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
CARLO CELLUCCI
Roma

1. Nel 1963 McCarthy scriveva:

> "E' ragionevole sperare che la relazione tra il computo e la logica matematica nel prossimo secolo sarà altrettanto fruttuosa di quella tra l'analisi e la fisica nel secolo scorso".[1]

Le speranze di McCarthy riguardano il prossimo secolo, e pertanto fino a quell'epoca è doveroso sospendere il giudizio. Ciò non impedisce di chiedersi se, negli sviluppi dell'informatica dalle origini fino ad oggi, esistano segni che incoraggino tali speranze. La domanda è tanto più legittima in quanto sono diffuse credenze che sembrano dare per scontato che il rapporto informatica-logica matematica non sia solo - per usare la felice espressione che compare nel titolo di questa tavola rotonda - un incidente di confine, ma abbia un carattere più organico, o addirittura abbia già dato sostanziosi frutti nella pratica informatica.

Una di queste credenze - o, per meglio dire, miti - è che una riprova della fecondità del rapporto informatica-logica matematica sia data dal LISP: un figlio legittimo - così si ritiene - del lambda-calcolo. Ora, questo mito è stato contestato recentemente dallo stesso McCarthy, il padre del LISP:

> "Per usare le funzioni come argomenti occorre una notazione per le funzioni, e mi sembrò naturale usare la notazione lambda di Church [...] Ora, avendo preso in prestito questa notazione, devo sfatare uno dei miti concernenti il LISP [...] e cioè che il LISP sia in qualche modo una realizzazione del lambda-calcolo, o che intendesse esserlo. La verità è che io, in realtà, non capivo il lambda-calcolo [...] Dunque il LISP non fu un tentativo di rendere pratico il lambda-calcolo - anche se ammetto che, se qualcuno fosse partito con quell'intenzione, avrebbe finito probabilmente per trovare qualcosa di simile al LISP".[2]

Spesso i miti sono innocui, e talora sono persino utili perché aiutano a sopportare il peso dell'esistenza. Ciò non toglie che in qualche caso essi possano essere pericolosi.

Cinquant'anni fa, nelle Osservazioni sopra i fondamenti della matematica (IV, 24), Wittgenstein parlava della "funesta irruzione" della logica nella matematica. Presa alla lettera l'affermazione di Wittgenstein è falsa. La pratica matematica possiede tali cinture protettive - tra le quali non ultima l'inerzia - da scoraggiare o comunque neutralizzare qualsiasi irruzione della logica. Cinture protettive ancor più valide - tra le quali non ultimo il collegamento con la produzione industriale - schermano l'informatica da influenze estranee indesiderate. Tutt'al più i tentativi di irruzione della logica nell'informatica possono ingenerare frustrazioni nei logici che si aspettino una facile conquista o, quanto meno, di essere presi sul serio dagli informatici - dagli informatici, intendo, con ricadute industriali.

Per far capire a che cosa alludo mi spiegherò con qualche esempio, del resto abbastanza scontato, ma non troppo

a giudicare dal tono di certi interventi a questo X Incontro di Logica Matematica.

I logici - o meglio, gli informatici di formazione logica - continuano a moltiplicare gli enti, inventando nuovi tipi di dati astratti. Ma nel far ciò si scontrano con ogni sorta - o forse dovrei dire tipo - di difficoltà: 1) Dal punto di vista semantico, che cos'è un tipo? E' una teoria, una classe di modelli, un'algebra, una categoria? 2) Qual è l'esatta relazione tra un oggetto e il suo tipo? 3) I tipi devono essere valori del linguaggio di programmazione considerato? o di qualche suo sottolinguaggio decidibile? 4) Tutte le nozioni della programmazione devono basarsi su una teoria costruttiva dei tipi?

Il risultato di tutto questo è la creazione di linguaggi di programmazione spesso artificiosi e poco maneggevoli, di cui solo i logici - e gli informatici di formazione logica - sembrano apprezzare la bellezza, e in cui è estremamente improbabile che vengano scritti altro che programmi-giocattolo, o comunque programmi di scarsa utilità pratica. E intanto i programmi di utilità pratica continuano ad essere scritti in FORTRAN, BASIC, C, Pascal, Modula-2 ecc., cioè in quelli che, ad un informatico di formazione logica, appaiono come delle mostruosità insensate: "nonsense monsters" è in effetti l'appellativo usato recentemente da Huet per riferirsi a tali linguaggi.[3]

Un secondo esempio. Gli informatici di formazione logica continuano ad inseguire il miraggio di formalismi assolutamente generali - o per lo meno molto generali -

un'eredità dell'ossessione dei logici per i linguaggi universali. Questo tipo di approccio ha dominato a lungo anche nel campo dell'intelligenza artificiale. Si pensi alla ricerca di modelli psicologici, euristici o linguistici di tipo globale (come quelli ispirati rispettivamente a Piaget, Polya e Chomski), o di metodi ultragenerali (come il principio di risoluzione di Robinson nel campo della dimostrazione automatica).

Il risultato di tutto questo è la creazione di sistemi per lo più 'intrattabili' dal punto di vista computazionale, che non lasciano alcuna possibilità di estendere le soluzioni trovate per problemi-giocattolo a problemi praticamente interessanti. E intanto si assiste all'esplosione di una miriade di sistemi esperti che sono l'esatto contrario della ricerca di formalismi molto generali. L'idea è che questi sistemi possiedano abbastanza conoscenze relative al particolare campo considerato da permettere di superare la barriera della complessità. Di conseguenza ogni sistema esperto introduce un suo proprio formato della base dei dati, un proprio metalinguaggio, propri algoritmi per la soluzione di problemi, pattern matchers ecc.. Si ha così un imprevisto sottoprodotto della torre di Babele dei linguaggi di programmazione - un labirinto di sistemi - al limite un sistema per ogni possibile campo di applicazione dei calcolatori.

2. Non vorrei, con questo, aver dato l'impressione di ritenere che le speranze di McCarthy circa i rapporti tra informatica e logica matematica nel prossimo secolo non ricevano alcun conforto dalla situazione attuale. Ci sono indizi di aree promettenti, o per lo meno che autorizzano qualche speranza. Poiché non avrei tempo di menzionarle tutte, mi limiterò ad indicarne due. L'unica giustificazione della mia scelta è che sono quelle che conosco meglio, e che tra esse, a mio modo di vedere, esistono interessanti connessioni. Curiosamente, inoltre, queste due aree hanno ricevuto scarsa attenzione in questo X Incontro di Logica Matematica.

La prima area è costituita dalla programmazione logica. Con questo termine non intendo riferirmi solo alla particolare versione che trae origine dalla scoperta di Kowalski e Kuehner nel 1970 che la procedura dimostrativa dell'eliminazione dei modelli di Loveland e una forma della regola di risoluzione di Robinson - la cosiddetta risoluzione lineare - sono l'una una variante dell'altra. Che lo si voglia o meno, la storia e il destino di questa particolare versione della programmazione logica si intrecciano strettamente con le vicende del Prolog.

Ora, è indubbio che alcune caratteristiche del Prolog lo rendono unico tra tutti i linguaggi di programmazione: 1) Programmi e dati vengono presentati in forme del tutto simili. 2) Le procedure possono contenere parametri che possono comparire sia come input che come output. 3) Esse possono dar luogo a risultati contenenti variabili non vincolate. 4) Il backtracking è parte integrante del linguaggio, e perciò consente di determinare soluzioni multiple di un dato problema. 5) Le capacità generali di pattern-matching operano in congiunzione con un meccanismo di ricerca goal-seeking.

Accanto a queste caratteristiche oggettive del linguaggio vanno menzionati anche alcun vantaggi soggetti-

vi: 1) Avendo il suo fondamento nel calcolo dei predicati, il Prolog incoraggia il programmatore a descrivere i problemi secondo i canoni della logica, il che facilita la verifica della correttezza dei programmi riducendo lo sforzo del debugging. 2) Gli algoritmi richiesti per interpretare i programmi Prolog sono spesso riconducibili ad elaborazioni parallele. 3) La relativa concisione dei programmi Prolog, con la conseguente riduzione dei tempi di sviluppo, lo rendono un linguaggio ideale per la sperimentazione di prototipi.

Il prezzo che occorre pagare per i vantaggi offerti dal linguaggio è costituito - ovviamente - dalle aumentate esigenze di memoria e di velocità di elaborazione. Tuttavia la storia dell'evoluzione dei linguaggi di programmazione dimostra che, col progredire delle tecniche VLSI, questo prezzo diventa non solo accettabile, ma perfino conveniente poiché i risparmi consentiti dalla concisione dei programmi e dal ridotto sforzo richiesto dalla programmazione - oltre alla facile leggibilità dei programmi - compensano ampiamente le maggiori esigenze di memoria e di tempo di esecuzione. Del resto l'algoritmo dell'unificazione di Robinson è stato implementato nel silicio, vari progetti di Prolog machines sono in un avanzato stadio di realizzazione, e per uno di essi si prevede addirittura una velocità di 425.000 LIPS (= inferenze logiche al secondo). Senza contare che le aumentate esigenze di efficienza imposte dai programmi Prolog stimolano le ricerche nel campo dell'ottimizzazione, e in particolare sul parallelismo.

Nonostante i vantaggi, però, il Prolog non è esente

da difetti, che derivano in gran parte dal fatto che esso è costruito intorno ad un dimostratore automatico di teoremi piuttosto rudimentale. I difetti più macroscopici, quali emergono dal confronto con la logica classica, sono i seguenti: 1) Il Prolog è sensibile all'ordine in cui sono immesse le clausole, cioè cambiando l'ordine lo stesso goal può dar luogo a risposte differenti. Per esempio, se la procedura 'append' è data dalle clausole append([],L2,L2), append([X|L1],L2,[X|L3]) :- append(L1,L2,L3), il goal append(L1,[a],L3) dà luogo ad infinite soluzioni, cioè L1 ← [] e L3 ← [a], L1 ← [X1] e L3 ← [X1,a], L1 ← [X1,X2] e L3 ← [X1,X2,a], ecc.. Se invece si inverte l'ordine delle due clausole precedenti, la procedura risultante cerca anzitutto l'ultima di tali infinite soluzioni, il che dà luogo ad una successione di chiamate ricorsive che non ha mai termine. 2) Il significato della negazione nel Prolog (non-A se il goal A non ha successo) non è quello vero-funzionale. Esso si basa sulla cosiddetta assunzione del mondo chiuso, secondo cui le clausole di cui consta un programma rappresentano tutta la conoscenza sulle relazioni nominate nel programma - una versione del principio del Tractatus logico-philosophicus: "I limiti del mio linguaggio significano i limiti del mio mondo". Ciò compromette la completezza. Per esempio, data la clausola P :- not(P), il goal P dà luogo ad un loop infinito. 3) Il Prolog non è monotonico, cioè un goal che, con date clausole, dà luogo ad una certa risposta, qualora a quelle clausole se ne aggiunga una nuova può dar luogo ad una risposta dif-

ferente. Ciò è evidente, ad esempio, in base al significato della negazione nel Prolog.

Questi difetti mostrano che il Prolog non possiede alcune tra le caratteristiche più fondamentali - e, a mio parere, irrinunciabili - della logica classica. Per ovviare ad essi sembra ragionevole rimpiazzare il dimostratore automatico di teoremi del Prolog con un altro più sofisticato. In linea di principio un tale dimostratore potrebbe anch'esso - come quello del Prolog - essere basato sul principio di risoluzione di Robinson. Ma in pratica tale soluzione non è soddisfacente, principalmente perché in generale la riduzione di una specificazione a forma clausale comporta una sostanziale perdita di compattezza e soprattutto di intellegibilità.

In vista di ciò risulta più conveniente rinunziare del tutto a ridurre le specificazioni a forma clausale, adottando piuttosto un approccio basato sui metodi di Gentzen (o loro derivati).

3. L'uso dei metodi di Gentzen, in particolare del calcolo delle sequenze, venne esplorato da Prawitz, Wang e Kanger fin dai primordi della dimostrazione automatica, e suscitò un certo interesse. Ne è una prova, ad esempio, il fatto che il manuale originario del LISP 1.5 di McCarthy e collaboratori fornisse come unico esempio articolato di programmazione un'implementazione dell'algoritmo di Wang per il calcolo proposizionale. Tale algoritmo, del resto, era destinato a diventare un classico dell'intelligenza artificiale, come mostrano i testi di Raphael e Shapiro. Per motivi non del tutto chiari - e che varrebbe la pena di indagare - a questo interesse

iniziale seguì, però, un ventennio di quasi totale oblio, e solo recentemente si è registrata una ripresa dell'interesse soprattutto grazie ad un lavoro di Bowen.[4]

Considerazioni di compattezza consigliano, tuttavia, di rimpiazzare il calcolo delle sequenze con qualche suo derivato più maneggevole, come gli alberi di verità di Smullyan, ma per ragioni di efficienza le usuali regole degli alberi di verità devono essere modificate. La regola critica è:

$$\begin{array}{c} \forall x A(x) \\ | \\ A(t) \end{array} .$$

Quale termine t scegliere? La soluzione adottata correntemente (nelle dimostrazioni di completezza) consiste in un'applicazione dell'algoritmo del British Museum: nello stadio n prova con ciascuno di $t_0,\ldots,t_n$ dove $t_0,t_1,\ldots$ è un'enumerazione fissata di tutti i termini.[5]

Per gli scopi tradizionali della logica matematica - che <u>studia</u> i sistemi formali, ma non li <u>usa</u> - questa soluzione estrema va bene, ma difficilmente la si potrebbe considerare accettabile per le applicazioni al mondo reale. Una soluzione più soddisfacente si ottiene ricorrendo al suggerimento di Prawitz e Kanger di introdurre <u>pseudovariabili</u>, affidando all'algoritmo dell'unificazione il compito di trovare un adeguato sostituto per esse. Ma, a differenza di quanto previsto da Prawitz e Kanger, la ricerca di un tale sostituto non verrà rimandata fino alla fine - cioè fino a quando l'intero pseudoalbero della dimostrazione sia stato sviluppato - poiché questa strategia darebbe alla ricerca un carattere glo-

bale a scapito dell'efficienza. Al contrario, la ricerca verrà localizzata alternando le applicazioni delle regole di inferenza con applicazioni dell'algoritmo dell'unificazione.

Problemi di efficienza sono sollevati anche dalle regole dell'eguaglianza, che saranno rimpiazzate, perciò, con un'opportuna versione della regola di paramodulazione.[6]

Naturalmente il sistema risultante non dev'essere inteso come un dimostratore automatico universale di teoremi: l'era delle illusioni dei procedimenti dimostrativi uniformi è tramontata da un pezzo. Come nel caso del Prolog il peso dell'implementazione di procedimenti dimostrativi efficienti relativi al particolare dominio considerato ricade in notevole misura sulle spalle del programmatore. E' singolare come questa circostanza sia vissuta dai sostenitori del Prolog come una limitazione da superare piuttosto che come un fatto con cui convivere:

> "Allo stato attuale dell'arte una programmazione logica efficace si basa in gran parte sulla scelta di componenti logiche appropriate che siano confacenti al limitato controllo disponibile. Un giorno potrebbe essere possibile basarsi sulla implementazione per realizzare il controllo più efficace per qualunque programma immesso, facendo gravare così l'onere dell'intelligenza sulla macchina piuttosto che sul programmatore".[7]

Un residuo dell'era delle illusioni dei procedimenti dimostrativi uniformi?

E' vero, però, che nel Prolog gli strumenti di controllo disponibili (principalmente: l'ordine delle clausole e il cut) sono piuttosto limitati. A maggior ragione è essenziale escogitarne di più efficaci in un sistema non basato sulla risoluzione come quello a cui si è accennato sopra. Contrariamente ad un'opinione diffusa la presenza di questi - come del resto di altri - elementi extralogici, lungi dal violare la natura autocontenuta della logica, la arricchisce di ciò che più le manca: la capacità di trattare, oltre agli aspetti statici, anche gli aspetti dinamici dell'informazione.[8] Una certa quantità di lavoro è stata svolta in questa direzione nel caso del Prolog, per esempio attraverso lo sviluppo del suo dialetto Epilog, ma presumibilmente molto di più ne occorrerà per un sistema basato sugli alberi di verità.

4. Una seconda area promettente è quella dell'<u>estrazione di programmi dalle dimostrazioni</u>. Una dimostrazione D di una formula della forma $\forall x \exists y A(x,y)$ sotto certe condizioni - per esempio, se è una dimostrazione costruttiva - fornisce un metodo per calcolare, per ogni x, un valore y tale che A(x,y), e perciò può essere considerata come la descrizione di un algoritmo f tale che, per ogni x, A(x,f(x)). Esistono vari procedimenti meccanici per estrarre f da D: l'eliminazione dei tagli (Gentzen), la normalizzazione (Prawitz), l'interpretazione di Dialectica (Gödel), la realizzabilità ricorsiva (Kleene e Nelson), la realizzabilità modificata (Kreisel). Tuttavia, in base al risultato di Mints sulla stabilità degli E-teoremi, e a quelli di Diller sulla commutatività, questi procedimenti producono programmi che computano tutti la stessa funzione f.

Tra essi il procedimento della normalizzazione presenta sostanziali vantaggi perché è il più semplice e fondamentale ed è anche facilmente implementabile. Il suo difetto principale - comune del resto agli altri procedimenti - è l'inefficienza. Le ragioni dell'inefficienza sono: 1) La rappresentazione corrente delle dimostrazioni nei sistemi di deduzione naturale non è la più adatta per la riduzione delle dimostrazioni. 2) Non sempre è necessario normalizzare l'intera dimostrazione. Spesso un numero di passi di riduzione molto minore è sufficiente per calcolare f. 3) Il repertorio delle regole di induzione è povero.

Un contributo alla soluzione di questa difficoltà è stato dato da Goad nella sua tesi di dottorato. La difficoltà 1) è stata risolta attraverso l'introduzione dei cosiddetti p-termini (un'estensione dei lambda-termini con tipi). La difficoltà 2) è stata risolta ammettendo come assiomi formule di Harrop qualsiasi. La difficoltà 3) è stata risolta attraverso l'introduzione delle cosiddette dimostrazioni ricorsive. Il procedimento risultante è stato adoperato con successo per estrarre un programma per l'algoritmo del bin-packing da una dimostrazione di una formalizzazione dell'algoritmo nella teoria del primo ordine dei numeri e delle liste di numeri.[9]

Naturalmente la normalizzazione permette di estrarre un programma soltanto da una dimostrazione data. Tuttavia sotto certe condizioni essa consente di costruire una dimostrazione completa di una formula a partire da una dimostrazione incompleta di quella stessa formula, per esempio da una dimostrazione contenente lemmi falsi. In questo senso essa può essere considerata come una sorta di ricerca automatica della dimostrazione di una data formula, il che la accosta alla programmazione logica e al Prolog. In particolare una dimostrazione costruttiva incompleta di una formula della forma $\forall x \exists y A(x,y)$ - che costituisce solo una descrizione parziale di un algoritmo f - una volta normalizzata può produrre una descrizione totale di f. In altri termini sotto certe condizioni la normalizzazione fornisce un metodo per ottenere un programma completo per il computo di f a partire da un frammento di programma.

L'accostamento della normalizzazione alla programmazione logica e al Prolog non è una forzatura. In effetti se la normalizzazione viene implementata - come fa Goad - nel modo call-by-value, allora la normalizzazione di una dimostrazione incompleta corrisponde esattamente alla ricerca di una dimostrazione nel modo backward chaining simile a quella implementata nell'interprete Prolog. E' vero che manca il backtracking, ma esso può essere aggiunto facilmente al meccanismo della normalizzazione. Viceversa la sintesi dei programmi costituisce una delle principali finalità della programmazione logica, che è stata perseguita con qualche successo, ad esempio, nel caso di algoritmi per la manipolazione di liste, l'unificazione o l'integrazione numerica.

5. Il fatto che le due aree sopra menzionate appaiano incoraggianti non significa che gli attuali metodi della logica matematica siano i più idonei per trattarle.

I sistemi formali della logica matematica furono introdotti originariamente per la ricerca sui fondamenti della matematica, i cui scopi prevedevano che si indagasse sulla possibilità di ridurre in linea di principio il ragionamento ad un calcolo meccanico, non certo che si elaborassero gli strumenti per effettuare praticamente tale riduzione, né tanto meno che si meccanizzasse fattibilmente la ricerca delle dimostrazioni.

I progressi futuri dell'informatica, e con essi anche quelli della logica matematica, presumibilmente dipenderanno in notevole misura dalla nostra capacità di ideare nuovi sistemi formali in cui si possano costruire dimostrazioni trattabili efficientemente dalla macchina, oltre che comprensibili per l'uomo. I sistemi in questione - a differenza, ad esempio, di quelli di deduzione naturale - non dovranno necessariamente ispirarsi al principio di analizzare fedelmente il comportamento umano - nella fattispecie, il comportamento dimostrativo. Le macchine di Turing, almeno nelle intenzioni del loro autore, si basavano su tale principio, ma non appena si è voluto programmare un calcolatore per svolgere un compito complesso si è imposta la necessità di sviluppare linguaggi come il FORTRAN, il LISP, ecc. ispirati ad una filosofia del tutto differente: prestare attenzione alla struttura dei dati.

Le accresciute richieste che la società attuale pone ai calcolatori esigono che si passi da macchine che calcolano a macchine che inferiscono. Sarebbe strano, oltre che triste, se in questo passaggio la logica matematica non svolgesse un ruolo rilevante.

NOTE

[1] J. McCarthy, "A basis for a mathematical theory of computation", in: P. Braffort & D. Hirschberg (a cura di), Computer programming and formal systems, Amsterdam (North-Holland) 1963, pp. 33-70.

[2] "LISP session", in: R.L. Wexelblat (a cura di), History of programming languages, New York (Academic Press) 1981, pp. 173-197.

[3] G. Huet, "In defence of programming language design", in: L. Steels & J.A. Campbell (a cura di), Progress in artificial intelligence, Chichester (Ellis Horwood) 1985, pp. 219-241.

[4] Cfr. K.A. Bowen, "Programming with full first-order logic", in: J.E. Hayes, D. Michie & Y-H Pao (a cura di), Machine intelligence 10, Chichester (Ellis Horwood) 1982, pp. 421-440.

[5] Cfr. J. Bell & M. Machover, A course in mathematical logic, Amsterdam (North-Holland) 1977, pp. 89-90.

[6] Per dettagli v. C. Cellucci, "Using full first-order logic as a programming language", workshop Logic and computer science: new trends and applications, Torino 13-15 ottobre 1986.

[7] C.J. Hogger, Introduction to logic programming, London (Academic Press) 1984, p. 73.

[8] Su questo limite della logica matematica cfr. M.J. Beeson, "Proving programs and programming proofs", in: R. Barcan Marcus, G.J.W. Dorn & P. Weingartner (a cura di), Logic, methodology and philosophy of science VII, Amsterdam (North-Holland) 1986, pp. 51-82.

[9] Cfr. C.A. Goad, Computational uses of the manipulation of formal proofs, Report No. STAN-CS-80-819, Dept. of Computer Science, Stanford University 1980.