri di parole, errori e imprecisioni di dubbio gusto.* *Penso riuscirete a sopravvivere ugualmente.*

Introduzione al concetto di *calcolabilità.*

Dato un problema, possiamo risolverlo attraverso un ***algoritmo***, ossia tramite una ***procedura che esegue meccanicamente un numero finito di passi arrivando alla soluzione***. Se una funzione ha un algoritmo che la calcola in un tempo *finito*, è detta *calcolabile*. Non un problema è risolvibile tramite un algoritmo: lo scopo di questa parte del corso è proprio quello di capire cos'è e cosa non è calcolabile. Alcuni problemi *non* risolvibili da un algoritmo sono:

- **Problema della terminazione**. Non esiste un programma che, dato un programma in input, riesce a stabilire se quest'ultimo converge sempre ad un risultato.
- **Problema della correttezza**. Non esiste un programma che, dato un programma in input, stabilisce se quest'ultimo è corretto.
- **Problema dell'autoriconoscimento**. Non esiste un programma che si autoriconosce, cioè che dato in input un programma restituisce *sì* se il programma dato in input è se stesso, *no* altrimenti. Motivo: nel momento in cui creo il programma che si autoriconosce, modifico anche l'input che dovrei dare in pasto alla computazione: questa dinamicità non permette d'avere un programma autoriconoscente[1].

Introduzione ai concetti di *programma*, *funzione*, *numero naturale*.

Dato l''**insieme-alfabeto A** possiamo definire **A* come l'insieme di tutte le stringhe generabili a partire da A**. Per esempio, se $A = \{a, b\}$, allora $A^* = \{\ a, aa, \dots,\ b, bb, \dots, ab, aba, \dots\}$. Ad ogni stringa possiamo **associare un numero naturale univoco**:

$$
\begin{aligned}
&0 \to a\\
&1 \to aa\\
&\dots\\
&10 \to ab\\
&\dots
\end{aligned}
$$

Dato che un **programma** può essere **visto come una stringa** più o meno corretta sintatticamente a seconda del linguaggio usato, allora anche i **programmi sono enumerabili**, cioè ad ogni programma si può associare un numero naturale univoco. Infine un **programma** calcola sempre una funzione $f: N^n \to N^n$, dove cioè input e output sono naturali; tale funzione può essere:

- **Totale**, ossia *f* ha *sempre* un output per qualsiasi valore dato in input; un esempio di funzione totale è $f(x) = x^2$: qualunque sia il valore di *x*, $f(x)$ darà sempre un preciso output.
- **Parziale**, ossia *f*, per uno o più input, non converge ad un output ma cicla per un tempo indefinito.

- P_n è il *programma* codificato dal naturale *n.*
- $\varphi_{\mathrm{n}} : N \to N$ è la *funzione* calcolata da P_n.

  $P_n \downarrow 5$ (**converge** a 5, cioè quando P_n riceve in input 5, termina la computazione su un certo

[1] In realtà è possibile farlo, ma si tratta di un programma molto complesso e che ha bisogno di strumenti matematici per essere scritto; per ora basti sapere che tale programma si ottiene unendo diversi algoritmi noti, che finiscono per comporre un algoritmo "unico".

output)

$P \uparrow 5$ (**non converge** a 5, cioè non termina la computazione su un output, alias continua a ciclare per un tempo indefinito)

Il problema della terminazione e il Paradosso di Russell.

Paradosso di Russell. Il paradosso di Russell è una delle antinomie più celebri della storia della matematica e scredita ***la teoria ingenua degli insiemi*** **di Georg Cantor** secondo cui *per ogni proprietà esiste un corrispondente insieme di elementi che soddisfano tale proprietà*. Russell parte dall'assunzione c'è sempre valido dire che un *elemento appartiene a se stesso*:

$$x \in x$$

…per esempio, "l'insieme di tutti i concetti astratti" è a sua volta un concetto astratto, cioè appartiene a se stesso. Lo stesso vale in versione negativa: esistono cioè insiemi che non appartengono a se stessi (proprio come l'insieme delle tazze da tè non è una tazza da tè):

$$x \notin x$$

Ora, se chiamiamo ***R*** **l'insieme di tutti gli insiemi che appartengono a se stessi…**

$$R = \{ x \mid x \in x \}$$

…R appartiene a se stesso? Russell arriva al paradosso dicendo che *R appartiene a se stesso se e solo se R non appartiene a se stesso*:

$$R \in R \quad sse \quad R \notin R$$

Perché il paradosso di Russell in informatica? Perché tra il *paradosso di Russell* e il concetto di *programma* nella teoria della calcolabilità ritroviamo un parallelismo: proprio come l'incognita *x*, in Russell, funziona sia da *elemento di un insieme* che da *insieme* stesso, così un programma, in informatica, funziona sia da "programma vero e proprio", sia da *input ad un programma*. Il motivo è semplice: dato che un programma è sempre codificato da un naturale e calcola una funzione $f: N \to N$, allora un programma può calcolare la funzione *f* su un naturale che è, in realtà, la codifica di un programma.

Il problema della terminazione. Sia data la coppia di numeri naturali (x, y): il programma P_x termina se converge su y, cioè se $P_x \downarrow y$. Vogliamo dimostrare che non esiste un programma che, data la coppia (x, y), restituisce "sì, P_x converge su y" o "no, P_x non converge su y". Prendiamo l'insieme Kcontenente tutti i programmi che convergono su se stessi:

$$K = \{x : P_x \downarrow x\} \quad (Programma\ che\ converge\ su\ se\ stesso)$$

K non è decidibile, ossia non esiste un programma in grado di dire se P_x termini o meno la computazione su se stesso; poniamo per assurdo che K sia decidibile: allora esiste una funzione ***f*** **calcolabile che *decide K***, ossia che può stabilire se un dato programma termina o non termina su se stesso:

$$f(x) = \begin{cases} 1, & P_x \downarrow x \\ 0, & P_x \uparrow 0 \end{cases}$$ //per comodità, ammettiamo che *f* sia fatta così.

Modifichiamo di poco *f* e otteniamo $g(x)$:

$$g(x) = \begin{cases} \uparrow, & P_x \downarrow x \\ 0, & P_x \uparrow 0 \end{cases}$$

Anche g è calcolabile da un programma codificato dal **numero naturale r**:

$$\exists\, r \in N \quad tale\ che \quad P_r\ calcola\ g(x)$$

Ma P_r termina la computazione su se stesso? **Se e solo se P_r *non termina la computazione su se stesso:***

$$P_r \downarrow r\ se\ e\ solo\ se\ P_r \uparrow r$$

Arriviamo ad una conclusione paradossale, da cui deriva che né g né f sono calcolabili e che, di conseguenza, K non è decidibile. ***Nota***: in questa dimostrazione abbiamo lavorato su *K*, cioè su programmi che hanno *un* solo input, ma il risultato vale anche nel caso vengano dati due input.

Decidibilità, semidecidibilità, non decidibilità.

Decidibilità. L'insieme $A \subseteq N$ è decidibile se la funzione caratteristica...

$$\varphi_A(x) = \begin{cases} 1, & x \in A \\ 0, & x \notin A \end{cases}$$ ...è calcolabile.

Cioè: un insieme A è decidibile se, preso un qualsiasi elemento, posso sempre *decidere se appartiene o se non appartiene* ad A. Per esempio, se $A = \{x : x \text{ è } pari\}$, allora posso sempre decidere se un certo y è o non è pari, cioè se x appartiene o non appartiene ad A.

Semidecidibilità. L'insieme $A \subseteq N$ è semidecidibile se la funzione *semi*caratteristica...

$$\chi_A(x) = \begin{cases} 1, & x \in A \\ \uparrow, & x \notin A \end{cases}$$ ...è calcolabile.

Cioè: un insieme A è semidecidibile se, preso un qualsiasi elemento, posso *solo* decidere se l'elemento appartiene ad A, ma *non* posso decidere se l'elemento non appartiene ad A. In altre parole posso avere conferma che l'elemento appartiene ad A, ma non che l'elemento non appartenga ad A.

Composizione della decidibilità. L'insieme A è decidibile *se e solo se* A e $\bar{A}$ sono semidecidibili. Infatti:

- Se A è decidibile, allora A e $\bar{A}$ sono semidecidibili.
- Se A e $\bar{A}$ sono semidecidibili allora, preso un certo *x*, posso *semidecidere se* appartiene a A o $\bar{A}$, di fatto *decidendo del tutto l'appartenenza di x ad A*. Sono infatti sicuro che uno dei due algoritmi componenti terminerà la computazione.

Caratteristiche dell'insieme K.

K è l'insieme dei programmi che convergono su se stessi: $K = \{x : P_x \downarrow x\}$; K gode di alcune proprietà:

- K **non è decidibile** ma è **semidecidibile**.
- $\overline{K}$ non è semidecidibile.
- K è $\boldsymbol{infinito}$, cioè è formato da infiniti programmi che terminano la computazione su se stessi (motivo: le funzioni costanti sono infinite e convergono tutte su se stesse; inoltre, se K fosse finito, diventerebbe decidibile).

Insiemi *ricorsivamente enumerabili.*

L'insieme A è *ricorsivamente enumerabile* se esiste un algoritmo che lo enumera); in particolare A è ricorsivamente enumerabile se:

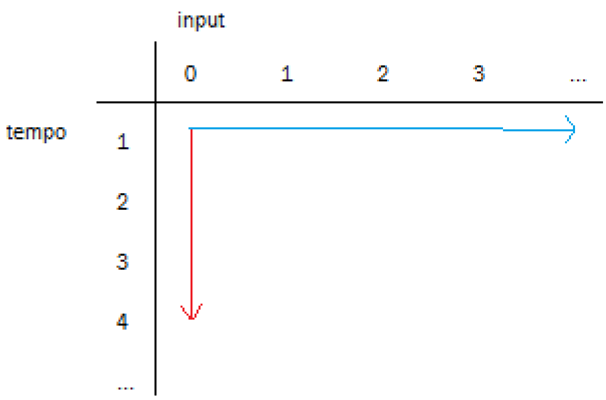
- $A = \emptyset$, cioè coincide con l'**insieme vuoto**.
- Esiste una ***f* calcolabile totale** $f: N \rightarrow A$, cioè tale che A è il suo codominio.

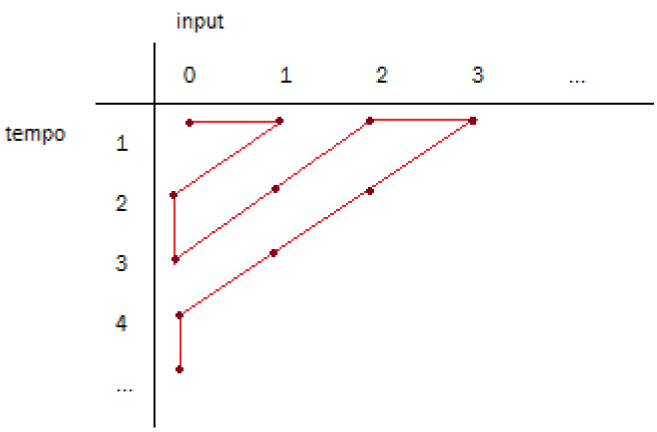
Sono ricorsivamente enumerabili l'insieme dei numeri naturali, primi, pari, dispari, etc.

Proprietà #1. ***A è ricorsivamente enumerabile se e solo se A è semidecidibile***. Infatti:

- $\boldsymbol{Se\ A}$ è $\boldsymbol{semidecidibile,\ A}$ è $\boldsymbol{un\ insieme\ ricorsivamente\ enumerabile}$.

Se A è semidecidibile allora possiamo ipotizzare che esista un algoritmo che, a partire da 0 e via via per tutti i naturali, risponde o "sì, il numero appartiene ad A" o cicla per un tempo infinito **(tempo infinito per un solo input)**. C'è un problema: non possiamo aspettare che l'algoritmo analizzi il numero per un tempo indefinito prima di dare l'eventuale risposta; per risolvere possiamo fissare uno "slot di tempo" entro il quale compiere l'analisi: scoperto che l'algoritmo impiega un certo tempo *t* per semidecidere 0, analizziamo i naturali successivi ciascuno per un tempo non superiore a *t*; terminato il tempo *t*, l'algoritmo o ha semideciso il naturale, o smette di ciclare e passa direttamente al naturale successivo, che verrà analizzato per un tempo non superiore a *t* **(input infiniti da analizzare)**. Anche in questo caso abbiamo un problema: il tempo *t* potrebbe non essere sufficiente a semidecidere un naturale. La soluzione è quella di **scorrere la tabella input-tempo a "zig-zag"**, passando più volte su ciascun input per un tempo via via crescente: ogni input viene così enumerato più volte, cioè ricorsivamente, su intervalli di tempo diversi.





Nota: uno stesso naturale viene analizzato più volte perché appartiene al codominio di una funzione *totale, f*. Infatti, se tale codominio è finito, più input possono essere mappati ad un solo output: per esempio, se

$A = \{0\}$, allora $f(0) = 0,\ f(1) = 0,\ f(2) = 0,$ ... e 0 sarà ripetuto tutte queste volte. Solo in questo modo sono sicuro che la mia funzione sia totale.

- $\boldsymbol{Se\ ho\ algoritmo\ che\ enumera\ A,\ allora\ A}$ è $\boldsymbol{semidecidibile}$.

Se ho un algoritmo che enumera gli elementi di A, basta eseguire l'algoritmo e verificare se un certo input da me cercato viene enumerato: in questo modo il mio input viene *semi*deciso, cioè A diventa semidecidibile.

Proprietà #2. **Un insieme infinito è *decidibile se e solo se è enumerabile in ordine strettamente crescente*.**

Infatti:

- **Se un insieme infinito è decidibile, allora è enumerabile in ordine strettamente crescente**: si pensi ad esempio all'insieme N, infinito e decidibile, che posso enumerare in ordine crescente attraverso un algoritmo.
- **Se un insieme è enumerabile in ordine strettamente crescente, allora tale insieme infinito è decidibile**. Se A è un insieme numerabile in ordine strettamente crescente, allora posso sempre controllare se un certo naturale *x* appartiene o non appartiene ad A, rendendo A decidibile.

Decidibilità e funzioni totali. Sia dato $\mathrm{A} = \{x \mid \varphi_x \text{ è } totale\}$, cioè l'insieme dei programmi che calcolano una funzione totale, cioè che qualsiasi sia l'input terminano sempre la computazione. A è decidibile? Preso un qualsiasi programma, posso decidere se calcola una funzione totale, cioè se per qualsiasi input ha sempre un output? Per quanto visto per il *problema della terminazione* (non possiamo decidere se un programma termina su se stesso), saremmo tentati di dire di *no*. Dimostriamolo supponendo per assurdo che A sia decidibile: allora esiste una programma che decide A, cioè che calcola la funzione $f(x)$:

$$f(x) = \begin{cases} \varphi_x(x) + 1, & se\ \varphi_x \text{ è } totale\ (x \in A) \\ 0, & se\ \varphi_x\ non \text{ è } totale\ (x \in \overline{A}) \end{cases}$$

Chiamiamo P_r il programma che calcola $f(x)$, cioè poniamo che $\boldsymbol{\varphi_r = f(x)}$. Perciò $f(r) = \varphi_r(r) + 1$ (f è una funzione totale, quindi non dà 0); sottraggo $f(r)$ da entrambe e ottengo $0 = 1$, che è un **assurdo**, quindi **A non è decidibile**. Notare che anche il complementare di A non è semidecidibile.

Teorema del parametro.

Sia data la funzione ***f* calcolabile e totale**:

$$f: N^2 \to N,\ f(x,y) = y^x.$$

Parametrizziamo x, cioè fissiamone un valore: otteniamo $f_x(y)$, cioè una funzione equivalente a $f(x,y)$ ma definita su una variabile in meno rispetto alla precedente:

$$\boldsymbol{f_x(y) = f(x,y)\ \ con\ un\ \ parametro\ fissato}$$

Per $x = 2,\ f_2(y) = f(2,y) = y^2$

Per $x = 3,\ f_3(y) = f(3,y) = y^3$

Secondo il *teorema del parametro*, esiste una ***funzione calcolabile totale*** $\boldsymbol{S: N \to N}$ che restituisce il programma che calcola $f_x(y)$:

$S(0)$ restituisce il programma che calcola $f_0(y) = y^0$

$S(1)$ restituisce il programma che calcola $f_1(y) = y^1$

…

Dato che $S(x)$ è calcolabile, a sua volta esiste un programma che la calcola, la cui funzione sarà $\varphi_{S(n)}(y)$. Perciò abbiamo che:

$$\varphi_{S(x)}(y) = f_x(y) = f(x,y) \rightarrow \quad \varphi_{S(x)}(y) = f(x,y)$$

Scopi del teorema. Il *teorema del parametro* ha una serie di scopi utili:

- Permette costruire **programmi che lavorano su programmi**. Basta infatti ricordare che $S(x)$ lavora sui naturali: se il naturale dato in input a $S(x)$ codifica un programma, allora $S(x)$ è una funzione che prende in input un programma e restituisce un programma che calcola $f_x(y)$.
- Permette di ***ritardare l'indefinito.*** Tramite il *teorema del parametro* si lavora su un programma ($f_x(y)$) equivalente a quello di partenza ($f(x,y)$), ma con un numero minore di variabili: nel caso il programma che calcola $f(x,y)$ ma non converga mai ad un risultato ma cicli per un tempo indefinito, si riesce a spostare in là nel tempo il momento in cui si dovrà avere a che fare con questo tempo indefinito.
- Permette di **lavorare su funzioni con un numero inferiore di variabili**. La funzione $S(x)$, infatti, restituisce un programma che calcola una funzione equivalente a quella di partenza ma definita su una variabile in meno (ch'è stata parametrizzata): il risultato è sempre lo stesso, ma si calcola $f_x(y)$ invece di $f(x,y)$.

Dimostrazione**.** Il *teorema del parametro* ci dice che, data una funzione $f\colon N^2 \rightarrow N$, se questa è ***calcolabile e totale*** allora esiste una ***funzione calcolabile totale*** $S\colon N \rightarrow N$ che parametrizza *f*, cioè che restituisce lo stesso output di *f* pur lavorando su un'incognita in meno:

$$\varphi_{S(x)}(y) = f(x,y) \quad \forall x,y \in N$$

Ma come possiamo esser sicuri che $\varphi_{S(x)}(y) = f(x,y)$, cioè le due funzioni danno lo stesso output? Scriviamo il programma P_S che calcola $\boldsymbol{S(x)}$:

$$
\begin{aligned}
P_s\colon \quad & S(x) = input(x) \quad (x\ \text{è l'input dato al programma}) \\
& r \coloneqq 0 \\
& r := \ r + 1; \\
& \ldots \qquad (x\ volte) \\
& r := r + 1;
\end{aligned}
$$

Dato che *x* è dato in input, suppongo d'avere un registro *r* dove vado a memorizzare x. Faccio quindi **"+1" x-volte**. Ora per avere il programma P_f che calcola $f(x,y)$, basta **modificare $\boldsymbol{P_s}$** (dato che l'input *x* è ora nel registro *r*).

Esempio #1 (struttura tipo). Immaginiamo di volere un programma che, presi in input due programmi, ne restituisce la composizione (= giustappone i risultati uno dopo l'altro): $P, Q \rightarrow P;\ Q$

Definiamo una funzione $h(x,y,z)$, dove:

- h è il programma P;
- y è il programma Q;
- z è il naturale input di Q.

Parametrizziamo sui due programmi, x e y:

$$h(x, y, z) = \varphi_x\left(\varphi_y(z)\right)$$

...dove $\varphi_x\left(\varphi_y(z)\right)$ è una funzione che calcola $P; Q$. Posso infatti decodificare x nel programma P_x e y nel programma P_y, quindi far lavorare P_y su z; se la computazione termina, posso passare il risultato ottenuto a P_x che, a sua volta, se termina, restituisce il valore di $h(x, y, z)$. Perciò:

$$\varphi_{S(x,y)}(z) = h(x, y, z) = \varphi_x\left(\varphi_y(z)\right)$$

Esempio #2 (esponenziazione). Siano date le funzioni $f: N \rightarrow N$ e $g: N^2 \rightarrow N$. $g(x, y)$ reitera la funzione $f(x)$ y-volte:

$$g(x, 0) = x \qquad (itero\ 0\ volte)$$
$$g(x, 1) = f(x) \qquad (itero\ 1\ volta)$$
$$g(x, 2) = f\left(f(x)\right) \qquad (itero\ 2\ \text{volte})$$

...

$$g(x, y) = f(f(f(f(\ldots . f(x) \ldots) = f^y(x)$$

Definiamo la funzione $h(x, y, z)$ dove:

- x è il programma che calcola *f;*
- z è l'input di *f;*
- y è il numero di volte che reitero il programma x;

$h(x, y, z)$ è quindi la funzione che reitera y-volte il *programma* x di input z:

$$h(x, y, z) = (\varphi_x)^y(z)$$

Per cui:

$$h(x, 0, z) = z$$
$$h(x, 1, z) = \varphi_x(z)$$
$$h(x, 2, z) = \varphi_x\left(\varphi_x(z)\right)$$

...

$$h(x, y, z) = (\varphi_x)^y(z)$$

Come al solito, parametrizzo sul programma, cioè su x:

$$\varphi_{S(x)}(y, z) = h(x, y, z) = (\varphi_x)^y(z)$$

...cioè $\varphi_{S(x)}(y, z)$ è *l'esponenziazione* di $h(x, y, z)$.

Esempio #3. Sia data la funzione che effettua la moltiplicazione per 2, $f(x) = 2 * x$. Vogliamo un programma che reitera y-volte tale funzione. Come prima, consideriamo la funzione $h(x, y, z)$, dove:

- x è il programma che calcola f;

- z è l'input di f;
- y è il naturale che indica quante volte è reiterato il programma che calcola f.

Perciò, dato $h(x, y, z)$, per esempio avrò:

$se\ y = 0 \quad h(x, 0, z) = 2^0 * z = z$

$se\ y = 1 \quad h(x, 0, z) = 2^1 * z = 2z$

$se\ y = 2 \quad h(x, 0, z) = 2^2 * z$

…

$h(x, y, z) = 2^y * z.$

Parametrizziamo sul programma, quindi su x:

$$\varphi_{S(x)}(y, z) = h(x, y, z) = 2^y(z)$$

Esempio #4. Prendiamo la funzione $f: N \to N$; vogliamo un programma che calcoli 2^f. Consideriamo la funzione $h(x, y)$, cioè che ha in input $f\,(x)$ e l'input di $f\,(z)$, quindi parametrizzo sul programma, *x:*

$$\mathrm{h}(x, z) = 2^{\varphi_x(z)}$$

Questa funzione *h* è calcolabile perché, dati in input *x* e *z*, posso decodificare *x* ottenendo il programma P_x a cui posso dare in input *z*; se P_x termina la computazione su z, uso questo valore e lo uso come esponente di 2:

$$\varphi_{s(x)}(z) = h(x, z) = 2^{\varphi_x(z)}$$

I costrutti di base *della programmazione.*

Definiamo ora in modo formale gli **operatori della programmazione**, cioè le funzioni sottostanti ai programmi e sufficienti a calcolare tutto ciò ch'è calcolabile. I costrutti più semplici sono:

Funzioni costanti. Sono i numeri naturali (0, 1, 2, 3…).

Successore $(\boldsymbol{s})$**.** Restituisce l'input incrementato di 1:

$s: N \to N \setminus \{0\}$ //attenzione: i codominio è $\{1, 2, 3, 4, \dots\}$

$s(x) = x + 1$

Proiezione $(\boldsymbol{P^n_{i,j}})$. Permette di **selezionare solo una porzione di memoria,** cancellando il resto: P^n_{ij} prende una sequenza lunga *n* e restituisce solo la sequenza che va *da i a j*, estremi compresi.

$$P^n_{ij}: N^n \to N^{j-i}$$

Per esempio, se la sequenza è 2,5,7, $P^3_{2,3}$ restituisce 5,7 cancellando 2:

$$2,5,7 \xrightarrow{P^3_{2,3}} 5,7$$

Se $i > j$ cancello tutta la memoria:

$$(5, 37, 45, 2) \xrightarrow{P^4_{4,3}} (\lambda)$$

Concatenazione (;). Date due funzioni *f* e *g*, concatenarle significa far diventare l'output dell'una l'input dell'altra:

$f: N^n \to N^k$

$g: \boldsymbol{N^k} \to N^r$ (codominio della 1 è dominio della 2)

$f;g : N^n \to N^r \qquad (f;g)(x_1, \dots, x_n) = g\big(f(x_1, \dots, x_n)\big)$

Per esempio, se f coincide con la funzione successore e $g(x) = 2 * x$, ottengo:

$$(s;g)(x) = g\big(s(x)\big) = 2 * (x + 1)$$

Esponenziazione (exp). Permette di ripetere y-volte un certo programma su un certo input, ed equivale all'iterazione. Data la funzione $f\colon N^n \to N^n$:

$exp(f)\colon N^{n+1} \to N^n$

$exp(f)(x_1, \dots, x_n, y) = f^y(x_1, \dots, x_n)$

…dove quindi *f* è la funzione ripetuta, *y* il numero di ripetizioni, $(x_1, \dots, x_n)$ l'input. Dato che, ripetizione dopo ripetizione, l'output diventa l'input, dominio e codominio di f devono avere la stessa lunghezza.

Cappuccio (^). Date f e g, ^ permette alle due funzioni di operare successivamente sulla stessa area di memoria, accodando poi i risultati in modo da ottenere un'unica memoria grande quanto la somma dei codomini delle due funzioni date. L'input dato a ***f e g è lo stesso***:

$f\colon N^n \to N^k$

g: $N^n \to N^r$

$f^g\colon N^n \to N^{k+r}$

$(f^\wedge g)(x_1, \dots, x_n) = f(x_1, \dots, x_n), g(x_1, \dots, x_n)$

Per esempio:

$$S \wedge S : \mathbb{N} \to \mathbb{N}^2 \text{ quindi: } 3 \xrightarrow{S \wedge S} 4,4$$

Parallelo (||). || consente di far lavorare due funzioni su due input diversi, separatamente, e di unificare il risultato in un'unica memoria grande quanto la somma dei codomini delle due funzioni date. Date le funzioni *f* e *g*:

$f: N^n \to N^k$

$g: N^r \to N^s$

$f||g : N^{\boldsymbol{n+r}} \to N^{\boldsymbol{k+s}}$

$(f||g)(x_1, \dots, x_n, y_1, \dots, y_n) = f(x_1, \dots, x_n), g(y_1, \dots, y_n)$

Per esempio $\qquad (0;S;S)||(0;S;S;S) = 2,3$

Esempio #1. Poniamo d'avere nella memoria la seguente sequenza di dati:

5,3,7,8,10,12

Vogliamo scrivere un programma *che sommi il terzo e il quarto elemento* della sequenza, scrivendo il risultato al posto del 7 e cancellando l'8. I passi per implementare il programma sono:

- Della stringa in input, copio il 5 e il 3;
- Effettuo la somma 7+8 e la metto nella terza locazione di memoria;
- Copio il 10 e il 12.

Per cui il programma P è:

$$P = P^2_{1,2}|| + || P^2_{1,2}$$

…dove || ha lo scopo di concatenare le tre parti del programma. Notare che nei tre casi suppongo di passare ai sottoprogrammi l'input adatto: $P^2_{1,2}$ riceverà in input la stringa (5,3), + riceverà direttamente in input 7 e 8, $P^2_{1,2}$ riceverà in input la stringa (10, 12).

Esempio #2. Come risolvere l'esempio precedente usando, invece, l'operatore ^? La differenza sostanziale è che l'input dei tre programmi *deve essere lo stesso*, ossia è necessario preparare gli input:

$$P = P^6_{1,2} \wedge (P^6_{3,3}; +) \wedge P^6_{5,6}$$

…si noti che l'input è sempre una stringa di *6* numeri naturali, cioè l'input è lo stesso.

Ricapitolando:

- **||** giustappone i risultati dei sottoprogrammi; ogni singolo sottoprogramma può avere un **input creato "ad hoc"**.
- **^** giustappone i risultati dei sottoprogrammi MA ogni singolo sottoprogramma ***deve avere lo stesso input***.

Costrutti derivati *della programmazione.*

Con i costrutti visti finora possiamo costruire costrutti più elaborati.

Somma $(+)$. Dati due addendi, la somma si ottiene applicando al primo addendo la funzione successore tante volte quant'è il secondo addendo:

$$+ =_{def} \exp(s)$$

Applichiamo questo programma su un paio d'esempi:

$3{,}2$ $-$ *reitero la* funzione *successore* 2 *volte sull'input* 3:

$$3 \xrightarrow{s} 4 \xrightarrow{s} \mathbf{5}$$

$3{,}0$ $-$ *reitero la funzione successore* 0 *volte sull'input* 3:

3

Moltiplicazione $(*)$. La moltiplicazione si ricostruisce a partire dalla somma: 3*4 si ottiene sommando 4 volte il numero 3 *allo 0*:

$$0{,}3 \xrightarrow{\text{sommo 3, memorizzo 3}} 3{,}3 \xrightarrow{\text{sommo 3, memorizzo 3}} 6{,}3 \rightarrow (\ldots) \rightarrow 12{,}3$$

Devo quindi:

- Inizializzare i dati da cui devo partire: lo 0 e la stringa di due input: (0 || $\boldsymbol{P^2_{2,2}}$)
- Ripetere la somma (+"secondo elemento della stringa"): $exp(+\wedge P^2_{22})$
- Rimuovere il valore da sommare una volta ottenuto il risultato voluto: $P^2_{1,1}$.

Perciò:

$$* =_{def} (0||P^2_{1,2});\ exp(+\wedge P^2_{22}); P^2_{1,1}$$

Nota: non possiamo definire la moltiplicazione come “esponenziazione della somma”, in quanto la funzione somma ha dominio e codominio con cardinalità diversa; l’esponenziazione, cioè, non si può applicare.

Esempio #1. Come viene effettuato $3 * 4$?

$$3,4 \xrightarrow{(0||P^2_{1,2})} 0,3,4$$

$$0,3 \xrightarrow{(+\wedge P^2_{2,2})} 3,3 \xrightarrow{(+\wedge P^2_{2,2})} 6,3 \xrightarrow{(+\wedge P^2_{2,2})} 9,3 \xrightarrow{(+\wedge P^2_{2,2})} 12,3 \xrightarrow{P^2_{1,1}} \mathbf{12}$$

$$-(+\wedge P^2_{2,2})\rightarrow \quad -(+\wedge P^2_{2,2})\rightarrow \quad -(+\wedge P^2_{2,2})\rightarrow$$

Sottrazione $(\dot{-}\,1)$. Notare che la sottrazione *deve fermarsi a 0*, dato che lavoriamo esclusivamente sui numeri naturali; per indicarlo, usiamo il simbolo $\dot{-}$.

$$\dot{-}\,1 =_{def} \left(0||0||P^1_{1,1}\right); exp((s||P_1\,)\wedge P^2_{1,1};P^2_{2,2}$$

Duplicazione di memoria $(\boldsymbol{D_2})$. È un costrutto che permette di duplicare una cella di memoria. Immaginiamo di avere una cella con il valore *x*: vogliamo un'altra cella con quel valore. Dobbiamo:

- Inizializzare le strutture: avere cioè due celle a 0 e prendere in input il valore di *x*;
- Ripetere l’esponenziazione su entrambe le celle tante volte quanto vale x. Perciò:

$$D_2 \;= (0||0||\,P^1_{1,1}); \exp(s||s);$$

Esercizio #8. È possibile definire ^ e | | uno nei termini dell'altro?

<u>*Cappuccio in termini del parallelo*</u>. Il cappuccio lavora su una *memoria condivisa*: fornisce cioè lo stesso input a due funzioni diverse, giustapponendo poi i risultati. Tramite il cappuccio possiamo anche definire il parallelo: per capirlo prendiamo il costrutto D_2, che duplica il suo input:

$$D_2\!: \;\; x \rightarrow \; x, x$$

Attraverso il cappuccio possiamo definirlo così:

$$D_2 \,: \; P^1_{1,1} \wedge P^1_{1,1}$$

Con il parallelo, invece, l’idea è un po’ più elaborata: dato il valore *x*, inizializziamo due celle della memoria a 0, quindi eseguiamo su entrambe la funzione “successore” tante volte quanto vale *x*:

$$D_2 \;: x \xrightarrow{0|\,|0|\,|P^1_{11}} 0,0,x \xrightarrow{\exp(S||S)} x, x$$

<u>*Parallelo in termini di cappuccio*</u>: immaginiamo d’avere due funzioni, *f* e *g*. Come si traduce $f||g$ usando solo il cappuccio?

$$f||g = (P^2_{1,1}; f)\wedge\left(P^2_{2,2}; g\right)$$

La ricorsione primitiva.

Una funzione *f* si può **definire per ricorsione primitiva** se esistono due funzioni *g* e *h* tali che:

$$g\!: N^n \rightarrow N^r$$

$$h\!: N^{n+2} \rightarrow N^r$$

$$f(x,y) = recprim(g,h) = \begin{cases} f(x,0) = \boldsymbol{g}(x) & caso\ base\ \ in\ cui\ y = 0 \\ f(x,y+1) = \boldsymbol{h}\big(x,y,f(x,y)\big) & passo\ induttivo \end{cases}$$

Nel caso l'input *x* sia una **stringa di valori** la situazione non cambia:

$$f = rec(g,h) = \begin{cases} f(x_1, \dots x_n, 0) = \boldsymbol{g}(x_1, \dots, x_n) \\ f(x_1, \dots, x_n, y+1) = \boldsymbol{h}\big(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)\big) \end{cases}$$

Per semplicità, molte volte $f(x,y)$ viene indicato semplicemente con z, per cui $\boldsymbol{h}\big(x,y,f(x,y)\big) = h(x,y,z)$.

Esempio #1 (funzione segno). La ricorsione primitiva è utile per ridefinire funzioni che già conosciamo, come la *funzione segno*. La definizione standard della funzione segno è:

$$sg\colon N \to N$$

$$sg(x) = \begin{cases} 0, & x = 0 \\ 1, & x \neq 0 \end{cases}$$

Con la *ricorsione primitiva*:

$$sg(x) = \begin{cases} sg(0) = 0 & caso\ base\ \ in\ cui\ x = 0 \\ sg(x+1) = \boldsymbol{h}\big(x, sg(x)\big) = h(x,z) = 1 & passo\ induttivo \end{cases}$$

Da cui:

$$g\colon N^0 \to N \qquad g = 0$$

$$h\colon N^{0+2} \to N \qquad h(x,y,z) = 1 \qquad \forall\, y,\ z$$

Esempio #2 (prodotto). Il prodotto si può definire in questo modo:

$$* = f(x,y) = x * y$$

$$f\colon\ N^2 \to N$$

Attraverso la ricorsione primitiva abbiamo:

$$x * y = \begin{cases} x * 0 = 0 & se\ y = 0 \\ x * (y+1) = (x * y) + x = z + x & passo\ induttivo \end{cases}$$

Definiamo *g* e *h*:

$$g\colon \mathrm{N}^1 \to N \qquad g(x) = 0$$

$$h\colon N^{1+2} \to N \qquad h(x,y,z) = z + x$$

Esempio #3 (esercizio inverso). Poniamo d'avere le seguenti funzioni:

$$g(x) = 1$$

$$\mathrm{h}(x,z) = 2 * z$$

Diamo allora la definizione di *f*:

$$f = \begin{cases} f(0) = g(x) = 1 \\ f(x+1) = \boldsymbol{h}(x,z) = h\big(x, f(x)\big) = 2 * f(x) \end{cases}$$

Esempio #4. Immaginiamo d'avere una funzione definita con ricorsione primitiva: il calcolo della funzione su un dato input segue il *pattern matching*: per esempio, se $x = 3\ \ e\ y = 5$, l'operazione $3 * 5$ diventa:

- $5 = 0?\ No\colon\ 5 = y + 1\ \to\ y = 4 \to\ 3 * 5 = (3 * 4) + 3$

- $4 = 0?\ No \ \rightarrow\ 4 = y + 1\ \rightarrow\ y = 3 \rightarrow\ 3 * 5 = (3 * 4) + 3\ =\ \big((3 * 3) + 3\big) + 3$

  Riapplicando di seguito la regola ottengo:

- $\dots\ =\ ((3 * 2) + 3) + 3) + 3\ =\ (((3 * 1) + 3) + 3) + 3$

  $=\ ((((3 * 0) + 3) + 3) + 3) + 3) + 3 = \mathbf{15}$

Esempio #5. Abbiamo già definito la funzione segno con o senza ricorsione primitiva; possiamo però anche calcolare il segno in modo iterativo, sfruttando i costrutti di base della programmazione. Il segno, infatti, restituisce 0 se l'input è 0, 1 in ogni altro caso. Definiamo quindi una funzione che restituisca sempre 1, qualunque sia l'input:

$$t(x) = 1$$

Allora: $$sg =_{def}\ (0\ ||P_{1,1}^{1});\ exp(t)$$

L'equivalenza di ricorsione primitiva ed esponenziazione.

L'esponenziazione è equivalente alla ricorsione primitiva, proprio come l'iterazione può essere espressa tramite la ricorsione e viceversa.

#1: Esponenziazione con la ricorsione primitiva. Sia data la funzione *f*: la sua esponenziazione è:

$f\colon N^n\ \rightarrow N^n$

$exp(f)\colon N^{n+1}\ \rightarrow N^n$

$exp(f)(x_1,\ \dots.,\ x_n,\ y) = f^y(x_1,\ \dots,\ x_n)$

Definiamo $exp(f)(x_1,\ \dots.,\ x_n,\ y)$ per induzione su y…

$$\exp(f)\ = \begin{cases} exp(f)\,(x_1,\ \dots.,\ x_n, 0) = x_1,\ \dots.,\ x_n \\ exp(f)\,(x_1,\ \dots.,\ x_n, y+1) = f(\exp(x_1,\ \dots.,\ x_n, y)\,) \end{cases}$$

…quindi definiamo le funzioni g e h:

$g\colon N^n \rightarrow N^n \qquad g(x_1, \dots,\ x_n) = x_1, \dots, x_n$

$h\colon N^{n+2} \rightarrow N^n \qquad h\big(x_1, \dots,\ x_n, y, \exp(f)(x_1, \dots, x_n, y)\big) = f(\exp(x_1, \dots, x_n, y))$

…dove $\exp(f)\,(x_1, \dots, x_n, y) = f\big(\exp(f)\,(x_1, \dots, x_n, y)\big)$.

Esempio #6. Sappiamo che la somma si definisce tramite l'esponenziazione così: $+ =_{def} \exp(S)$. Come si definisce, invece, tramite la ricorsione primitiva? Compiamo l'induzione su y:

$$+ = \begin{cases} \exp(S)\,(x, 0) = x \\ \exp(S)\,(x, y+1) = s(\exp(x, y)) \end{cases}$$

Con la nuova definizione, la somma $3 + 2$ diventa:

$$exp(S)(3{,}2) = S(\exp(S)(3{,}1)) = (S(S(\exp(S)(3{,}0))) = S(S\big((3)\big) = 5$$

…ch'è il risultato corretto ottenuto applicando con "pattern matching" la definizione ricorsiva primitiva della somma.

#2: ricorsione primitiva con l'esponenziazione. Date le funzioni g e h, vogliamo implementare $exp(f)(x, y) = t(x, y)$ in termini di g e h:

$$t(x,y) = recprim(g,h) = \begin{cases} t(x,\ 0) = g(\mathrm{x}) \\ t(x,\ y+1) = \mathrm{h}\big(x,\ y\ ,t(x,y)\big) \end{cases}$$

Ora, cosa dobbiamo iterare y-volte per ottenere $t(x,y)$? Ossia: che cosa dobbiamo iterare per definire il passo induttivo? L'idea è quella di reiterare la funzione h. Il primo passo dell'iterazione è **preparare** quanto dobbiamo reiterare:

$$x, y \to x, 0, g(x), y \to \ \ \dots \ \ \to\ t(x,y)$$

…dove:

- x è il primo input;
- 0 è il numero di reiterazioni eseguite;
- $g(x)$ è la funzione eseguita nel caso base;
- y è il numero di iterazioni che mancano.

1° iterazione (che corrisponde al caso base):

$$x,\ 1,\ h\big(x, 0, t(x, 0)\big),\ y-1 = x,\ 1,\ h\big(x, 0, g(x)\big),\ y-1$$

2° iterazione:

$$x\ , 2, h\big(x, 1, t(x, 1)\big), y-2$$

3° iterazione:

$$x, 3, h\big(x, 2, t(x, 2)\big), y-3$$

Il programma che calcola t iterativamente:

- Prende tre input e tiene solo il primo: $P^3_{1,1}$
- Prende il secondo e gli si somma 1: $\left(P^3_{2,2}; S\right)$
- Esegue h

Perciò la funzione che itero è:

$$\mathrm{P}^3_{1,1}\ \hat{}\ \left(\mathrm{P}^3_{2,2}; \mathrm{S}\right)\ \hat{}\ \mathrm{h}$$

Per arrivare a $t(x,y)$ dobbiamo infine inserire uno 0 e valutare g nel mezzo:

$$(\mathrm{P}^2_{1.1}\,||\,0)\ \hat{}\ (\mathrm{P}^2_{1,1}, \mathrm{g})\ \hat{}\ \mathrm{P}^2_{2,2}$$

Perciò ottengo:

$$rec(g,h) = \ (P^2_{1.1}||0)\ \hat{}\ (P^2_{1,1}, g)\ \hat{}\ \ P^2_{2,2};\ exp(P^3_{1,1}\hat{}\ (P^3_{2,2},\ S)\hat{}\ h)$$

Prendiamo un caso semplice: la somma definita per ricorsione primitiva:

$$\begin{cases} \mathrm{x} + 0 = \mathrm{x} \\ \mathrm{x} + (\mathrm{y}+1) = (\mathrm{x}+\mathrm{y}) + 1 \end{cases}$$

In questo caso abbiamo:

$$g = \ \mathrm{P}^1_{1,1}$$

$h = P^3_{3,3}; S$ (proietto sull'ultimo elemento e faccio il successore!)

Con un caso concreto, proviamo ad eseguire $3+2$ usando il programma $REC(g,h)$. Allora:

- Prima parte: preparo l'input per l'esponenziazione:

- Proiettto sulla prima, e parallelamente 0: 3,0
- Proietto sulla prima ed eseguo g (che prende 3 e lo riporta): 3
- Proietto sulla seconda: 2

$3{,}2 \ \rightarrow 3{,}0{,}3{,}2$

- Seconda parte: prendiamo tutti gli argomenti escluso l'ultimo (3,0,3) e applichiamo la funzione tante volte questa funzione quant'è l'ultimo valore (2);
- $\rightarrow 3{,}1{,}4{,}1 \rightarrow 3{,}2{,}5{,}0$
- Terza parte: eseguo h (cioè proietto sul terzo argomento)

  $\rightarrow 5$

Funzione di Ackermann e minimizzazione.

Funzione di Ackermann. È una **funzione *calcolabile e totale*** ma che **non può essere calcolata da nessuno dei costrutti** visti precedentemente – il che dimostra che non è possibile calcolare *ogni tipo di funzione usando costrutti che passano da funzioni totali a funzioni totali*. La *funzione di Ackermann* è formata da tanti esponenziali "uno dentro l'altro" (tanti livelli quant'è la lunghezza degli input: se l'input è lungo 2, ad esempio, avrò 2 esponenziali uno dentro l'altro).

Definiamo induttivamente una famiglia di funzioni *totali*:

$f_0 : N \rightarrow N \quad f_0 = S$ (funzione successore)

...

$f_{n+1} : N \rightarrow N \quad f_{n+1} = ?$

Vogliamo definire f_{n+1} in termini di f_n. Perciò:

- Sdoppiamo tutti gli input: $x \ \rightarrow x, x$
- applichiamo l'esponenziale della funzione precedente.

Il risultato è:

$f_0 = S$ (funzione successore)

...

$f_{n+1} = D_2; exp(f_n)$

Il programma è:

$$0||0||P^1_{1,1} \rightarrow 0{,}0, x \ \rightarrow exp(S||S) \rightarrow x,\ x$$

Ora, definiamo la famiglia di funzioni $g(x) = f_x(x)$. Se costruiamo il grafico di tutte queste funzioni, scopriamo che esse vanno ad infinito ad una velocità estremamente alta, superando per *x* sufficientemente grande, tutte le funzioni calcolabili che conosciamo. Perciò g(x) non è calcolabile: il numero di esponenziali che costruisco è tale da farla crescere in maniera spropositata. Possiamo però calcolarla se aggiungiamo ai nostri costrutti il *while*, che obbliga a **passare dalle funzioni totali alle funzioni parziali** e può essere introdotto iterando una certa funzione f un numero di volte che dipende dal risultato del test, "imprevedibile" in linea di principio:

$$while \ \mathrm{h} \neq 0$$
$$do \ f$$

...dove:

$\mathrm{h}: N^n \rightarrow N$

$f: N^n \rightarrow N^n$

Il numero di volte che eseguiamo f non è prevedibile, perché non possiamo prevedere quando *h* sarà uguale a 0.

Il costrutto di *minimalizzazione*.

Data la funzione $f(x, y)$, la sua definizione tramite minimalizzazione è:

$$\mu_y f(x, y) = \begin{cases} \text{più piccolo valore di y tale che } f(x, y) = 0 \\ f(x, 0), \dots, f(x, y-1) \text{ definito} \end{cases}$$

Ossia $f(x, y)$ definita per minimalizzazione è:

- Minimo valore di *y* per cui $f(x, y) = 0$;
- Tutti i valori di $f(x, y)$ quando $y \ va \ da \ 0 \ a \ y - 1$.

Il ***while* può essere pensato tramite la minimalizzazione**: il *while* consiste nella ripetizione di una funzione tante volte finché non trovo un valore di y che termina l'iterazione (cioè finché non arrivo ad un y tale per cui $f(x, y) = 0$). Il *while* è anche definibile attraverso la *ricorsione primitiva*:

$$x_1, \dots x_n \quad se \ \mathrm{h} = 0 \qquad \text{(caso base)}$$
$$f; \ while \ (\mathrm{h} \neq 0 \ do \ f)(x_1, \dots, x_n) \quad altrimenti$$

Perciò ***la piena potenza della calcolabilità è equivalente alla piena potenza della ricorsione***.

Come giustificare una ricorsione più generale come questa? Che basi matematiche ha e perché funziona? Si possono usare due approcci:

a. **Semantica operazionale**, che dà un significato alle equazioni ricorsive in termine di sistema di calcolo (cioè individuando uno strumento che calcoli il valore della ricorsione).
b. **Semantica denotazionale**, Nata anni '60 per dare significato matematico preciso ai costrutti della programmazione.

Con la minimalizzazione si conclude il capitolo relativo ai costrutti della programmazione: abbiamo cioè a disposizione tutti i costrutti capaci di calcolare completamente tutto ciò ch'è calcolabile.

Macchina di Turing.

La macchina di Turing non solo è importante per motivi storici, ma soprattutto perché **permette di studiare il calcolo a livello *atomico***. È formata da una ***testina di lettura e scrittura*** che guarda **una sola cella** di memoria in cui può essere inserito solo un carattere. In base allo ***stato*** della macchina e del carattere letto, la macchina esegue un'operazione, quindi cambia stato. Permette di rappresentare perfettamente la complessità "a livello fondamentale" di ogni calcolo o programma. Fondamentali, per la macchina di Turing, sono:

Q Insieme finito di **stati** in cui può trovarsi la macchina. q_0 è lo stato iniziale.

A Alfabeto finito (*blank* indica "cella vuota")

$Programma$ Insieme di **quintuple**; la struttura è sempre questa:

$$q\ \ a\ \ b\ \ q'\ D$$

...dove D, S, F indicano "Destra", "Sinistra", "Fermo". Il programma va interpretato così:

$If\ (q\ a)$ se sono nello stato q e leggo a...

$then\ (b\ q')$ ...allora scrivo b al posto di a e passo nello stato q'...

D ...quindi vado a destra.

Esempio. Supponiamo d'avere un nastro unidimensionale suddiviso in celle:.

		1	0	1	0				

...dove la cella rossa rappresenta ciò ch'è letto dalla testina. La macchina si trova in uno stato iniziale q_0. Vogliamo scrivere un programma che, quando legge 0 lo cancella, quando legge 1 lo lascia com'è. Perciò:

- leggo 0, scrivo *blank* e resto nello stesso stato, quindi vado a destra.

$$q_0\ 0\ blank\ \ q_0\ \ D$$

- leggo 1, lascio l'1, resto nello stato corrente e vado a destra.

$$q_0\ 1\ \ 1\ \ q_0\ D$$

	1	0	1	0				

	1		1	0				

	1		1					

In tutti questi passaggi, la macchina rimane sempre nello stato q_0. A volte può essere comodo avere due condizioni, alias "se leggo 1 o 0":

$$q_0 \begin{matrix} 1\ 1 \\ 0\ 0 \end{matrix} q_1\ S$$

...ossia: se sono nello stato q_0 e leggo o 1 o 0, scrivi o 1 o 0, passa nello stato q_1 e mi fermo.

Esempio. Vogliamo scrivere l'algoritmo della somma data una *memoria bidimensionale.*

		5	2	8
	6	8	1	0

La situazione si fa più complessa perché abbiamo bisogno di *memorizzare il riporto*. Se poniamo che $q_0 = 8$, allora dobbiamo:

- Se sono nello stato q_0 e leggo una cifra C, lascio lì la cifra C, passo nello stato di memorizzazione della cifra letta q_c, ricordo che il riporto è 0, mi sposto in giù S:

$$q_0 \;\; C \;\; C \;\; q_C, 0 \;\; S$$

- Nello stato q_c, 0 (è il riporto) leggo la cifra d, lascio d dov'è, quindi passo nell'altro stato q_d, ricordo la cifra C e il riporto 0.

$$q_c, 0 \; d \;\; d \;\; q_d, C, 0 \;\; S$$

- Quando siamo nella casella corretta (cioè in stato $q_d, C, 0$), se leggiamo *blank* allora scriviamo $d + c + 0)\%10$:

$$q_d, c, 0 \;\; blank \;\; (d + c + 0) mod\ 10\ q_t$$

Note: la *macchina di Turing* è strutturata per lavorare in uno spazio dove vige la *geometria euclidea*: la singola computazione si svolge, cioè, in una porzione limitata del piano. Come cambia, però, la macchina se allarghiamo lo spazio fino a comprendere uno spazio non euclidea? Se consideriamo l'universo globalmente, come Funziona la *macchina di Turing*? È in generale, come cambia la nozione di calcolo?

Esercizio #1: applicazione del *Teorema del parametro* nella *Ricorsione primitiva*. Consideriamo le due funzioni *g* e *h* e vediamo il caso più semplice di ricorsione primitiva, come il seguente:

$g: N \rightarrow N$

$h: N^3 \rightarrow N$

$f = Rec(g, h)$

La funzione *f*, cioè, ha un significato matematico ben preciso:

$f : N^2 \rightarrow N$

$f(x, 0) = g(x) \qquad (caso\ base)$

$f(x, y + 1) = h(x, y, f(x)) \qquad (chiamata\ ricorsiva)$

Si può anche dare una **definizione matematica** della ricorsione primitiva, effettuando un ***test sul valore di y***:

$f(x, y) = g(x) \qquad se\ y = 0$

$f(x, y) = h\big(x, y, f(x, y - 1)\big) \qquad se\ y \neq 0$

Vorremmo ora scrivere un programma che, dato in input un programma per *g* e uno per h, restituisce un programma per *f*, "automatizzando" la ricorsione primitiva. Mi viene in soccorso il *teorema del parametro*: definisco una nuova funzione *t* che prende in input un programma per *g* (detto u), un programma per h (detto w) e i due input (x, y):

$t(u, w, x, y)$

…e restituisce:

$\varphi_u(x) \qquad se\ y = 0$ (cioè, se y è uguale a 0, restituisce la funzione calcolata dal programma codificato dal valore *u*.

$$t(u,w,x,y) = \varphi_w(x,y,t(u,w,x,\ y-1)) \quad se\ y \neq 0$$

Ora, $t(u,w,x,y)$ restituisce un numero, non un programma che automatizzi *f*. Sapendo però che $t(u,w,x,\ y-1)$ è calcolabile[2], posso parametrizzare tramite il teorema del parametro: posso cioè trovare una funzione $S: N \rightarrow N\ calcolabile\ totale$ tale che:

$$\varphi_{s(u,w)}(x,y) = t(u,w,x,y)$$

…dove $S(u,w)$ è la funzione calcolata da un programma per *f*. Perciò il *teorema del parametro* permette di ottenere **funzioni calcolabili totali da programmi in programmi**.

Esercizio #2. Consideriamo l'insieme E_n così definito:

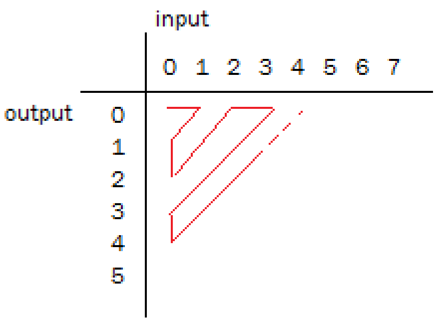
$$E_n = \{y : \exists\, x\ (\varphi_n(x) = y)\}$$

Vogliamo sapere se E_n è decidibile, semidecidibile o non decidibile. Quando abbiamo di fronte esercizi del genere, conviene prima di tutto **interpretare quanto scritto**, dando un valore a *x* e y**:**

- x è l'input;
- y è l'output
- n è il naturale che codifica il programma e la funzione calcolata da quest'ultimo

Perciò E_n è l'insieme degli output generati da tutti i possibili input; matematicamente, E_n è il *codominio di* φ_n. Perciò il problema diventa: *il codominio di una funzione calcolabile è semidecidibile, semidecidibile o non decidibile?* Posso cioè *decidere o semidecidere* se un numero è un output di un programma?

Semidecidibilità. Una prima soluzione è *stampare tutti gli output del programma*, dando in input al programma dei numeri ordinati in modo crescente e per un tempo finito. Usiamo quindi il solito piano:



[2] Basta prendere un programma P_u che decodifica u e un programma P_w che decodifica w; Rispettivamente P_2 prende un input, P_w tre input. Guardiamo y:

- Se $y = 0$: prendo il programma P_u e do in input (x). I casi sono due: P_u può convergere ad x, e quindi $\varphi_u(x)$ è calcolabile, oppure $\varphi_u(x)$ può non convergere ad x.
- Se $y \neq 0$: prendo il programma P_w e do in input $(x,y,t(u,w,x,y-1))$ - dove t(…) è la chiamata ricorsiva. Quando ottengo il risultato delle varie chiamate ricorsive, se P_w converge ottengo $\varphi_w(x,y,t(u,w,x,\ y-1))$.

Perciò, *t* è calcolabile.

Posso quindi testare un qualsiasi incrocio (ad esempio, valuto l'input 3 per un'ora: controllo quindi l'output e lo confronto con il valore arbitrario che stiamo cercando; se l'output e il valore cercato non coincidono, continuo percorrendo il piano, ch'è infinito).

$\rightarrow E_n$ è $semidecidibile$.

È decidibile? Posso decidere se un valore è o non è un output di un programma? Dipende dalla natura del codominio: se ad esempio E_n coincide con K (che è semidecidibile ma non decidibile), allora il codominio non è decidibile. In generale, quindi, E_n non è decidibile.

$\rightarrow E_n$ non è $decidibile$.

Perciò, E_n è *semidecidibile* ma *non è decidibile*.

Esercizio #3. Viene dato il seguente insieme:

$$A = \{n : E_n \text{è } infinito\}$$

Come prima, interpretiamo quanto vediamo:

- n è il naturale che codifica il programma e la funzione calcolata da quest'ultimo[3];

Perciò A è *l'insieme dei programmi tali che il codominio delle funzioni da loro calcolate sono infiniti* – cioè, in altre parole, è l'insieme dei programmi che restituiscono output infiniti. Per esempio, se prendiamo il programma P_1 (con $n = 1$) che calcola la seguente funzione…

$$f(x) = \begin{cases} 1 & se\ x = 0 \\ \uparrow & altrimenti \end{cases}$$

…allora $f(x)$ ***non*** appartiene ad A, dato che il suo codominio è formato solo dal valore 1. Questa funzione, invece,…

$$f(x) = \begin{cases} 1 & se\ x \in K \\ \uparrow & altrimenti \end{cases}$$

...appartiene a A, dato che il suo codominio è formato da tanti 1 quanti sono gli elementi dell'insieme K, che è infinito.

$$codominio(f) = K$$

Perciò possiamo anche riscrivere l'insieme A in questo modo:

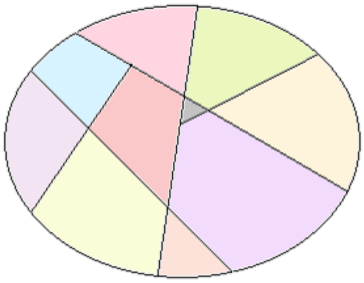
$$A = \{n : E_n \text{è } infinito\} = \{n: codom(\varphi_n) \text{è } infinito\}$$

Ora, possiamo semidecidere o decidere i programmi che hanno output infinito? Con le nostre attuali conoscenze no, ma possiamo ipotizzare che A non sia né decidibile né semidecidibile.

Il metodo della riduzione.

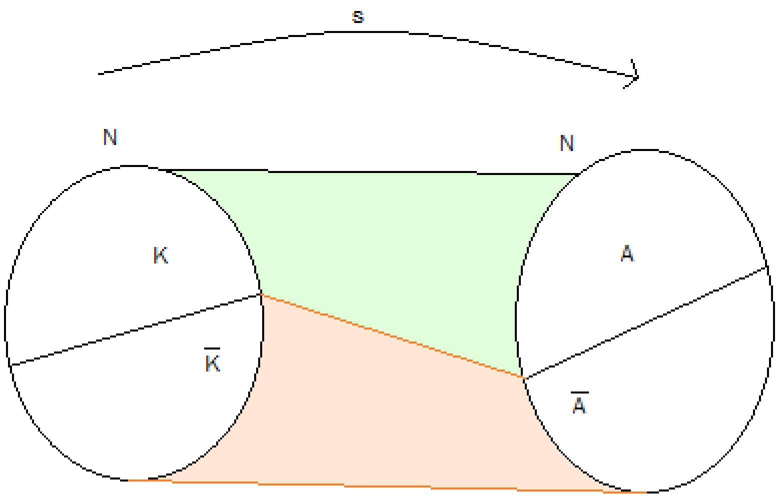
Classi d'equivalenza e programmi. Consideriamo l'insieme dei programmi (operazione equivalente a considerare l'insieme dei *numeri naturali*). Possiamo definire una *relazione d'equivalenza* interessante: ***due programmi sono equivalenti se hanno lo stesso comportamento input-output***, ossia se risolvono lo stesso problema. In questo modo l'insieme dei programmi è *partizionato in infinite classi d'equivalenza*, ciascuna formata dagli *infiniti* programmi che mostrano lo stesso comportamento input-output:

[3] ***n* è *sempre un programma***, perché così i programmi sono codificati da numeri naturali.



Presa una certa classe d'equivalenza, posso *decidere* o *semidecidere* se un dato programma appartiene a questa classe d'equivalenza? **No**: in generale non è possibile né decidere né semidecidere in merito all'appartenenza di un programma (è il problema della correttezza!); al massimo posso farlo per particolari sottoclassi di programmi, ma si tratta di casi specifici.

Tecnica della riduzione. Supponiamo d'avere un certo insieme, A. Come dimostrare velocemente che A non è decidibile? Possiamo usare la *tecnica della riduzione* che, in parole povere, consiste nell'espandere la *non decidibilità* dell'insieme K all'insieme A. Andiamo per punti:



- Supponiamo d'avere una funzione $\boldsymbol{S}$: $\boldsymbol{N} \rightarrow \boldsymbol{N}$ **calcolabile e totale**, cioè tale per cui *per ogni input esiste sempre un output*;
- Prendiamo il **dominio di S**, N, e partizioniamolo in due insiemi, K e $\overline{K}$. Dato che K è semidecidibile, potrò sempre semidecidere se un dato elemento vi appartenga; nel caso non possa decidere, allora tale elemento apparterrà al complementare di K.
- Prendiamo il **codominio di S**, N, e partizioniamolo in A e $\bar{A}$.

S, allora, è la funzione che associa ad ogni elemento del dominio uno e uno solo elemento del codominio. Poniamo, per assurdo, che K sia decidibile: allora, dato un qualsiasi valore $x \in K$, posso sempre decidere se $S(x) \in A$, rendendo di fatto A un insieme decidibile. Sappiamo però che K non è decidibile: di conseguenza A non è decidibile. Riusciamo così ad allargare l'indecidibilità di K ad altri insiemi.

Ci manca, però, capire **come ottenere la funzione *S, calcolabile e totale***: possiamo usare il *teorema del parametro*. Consideriamo la funzione $f(x, y)$, **dove x è un programma tramite cui vogliamo semidecidere A, y è il suo input**. Tale funzione assume due valori in base dell'appartenenza di x a K:

$$f(x,y) = \begin{cases} ? & se\ x\ \in K \\ ? & se\ x\ \in \overline{K} \end{cases}$$

Abbiamo bisogno che f sia calcolabile: scegliamone una che sappiamo essere calcolabile, ad esempio la **funzione identità**.

$$f(x,y) = \begin{cases} y & se\ x\ \in K \\ \uparrow & se\ x\ \in \overline{K} \end{cases}$$

Ora possiamo parametrizzare su *x*: per il *teorema del parametro* possiamo allora dire che esiste una funzione $\varphi_{S(x)}(y)$ tale che:

$$\varphi_{S(x)}(y) = f_x(y) = f(x,y) = \begin{cases} y & se\ x\ \in K \\ \uparrow & se\ x\ \in \overline{K} \end{cases}$$

…da cui otteniamo la funzione calcolabile e totale che stavamo cercando, $S(x)$. Infatti:

- $\boldsymbol{se\ x\ \in K} \quad \rightarrow \quad \varphi_{S(x)}(y) = y \quad \forall y$

  Dato un qualsiasi input $\in K$, $S(x)$ restituisce sempre un output, cioè è calcolabile e totale e restituisce un valore appartenente ad A;

- $\boldsymbol{se\ x\ \in \overline{K}} \quad \rightarrow \quad \varphi_{S(x)}(y) = \uparrow\ \forall y.$

  Dato un qualsiasi input $\in \overline{K}$, S(x) cicla per un tempo indefinito, perciò $S(x)$ non sta in A.

Concludendo, tramite la funzione S(x) riusciamo ad espandere la *non* decidibilità di K ad A. Per sicurezza, dimostriamo che questa "espansione" delle proprietà di K ad A funziona:

- Poniamo per assurdo che A sia decidibile; dato un arbitrario *x*, potrò sempre decidere se $S(x)$ appartiene ad A o a $\bar{A}$. Ma arriviamo ad una contraddizione: se $S(x) \in A$ allora $x \in K$, mentre se $S(x) \in \bar{A}$ allora $x \in \bar{K}$. In questo modo, cioè, K diventa decidibile – e sappiamo che è *solo* semidecidibile.
- Proviamo a lavorare su $\bar{A}$, ponendo per assurdo che sia semidecidibile: esiste allora un algoritmo che, dato un arbitrario *x*, stabilisce se $S(x)$ appartiene ad $\bar{A}$ (mentre non può dire nulla in merito all'appartenenza di $S(x)$ ad $\bar{A}$): se $S(x)$ appartiene ad $\bar{A}$, allora *x* proviene da $\bar{K}$ che diventa semidecidibile. Sappiamo però che $\bar{K}$ non è semidecidibile, quindi nemmeno $\bar{A}$ è semidecidibile.

Perciò, riducendo K ad A, siamo riusciti a spostare le proprietà di K su A: così come K è semidecidibile, così anche A è semidecidibile; così come $\bar{K}$ non è semidecidibile, anche $\bar{A}$ non è semidecidibile. **Ricapitolando**, il *metodo della riduzione* funziona in questo modo: l'insieme A è riducibile all'insieme B se esiste una funzione calcolabile totale S: $N \rightarrow N$ tale che:

$$x\ \in A \quad sse \quad S(x) \in B$$
$$x \in \bar{A} \quad sse \quad S(x) \in \bar{B}$$

Esercizi sulla *tecnica della riduzione.*

Esercizio #1 (*problema della correttezza*). Sia data la seguente funzione:

$$f(x) = 0 \quad \forall x$$

…ossia la funzione che, qualunque sia l'input, restituisce sempre 0. Si tratta di una funzione *calcolabile e totale*, visto che restituisce *sempre* un output. Utilizziamo questa funzione per definire un insieme di programmi:

$$A = \{x : \varphi_x(y) = 0\ \forall y\}$$

…ossia A è l'insieme dei programmi che restituiscono 0 qualunque sia l'input. Ora, A è decidibile, semidecidibile o non decidibile? Ossia: preso un qualsiasi programma, posso decidere, semidecidere o non decidere se appartiene ad A? Una **prima soluzione "grezza"** consiste nel dare a questo programma tutti i possibili input e verificare, per ciascuno, se l'output è 0. Si tratta però di un test utile perché ci fa intuire che probabilmente A è semidecidibile, ma porta via un *tempo infinito*: non è quindi una vera soluzione al nostro problema. Forse, allora, conviene dimostrare che A è non decidibile. Per farlo, dobbiamo ridurre K (che sappiamo essere non decidibile) ad A, trasferendo su quest'ultimo la non decidibilità.

Poniamo d'avere il programma P che prende in input il secondo programma Q e che restituisce "sì, Q calcola la funzione $f(x) = 0\ \forall x$" o "no, Q non calcola la funzione $f(x) = 0\ \forall x$". Allora esiste la funzione $t(y, z)$ calcolata da P, dove y è il naturale che codifica Q, z l'input di Q:

$$t(y,z) = \begin{cases} 0 & se\ y \in K \\ \uparrow & se\ y \in \overline{K} \end{cases}$$

t è una funzione calcolabile, per cui possiamo parametrizzare rispetto a y:

$$\varphi_{S(y)}(z) = t_y(\mathrm{z}) = t(y,z) = \begin{cases} 0 & se\ y \in K \\ \uparrow & se\ y \in \overline{K} \end{cases}$$

I casi sono quindi due:

- $y \in K :\ \ \varphi_{S(y)}(z) = 0\ \forall z$

  …cioè $S(y)$ calcola la funzione costante 0, cioè $S(y) \in A$.
- $y \in \overline{K} : :\ \ \varphi_{S(y)}(z) =\uparrow \forall z$

  …cioè $S(y)$ non calcola la funzione costante 0, perciò $S(y) \in \bar{A}$.

In questo modo, A risulta semidecidibile proprio come l'insieme K: perciò A non è decidibile.

Nota: quest'esercizio, in generale, è equivalente a dimostrare che il problema della correttezza non è decidibile, cioè che **non esiste un programma capace di decidere se un secondo programma restituisce o meno un certo output.**

Esercizio #2. Consideriamo il seguente insieme:

$$I = \{x : \exists y\ (\varphi_x(y) = 5)\}$$

…cioè l'insieme dei programmi itali che esiste un input per cui l'output è 5.

Domanda: I è semidecidibile? Sì – e per provarlo non serve usare la tecnica della riduzione, ma basta provare tutti gli input e verificare se ne esiste almeno uno per cui l'output è 5. Il metodo per farlo è usare il "piano a zig-zag".

Domanda: I è decidibile? Di primo impatto la risposta è *no*: per deciderlo dovrei consultare tutto il "piano a zig-zag", impiegando un tempo indefinito. Per dimostrarlo posso ridurre K a I. Come prima, consideriamo una funzione $t(y, z)$ dove y è il programma che vogliamo analizzare per capire se I è o meno decidibile, z il suo input:

$$t(y,z) = \begin{cases} 5 & se\ y \in K \\ \uparrow & se\ y \in \overline{K} \end{cases}$$

$t(y, z)$ è calcolabile, quindi parametrizziamo su y: esiste una funzione calcolabile totale $\varphi_{S(y)}(z)$...

$$\varphi_{S(y)}(z) = t(y,z) = \begin{cases} 5 & se\ y \in K \\ \uparrow & se\ y \neg\in K \end{cases}$$

I casi sono quindi due:

- $y \in K$: $\varphi_{S(y)}(z) = 5$ $\forall z$ e quindi, in particolare, per un certo z.
- $y \in \overline{K}$: $\varphi_{S(y)}(z) = \uparrow \forall z$

Abbiamo ridotto a K a I, rendendo quest'ultimo non decidibile.

Domanda: avrei potuto ridurre anche il $\overline{K}$ ad I? No, perché riducendo K a I automaticamente si riduce anche $\overline{K}$a $\overline{I}$.

Esempio #3. Consideriamo le due funzioni calcolabili φ_x e φ_y. Posso decidere se x e y hanno lo stesso comportamento input/output (**problema *generale* della correttezza**)? No: se sì, infatti, potrei fissare il comportamento input-output di y e decidere in merito alla correttezza di x, che però come sappiamo non è decidibile.

Esercizio #4. Prendiamo un insieme A semidecidibile: allora A può essere ridotto a K.

$$\varphi_{S(y)}(z) = t(y,z) = \begin{cases} 0 & se\ \ y\ \in A \\ \uparrow & se\ \ y\ \in\ \overline{A} \end{cases}$$

Ma allora:

- Se $y\ \in A \rightarrow \varphi_{S(y)}(z) = 0\ \ \forall z \rightarrow S(x) \in K$
- $Se\ \ y\ \in\ \overline{A} \rightarrow \varphi_{S(y)}(z) = \uparrow\ \ \rightarrow\ \ S(x) \in \overline{K}$

...cioè A è stato ridotto a K.

Non sempre il meccanismo di riducibilità è applicabile; per esempio, prendiamo questi tre insiemi:

$I_1 = \{x: x \text{ è } primo\}$

$I_2 = \{x: \varphi_x \text{ è } totale\}$

$I_3 = \{x: \exists y\ (\varphi_x(y) \geq 5)\}$

Traduciamoli:

$I_1 = l'insieme\ dei\ numeri\ primi;$

$I_2 = l'insieme\ dei\ programmi\ che\ calcolano\ una\ \boldsymbol{funzione\ totale}.$

$I_3 = l'insieme\ dei\ programmi\ tali\ per\ cui\ esiste\ un\ input\ che\ d\grave{a}\ l'output\ \geq 5.$

Domanda #1: *posso ridurre K a I_1*, ossia $K \leq I_1$? $\boldsymbol{No}$ perché I_1 è decidibile mentre K è solo semidecidibile.

Domanda #2: *posso ridurre K a I_2?* Per rispondere affermativamente avremmo bisogno di una funzione *calcolabile e totale* $S(x)$ che, dato un qualsiasi *x*, permetta di stabilire se:

- $S(x) \in I_2 \quad sse \quad x \in K$ ($S(x)$ in questo caso calcola una funzione *totale:* si può sempre decidere se x appartiene a K dato che K è semidecidibile)
- $S(x) \neg\in I_2 \quad sse \quad x \neg\in K$ ($S(x)$, in questo caso, non è totale: K infatti è solo *semidecidibile*)

Per definire $S(x)$ usiamo il *teorema da parametro*. Poniamo d'avere una funzione *f* in due variabili, $f(x,y)$, ***calcolabile***. Posso allora applicare il *teorema del parametro*, parametrizzandola su *y* (senza sapere, almeno a questo punto, quale sia il "contenuto" di *f*):

$$\varphi_{s(x)}(y) = f(x,y) = \ ?$$

Ora, questa funzione può essere definita in questo modo:

$$\varphi_{s(x)}(y) = f(x,y) = \begin{cases} y & se \ \ x \ \in K \\ \uparrow & se \ \ x \neg\in K \ (\text{è } solo \ semidec!) \end{cases}$$

Verifichiamo i due casi:

$$x \ \in K \ \rightarrow \varphi_{s(x)}(y) = y \quad \forall y \quad \rightarrow \quad S(x) \in I_2$$

$$x \neg\in K \ \rightarrow \varphi_{s(x)}(y) = \uparrow \quad \forall y \quad \rightarrow \quad S(x) \neg\in \ I_2$$

…ch'è esattamente quanto volevamo ottenere: se x appartiene a K, $S(x)$ è totale (è la funzione identica, ch'è totale); se x invece non appartiene a K, $S(x)$ è indefinita. Perciò I_2 non è decidibile e, di conseguenza, K è riducibile a I_2.

Domanda #3. Cosa cambia tra I_2 e I_3 ? In entrambi x è indice di una funzione calcolabile. Tuttavia:

- I_2 è l'insieme dei programmi la cui φ_{x} è una funzione totale;
- I_3 è l'insieme dei programmi per cui esiste un input *y* il cui output è ≥ 5.

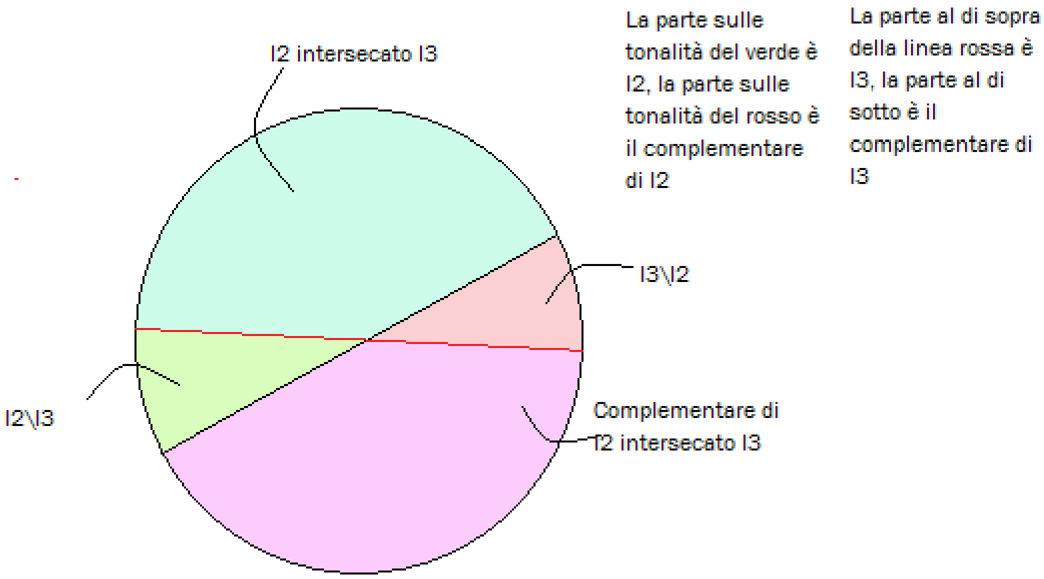
Questo mi suggerisce che I_3 non sia decidibile: dato un programma, cioè, non posso sempre stabilire se esista un input il cui output è ≥ 5 (potrebbe anche ciclare per un tempo indefinito!). Posso ridurre K a I_3? Sì, perché poco cambia rispetto alla domanda precedente: basta infatti prendere la funzione f definita precedentemente:

$$\varphi_{s(x)}(y) = f(x,y) = \begin{cases} y & se \ \ x \ \in K \\ \uparrow & se \ \ x \neg\in K \ (\text{è } solo \ semidec!) \end{cases}$$

Per una *f* così definita, infatti, esiste un input (5) tale per cui $f(x,y)$ restituisce y (5). Perciò, posso ridurre K a I_3 perché sono entrambi semidecidibili, e $S(x)$ riesce a "legare" K a I_3.

Domanda #4. Un programma per la *funzione vuoto* $f_\emptyset$ è totale? No: è un programma che cicla per un tempo indefinito *qualunque sia il suo input*. Perciò è un buon esempio di funzione parziale (ma non totale). Rispetto ai tre insiemi di quest'esempio, ogni programma che calcoli la funzione vuoto appartiene a $\overline{I_2}$. Notare che per ogni programma che calcoli la funzione vuoto non esisterà mai un input tale da produrre un output ≥ 5, quindi ogni programma che calcoli la funzione vuoto appartiene anche a $\overline{I_3}$.

Possiamo dare una rappresentazione di questa situazione: se prendiamo l'insieme dei numeri naturali (cioè l'insieme dei programmi)…



Notare che i programmi della funzione vuota stanno dentro i *due complementari intersecati*; i programmi della funzione identità, invece, stanno *nell'intersezione dei due insiemi* (la funzione identica è totale e dà un output ≥ 5, dato un certo input).

Domanda #5. Quando elenchiamo i numeri naturali, sappiamo che ciascuno codifica un programma. Prendiamo lo 0: sta in I_2? Ossia: il programma P_0 sta in 0? Se P_0, per *ogni* input, restituisce un output ((cioè *calcola una funzione totale*), allora 0 appartiene ad I_2.

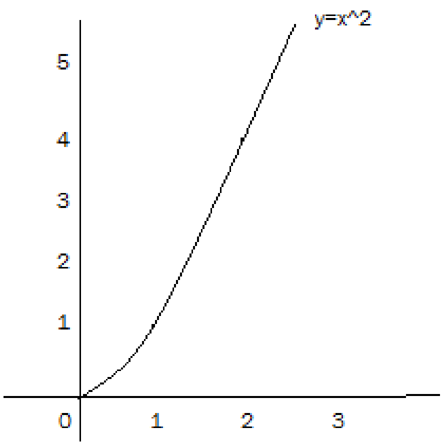
Domanda #6. Stessa questione: 0 sta in I_3? Anche qui dobbiamo considerare P_0 e verificare se, per *un qualche* input, dà un output ≥ 5.

Nota: esistono insiemi di cui non sappiamo dire se siano decidibili, semidecidibili o non decidibili e per cui, di conseguenza, non sappiamo applicare il *metodo della riduzione.*

Esempio #1. Sia *f* una funzione calcolabile parziale tale che $f: N \rightarrow N$ e definiamo l'insieme seguente:

$$A = \{(x, y): y = f(x)\}$$

Traduciamo il significato dell'insieme: è l'insieme dei punti del piano cartesiano che tracciano il grafico di *f*. Cosa possiamo dire di A? Per esempio, prendiamo il caso $y = x^2$ e ne tracciamo il grafico:



Ora, il grafo della funzione "elevamento al quadrato" è *decidibile*: posso sempre stabilire, data una coppia (x, y), se essa appartiene al grafico o non appartiene al grafico. Ma possiamo pensare ad una funzione il cui grafico non è decidibile? Sì, se ad esempio immaginiamo che i valori assumibili da *x* siano solo quelli dell'insieme K:

$$f(x) = \begin{cases} 1 & se\ x\ \in K \\ 0 & altrimenti \end{cases}$$

Data una coppia (x, y), cioè, possiamo solo semideciderla perché K è *solo* semidecidibile. Supponiamo che il grafico di *f* sia decidibile: dato in input *x*, possiamo trasformarlo nella coppia (x,1) e passarlo in input al programma che decide il grafico di *f*. Tale programma può darci due risposte:

- Sì, $(x, 1)$ appartiene a grafico di *f*;
- No, $(x, 1)$ non appartiene al grafico di *f*.

Se vale il primo caso, allora $1 = f(x)$, cioè $x\ \in K$. Se vale il secondo caso 2, allora $f(x) = 0$, cioè $x\ \neg\in K$. In questo modo, K diventa *decidibile*! Sappiamo però che K è *solo* semidecidibile, perciò il grafico di *f* non è decidibile.
Possiamo anche valutare se il grafico di *f* è semidecidibile; supponiamo, per assurdo, che lo sia. Dato in input $x \neg\in K$, posso trasformarlo nella coppia (x,0) e passarlo in input al programma che *semi*decide il grafico su *f.* Questo programma può darci queste risposte:

- Sì, $(x, 0)$ appartiene al grafico di *f*
- indefinito, cioè non da risposta.

In questo modo, il complementare di K diventa semidecidibile – e sappiamo che il complementare di K non ha questa proprietà! Quindi il grafico di *f* non è nemmeno semidecidibile.

Primo *Teorema di Rice*.

Il *primo teorema di Rice* ci permette di *decidere* proprietà relative al *comportamento input-output* dei programmi, indipendentemente da come questi programmi sono scritti: decidere se un certo programma calcola, per un dato input, un output uguale a 3, se l'insieme dei programmi che convergono ad un output è decidibile, etc.
Sia I un insieme che rispetta[4] le funzioni; allora:

---

[4] ***Rispettare le funzioni***. Un sottoinsieme di numeri naturali rispetta le funzioni se, quando x appartiene all'insieme e $\varphi_x = \varphi_y$, allora y appartiene all'insieme:

$$x\ \in I\ e\ \varphi_x = \varphi_y \quad \rightarrow \quad y\ \in I$$

Cioè: un insieme rispetta le funzioni se *tutte* le funzioni che hanno lo stesso *comportamento input-output* appartengono all'insieme; in altre parole, se ho il programma *x* appartenente ad I e un programma *y* ad esso equivalente, I rispetta le funzioni se anche y appartiene ad I. Per esempio, consideriamo quest'insieme:

1. ***I* è decidibile** se e solo se $I = \emptyset$ oppure $I = N$;

   Ciò significa che comunque io prenda un insieme, se esso non coincide con $\emptyset$ o con N (cioè non è banale), allora ***non è decidibile*.**
2. Se l'insieme dei programmi che calcolano $f_\emptyset$ è sottoinsieme di I, allora ***I* non è semidecidibile**; se invece i programmi che calcolano $f_\emptyset$ è sottoinsieme di $\overline{\boldsymbol{I}}$, allora $\overline{\boldsymbol{I}}$ non è semidecidibile.

   $$\{x : \varphi_x = f_\emptyset\} \subseteq I \rightarrow I \; non \text{ è } semidec$$

   Questo significa che se I non è banale, allora bisogna vedere dove si trovano i programmi che calcolano la funzione vuoto: se stanno in I, allora I non è semidecidibile; se invece si trovano in $\bar{I}$, $\bar{I}$ non è semidecidibile.

Osservazione #1. Se I rispetta le funzioni, allora anche $\overline{\boldsymbol{I}}$ rispetta le funzioni.

Osservazione #2. Se un programma appartiene ad I, allora appartengono ad I anche tutti i programmi equivalenti a tale programma.

Osservazione #3. La funzione vuota è definita in questo modo:

$$f_\emptyset : \emptyset \rightarrow \emptyset$$

…cioè non converge mai su un input.

Applicazione-tipo. Consideriamo questo insieme:

$$I = \{x : \varphi_x(5) = 3\}$$

Allora:

- I è diverso dal vuoto (esiste almeno un programma che, su input 5, restituisce 3) e non coincide con N (non *tutti* i programmi, se hanno input 5, convergono a 3), quindi di sicuro I non è decidibile.
- Consideriamo la funzione vuota: sull'input 5 non dà come risultato 3, dato che la funzione vuota non converge mai; perciò la funzione vuota sta nel complementare di I, quindi $\overline{\boldsymbol{I}}$ non è semidecidibile.

$$I = \{x : \varphi_x(5) = 3\}$$

…cioè l'insieme dei programmi che sull'input 5 danno come output 3. Se prendo un programma qualsiasi appartenente ad I (cioè che, dato in input 5, restituisce 3) e un secondo programma che, dato in input 5, restituisce 3, allora anche il secondo programma apparterrà ad I. Notare che, nel caso si specifichi il "tempo" della computazione ("in 5 unità di tempo…"), un insieme non rispetta le funzioni (il tempo è intrinseco al programma: due programmi possono avere stesso input e output, ma calcolarlo in tempi differenti).

Esempio #1. Consideriamo questo insieme:

$$\{x : \exists y \, \forall \, t \; (P_x \downarrow t \quad in \leq y \, passi)$$

È l'insieme dei programmi per cui esiste un tempo *y* tale che comunque scelga l'input, P_x converge all'input in al massimo y-passi. Questo insieme *non* rispetta le funzioni – proprio perché è implicato il tempo.

Dimostrazione: ***I è decidibile se e solo se*** $\boldsymbol{I = \emptyset}$ ***oppure*** $\boldsymbol{I = N}$. La dimostrazione vai in due direzioni:

- Se $I = \emptyset$ $\boldsymbol{o}$ $I = N$, allora I è banalmente decidibile, ossia è facile trovare sempre un programma che decida I.
- Se I è decidibile, allora $I = \emptyset \; oppure \; I = N$. La dimostrazione, in questo caso, si avvale dell'implicazione logica, che funziona in questo modo:

$$A \rightarrow B \quad sse \quad \neg B \rightarrow \neg A$$

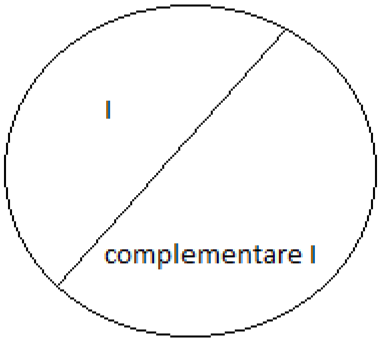
Dalla *legge di DeMorgan* sappiamo però che:

$$\neg(A \; or \; B) \quad = \quad (\neg A \; and \; \neg B).$$

Quindi dobbiamo dimostrare che:

$$I \neq \emptyset \text{ and } I \neq N \;\rightarrow I \; non \; decidibile$$

Per dimostrare che I non è decidibile, ***riduciamo*** $\boldsymbol{K}$ ***ad*** $\boldsymbol{I}$. Se I, per ipotesi, non coincide né con $\emptyset$ né con N, allora possiamo partizionare N in I e in $\bar{I}$:



Dato che $I \neq \emptyset$, avrà almeno un elemento; a sua volta, anche $\bar{I}$ avrà almeno un elemento. Prendiamo allora un $z \in \bar{I} \; tale \; che \; \varphi_z \neq f_\emptyset$ e definiamo la funzione

$$f(x,y) = \begin{cases} \varphi_z(y) & se \; x \in K \\ \uparrow & se \; x \in \bar{K} \end{cases}$$

Notare che se $x \in K, f(x,y) = \; \varphi_z(y)$ perché, per ipotesi, I e $\bar{I}$ rispettano le funzioni. *f(x,y)* è calcolabile, quindi posso parametrizzarla:

$$\varphi_{S(x)}(y) = f(x,y) = \begin{cases} \varphi_z(y) & se \; x \in K \\ \uparrow & se \; x \in \bar{K} \end{cases}$$

…dove S(x) è una funzione calcolabile e totale che "matcha" K a $\bar{I}$. Analizziamo i due casi:

- Se $x \in K$ allora $\varphi_{S(x)}(y) = \; \varphi_z(y) \; \forall y, cioè \; S(x) \in \bar{K}$;
- Se $x \in \bar{K}$ allora $\varphi_{S(x)}(y) = \uparrow \; \forall y, cioè \; S(x) \in K$.

Abbiamo ridotto K a $\bar{I}$, quindi né I né $\bar{I}$ sono decidibili.

Dimostrazione: ***I non è semidecidibile.***

Ho ridotto I a K, ma posso anche ridurre I a $\bar{K}$. Ma dato che $\bar{K}$ complementare non è semidecidibile, allora neanche I è semidecidibile.

Note sui quantificatori, per lavorare su insiemi e loro complementari.

Conviene fare un breve richiamo su come si definire il complementare di un insieme dato. Per esempio, prendiamo il seguente insieme:

$$I = \{x : \varphi_x(5) = 3\}$$

$\bar{I}$ è l'insieme dei programmi che, dato 5 in input, non danno input 3, cioè o convergono su 5 ma danno in output un valore diverso da 3, o non convergono su 5.

$$\bar{I} = \{x : \varphi_x(5) \neq 3\} \to \bar{I} = \{x\colon P_x \uparrow 5 \; oppure \; P_x \downarrow 5 \;\&\; \varphi_x(5) \neq 3]$$

Nel caso vi siano i **quantificatori** la situazione è diversa e valgono queste equivalenze logiche:

$\neg[\forall x\, P(x)] \quad \leftrightarrow \quad \exists x\, \neg P(x)$

$\neg\, \exists\, x\, P(x) \quad \leftrightarrow \quad \forall\, x\, \neg P(x)$

Per esempio:

$$\neg\, \exists y\big(\varphi_x(y) = 3\big) \qquad \leftrightarrow \qquad \forall\, y\; \neg\big(\varphi_x(y) = 3\big)$$

...cioè per ogni input il programma o non convergerà a nessun risultato, oppure convergerà su y non avendo come output 3.

Esempio #1. Consideriamo questo insieme:

$$\{x\colon \;\; \forall y\, \exists\, t : \, (\varphi_x(y) = t)\}$$

...cioè è l'insieme dei programmi tali che per ogni input esiste un output (cioè $\varphi_x(y)$ è una funzione totale, che converge su tutti gli input). La sua negazione è:

$$\neg\big\{x\colon \forall y \exists t \; tale \; \text{che} \; \big(\varphi_x(y) = t\big)\big\} \to \; \exists y\, \forall t\, \neg\big(\varphi_x(y) = t\big)$$

...cioè esiste un input su cui il programma non converge (i quantificatori cambiano, ma il “non” passa).

Esercizio #1. Sia dato questo insieme:

$$K = \{x : P_x \; \downarrow x\}$$

K rispetta le funzioni? Ossia: presi due programmi qualsiasi *x* e *y* equivalenti, entrambi stanno in K (o in $\overline{K}$) oppure no? La risposta è *no*, ma lo dimostreremo solo quando riusciremo a dimostrare che *esiste sempre* un *programma che si autoriconosce* (cioè che termina la computazione se e solo se, dato se stesso in input, sa dire "sì, l'input sono io" o "no, l'input non sono io":

$$P \downarrow Q \quad sse \quad Q \equiv P$$

Scrivere un programma che si autoriconosce non è possibile "a tavolino", in quanto nel "scriverlo" modifichiamo dinamicamente l'input e ciò che gestisce l'input. Dobbiamo quindi usare le tecniche ricorsive - ne parleremo poi. Supponiamo però che esista un programma che si autoriconosce, P. P termina la computazione su se stesso? Sì, perché si autoriconosce:

$$P \downarrow P \;\to\; P \in K$$

Notare che se aggiungo un’istruzione in P che non impatta il suo output, come ad esempio…

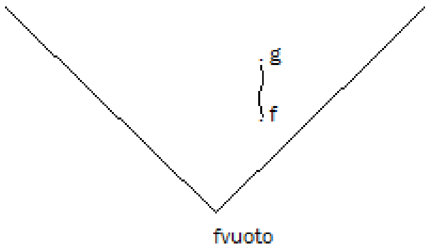
$P;$

$x := x$

…ottenendo P’, ottengo un nuovo programma che ha lo *stesso comportamento input-output di P*, ma se P cerca di “riconoscere” P’ come se stesso, fallisce. Perciò, se P sta in K, P’ non sta in K: quindi **K non rispetta le funzioni**: $P' \uparrow P', P' \downarrow P$

In questo modo P e P’ sono due degli infiniti nomi dati alla stessa funzione, ma P’ non riconosce se stesso in P’.

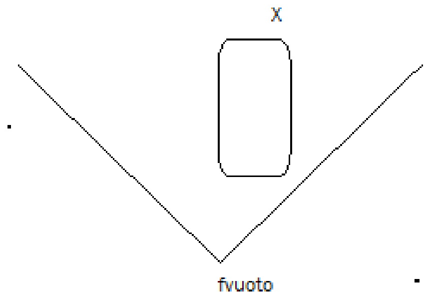
Esempio #2. Consideriamo tutte le funzioni calcolabili; quella *meno definita*[5] è sicuramente la funzione vuoto: $f_\emptyset$. Possiamo poi costruire uno schema in cui posizionare tutte le funzioni, dalla meno definita alla più definita:



Otteniamo così un **ordinamento parziale**, cioè un insieme di funzioni che godono delle proprietà:

- **Riflessiva**: una funzione è definita al massimo quanto se stessa.
- **Antisimmetrica**: Se una funzione è meno definita di *g*, allora *g* non è meno definita di *f*.
- **Transitiva**: *se* $f \leq g \leq h$ *allora* $f \leq h$.

Scegliamo un sottoinsieme X di queste funzioni calcolabili “ordinate”, diverso da vuoto e diverso dall’insieme di *tutte* le funzioni calcolabili esistenti:



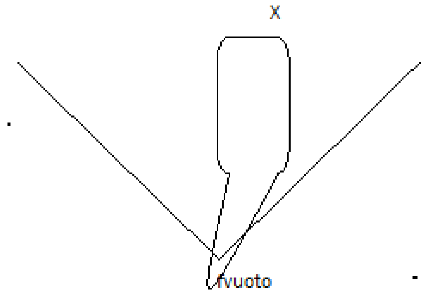
$\{y : \varphi_y \in X\}$

Allora, per il teorema di Rice, l’insieme dei programmi appartenenti ad X ***non è decidibile***. Se poi anche la funzione vuota è compresa in X…

[5] Prendiamo due funzioni, *f* e *g*. Allora g è più definita di *f* se:

- *g* converge su tutti gli input su cui converge *f*, dando lo stesso output;
- *g* converge anche su ulteriori input, dando altrettanti output.

Si può anche dire che *il comportamento input-output di g contiene il comportamento input-output di f*, oppure che *il grafico di g contiene il grafico di f.*



...l'insieme X diventa anche ***non semidecidibile***. Perciò, comunque vada, secondo il *Teorema di Rice* **il problema della correttezza non è decidibile**: dato che X contiene "tutto", dalla funzione che non converge mai alla funzione che converge su *ogni* input, allora se X non è né decidibile né semidecidibile, il problema della correttezza in generale non è decidibile né semidecidibile.

Domanda: *e nel caso X contenesse solo una funzione calcolabile?* Fissiamo *f* calcolabile: avendo solamente una funzione, posso comunque avere *infiniti* programmi che calcolano *f*. Perciò quanto detto vale ancora: il problema della correttezza *non è decidibile*.

Esercizio #3. Sia dato quest'insieme:

$$I = \{x : \forall y(P_y \downarrow x \quad in\ tempo \geq y + 1)\}$$

Determinare:

- Se I è semidecidibile, semidecidibile, ...;
- Se I rispetta le funzioni oppure no.

Come al solito, cerchiamo di capire il significato di I: è l'insieme degli input per cui, comunque scelga un programma, quel programma converge sull'input in un tempo $\geq\ y + 1$. Ora, questo insieme ha almeno un elemento? Ipotizziamo che contenga $x = 0$: qualsiasi programma converge su 0 in un tempo $\geq y + 1$? No: visto che possiamo scegliere *qualsiasi* programma, allora possiamo scegliere anche il programma che calcola la funzione vuoto, che diverge sempre (e cioè diverge anche per $x = 0$). Perciò, in generale, l'insieme è vuoto. Dato quindi che I coincide con il vuoto, per il *Teorema di Rice* l'insieme è decidibile (quindi non è semidecidibile).

Esercizio #4. Consideriamo l'insieme die programmi che convergono sullo 0:

$$I = \{x : \ P_x \downarrow 0\}$$

Determinare se I è decidibile, semidecidibile, ... e se I rispetta o meno le funzioni.

Per prima cosa, I rispetta le funzioni? Per capirlo analizziamo la **proprietà di convergenza**: è una proprietà **della *funzione* calcolata dal programma**, **non del programma in sé**. Quindi comunque io prenda due programmi P_x e P_y tali che $\varphi_x = \varphi_y$, sicuramente sia x che y apparterranno ad I – da cui I rispetta le funzioni:

$$x \in I \quad e \quad \varphi_x = \varphi_y \quad \rightarrow \quad se\ P_x \downarrow 0 \quad allora \quad \mathrm{P_y} \downarrow 0$$

Esercizio #5. Prendiamo il programma P_x costituito da 1000 caratteri e l'insieme I formato da tutti i programmi costituiti da 1000 caratteri. Questa, a differenza di prima, è una **proprietà del programma**, *non* della funzione calcolata dal programma. Se prendo un programma P_y che calcola la stessa funzione di P_x, non è detto che anche P_y sia di 1000 caratteri: P_y, cioè, non sta in I.

Esercizio #6. Sia I l'insieme dei programmi che, sugli input di lunghezza ≤ 1000, occupano al più 5GB di memoria. Anche questa è una *proprietà dei programmi*, non della funzione: per calcolare una certa funzione posso avere programmi estremamente grandi o estremamente piccoli, ma la cosa non ha conseguenze sulla funzione calcolata. Altre proprietà, ad esempio, sono:

- **Delle funzioni**: ovunque si parli di φ_x; *l'insieme dei programmi che , su input 7, danno output 3* (l'output è calcolato dalla funzione); *l'insieme dei programmi che terminano la computazione su un numero finito di input* (il dominio della funzione, cioè, è finito);
- **Dei programmi**:

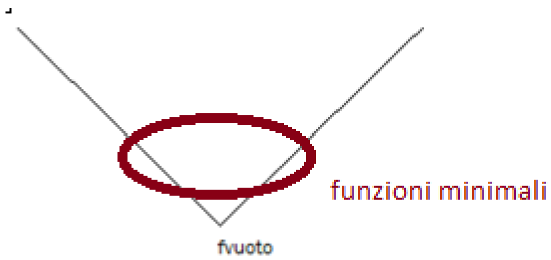
Esercizio #7. L'insieme dei programmi identificati dai numeri pari rispetta le funzioni? No, perché l'insieme dei numeri pari è diverso dal vuoto, diverso da N e decidibile: se rispettasse le funzioni, per il teorema di Rice sarebbe *indecidibile*.

Esercizio #8. L'insieme dei programmi identificati dai numeri primi rispetta le funzioni? No, perché l'insieme dei numeri primi è diverso dal vuoto, è diverso da N ed è decidibile; se rispettasse le funzioni, per il teorema di Rice sarebbe *indecidibile*.

Quindi: ogni insieme che rispetta le funzioni – ad esempio, $I = \{x \,|P_x \downarrow 0)$ - può essere riscritto così:

$$I = \{x\,|P_x \, frecciagiu\; 0) = \{y : \varphi_y \in \mathrm{X}\}$$

…dove $\mathrm{X} = \{\mathrm{f}: \mathrm{f} \downarrow 0\}$, cioè X è l'insieme che terminano la computazione su 0. **X è semidecidibile** se è fatto come in questa figura:



…cioè contiene solamente *funzioni minimali* (per esempio, che dà un output per un solo specifico input: per esempio, una funzione che sullo 0 restituisce 0, e basta). Il motivo per cui X diventa semidecidibile è che, essendo X formato da funzioni che danno *un solo* output preciso, possiamo semidecidere una qualsiasi funzione (ossia vedere se quest'ultima, dopo un tot di tempo, dà in output l'output delle funzioni di X). Perciò, ricapitolando: ***il problema della correttezza è, in generale, indecidibile; se però la proprietà che definisce l'insieme può essere controllata in un tempo finito, allora il problema della correttezza è semidecidibile***.

Esercizio #9. Consideriamo il seguente insieme: $I = \{x : \varphi_x = Id\}$, dove *Id* indica la funzione identità. Traducendo, I è l'insieme dei programmi che restituiscono l'input (se l'input è 1000, il programma restituisce 1000). I rispetta le funzioni, non è decidibile e non è nemmeno semidecidibile per il *teorema di*

Rice, punto 3. D'altronde, per scoprire se un qualsiasi programma sta in *I*, dovrei fare una serie di controlli infiniti (se su 0 mi da 0, se su 1 mi da 1, etc); il controllo, anche a livello intuitivo, richiede cioè un tempo infinito.

Secondo *Teorema di Rice*.

Il *secondo teorema di Rice* permette di ricavare informazioni sulla *non semidecidibilità* di un insieme.

Sia I un insieme che rispetta le funzioni. Allora se sono date le funzioni f e g tali che:

- $f < g$, cioè il grafico di f è cioè contenuto nel grafico di g, o anche g estende f.
- $\{x : \varphi_x = f\} \subseteq I$ cioè l'insieme dei programmi che calcolano f è sottoinsieme di I
- $\{x : \varphi_x = g\} \subseteq \bar{I}$ cioè l'insieme dei programmi che calcolano g non sono sottoinsieme di I.

Allora $\boldsymbol{I}$ **non è semidecidibile**.

Dimostrazione. Per dimostrare che I non è semidecidibile riduciamo $\bar{K}$ a I. Definiamo la funzione $h(x,y)$ calcolabile:

$$h(x,y) = \begin{cases} g(y) & x \in K \quad o \quad y \in dom(f) \\ \uparrow & altrimenti \end{cases}$$

Dato che $h(x,y)$ presenta condizioni alternative, prendiamo tre programmi P_f, P_g e P_K, tali che:

P_f calcola f. Permette cioè di decidere se il proprio input appartenga a $dom(f)$.

P_g calcola g.

P_k semidecide K.

Lanciamo in parallelo $P_f(y)$ e $P_K(x)$; abbiamo tre possibili soluzioni:

- Termina per primo $P_f(y)$: allora $y \in dom(f)$, per cui $h(x,y) = g(y) = f(y)$ dato che $f < g$.
- Termina per primo $P_K(x)$: allora $x \in K$.
    - Facciamo partire $P_g(y)$: se quest'ultimo termina allora $h(x,y) = g(y)$, altrimenti $h(x,y) = \uparrow$.
- Se non terminano entrambi, allora $h(x,y) = \uparrow$.

Dato che h è una funzione calcolabile, applichiamo il ***teorema del parametro***: esiste cioè la funzione calcolabile e totale $S: N \rightarrow N$ tale che:

$$\varphi_{S(x)}(y) = h(x,y) = \begin{cases} g(y) & x \in K \quad o \quad y \in dom(f) \\ \uparrow & altrimenti \end{cases}$$

Ma allora:

- Se $x \in \bar{K}$ allora $\varphi_{S(x)}(y) = g(y)$ se $y \in dom(f)$ e $\varphi_{S(x)}(y) = \uparrow$ se $y \notin dom(f)$. Ma allora $\varphi_{S(x)}(y) = f(y)$ $\forall\, y$, da cui $S(x) \in I$.
- Se $x \in K$ allora $\varphi_{S(x)}(y) = g(y)$ $\forall\, y \in dom(f)$, da cui $S(x) \in \bar{I}$.

Ho ridotto $\bar{K}$ a I, quindi I non è semidecidibile.

Terzo *Teorema di Rice*

Sia I un insieme che rispetta le funzioni. Se esistono:

- Una funzione calcolabile f di dominio infinito tale che $\{x : \varphi_x = f\} \subseteq I$.
- Tutte le approssimazioni di *f* di dominio finito, θ, tali che $\{x : \varphi_x = \theta\} \subseteq \overline{I}$.

Allora I non è semidecidibile.

Secondo teorema di ricorsione.

Sia f una funzione calcolabile totale. Allora esiste sempre n tale che $\varphi_n = \varphi_{h(n)}$. Ciò significa che comunque si scelga un programma che termina la computazione su ogni input, esiste almeno 1 input tale per cui $\varphi_n = \varphi_{h(n)}$.

Applicazione: programmi che si autoriconoscono. Il secondo teorema di ricorsione viene usato per costruire programmi che sia autoriconoscono. Consideriamo la funzione *f* calcolabile:

$$f(x,y) = \begin{cases} 1 & x = y \\ \uparrow & altrimenti \end{cases}$$

Applico il *teorema del parametro*: esiste una funzione $S: N \to N$ calcolabile e totale tale che:

$$\varphi_{S(x)}(y) = f(x,y) = \begin{cases} 1 & x = y \\ \uparrow & altrimenti \end{cases}$$

$S(x)$ converge solamente su x. Possiamo allora applicare il *Secondo teorema di ricorsione:* esiste n tale che $\varphi_n = \varphi_{S(n)}$.

$$\varphi_{S(n)} = f(n,y) = \begin{cases} 1 & n = y \\ \uparrow & altrimenti \end{cases}$$

Ma allora $\varphi_n(n) = \varphi_{S(n)}(n) = f(n,n) = 1$ che permette di avere un programma che si riconosce.

Dimostrazione del Teorema 1 di Rice attraverso il *Secondo teorema di Ricorsione*. Sia I un insieme che rispetta le funzioni: il primo teorema di Rice dice che se $I \neq \emptyset, N$ I non è decidibile.

Supponiamo per assurdo che I sia decidibile: allora esiste una funzione calcolabile totale che decide *I:*

$$f(x) = \begin{cases} 1 & x \in I \\ 0 & x \notin I \end{cases}$$

Prendiamo due valori: $c_0 \in I$, $c_1 \in \overline{I}$. Posso allora modificare $f(x)$ in questo modo:

$$f(x) = \begin{cases} c_1 & x \in I \\ c_0 & x \notin I \end{cases}$$

Applico il *Secondo teorema di ricorsione*: esiste un n tale per cui $\varphi_n = \varphi_{F(n)}$. Abbiamo due casi:

- $n \in I \to \varphi_n = \varphi_{f(n)} = \varphi_{c_1}$. In questo modo c_1 dovrebbe appartenere a I, cosa falsa perché I rispetta le funzioni.
- $n \notin I \to \varphi_n = \varphi_{f(n)} = \varphi_{c_0}$. In questo modo c_0 non dovrebbe appartenere a I, cosa falsa perché I rispetta le funzioni.

Siamo quindi arrivati all'assurdo e il teorema è dimostrato.