di Andrea de Prisco

Terza puntata

Teoria della computabilità:

Formalismi e funzioni totali

Terzo «Appuntamento» con la teoria dell'informatica. Questo mese è la volta di un teorema della calcolabilità che riguarda i formalismi che calcolano solo algoritmi che «terminano». Nel dimostrarlo avremo modo di parlarvi ancora un po' di questo affascinante mondo, mostrandovi un tipico approccio alla risoluzione di problemi di questo tipo.
In appendice (riquadro) per concludere questo piccolo viaggio mostreremo alcuni risultati raggiunti nell'ambito della computabilità, taluni addirittura di sapore vagamente magico.

Breve riassunto

Prima di procedere nel nostro viaggio dentro le basi teoriche dell'informatica, è d'uopo fare il punto della situazione giusto per ricordare a tutti i lettori ciò che stiamo facendo da tre mesi a questa parte.

Anche se in queste pagine non potremo affrontare troppo nei minimi dettagli il problema (chi è maggiormente interessato può iscriversi al corso di laurea in Informatica, ndr), per studiare adeguatamente la teoria della computabilità ovvero riguardo ciò che è calcolabile o non calcolabile con una macchina, per prima cosa è necessario stabilire con sufficiente esattezza cosa intendiamo per algoritmo e cosa per agente di calcolo, l'oggetto che dovrà eseguire l'algoritmo. Nelle prime due puntate sono stati esposti i requisiti necessari alla definizione di qualsiasi algoritmo e come agente di calcolo «minimo» abbiamo parlato dalla Macchina di Turing ideata nel 1939, quando i calcolatori, pur sentendone la necessità, non si sapeva ancora come costruirli.

Macchina di Turing a parte, la quale anche se nessuno è ancora riuscito a dimostrarlo, nell'ambito della calcolabilità non probabilistica è universalmente riconosciuta come l'agente di calcolo più potente (a parimerito con altri formalismi, naturalmente), dicevamo macchina di Turing a parte, abbiamo mostrato come nessun calcolatore sia in grado di calcolare tutte le funzioni possibili e immaginabili. Non per mancanza di tempo o di memoria necessaria per il calcolo ma proprio perché si dimostra che alcune funzioni non possono essere computate... e basta!

Di memoria e di tempo, infatti, secondo i requisiti necessari per la definizione di algoritmi non abbiamo limiti teorici: una funzione è considerata calcolabile anche se necessita di qualche miliardo di anni di computazione e un numero pari o maggiore di celle di memoria per mantenere risultati intermedi al calcolo. La teoria è teoria... e ricordo che gli informatici sono molto più matematici che ingegneri (complimento o «scomplimento» che sia dipende come noto da punti di vista).

Oltre a ciò, aggiungiamo, una computazione può anche richiedere un numero infinito di passi: sono ammesse, in altre parole, esecuzioni che non terminano mai. Sembrerebbe che dalla teoria stiamo passando all'assurdo, ma non è così. Si dimostra, ed è quello che faremo questo mese, che formalismi che definiscono solo funzioni totali, ovvero funzioni che qualsiasi input gli passiamo siamo certi che prima o poi perverremo a un risultato, non riescono ad esprimere la stessa potenzialità di formalismi che definiscono funzioni anche non sempre definite. Ovvero, ed è qui che sembra proprio toccare il fondo, se un formalismo definisce solo funzioni totali non potrà definirle tutte. E non ci stiamo riferendo a funzioni parziali: quel «non potrà definirle tutte» della frase precedente si riferisce sempre alle sole funzioni totali. Bah!

Dell'altro

Giusto per essere sicuri di aver esposto correttamente il problema, poniamoci da una diversa ottica e reim-

postiamolo. Come detto nella prima puntata dedicata alla teoria della computabilità lo studio della calcolabilità di generici algoritmi, per semplicità, viene ricondotto allo studio delle sole funzioni dai naturali ai naturali. Ovvero preso un algoritmo che prende i più disparati input e restituisce output qualsiasi passiamo ad una corrispodente funzione da un naturale a un naturale. Ciò è possibile grazie a due passaggi; il primo da input e output generici (stringhe, record, booleani, file, interi, reali, complessi ecc.) a insiemi di naturali, il secondo da insiemi di naturali a un unico numerone che li indentifica univocamente. A questo punto, l'esecuzione di un generico algoritmo viene ricondotta alla valutazione di una determinata funzione dai naturali ai naturali, dopo avere applicato la trasformazione di codifica degli input e col risultato così ottenuto decodificandolo nei normali output del problema di partenza.

Stabilito che studiare gli algoritmi in genere equivale a studiare le funzioni dai naturali ai naturali, ci proponiamo di dimostrare che prendendo un formalismo (agente di calcolo più specifiche per definire gli algoritmi) in grado di calcolare solo funzioni totali, che restituiscono un risultato qualsiasi input gli passiamo, sicuramente non riuscirà a calcolarle tutte: ci sarà sempre un numero di funzioni che pur essendo totali come le altre non sarà calcolabile dal formalismo in questione.

La dimostrazione , assai semplice, di questo fatto parte dalla Goedelizzazione di tutti gli algoritmo scritti nel formalismo che stiamo analizzando. Di Goedelizzazione ne abbiamo già parlato sempre nella prima puntata, ma riconsco che sarebbe bene ritornare un attimo sull'argomento. Goedelizzare vuol dire ordinare, mettere in corrispondenza biunivoca tutti gli algoritmi esprimibili con un dato formalismo con i numeri naturali: sapere qual'è il primo, il centotrentesimo, il millesimo e così via. Analogamente, preso un algoritmo, siamo in grado di conoscere quale posizione occupa nella nostra ipotetica «tabella degli algoritmi». Fare questo è banale: si prendono prima di tutti gli algoritmi (se esistono) composti da un solo simbolo (ricordiamo per scrivere un algoritmo in ogni caso è necessario servirsi di simboli, siano questi zeri e uno, caratteri o parole chiave del linguaggio) e nell'ambito di questi eseguo un ordinamento alfabetico, poi passo a quelli composti da due simboli e faccio altrettanto e così via. Conoscere qual è il 130-esimo algoritmo del mio formalismo a questo punto è molto semplice: inizio la mia costruzione e proseguo sino alla 130-esima posizione. Allo stesso modo, per conoscere quale posizione occupa un dato algoritmo, basta catalogare nel modo sopra esposto i vari algoritmi del nostro formalismo fino al turno dell'algoritmo in questione. È vero, anche questa sembra arte dei pazzi: non fa niente, l'importante è che esista un procedimento effettivo per farlo, il resto (non ci stancheremo di ripeterlo) in qualsiasi disciplina teorica ha poco conto.

Dimostrazione

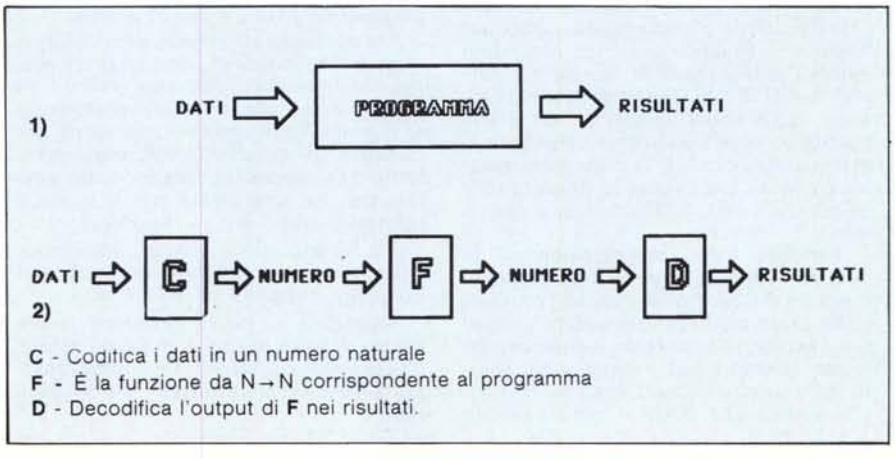
Posto che sia chiaro (!) tutto ciò che abbiamo detto finora, la dimostrazione del fatto (ripetiamo) che qualsiasi formalismo prendiamo capace di calcolare solo funzioni totali non sarà in grado di calcolarle tutte rasenta la banalità. Si fa per dire. Ricordiamo inoltre che per funzione totale o se preferite algoritmo sempre terminante si intende una funzione che preso qualsiasi input si restituisce sempre un risultato. Mentre sappiamo bene che di solito l'immettere dati a caso in un generico programma, può anche portare alla non terminazione del programma in questione. Ad esempio, un programma che calcola (senza alcun controllo sugli input, chi l'ha detto che sono obbligatori?) tutti i numeri primi che finiscono per la cifra passata come input. Provate a passargli 4: lo stupido programma (pur sempre programma, ndr) non restituirà mai un risultato, pur continuando a calcolare infinitamente.

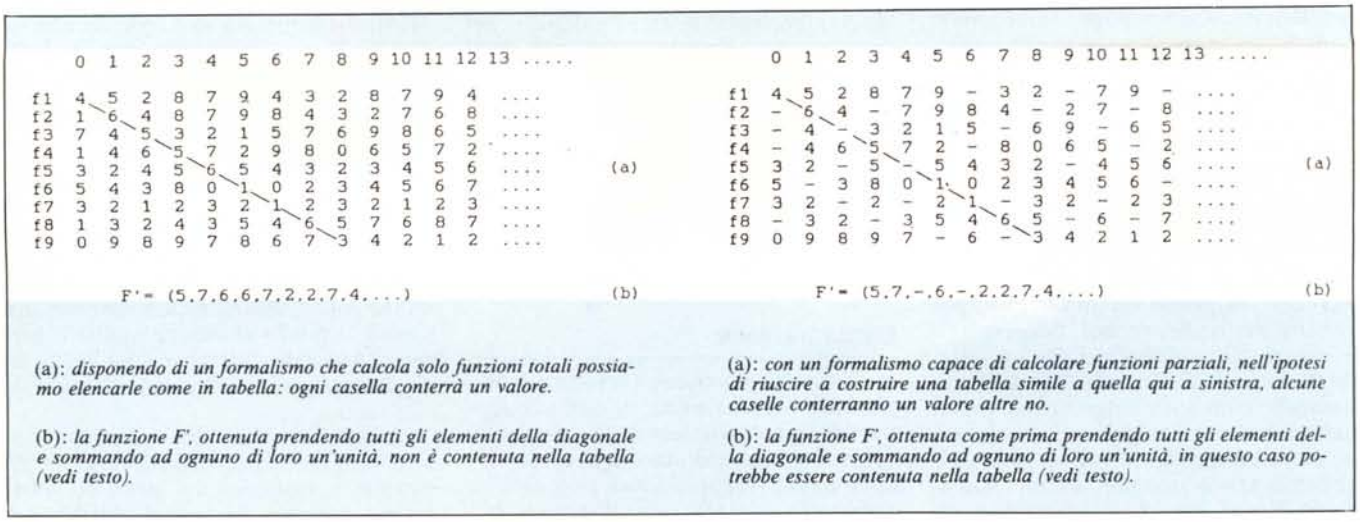
Riassumendo, ponendoci nell'ottica di un formalismo (chiamatelo pure linguaggio di programmazione più calcolatore capace di eseguirlo) che calcola solo algoritmi che terminano sempre e ci accorgeremo di aver fatto un buco nell'acqua: prima o poi ci troveremo nella situazione di non sapere più come esprimere un problema che, giureremmo, come gli altri ad ogni input restituisce un preciso output.

Allora, dal momento che abbiamo capito cos'è la Goedelizzazione prendiamo il nostro formalismo, i programmi esprimibili con questo e mettiamoli ipoteticamente in ordine. Fisicamente non è possibile essendo in ogni caso in numero infinito, ma avendo fornito un procedimento effettivo per farlo possiamo usare questo risultato per dimostrare il nostro teorema. Fatto? Bene, il passo successivo consiste nel costruire una tabella le cui righe sono etichettate dai nostri programmi scritti nel formalismo in questione (in seguito alla Goedelizzazione siamo in grado di sapere qual è il primo, il secondo, il terzo ecc.) e le colonne semplicemente dai numeri naturali 0, 1, 2, 3 ...

A questo punto, dato che ogni nostro programma, qualsiasi naturale gli passiamo restituirà un risultato, possiamo riempire la nostra tabella coi valori restituiti da ogni programma per ogni input. Per essere più chiari, nella casella (4,12) immetteremo il risultato ottenuto passando al programma 4 il valore 12. Nella casella (3424,1231232) il valore restituito dal programma numero 3424 una volta passatogli il valore 1231232 e così via per tutte le caselle. Ciò è possibile ed effettivo proprio per il fatto che per ipotesi tutti i programmi terminano (forniscono un risultato) qualunque input gli passiamo. La tabella così ottenuta, di dimensioni infinite, è assai simile a quella usata due mesi orsono per dimostrare che i naturali non possono essere messi in corrispondenza biunivoca con i suoi sottoinsiemi. Anzi, a dire il vero questo schema di dimostrazione è molto diffuso in tali aree ed ha anche un nome: dimostrazione diagonale. Si dimostra cioè che una tabella siffatta non potrà mai essere completa. Vediamo come.

La nostra tabella dicevamo, di dimensioni infinite, è una raccolta di righe dove ogni riga è un preciso algoritmo corredato da tutti i suoi risultati, un risultato per ogni possibile naturale in ingresso. Ogni riga sarà ovviamente di lunghezza infinita. Per dimostrare che la nostra tabella non è completa,





(a): *disponendo di un formalismo che calcola solo funzioni totali possiamo elencarle come in tabella: ogni casella conterrà un valore.*

(b): *la funzione F', ottenuta prendendo tutti gli elementi della diagonale e sommando ad ognuno di loro un'unità, non è contenuta nella tabella (vedi testo).*

(a): *con un formalismo capace di calcolare funzioni parziali, nell'ipotesi di riuscire a costruire una tabella simile a quella qui a sinistra, alcune caselle conterranno un valore altre no.*

(b): *la funzione F', ottenuta come prima prendendo tutti gli elementi della diagonale e sommando ad ognuno di loro un'unità, in questo caso potrebbe essere contenuta nella tabella (vedi testo).*

basta trovare una riga non contenuta in essa e, matematicamente parlando, trovata una, trovate infinite. Prendiamo come al solito tutti gli elementi della diagonale: sono tutti i numeri, sono in numero infinito ottenendo così una nuova riga, simile alle altre presenti in tabella. Sommiamo ad ogni elemento di questa nuova riga un'unità ottenendone un'antra. Signori e signore la riga così ottenuta, pur essendo qualitativamente simile alle altre (rappresenta un algoritmo, una funzione totale: sappiamo cioè quanto vale per input 0, 1, 2, 3 ...) non è presente in tabella: dovunque proviamo a confrontarla con un'altra sicuramente arrivati alla diagonale non potrà coincidere a causa del tipo di costruzione della riga stessa. Fuori uno (funzione totale come le altre ma non contenuta nella tabella costruita con gli algoritmi del formalismo dal quale siamo partiti). Ma possiamo sommare invece che 1, 2 oppure moltiplicare per qualsiasi numero dividere (intero)... insomma abbiamo infiniti modi per trovare infinite righe (funzioni totali) non calcolate dal formalismo di cui sopra. Contenti?

E allora?

Abbiamo dimostrato che un formalismo capace di calcolare solo algoritmi definiti su un qualsiasi input non è in grado di calcolarli tutti. Quindi se aspiriamo all'equipotenza con le macchine di Turing siamo costretti ad ammettere computazioni infinite come indicato nei requisiti di ogni definizione di algoritmo. Tutto qui.

La classica obiezione a questo punto riguarda le differenze coi formalismi che calcolano funzioni parziali. Per funzione parziale si intende, lo ricordiamo, funzioni non definite per tutti gli input, ovvero per alcuni valori in ingresso non forniscono un risultato. Casi particolari di funzioni parziali sono le funzioni ovunque indefinite (tipo 10 goto 10 del basic) cioè che non forniscono alcun risultato qualsiasi input gli proponiamo e le funzioni totali sempre definite. Si potrebbe obiettare che il procedimento visto sopra per i formalismi che calcolano solo funzioni totali può essere applicato anche alle macchine di Turing così come a qualsiasi altro formalismo equipotente a questo per ottenere lo stesso risultato: trovare una funzione non contenuta «in tabella». Spiacente ma non funzionerebbe.

Il teorema del prestigiatore

Avete un prestigiatore? Immaginate che questo abbia in mano un mazzo di carte nuovo di zecca ovvero ancora sigillato. Davanti a voi lo scarta e dà una abbondante mischiata. Come sapete le carte appena acquistate sono in ordine, dall'Asso di Cuori sino al Re di Picche. Al termine del rimescolamento, eseguito in tutta la regolarità, il prestigiatore mostra ai presenti che le carte sono tutt'altro che in disordine: Asso di Cuori, Due di Cuori, Tre di Cuori...

Come per magia... o informatica che dir si voglia.

Non stiamo dando i numeri: era solo per introdurvi il Teorema di Kleene che fa praticamente lo stesso con i programmi. Matematicamente parlando questo teorema afferma che, presa la Goedelizzazione delle Macchine di Turing, ovvero associato un indice numerico ad ogni MDT possibile e immaginabile, qualsiasi funzione calcolabile totale dai naturali ai naturali prendiamo e la usiamo per passare da MDT a MDT (applicando la funzione di cui sopra all'indice della prima otteniamo un altro indice e conseguentemente una nuova MDT) esiste sicuramente una MDT che calcola la stessa funzione calcolata dalla MDT ottenuta con questa trasformazione.

Parlando meno matematicamente se noi prendiamo un linguaggio di programmazione e i programmi scritti con esso, stabilito un modo per mescolare linee di programma, se applichiamo il medesimo mescolamento a tutti i programmi, Kleene afferma che tra tutti esiste almeno un programma che dopo il mescolamento continua a funzionare esattamente come prima anche se è effettivamente un altro programma. Come per magia...

Anche se sembra proprio che questo teorema per quanto simpatico non serve a nulla, è usato da un altro insigne matematico, Rice, per dimostrare il suo.

Rice dice che presa una qualsiasi proprietà non banale delle funzioni calcolabili (es. risultati sempre pari, sempre costanti, tutti diversi tra di loro, ecc.) non è possibile stabilire se una generica funzione ha o non ha la proprietà di cui sopra. Attenzione, generica non particolare. In altre parole si può ad esempio dire che una particolare funzione gode di quella proprietà ma, in pratica, non è possibile scrivere un programma che accetti in ingresso la proprietà e qualsiasi programma, e risponda sì o no a seconda che il programma goda o meno di questa.

Un'applicazione? Presto detto: Rice ci vieta di fare ipotesi di correttezza sui programmi. Nessuno potrà mai scrivere un programmone che ci dica con sicurezza se il programma che abbiamo scritto noi calcola effettivamente la funzione che volevamo calcolasse. O, meglio, esiste sicuramente un programma per il quale il «Programmone» non sa rispondere.

Da Albano, Aiello, Attardi, Montanari «Teoria della Computabilità Teoria dei Linguaggi Formali» ed. ETS Pisa:

«L'origine di questa impotenza risiede nel fatto che gli oggetti della nostra azione, gli algoritmi, coincidono con i nostri attrezzi di lavoro, che sono ancora e solo gli algoritmi».

Anche perchè bisognerebbe compiere un falso ideologico nel costruire la tabella di prima: non saprei infatti come riempire le caselle dove la macchina di Turing X, sul dato Y non termina. Nel senso che non posso nemmeno metterci un trattino ad indicare ciò, per il semplice fatto che... se faccio partire la MDT X sul dato Y e dopo un'ora di elaborazione non si è ancora fermata nessuno mi autorizza a dire che non si fermerà mai più: potrà magari accadere tra un'altra ora, tra un anno, tra un miliardo di anni... chi lo sa.

Ma anche commettendo il falso e, diciamo, riuscendo a costruire la tabella mettendo dei trattini dove le MTD sul dato in ingresso non terminano, non potrei utilizzare ugualmente il procedimento di cui sopra.

Proviamo: prendiamo gli elementi della diagonale, questa volta composti da numeri e «trattini» e sommiamo uno ad ogni elemento per ottenere una nuova riga. Secondo problema: quanto fa trattino più uno? Difficile rispondere. Possiamo però assumere che faccia ancora «trattino» se intendiamo che la somma è effettuata dopo che la MDT X sul dato X (siamo sulla diagonale) ci fornisce il suo risultato. Non sapendo a quale risultato sommare 1 non sappiamo quanto vale la somma, quindi trattino.

Ok!, pur dopo esserci ripetutamente tappati il naso, ottenuta la nostra nuova riga formata come le altre da numeri e trattini sorge l'ultimo problema: non possiamo dire se questo appartiene o meno alla tabella. Potrebbe infatti collimare sulla diagonale con un altro trattino e quindi esserci, oppure non esserci.

Se c'è tanto di guadagnato; se non c'è non dobbiamo stupirci: nella prima puntata sulla teoria della calcolabilità abbiamo appunto dimostrato che nessun formalismo è in grado di calcolare tutte le funzioni dai naturali ai naturali.

Ma Church e la sua tesi ci assicurano che la troveremo: il procedimento è, infatti, del tutto calcolabile. Valutare la nostra funzione «diagonale più uno» non vuol dire altro che: preso l'input X, troviamo la MDT X (questo lo sappiamo fare essendo ormai esperti di Goedelizzazione); facciamo partire la Macchina di Turing numero X scrivendo sul nastro il suo dato iniziale X (questo vuol dire prendere la diagonale della ipotetica tabella). Se la MDT termina sommiamo 1 al risultato (e questo è certamente calcolabile) se non termina, semplicemente stiamo lì ad aspettare che termini ottenendo l'effetto voluto: la non terminazione anche della somma. Valutare la diagonale più uno è algoritmicamente possibile, quindi calcolabile, dunque per la tesi di Church esiste una macchina di Turing che calcola questa funzione... implica la corrispondente riga è presente «in tabella». A questo punto, una buona dose di pillole per il mal di testa. Arrivederci.

MC

BIBLIOGRAFIA

Aiello, Albano, Attardi, Montanari:
Teoria della computabilità logica,
teoria dei linguaggi formali
Editrice ETS Pisa, 1976.