Indice

Capitolo 1

Introduzione

1.1 La triplice faccia della teoria dei tipi

La teoria dei tipi offre una base teorica a fondamento dello sviluppo di:

- **Matematica**: nella teoria degli insiemi;
- **Logica**: come fondamento dei connettivi logici e dei quantificatori, con trattazione mediate tecniche di *proof-teory* per dimostrarne la non falsità o non contradditorietà;
- **Informatica**: per la correttezza dei programmi, da una semantica operazionale a un certo tipo di operazioni.
  Con riferimento alla teoria degli insiemi, visto come linguaggio di programmazione funzionale, è possibile specificare con formule l'obiettivo di un programma e dimostrarne la correttezza attraverso la specifica.

La teoria dei tipi nasce per garantire la *Certified Proof Correctness*. Ovvero la correttezza dei programmi, volta a costruire gli assistenti automatici per le formalizzazioni.

1.2 Come nasce la teoria dei tipi

Gli errori di programmazione sono stati preponderanti alla nascita di metodi automatici, che assicurassero la correttezza del *software*. Alcuni di questi, degni di nota, sono stati:

- Incidente nel lancio dell'Apollo 11;
- Tragedie sanitarie: incidenti avvenuti tra il 1985-1987, in cui dei pazienti ricevettero una massiccia *overdose* di radiazioni e per la quali alcuni morirono;
- Errori di vita civile: riserva di solo due cifre per il campo età all'interno dei *database*. Ecco che una signora danese ricevette per il suo 107-esimo compleanno, una mail dalle autorità della scuola locale per iscriversi alla prima elementare.

Per la matematica la correttezza delle dimostrazioni è irrilevante solo quando la soluzione è certa (come accade con il cubo di *Rubik*, dove so che la soluzione è corretta quando ognuno dei lati è uniformemente colorato); e in generale questo è difficile che accada.
Un'esempio di problema, dove la soluzione non è certa, è il Teorema dei Quattro Colori, risolto da un *computer* e la cui prova di correttezza della dimostrazione fu data dal *proof assistant* Coq. Quest'ultimo basato sulla teoria dei tipi e intellegibile dall'essere umano.

Una citazione importante va al matematico Russo V.V. *Voevodsky*, vincitore della medaglia *Fields*. Esso si battè per la creazione di un *proof assistant*, per rendere le dimostrazioni da informali, per problemi complessi, a completamente formalizzate, con l'impiego della teoria dei tipi. I suoi studi trovano principale applicazione in campo algebrico e geometrico; ma i concetti emersi assunsero delle 1connotazioni più ampie. *Voevodsky*, difatti, si rese conto che formalizzare equivale a programmare. Ciò significa che la teoria dei tipi permette di vedere una dimostrazione come un programma.

Esiste la certezza assoluta per una certa teoria, esclusivamente, quando ha un numero di assiomi, accettati per fede, molto limitato. In quanto assiomalizzabile da un calcolatore.

Concetto chiave: formalizzare in una teoria dei tipi (come quella degli insiemi) equivale a programmare un programma.

1.3 Il Paradosso di Russell

La base della teoria dei tipi, compresa quella di *Martin-Löf*, si deve a B. *Russell*. Siamo nel 1907 quando nasce la teoria dei tipi, sviluppata nei *Principia Mathematica* da B. *Russel* assieme ad A.N. *Whitehead*. Tale teoria, intesa come logica e non informatica, nasce come soluzione alternativa alla teoria degli insiemi, di allora, con lo scopo di fondare la matematica su un sistema formale accettabile e non contradditorio.
Di seguito espongo un sistema contradditorio della teoria degli insiemi.

Linguaggio *L* di una teoria degli insiemi *F*

- L linguaggio del primo ordine (=, &, $\rightarrow$, $\vee$, $\forall x$, $\exists x$), con l'aggiunta del predicato $\in$ "appartiene"
- variabili VAR $\ni$ {x, y, z, w,...}

dove x, y, z sono da intendersi come insiemi e x ϵ y = "x appartiene a y".

All'interno di L c'è una teoria degli insiemi. Tra cui prende posto l'**assioma di comprensione di *Frege***, definito nel modo seguente:
Per ogni formula ϕ(x) vale che $\exists$z $\forall$y (y$\in$z $\Leftrightarrow$ ϕ(y)) [$\equiv \exists z\ z = \{x \mid \phi(x)\}$]

Teorema (o Paradosso) di *Russell*: la teoria F è contradditoria.

Dimostrazione:
ϕ(x) = x$\notin$x ($\equiv \neg$ (x $\in$ x))
Per l'assioma di comprensione $\exists$z z = {x | x$\notin$x} ($\exists$z $\forall$y (y$\in$z $\Leftrightarrow$ y$\notin$y))
Ponendo y=z ottengo che z$\in$z $\Leftrightarrow$ z$\notin$z, risulta una **contraddizione**.

L'assioma di comprensione è contradditorio perchè permette di formare insiemi che non appartengono a se stessi.

Come correggere la contraddizione?
La soluzione accettabile è porre agli insiemi una **gerarchia di tipi**. In questo modo l'assioma di comprensione diventa:
$\exists$z $\forall$y (y$\in$a $\rightarrow$ (y$\in$z $\Leftrightarrow$ ϕ(y)) $\equiv$ z = {x$\in$a $\mid$ ϕ(x)}

In questo modo non posso più creare il Paradosso di *Russell*.

Al momento questa teoria dei tipi non è utilizzata. Una sua evoluzione diretta è la teoria dei tipi di *Martin-Löf*.

Concetto chiave: l'idea di *Russell* è quello di costruire insiemi partendo da una gerarchia.

1.4 Idee principali nelle teorie di tipo moderne

Le teorie di tipo moderne (chiamate λ-calcolo tipato) nascono, nel corso degli anni '30, dalla combinazione della teoria di tipo di *Russell* con il λ-calcolo di *Church*.

1.4.1 Richiamo della teoria del λ-calcolo di *Church*

Ha origine dalla logica, è un linguaggio in grado di trattare le funzioni e rivolto alla loro formalizzazione. Consiste in un linguaggio formale, le cui componenti principali sono programmi chiamati termini (pensati come funzioni).
La grammatica è la seguente:

$$\text{t} := \text{x} \mid \text{b}_1(\text{b}_2) \mid \lambda\text{x.t}$$

Esempio di applicazione: tg(x) $\equiv \lambda$x.tg(x)

Regole di computazione di base

$$(\lambda x.t)(b) \rightarrow t[\frac{x}{b}] \qquad \frac{b_1 \rightarrow b_2 \qquad a_1 \rightarrow a_2}{b_1(a_1) \rightarrow b_2(a_2)} \qquad \frac{b_1 \rightarrow b_2}{\lambda x.b_1 \rightarrow \lambda x.b_2}$$

Si dice che un programma si riduca a un altro, cioè converge, solo se c'è una sequenza di riduzioni (applicazione di regole e/o assiomi), che connettono il primo programma con l'ultimo. Si parla, in questo modo, di **chiusura transitiva e simmetrica**, che si conclude quando il programma non è più riducibile. Quanto appena descritto può venire espresso nel seguente modo:

$t \rightarrow t'$ sse esiste un numero finito di passi per cui t si riduce a t', ovvero esiste

$b_1 \dots b_m$ t.c. $t \to b_1 \to b_2 \dots \to b_m \to t'$.

Il λ calcolo permette di codificare qualsiasi programma scritto in qualunque linguaggio (imperativo, dichiarativo, Java, C++, BASIC, ...). Tuttavia tale linguaggio non codifica solo programmi che terminano, ma anche programmi che non lo fanno. Un esempio di applicazione, per quest'ultima categoria, è un programma con computazione infinita: $\lambda x.x(x)$
$\lambda x.x(x)$ lo applichiamo a se stesso. Perció diventa $\Lambda \equiv (\lambda x.x(x))(\lambda x.x(x))$ che seguendo la computazione si riduce a:

$$x(x)[\frac{x}{\lambda x.x(x)}] \equiv (\lambda x.x(x))(\lambda x.x(x))$$

Dunque esiste una catena di $(t_i)_{i\in\mathbb{N}}$ di termini $t_i \to t_{i+1}$. Ciò significa che Λ non termina in qualunque linguaggio sia interpretato.

Λ risulta un buon metodo per rappresentare le funzioni, ma non è completo, rispetto all'intuizione matematica di funzione. È necessario, per questo, tipare le variabili; ovvero λx.x∈A→B(x∈A).

Concetto chiave: il λ-cacolo tipato, nato dal λ-calcolo "puro", è anch'esso un linguaggio di programmazione. Essendo tipato può essere trattato come una teoria degli insiemi.

1.5 Che cosa è un tipo?

Per rispondere a questa domanda è necessario fornire la semantica intuitiva di tipo. Per farlo è utile pensare alla teoria dei tipi come paradigma di fondazione sia logico che matematico che informatico.

Sintassi in teoria dei tipi moderna	**Sintassi in teoria degli insiemi**	**Sintassi in un linguaggio logico/per una logica (anche predicativo)**	**Sintassi in un linguaggio di programmazione**
A type	A set	A prop	A data type
a∈A	a∈A set	a∈A	a∈A

Tabella 1.1: Sintassi per i diversi paradigmi funzionali.

Per la sintassi:

- nella **teoria dei tipi moderna** a rappresenta un termine e A un tipo;
- nella sintassi in una **teoria degli insiemi** a è un elemento e A un insieme. Coincidendo con la corrispondenza originale in mente da *Russell*.
- nella sitassi in un **linguaggio logico** a rappresenta una proposizione di A, inteso come insieme di tipo delle sue dimostrazioni. Perciò un *proof-term* affermante come la proposizione di A sia vera.

- nella teoria in una sintassi di un **linguaggio di programmazione** a rappresenta un programma e A una specifica.

Dunque quando parliamo di tipo ci riferiamo a un insieme, una proposizione o *data type*, a seconda dell'applicazione di tipo che si ha in mente.

Dal punto di vista logico non si hanno solo proposizioni, ma anche predicati. Parlare solo di tipo non risulta quindi sufficiente. Per questo se si vuole rappresentare non una proposizione, ma un predicato A(x) si usa la seguente sitassi: **A(x) prop[x∈D]**.
Dalla logica si sa che i predicati ϕ(x) hanno x senza un dominio specifico, perchè la sintasssi non determina che cosa è in x. Al seguito di tutto questo i predicati hanno una variabile che deve essere tipata come **ϕ(x) prop[x∈D]**.
Dunque (definizione di predicato)

$$\exists\ \mathbf{z} \qquad \mathbf{z=\{x\in a|\ \phi(x)\}} \qquad \equiv \qquad \mathbf{\phi(x) prop[x\in a]}$$

Quanto appena definito da origine al concetto di **tipo dipendente**, nel quale vengono tipate tutte le variabili che appartengono ad una **famiglia di tipo**.

Concetto chiave: le famiglie di tipo sono indispensabili per rappresentare il concetto di predicato. Di seguito ho riassunto in forma tabellare le diverse famiglie.

di tipo	negli insiemi	in logica	dati dipendenti
A(x) prop[x∈D]	A(x) set[x∈D]	A(x) prop[x∈D]	A(x) datatype[x∈D]

Tabella 1.2: Famiglia di tipi.

Il concetto di tipo dipendente è stato introdotto per la prima volta da *Martin-Löf*. *Russell* si era limitato a definire esclusivamente il concetto di funzione proposizionale dipendente da un tipo.

1.6 Esempi di tipi

A type	A set	A prop	A data type
N_1 singoletto	l'insieme singoletto	*tt* costante vero	tipo Unit
N_0 vuoto	l'insieme vuoto	$\perp$ costante falso	vuoto come data-type
BxC (tipo prodotto)	l'insieme prodotto cartesiano dell'insieme B con l'insieme C	B&C congiunzione della proposizione B e della proposizione C	tipo prodotto cartesiamo (come in *set theory*)
B+C (tipo somma binaria)	l'insieme unione disgiunta dell'insieme B con l'insieme C	B$\vee$C disgiunta della proposizione B e della proposizione C	tipo unione disgiunta con codifica
B$\rightarrow$C	l'insieme delle funzioni dall'insieme B verso l'insieme C: A$\rightarrow$B $\equiv$ {f | f: B$\rightarrow$C}	B$\rightarrow$C, implicazione della proposizo-ne B e della proposizione C	insieme delle funzioni dal *datatype* B al *datatype* C

Tabella 1.3: Famiglia di tipi.

1.6.1 I tipi dipendenti

A(x)type[x$\in$B]
tipo indiciato $\prod_{x \epsilon B} C(x)$
tipo somma disgiunta indiciata $\sum_{x \epsilon B} C(x)$

A(x)set[x$\in$B]	A(x)prop[x$\in$B]	A(x)datatype[x$\in$B]
$\{f : B \rightarrow \coprod_{x \epsilon B} C(x)\}$ $\coprod_{x \epsilon B} C(x) = \{b, c \mid b \epsilon B \quad c \epsilon C(b)\}$	$\forall x \epsilon B \quad C(x)$	tipo prodotto indiciato come in *set theory* (non esiste un *data-type* specifico)
$\dot{\bigcup}_{x \epsilon B} C(x)$ $\coprod_{x \epsilon B} C(x) = \{b, c \mid b \epsilon B \quad c \epsilon C(b)\}$	$\exists x \epsilon B \quad C(x)$	non è primitivo deriva sempre dalla *set theory*

Tabella 1.4: Tipi dipendenti.

Concetto chiave: lo slogan principale della teoria dei tipi è quello di tipare le variabili in un linguaggio formale set teorico/computazionale.

Esite anche il **tipo uguaglianza**:

- intensionale: Id(B,c,d);
- estensionale: Eq(B,c,d).

Introdotte da *Martin-Löf*.
E i costrutti degli **universi**, in cui U è universo di proposizioni e di insiemi.

1.7 Regole paradigmatiche per caratterizzare la teoria dei tipi

La teoria dei tipi è stata formalizzata usando la nozione di **giudizio**, dove si asserisce qualcosa come vero.
Ci sono quattro forme di giudizio (nelle quali Γ identifica il contesto):

- **A type**[Γ]: A è un tipo, possibilmente indicato da variabili nel contesto Γ, dipendente da Γ stesso. Rappresenta il giudizio di tipo.
- **A = B type**[Γ]: il tipo A dipendente da Γ è uguale al tipo B dipendente da Γ. Rappresenta il giudizio di uguaglianza di tipo.
- **a** $\in$ **A type**[Γ]: a è un elemento del tipo A, possibilmente indiciato, ovvero dipendente da Γ e dalle sue variabili di contesto. Un esempio di tipo dipendente è l'array, che ha termini di funzioni che dipendono da Γ. Invece il termine non è dipendente quando si parla di funzione costante senza variabili.
- **a = b** $\in$ **A type**[Γ]: a come elemento del tipo A dipende da Γ ed è uguale in modo definizionale/computazionale al termine b. Quest'ultimo, difatti, è elemento del tipo A dipendente da Γ.

All'interno di ogni singolo giudizio si lavora con la teoria dei tipi.
I giudizi solo esclusivamente asserzioni, dicono solo qualcosa quando è vero (non si usano i quantificatori). Essi limitano le frasi che si possono fare per codificare la logica intuizionistica.

1.7.1 Simbolo $\in$

Il significato di a$\in$A in teoria dei tipi è differente da quello insiemistico. Espongo il concetto con un esempio trattato a lezione:

$$1 \in Nat \tag{1.1}$$

- In ***set theory*** usuale $\in$ è tra insiemi. Nell'equazione 1.1, 1 rappresenta lui stesso un'insieme e Nat l'insieme dei numeri Naturali. Risulta vero che $1 \equiv \{\varnothing\}$, poichè $0 \equiv \varnothing$.
- Invece in **teoria dei tipi** (di *Martin-Löf* come di *Russell*) 1 rappresenta un elemento ma non un tipo e Nat il tipo dei Naturali. Vi è dunque la distinzione tra elemento e tipo (come esiste quella tra programmi e tipi).

1.7.2 Uguaglianza estensionale

Specifico $\mathbf{a = b} \in \mathbf{A}[\Gamma]$ come l'uguaglianza computazionale/definizionale, che viene data come primitiva e non va confusa con l'uguaglianza proposizionale/estensionale tra a e b.
L'uguaglianza proposizionale a = b è rappresentata non da un giudizio, che asserisce solo ciò che è vero, ma bensì da un tipo Eq(A,a,b) che può anche essere senza termini e/o essere falso, dal punto di vista logico.

Visti come programmi, a e b rappresentano lo stesso programma. In λ-calcolo a$\rightarrow$b oppure b$\rightarrow$a (si riducono). Inoltre a e b possono essere sia termini finali che trovarsi in mezzo alla computazione.

1.7.3 Generazione di contesti

Esiste anche un quinto giudizio ausiliario:

$$\varnothing \ \ contesto \qquad\qquad F-C) \quad \frac{A \ \ type[\Gamma]}{\Gamma, x \in A \ \ cont}$$

Il giudizio *F-C* permette di generare i contesti. Tale giudizio, a differenza dei primi quattro, rimane immutato in ogni teoria dei tipi.

1.8 Esempio di tipo: il tipo singoletto

Di seguito espongo alcune regole principali del tipo singoletto.

Formazione

$$S) \quad \frac{[\Gamma] \ \ cont}{N1 \ \ type[\Gamma]}$$

Permette di dire che cosa è un tipo.

Introduzione

$$I-S) \quad \frac{[\Gamma] \ \ cont}{* \in N1 \ \ type[\Gamma]}$$

Sia N_1 in ogni contesto Γ, partendo da contesto $\varnothing$, permette di formare i termini.

Eliminazione

$$E-S) \quad \frac{t \in N1 type[\Gamma] \quad M(z) type[\Gamma, z \in N_1] \quad c \in M(*) type[\Gamma]}{El_{N1}(t,c) \in M(t) type[\Gamma]}$$

El trattasi di costruttore di funzioni e $M[t] = M(z)[\frac{z}{t}]$.

Conversione

$$C - D) \quad \frac{M(z)type[\Gamma, z \in N_1] \quad c \in M(*)type[\Gamma]}{El_{N1}(*, c) = C \in M(*)type[\Gamma]}$$

La conversione rende possibile l'applicazione della regola di eliminazione introducendo delle uguaglianze.

Concetto chiave: le regole (S), (I-S), (E-S) e (C-D) hanno una spiegazione computazionale, e riguardano la compatibilità tra tipi, ma non da caratterizzare il tipo dei tipi.
Inoltre il tipo singoletto non è dipendente.