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

APPLICAZIONI DELLA TEORIA INTUIZIONISTICA DEI TIPI IN THEORETICAL COMPUTER SCIENCE

SILVIO VALENTINI
Padova

0. Introduzione

In questa relazione, non tecnica, voglio presentare gli argomenti che il gruppo di cui faccio parte, insieme a Bossi e Garzotto, sta studiando. Tutto il lavoro e' fondalmentalmente ispirato dagli studi di Martin-Lof ed altri (vedi bibliografia) sulla teoria intuizionistica dei tipi e sulle espressioni con arieta'.

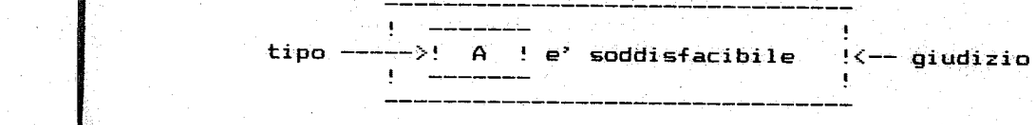
Il nostro interesse attuale e' prevalentemente rivolto alla theoretical computer science dove pensiamo che la teoria dei tipi possa servire come approccio ai problemi di correttezza dei programmi. Tuttavia lo studio teorico della teoria delle espressioni, di cui abbiamo presentato una nostra versione in [7], sembra essere un passo fondamentale verso un qualsiasi tentativo di implementazione della teoria dei tipi su elaboratore.

1. Una breve introduzione alla teoria intuizionistica dei tipi.

Nella teoria intuizionistica dei tipi si e' interessati ad esprimere giudizi su tipi. Per questo motivo, l'unico requisito che pretendiamo da una espressione per poterla chiamare tipo e' che si conosca un metodo per decidere sui giudizi che la riguardano.

In questo approccio saranno percio' tipi le usuali formule del calcolo dei predicati del primo ordine, una volta che si sia deciso cosa si e' disposti ad accettare per dimostrazione della loro verita', ma anche le specifiche che un programma (per calcolatore) deve soddisfare, se si puo' spiegare quando un programma le soddisfa.

La situazione si puo' rappresentare con la seguente figura



La teoria intuizionistica dei tipi si propone come sistema di regole che permettono di dedurre giudizi sui tipi.

La nozione di regola e' quella standard (per esempio mutuata dalla deduzione naturale): una regola e' il "riassunto" della giustificazione che ci permette di passare da alcuni giudizi (espressi nelle ipotesi) ad un nuovo giudizio (espresso nella conclusione).

Le forme dei giudizi che si possono esprimere su un tipo, al di la' delle interpretazioni che al tipo stesso si vogliono dare, sono

	Giudizio	Interpretazione logica	Interpretazione computazionale	Interpretazione insiemistica
A type	A e' un tipo	A e' una proposizione	A e' un problema	A e' un insieme
A=B type	A e B sono tipi uguali	A e B sono proposizioni uguali	A e B sono problemi uguali	A e B sono insiemi uguali
a∈A	a e' un elemento del tipo A	a e' una prova della proposizione A	a e' una soluzione del problema A	a appartiene all' insieme A
a=b A	a e b sono elementi uguali del tipo A	a e b sono prove uguali della proposizione A	a e b sono soluzioni uguali del problema A	a e b sono elementi uguali dell'insieme A

L'interpretazione insiemistica puo' essere esemplificata nel riconoscere l'insieme dei numeri naturali come tipo, cioe' nell'esprimere il giudizio N-type, e nel riconoscere come elementi "canonici" di N lo zero, cioe' nell'esprimere il giudizio $0 \in N$, e nel riconoscere la chiusura di N rispetto all'operazione di successore che si puo' esprimere nel giudizio $s(a) \in N$ se $a \in N$.

Le capacita' espressive del concetto di tipo si possono pero' vedere meglio quando si decide di "fare confusione" tra le varie interpretazioni. Ad esempio la proposizione $\forall x\ B(x)$ puo' essere riconosciuta nel tipo $(\Pi\, x \in A)\ B(x)$ il cui significato inteso e' "per ogni elemento a di A vale B(a)": viene usata contemporaneamente l'interpretazione insiemistica e quella logica.

$(\Pi\, x \in A)\ B(x)$ type e' un giudizio vero (sotto l'ipotesi A-type e B(x)-type$[x \in A]$) perche' sappiamo quali sono i suoi elementi "canonici": una sua

dimostrazione e' una qualsiasi funzione (costruibile) $\lambda(b)$ che mappi un elemento a di A in una prova b(a) di B(a).

Ne consegue la verita' del giudizio $\lambda(b) \in (\Pi x \in A)\ B(x)$ (sotto l'ipotesi che $b(x) \in B(x)$ $[x \in A]$).

E' evidente che gli elementi canonici non esauriscono (in genere) gli elementi di un tipo: ad esempio 2^2 va riconosciuto come elemento di N pur non essendo 0 o della forma "successore"; diremo allora elemento "non canonico" di un tipo un metodo che, una volta eseguito, produce un elemento canonico (del tipo).

Quando si definisce un tipo si potranno allora introdurre nuovi operatori che specifichino come operare sui suoi elementi: essi daranno luogo ad elementi non canonici, cioe' metodi; allo stesso tempo si introdurranno delle regole di computazione che specifichino 'quale' metodo: esse diranno come (semantica operazionale) puo' venire computato l'elemento canonico. Un elemento gia' canonico verra' naturalmente sempre computato in se stesso.

Ad esempio associato al tipo $(\Pi x \in A)\ B(x)$ si introdurra' un operatore Ap per applicare all'elemento a di A la funzione $c \in (\Pi x \in A)\ B(x)$ ottenendo un elemento g del tipo B(a). La regola di computazione e'

$$\frac{c \Longrightarrow \lambda(b) \qquad b(a) \Longrightarrow g}{Ap(c,a) \Longrightarrow g}$$

per computare Ap(c,a) si deve prima computare c ottenendo l'elemento canonico $\lambda(b)$ e poi calcolare b(a).

Allo stesso modo agli elementi di N si puo' associare l'operatore di recursione rec con regole di computazione

$$\frac{a \Longrightarrow 0 \qquad d \Longrightarrow g}{rec(a,d,e) \Longrightarrow g} \qquad \frac{a \Longrightarrow s(b) \qquad e(b,rec(b,d,e)) \Longrightarrow g}{rec(a,d,e) \Longrightarrow g}$$

per computare rec(a,d,e) si valuta prima a ottenendo 0 o s(b) per qualche $b \in N$. Poi, nel primo caso si prosegue valutando d (base), nel secondo si valuta e(b,rec(b,d,e)) (passo induttivo)

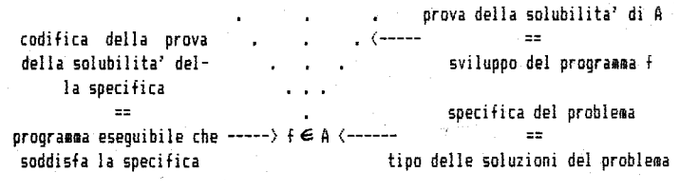
2. Tipi come specifiche di programmi.

Specificare un problema in type theory equivale a definire il tipo delle soluzioni del problema. Se poi si

riuscira' a dimostrare che il tipo A cosi' ottenuto non e' vuoto, cioe' se si potra' dimostrare che il giudizio $f \in A$ e' vero per qualche f, allora, essendo la dimostrazione costruttiva, avremo effettivamente costruito f ed al tempo stesso dimostrato che esso risolve il problema specificato con A.

Naturalmente la specifica di un problema non e' la soluzione del problema, ma la definizione di cosa puo' essere accettato come soluzione del problema. Si possono infatti specificare problemi non solubili, cioe' tipi vuoti.

Una figura illustra il modo di procedere:



L'esecuzione del programma f consistera' quindi nell'applicazione delle regole di computazione sino ad ottenere un elemento canonico g del tipo A, o, alternativamente, potra' proseguire con la valutazione degli argomenti di g (fully evaluation).

In questo approccio due problemi si pongono immediatamente

1) c'e' un programma che soddisfa la specifica A ($?\in A$)?

2) il programma a soddisfa la specifica A ($a \in A$)?

Entrambi questi problemi risultano essere, in generale, non decidibili in type theory.

3.Le espressioni con arieta'.

Se si esaminano le espressioni usate per descrivere le regole di computazione si nota che esse sono costruite a partire da costanti e variabili tramite applicazioni. Ad esempio rec(a,d,e) e' ottenuta applicando la costante rec alle espressioni a,d ed e.

Inoltre ad ogni espressione e' associata una sua specifica funzionalita': il numero dei suoi argomenti; ad esempio rec-(.,.,.), a-(), d-(), e-(.,.).

Si puo' specificare meglio se per ogni espressione si introduce non solo il numero ma anche la funzionalita' dei suoi argomenti, cioe' la sua arieta'. Cosi' a rec

sara' associata l'arieta' $((),(),((),()))$.

A partire da costanti e variabili con arieta' si puo' costruire un sistema formale per trattare di espressioni con arieta'. Il risultato e' in qualche modo simile ad un lambda calcolo con tipi: in questo caso l'unico tipo e' l'arieta'. Tuttavia se ne differenzia fondalmentalmente perche' l'astrazione non viene introdotta come operazione primitiva ma tramite una definizione abbreviatoria. Infatti la nozione di definizione abbreviatoria svolge un ruolo centrale nello sviluppo del sistema formale.

Una definizione abbreviatoria e' una relazione binaria tra stringhe

$$\text{definiendum} \underset{def}{=} \text{definiens}$$

Si puo' pensare al definiendum (cio' che si vuole definire) come ad una nuova denotazione per l'oggetto denotato dal definiens (cio' che gia' si possiede). La forma generale di un definiendum e'

$$d\ (y_1,\ldots,y_m)$$

dove in d (il nome parametrico) possono comparire delle variabili $x_1,\ldots,x_n$ (i parametri) di arieta' $\alpha_1,\ldots,\alpha_n$ e $y_1,\ldots,y_m$ (gli argomenti) sono variabili di arieta' $\beta_1,\ldots,\beta_m$.

Ad esempio, sia x una variabile di arieta' (), f e g siano variabili di arieta' $(())()$, allora $o[f,g]\ (x) \underset{def}{=} f(g(x))$ e' una definizione abbreviatoria: o[f,g] e' il nome parametrico, x e' l'argomento. Il significato inteso e' di definire la funzione composizione di f e g; il nome parametrico e' la notazione prefissa per la piu' usuale notazione infissa.

L'astrazione si puo' allora introdurre tramite una schema di definizione abbreviatoria

$$((x_1)..(x_n)\ c)(x_1,\ldots,x_n,y_1,\ldots,y_m) \underset{def}{=} c(y_1,\ldots,y_m)$$

dove c sta per una qualsiasi espressione.

L'introduzione delle definizioni abbreviatorie richiede di estendere la relazione di equivalenza testuale tra stringhe alla minima congruenza che ponga uguali definiendum e definiens. Una volta definita tale congruenza si possono dimostrare sulle espressioni i risultati standard relativi al lambda calcolo con tipi: il teorema di forma normale (esistenza ed unicita') e la decidibilita' dell'eguaglianza.

E' inoltre possibile introdurre un nuovo sistema formale, equivalente al primo proposto, che permette di dimostrare piu' facilmente la chiusura per sostituzione e di dare le condizioni cui deve sottostare l'insieme delle definizioni abbreviatorie per rendere decidibile

il problema se una stringa e' una espressione con arieta'.

Bibliografia

[1] P. Martin-Lof: 'An Intuitionistic theory of types: predicative part'
Proc. Logic Colloquium 1973,
ed. H.E. Rose and J.C.Shepherson,
North-Holland, Amsterdam (1975)
pp. 73-118.

[2] P. Martin-Lof: 'Constructive Matematics and Computer Programming'
Logic, Methodology and Philosophy of Science. VI.,
North-Holland, Amsterdam (1982).

[3] B. Nordstrom : 'Programming in Constructive Set Theory, Some Examples'
Proc. of the 1981 Conference on Functional Programming Languages and Computer Architectures.
ACM. pp 141-153.

[4] B. Nordstrom,K.Petersson:'Types and Specifications'
Congresso IFIP 83. pp 915-920.

[5] K. Petersson : 'An Introduction to the Programming System for Type Theory'
Proc. of the SEARC-Chalmers Workshop on Declarative Programming,April 83.

[6] P. Martin-Lof: 'Intuitionistic Type Theory'
Notes by G.Sambin of a series of Lectures given in Padova, June 1980.
Bibliopolis 1984.

[7] A. Bossi,S. Valentini: 'The expressions with arity'
di prossima pubblicazione