Estratto da

R. Ferro e A. Zanardo (a cura di), *Atti degli incontri di logica matematica* Volume 3, Siena 8-11 gennaio 1985, Padova 24-27 ottobre 1985, Siena 2-5 aprile 1986.

Disponibile in rete su http://www.ailalogica.it

TAVOLA ROTONDA «LOGICA E INFORMATICA: DA INCIDENTI DI CONFINE A UN'ALLEANZA DI INTERESSE?»

INTERVENTO DI
CORRADO BÖHM
Dipartimento di Matematica Università di Roma «La Sapienza»

Per potere diventare un buon matematico o informatico, per potere insegnare materie informatiche in maniera concettualmente economica occorre avere a disposizione dei buoni modelli computazionali. Vediamone la breve storia nel contesto dello sviluppo dell'informatica teorica negli ultimi quarant'anni.

Agli inizi degli anni '50 i modelli erano tutti presi a prestito dalla logica matematica: macchine di Turing, teoria delle funzioni ricorsive,algoritmi di Marcof, sistemi di Post erano in primo piano mentre il λ-calcolo di Church e la teoria dei combinatori di Curry e Fitch stavano nella penombra. Vi erano teoremi che affermavano l'equivalenza di tutti questi modelli senza privilegiarne alcuno. Negli anni '60, mentre si sviluppavano rigogliosamente la teoria degli automi e dei linguaggi regolari e più in generale dei linguaggi di programmazione, questi ultimi venivano anche implementati tenendo più o meno conto dei contemporanei risultati teorici (vedi LISP, Algol, PL1, APL, BASIC).

Contemporaneamente si sviluppava la semantica di tali linguaggi basata appunto sul λ-calcolo e si cercava una teoria generale delle funzioni ricorsive che prescindesse dalla natura dei dati (numeri, booleani, stringhe, matrici, liste e alberi). Un primo tentativo fu fatto colle "Uniform Reflexive

Structures" da ricercatori di York Town che presto si concentrarono sui tipi algebrici di dati sfociando nella disciplina creata dal gruppo ADJ.

Negli anni '70 si sviluppava la teoria della complessità di calcolo, come raffinamento della teoria delle funzioni subricorsive totali, giungendo alla problematica P e NP. Mentre nasceva il λ-calcolo con tipi del 2º ordine per merito di Girard e Reynolds, dilagava la moda della semantica denotazionale di Scott ed altri, che solo negli anni '80 veniva gradualmente ridimensionata con l'uso di semantiche operazionali.

Se noi prescindiamo dal main stream della ricerca logico-informatica degli anni '80 che potrebbe essere individuato nella semantica dei processi concorrenti , nella programmazione logica, nei modelli di logica modale temporale dinamica e nei modelli del λ-calcolo, mi sembra che dobbiamo ancora saldare i conti col passato riprendendo la problematica delle "Uniform Reflexive structures", ma restringendola alle funzioni totali e corredandola con la concezione dei tipi algebrici di dati (in cui la nozione di tipo diventa essenziale).

Mi scuso in anticipo se d'ora in poi dovrò riferirmi a idee o lavori quasi esclusivamente personali.

E' dell'80 un lavoro in cui si esibiscono rappresentazioni di liste lineari e di vari tipi di alberi (concepiti come liste ereditariamente finite) mediante combinatori con un approccio astratto, molto simile a quello algebrico [1].

Rammento che già da Böhm e Gross nel 1966 [2] e da Curry nel '72 [3] venivano ipotizzati sistemi numerici "generali", definiti da una quadrupla (fra infinite) <[0],[s],[p],[d]> di combinatori cioè rappresentanti lo zero , le funzioni successore e predecessore e il predicato che distingue lo zero dagli altri numeri, e soddisfacenti un certo numero di relazioni (corrispondenti agli assiomi di Peano escluso quello di induzione).

Nell'80 le liste venivano rappresentate mediante una sestupla di combinatori <nil, unit,cat,head ,tail,null > (anche qui si poteva scegliere fra un'infinità di sestuple) obbedienti a un certo numero di relazioni.

Nell'82 una di queste sestuple veniva scelta per fondare su un sistema di combinatori la programmazione funzionale proposta da Backus nel '78 [4]. Naturalmente anche gli operatori Map (apply to all) e Lit (iterazione su liste o*insert to the right* ovvero anche chiamato *tail recursion)* venivano rappresentati da combinatori in cui, ovviamente, era coinvolto il combinatore del punto fisso **Y**.

Talvolta le cose che si ritengono ovvie non lo sono affatto e in questo caso era proprio così. Sembrava "antidemocratico" (è nota l'estrema democrazia dei combinatori che trattano nello stesso modo funzioni ed argomenti) che nel caso dei numeri naturali, (rappresentati da Church come λ-termini) la rappresentazione di una funzione definita iterativamente (come per esempio l'addizione) si potesse esprimere senza l'uso del combinatore **Y**, ovvero senza iterazione o ricursione, mentre ciò non riusciva per la tail-recursion nel caso delle funzioni di lista. Appena mi posi questa domanda, nel corso dell'82, mi accorsi che ciò dipendeva dall'aver scelto male i primi tre combinatori della sestupla cioè nil, unit e cat. Ben presto trovai la soluzione e sviluppando una discussione avuta con Kozen fui in grado di ottenere lo stesso risultato anche con le strutture di dati chiamate alberi [5].

Nell'83 Leivant [6] trasformava dimostrazioni di totalità di funzioni su naturali (usando la logica di Peano del 2º ordine e la corrispondenza di Curry-Howard fra formule e tipi) nella costruzione di λ-termini del 2º ordine rappresentanti tali funzioni. Ciò diede a me e Berarducci la speranza di potere definire una famiglia di algebre in cui tali dimostrazioni venissero per

così dire fatte una volta per tutte, in modo che bastasse possedere una definizione ricorsiva (o meglio iterativa) di una funzione per dedurne, in modo assolutamente automatico, un λ-termine o combinatore in forma normale (non espresso con **Y** o in modo equivalente) che la rappresenti.

I risultati, pubblicati recentemente sul TCS [7] , sono andati aldilà delle speranze cui ho appena accennato, aprendo un nuovo campo di ricerca sia teorico che applicativo, e chiudendo le annose incertezze su quali modelli di computazione scegliere, anche soltanto allo scopo di insegnare una sintetica teoria dei programmi.

Mi propongo di esaminare i risultati guardandoli attraverso una serie di filtri che per brevità potremo chiamare:

(i) Tipi algebrici di dati e sistemi di riscrittura

(ii) Programmazione funzionale

(iii) Funzioni subricorsive

(iv) Programmazione strutturata

(i) La principale restrizione sulle algebre, che permette di ottenere i risultati cui si è già accennato e che verranno ribaditi fra poco, è quella di limitarsi esclusivamente a particolari algebre iniziali, dette ***algebre assolutamente libere*** o ***anarchiche*** .

Ciò significa che ogni elemento di ognuna di tali algebre può essere scritto in un unico modo mediante i costruttori dell'algebra in questione. Per fortuna ciò non costituisce una seria limitazione in quanto molti sistemi di dati usati in informatica (Booleani, Insiemi finiti, Naturali, Liste parametriche, cioè sequenze finite di...,Stringhe binarie, Alberi binari ecc.) possono venire così descritti. Un esempio di problema ancora aperto è quello di riuscire a descrivere come algebra anarchica l'insieme dei grafi finiti aciclici.

Consideriamo ora la definizione iterativa di una funzione che mappa una di tali algebre (supposte per semplicità omogenee) su un'altra. In altre parole il valore di tale funzione per un qualsiasi argomento dipende in modo noto (mediante parametri funzionali noti) esclusivamente dal (dai) valore (i) della funzione calcolata per la (le) componente (i) algebrica(che) di tale argomento.

E' abbastanza banale trasformare la definizione iterativa in un sistema di riscrittura dotato di un'espressione iniziale (significante il valore della funzione da calcolare per un dato elemento dell'algebra) ed il cui calcolo consiste in una reiterata applicazione delle regole di riscrittura fino a che il simbolo di tale funzione sia sparito.

Ebbene, un risultato fondamentale del lavoro che sto descrivendo consiste nel potere sostituire al sistema di riscrittura appena descritto un sistema di riscrittura "molto più semplice" del precedente, dove cioè l'espressione iniziale non è altro che l'elemento dato dell'algebra (ovvero l'argomento della funzione da calcolare) e le regole di riscrittura sono tante quante sono i costruttori dell'algebra: in effetti ogni regola consiste nel sostituire un costruttore col parametro funzionale che si trova nel corrispondente membro destro della definizione iterativa.

(ii) Il rimpiazzamento appena descritto di simboli funzionali con altri simboli funzionali mette sufficientemente in luce che il risultato di cui si parla appartiene propriamente a quella disciplina che viene chiamata programmazione funzionale. In effetti ogni dato o elemento di un'algebra anarchica è descritto come una composizione di costruttori che soddisfa i vincoli dei tipi su cui tale algebra è definita. Per ottenere il valore di una funzione su tale algebra i costruttori vengono sostituiti da altri costruttori o da funzioni più semplici (come funzioni costanti o proiettori) o da funzioni

precedentemente ottenute (ovvero definite) da schemi iterativi. Quanto appena detto definisce induttivamente la famiglia **I** delle funzioni definite iterativamente sulle algebre anarchiche. Un altro importante risultato del lavoro è che la famiglia **PR** delle funzioni definite mediante ricursione primitiva coincide con **I** se ci riferisce all'intera classe delle algebre anarchiche.

(iii) Una versione costruttiva di tale risultato è stata recentemente presentata al convegno ESOP 86 [8] sotto forma di un algoritmo uniformemente valido per ogni algebra anarchica mirante a ridurre la ricursione primitiva all'iterazione. Naturalmente il metodo esposto, pur essendo originale, non è l'unico. E' in corso di studio uno sfruttamento sistematico del più noto metodo delle coppie per ridurre altri tipi di ricursione all'iterazione. Poichè nel lavoro pubblicato dal TCS si mostra come il metodo colà esposto si possa applicare anche alla programmazione della funzione di Ackermann (che non è primitiva ricorsiva) ci si può chiedere quale sia la classe delle funzioni cui tale metodo si applica. La risposta non è ancora chiara anche se si sa che tale classe è certamente limitata, per quanto riguarda i naturali, a quella delle ricorsive totali provabili nell'aritmetica del 2º ordine di Peano (comunicazione di Girard).

(iv) Ci si può chiedere in che relazione i risultati esposti stiano con la programmazione strutturata, la cui disciplina, instaurata a partire dal 1970 e basata sul teorema di Böhm-Jacopini (1966) [9], negli anni '80 viene ancora insegnata in molti corsi scolastici ed aziendali. Come è noto essa consta nell'evitare il comando GOTO limitandosi alle chiamate sequenziali di procedure (operatore di composizione) e usando unicamente le strutture di controllo IF-THEN-ELSE (definizione per casi) e WHILE (Iterare una procedura finchè un certo predicato è vero, uscirne quando è falso). Se si

considera che l'uso della definizione per casi è facilmente riducibile a composizioni funzionali in cui interviene l'algebra dei booleani e che tutte le procedure, usando la disciplina delle algebre anarchiche, terminano, ci si rende conto che tale disciplina consiste nell'eliminare i cicli, persino nel concepire gli algoritmi. In altri termini ciò equivale ad usare esclusivamente dei *while limitati* nel numero di cicli e poi eliminarli, incorporandoli nelle dimensioni degli elementi dell'algebra su cui si calcola.Quindi, riassumendo, la nuova disciplina elimina il WHILE. Tutti i programmi diventano aciclici e quindi si possono disegnare con diagrammi aciclici: l'unico operatore che resta è quello di composizione di procedure con più argomenti.

Non si deve credere tuttavia che possedendo uno strumento automatico di sintesi dei programmi in termini di specifiche di funzioni ricorsive su algebre anarchiche tutti i problemi siano risolti. E' indubbiamente vero che questa disciplina può contribuire molto alla scrittura di programmi corretti, specialmente se aiutata da controlli statici di tipi del 2º ordine. Vi sono tuttavia ancora molti problemi insoluti sugli interpreti e sui compilatori per tali linguaggi algebrici. Per non parlarne che di uno, ritorniamo al WHILE, che, eliminato fisicamente da un suo surrogato, potrebbe *vendicarsi* facendo allungare i tempi di calcolo non permettendo cioè di utilizzare un risultato appena prodotto,nel cuore di un elemento algebrico, ma costringendo ad operazioni ulteriori essenzialmente inutili che hanno il solo scopo di farlo emergere.

Una soluzione a questo problema sembra essere la medesima che permette di estendere la classe delle funzioni calcolabili, in analogia a quanto detto per la funzione di Ackermann.

E' venuto il momento di trarre le conclusioni di questo intervento. Spero di avere illustrato l'insostituibilità della logica per lo sviluppo dell'informatica

e non solo per quello dell'informatica teorica. Il punto di vista sostenuto, che privilegia cioè fra tutti i modelli di calcolo deterministici, quello delle algebre assolutamente libere o anarchiche, offre almeno due vantaggi concettuali, provenienti dall'avere concentrato tutta la ricorsività nella costruzione induttiva dei dati algebrici: 1) Lo stile della programmazione funzionale e' reso accessibile ad una pluralità di algebre.2) I costrutti ripetitivi come il WHILE e il FOR risultano superflui mentre viene lasciata libera la fantasia di creare le algebre più adatte agli algoritmi che ci si propone di sviluppare.

BIBLIOGRAFIA

[1] Böhm, C., An Abstract Approach to (Hereditary) Finite Sequences of Combinators, in :J.P.Seldin and R. Hindley, eds., To H.B. Curry: Essay on Combinatory Logic,Lambda Calculus and Formalism, Academic Press, London, 231-242.

[2] Böhm,C. and Gross, W.,Introduction to the CUCH, in E.R. Caianiello (ed.), Automata Theory, Academic Press, New York, 1966, 35-66.

Curry, H.B., Hindley, R. and Seldin,J.P., Combinatoric Logic, Vol. 2, North - Holland, Amsterdam, 1972.

[4] Backus, J., Can programming be liberate from Von Neumann style? A functional stile and its algebra of programs, CACM **21**, 8, 1978, 613-641.

[5] Böhm,C. and Kozen, D., Eliminating recursion over acyclic data structures in functional programs, in 4th Intern. Workshop on the Semantics of Programming, Bad Honnef, March 1983;Summary in Bull. EATCS (1983), 205.

[6] Leivant, D., Reasoning about functional programs and complexity classes associated with type disciplines, 24th Ann. Symp. on Foundation of Computer Science (1983), 460-469.

[7] Böhm,C. and Berarducci, A., Automatic Synthesis of Typed λ-Programs

on Term - Algebras, Theor. Comput. Science **39** (1985), 135-154.

[8] Böhm, C., Reducing Recursion to Iteration by Algebraic Extension, in: Robinet,B. and Wilhelm,R.,eds., ESOP 86 European Symposium on Programming , March 1986, Saarbrücken, LNCS **213**,Springer Verlag , Berlin, 111-118.

[9] Böhm, C. and Jacopini, G., Flow-Diagrams, Turing Machines and Languages with Only Two Formation Rules, Comm. ACM **9** (1966), 366-371.