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



# TIPI E POLIMORFISMO NEI LINGUAGGI DI PROGRAMMAZIONE

MARIO COPPO
Dipartimento di Informatica Università di Torino

## 1. Introduzione.

In queste note analizzeremo alcuni aspetti della nozione di tipo nei linguaggi di programmazione, discutendo i collegamenti con lo sviluppo di tale nozione in logica matematica. Quello che si vuole mettere in rilievo e' come alcuni sistemi e risultati della logica abbiano avuto influenza, diretta o indiretta, sullo sviluppo della nozione di tipo in informatica e come certe metodologie, tipiche della logica matematica, si possano applicare allo studio di concetti di linguaggi di programmazione, come appunto i tipi.

L'utilita' di una partizione dei valori in tipi viene evidenziata , in informatica, da problemi strettamente legati alla pratica della programmazione, come quella di assicurare che certi operatori vengano sempre applicate ad argomenti su cui sono in grado di agire.

Notiamo che la struttura base di un elaboratore tradizionale e' tipicamente senza tipi . Una nozione di tipo era pero' presente gia' nei primi linguaggi ad alto livello come il FORTRAN anche se in modo ancora piuttosto primitivo. La tendenza a dare sempre maggiore enfasi alla struttura di tipi si e' pero' venuta accentuando in tempi piu' recenti nei linguaggi come ALGOL o PASCAL e, infine ADA. Una forte struttura di tipi, infatti, favorisce un approccio meno operazionale e piu' astratto alla programmazione, che ha spesso benefiche influenze sullo stile di programmazione e sulla correttezza dei programmi.

Un campo in cui la nozione di tipo viene ad assumere un aspetto particolarmente interessante e' quello dei linguaggi funzionali. Un aspetto caratteristico di questi linguaggi e' quello di consentire lo

sviluppo di funzioni di ordine superiore (in grado cioe' di avere funzioni come argomenti e di restituire funzioni come risultati), fornendo cosi' degli strumenti alternativi (e piu' potenti) a nozioni tipiche dei linguaggi imperativi come variabili e assegnazioni.

Il primo di tali linguaggi, il LISP (che venne suggerito dal λ-calcolo di Church(1941)), e' un linguaggio senza tipi in cui non esiste alcuna differenza tra funzioni ed argomenti ed ogni applicazione, purche' operativamente accettabile, e' possibile. Questo permette una grande liberta' di programmazione ma ha anche degli aspetti negativi, simili a quelli gia' discussi nel caso dei linguaggi tradizionali. Un problema interessante e' quindi quello di sviluppare nozioni di tipo che consentano di recuperare i vantaggi dei linguaggi tipati senza perdere, in modo essenziale, la flessibilita' e generalita' della programmazione funzionale senza tipi. Questo ha portato allo studio di sistemi di tipi sofisticati che sono stati a volte suggeriti da nozioni sviluppate nell'ambito della logica matematica o ne hanno comunque subito l'influenza. E' il caso, per esempio, del linguaggio ML (Gordon ed al.(1979)).

La nozione di tipo in logica e' stata introdotta da Whitehead e Russell, nei *Principia Mathematica* , per stratificare in modo gerarchico l'universo degli insiemi , onde evitare i ben noti paradossi dovuti all'esistenza di insiemi come $\{X|X\in X\}$. Una stratificazione in tipi, sulla linea quella di Whitehead e Russel, puo' anche essere applicata alla nozione di funzione definita in modo costruttivo. Church(1940) fu forse il primo a sviluppare un sistema in cui e' presente una gerarchia di funzionali λ-definibili organizzati secondo una struttura di tipi. Anche Curry(1934) aveva sviluppato un sistema per organizzare gli oggetti della logica combinatoria secondo una nozione di funzionalita'.

La nozione di gerarchica di funzionali costruttivi di tipo finito venne poi sfruttata da Godel(1958) per dimosrare la consistenza dell'aritmetica attraverso una interpretazione nel linguaggio dei funzionali. Questo approccio venne esteso successivamente a sistemi di ordine superiore (si veda Kreisel(1959)).

Un altro aspetto che stimolo' molto lo studio dei tipi in logica e' l'analogia (scoperta da Curry e Howard) tra tipi e formule logiche, secondo cui una formula e' vista come il tipo della sua dimostrazione.



Questa analogia e' stata sfruttata nella dimostrazione di proprieta' delle deduzioni come quella di normalizzazione.

Uno dei sistemi di tipi piu' interessanti da un punto di vista informatico, il λ-calcolo tipato del II ordine, e' una restrizione (non essenziale) del sistema F di Girard(1972) che fu sviluppato per dare un'interpretazione funzionale dell'analisi e dimostrarne la consistenza attraverso un teorema di normalizzazione..

E' interessante notare, a questo proposito, come il λ-calcolo del II-ordine (cosi' come altri sistemi) sia stato poi reintrodotto, indipendentemente, partendo da motivazioni puramente informatiche (Reynolds(1974)) .

In queste note, partendo da un linguaggio funzionale senza tipi (il λ-calcolo di Church(1941)) introdurremo alcune nozioni di tipo (di complessita' crescente) di cui studieremo le principali proprieta' sintattiche e semantiche, viste soprattutto in funzione della loro applicazione informatica.

Bisogna puntualizzare che questa rassegna non vuol assolutamente essere esaustiva. In particolare la nozione di tipo che viene sviluppata in queste note riguarda solamente sistemi in cui i tipi non dipendono esplicitamente dai termini. Vengono quindi esclusi tutti quei sistemi che incorporano questa nozione come l'approccio di Martin-Lof (1975) e i sistemi derivati (vedi, per esempio Constable e Zlatin(1984)) e, piu' recentemente, la teoria delle costruzioni di Coquand e Huet(1985).

## 2. λ-calcolo senza tipi

Introduciamo il nostro linguaggio nella sua versione senza tipi, richiamandone brevemente le principali proprieta' sintattiche e semantiche. Per una trattazione completa e rigorosa rimandiamo a Hindley e Seldin(1986) o Barendregt(1981/4).

### 2.1 Linguaggio.

#### Sintassi

Sia $\mathcal{C}$ un'insieme di <u>costanti</u> e $\mathcal{V}$ un'insieme di <u>variabili</u> . L'insieme $\Lambda$ dei <u>termini</u> e' definito da:

$c \in \Lambda$ se $c \in \mathcal{C}$

$x \in \Lambda$ se $x \in \mathcal{V}$

$(MN) \in \Lambda$ se $M,N \in \Lambda$ (applicazione)

$\lambda x.M \in \Lambda$ se $x \in \mathcal{V}$ e $M \in \Lambda$. (astrazione)

Useremo le seguenti abbreviazioni standard:

$\lambda x_1 ... \lambda x_n.M$ per $\lambda x_1.(...(\lambda x_n.M)...)$ .

$MN_1...N_n$ per $((...(MN_1)...)N_n)$

La scelta delle costanti e' arbitraria ,sia pure con qualche cautela (vedi piu' avanti). Supporremo, negli esempi, che $\mathcal{C}$ contenga le costanti intere ( <u>0</u> , <u>1</u> , <u>2</u> ,...) e booleane ( <u>true</u> , <u>false</u>) e le operazioni base su di esse ( + , * , ..., <u>and</u> , <u>or</u>, <u>not</u>, ..).

<u>Riduzione</u>

La regola fondamentale di valutazione e' la β-regola rappresentata dallo schema di assioma

(β) $(\lambda x.M)N \rightarrow_\beta M[N/x]$

dove M[N/x] rappresenta la sostituzione di x con N in M, a meno di un'eventuale ridenominazione delle variabili legate in M in caso di conflitto di nomi. Indicheremo con $\rightarrow_\beta$ la relazione ( in $\Lambda \times \Lambda$) ottenuta da (β) per chiusura sui contesti. Indicheremo inoltre con $\rightarrow\!\!\rightarrow_\beta$ la chiusura riflessiva e transitiva di $\rightarrow_\beta$ e con $=_\beta$ la relazione di equivalenza generata da $\rightarrow\!\!\rightarrow_\beta$ . Se $M \rightarrow\!\!\rightarrow_\beta N$ diremo che M <u>si riduce</u> a N . Per una definizione formale di $\rightarrow_\beta$ e $=_\beta$ si vedano i riferimenti citati.

Indicheremo con (λ) la teoria equazionale generata da $=_\beta$ e,in generale, il relativo sistema formale.

Un altro assioma che considereremo e' il seguente

(η) $\lambda x.Mx \rightarrow_\eta M$ (se x non occorre libero in M)

che rappresenta l'estensionalita'. $\rightarrow\!\!\rightarrow_{\beta\eta}$ e $=_{\beta\eta}$ sono le relazioni di riduzione ed eguaglinza generate da (βη) . Indicheremo con (λη) il sistema corrispondente.



Una proprieta' fondamentale della riduzione e' data dal seguente *Teorema* (Church-Rosser). Se $M \rightarrow\!\!\rightarrow_{\beta(\eta)} P$ ed $M \rightarrow\!\!\rightarrow_{\beta(\eta)} Q$ esiste un termine Z tale che $P \rightarrow\!\!\rightarrow_{\beta(\eta)} Z$ e $Q \rightarrow\!\!\rightarrow_{\beta(\eta)} Z$ .

Per il teorema di Church-Rosser, la forma normale di un termine, se esiste, e' unica. Infatti,siano P e Q due forme normali di M : essendo P e Q irriducibili si deve avere P≡Q). Quindi due forme normali distinte non sono eguagliabili . Cio' garantisce che la riduzione e' una buone "regola di calcolo". Inoltre, poiche' esitono infinite forme normali distinte (ex: $\lambda x.\lambda y.x$ e $\lambda x.\lambda y.y$ ) il sistema e' consistente (nel senso che non tutte le eguaglianze sono vere).

Possiamo anche considerare delle regole di riduzione per le costanti, del tipo <u>succ n</u> --> <u>n+1</u> , dove <u>succ</u> rappresenta la funzione successore. Poniamo pero' il vincolo che tali regole, come nel caso di quella dell'esempio, non distruggano la proprieta' di Church-Rosser e non introducano la possibilita' di riduzioni infinite (come, per esempio, una regola del tipo $c_1c_2 \rightarrow c_1c_2c_2$ ).

Con --> indicheremo, in generale, l'unione di $\rightarrow_\beta$ con le eventuali regole per costanti . Inoltre indicheremo con $\rightarrow\!\!\rightarrow$ la sua chiusura riflessiva, transitiva e con = la relazione di equivalenza generata da $\rightarrow\!\!\rightarrow$.

Introduciamo alcune definizioni di base. Un termine e' <u>riducibile</u> se contiene qualche sottotermine della forma $(\lambda x.M)N$ ed e' in <u>forma normale</u> se non e' riducibile. Un termine M e' <u>normalizzabile</u> se $M \rightarrow N$ dove N e' in forma normale ed e' <u>fortemente normalizzabile</u> se non e' possibile definire una sequenza infinita di riduzioni partendo da M ( cioe' se ogni possibile riduzione che parte da M conduce alla sua forma normale).

Per esempio:

$\lambda x.\lambda y.x(xy)$ e' in forma normale ,

$(\lambda x.x)(\lambda x.\lambda y.x(xy))$ no , ma e' fortemente normalizzabile

(infatti $(\lambda x.x)(\lambda x.\lambda y.x(xy)) \rightarrow \lambda x.\lambda y.x(xy)$ ).

$(\lambda x.xx)(\lambda x.xx)$ non e' normalizzabile

(infatti $(\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx) \rightarrow ...$ )

$(\lambda x.\lambda y.y)((\lambda x.xx)(\lambda x.xx))$ e' normalizzabile (poiche' si riduce a $\lambda y.y$ applicando ($\beta$) all'intero termine), ma non fortemente perche' $(\lambda x.\lambda y.y)(\underline{(\lambda x.xx)(\lambda x.xx)}) \dashrightarrow (\lambda x.\lambda y.y)((\lambda x.xx)(\lambda x.xx)) \dashrightarrow \ldots$ indefinitivamente (riducendo il sottotermine sottolineato).

## 2.2 Alcune proprieta'.

### Punti fissi

Si definisca $Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$. E' un semplice esercizio verificare che $YM =_\beta M(YM)$ per ogni termine M ( in particolare $(YM) \dashrightarrow_\beta M(YM)$) . Quindi , dato un termine M, esiste sempre un termine P che rappresenta un punto fisso di M ( cioe' tale che $P =_\beta (MP)$ ). In particolare si ha $P \equiv (YM)$. Y viene detto combinatore di punto fisso . Si puo' dire, inoltre, che YM rappresenta il minimo punto fisso di M, in un senso che verra chiarito da considerazioni semantiche. Y consente quindi la definizione di funzioni per ricorsione (si veda 2.4). Notiamo infine che Y non e' un termine normalizzabile.

### Funzioni rappresentabili

E' possibile rappresentare i naturali e funzioni su di essi nel $\lambda$-calcolo puro ( cioe' senza l'uso delle costanti). Sia $n \in N$ , dove N denota l'insieme dei naturali. Si definisca

$$\mathbf{n} \equiv \lambda f.\lambda x.f^n(x)$$

dove $f^n(x)$ denota $f(\ldots(f\ x)\ldots)$ (n volte). In particolare $\mathbf{0} \equiv \lambda f.\lambda x.x$ .

Un termine M rappresenta una funzione parziale $f:N^k \to N$ $(k>0)$ se

1. $f(n_1,\ldots n_k) = m \Rightarrow M\mathbf{n}_1\ldots\mathbf{n}_k \dashrightarrow_\beta \mathbf{m}$
2. $f(n_1,\ldots n_k)$ indefinita $\Rightarrow M\mathbf{n}_1\ldots\mathbf{n}_k$ non e' normalizzabile

Per esempio

- + e' rappresentato da $+ \equiv \lambda p.\lambda q.\lambda f.\lambda x.\ pf(qfx)$
- * e' rappresentato da $* \equiv \lambda p.\lambda q.\lambda f.\lambda x.\ p(qfx)$
- $\mathbf{c} \equiv \lambda p.\lambda q.\lambda r.\lambda f.\lambda x.\ p(\lambda y.qfx)(rfx)$ rappresenta il condizionale tale che $\mathbf{c}(0,m,n) = n$ e $\mathbf{c}(k+1,m,n) = m$ .

Esercizio: verificarlo. In generale abbiamo il seguente risultato.



*Teorema* (Kleene). Le funzioni rappresentabili in ($\lambda$) (e in ($\lambda\eta$) ) sono esattamente le funzioni ricorsive parziali.

Ogni funzione parziale ricorsiva , inoltre, e' rappresentabile mediante un termine fortemente normalizzabile.

## 2.3 Semantica

### Domini

Data la natura "senza tipi" di ($\lambda$), un dominio D adatto ad interpretare ($\lambda$) deve soddisfare l'equazione

$$D \cong A + [D \to D]$$

dove + rappresenta la somma disgiunta, A e' un dominio in cui vengono interpretate le (eventuali) costanti e $[D \to D]$ e' un insieme di funzioni da D in D. Per evidenti ragioni di cardinalita', D non puo' contenere tutte le funzioni da D in D.

La costruzione di un dominio di questo tipo e' stata proposta per la prima volta da D. Scott (Scott(1972)). La soluzione di Scott si basa essenzialmente su queste idee:

- Considerare strutture dotate di un ordine parziale ($\sqsubseteq$) con un elemento minimo ($\bot$) e tali che ogni catena crescente ammetta un limite (c.p.o.) $\bot$ rappresenta, in tal caso, "indefinito" mentre $d \sqsubseteq e$ ha il significato di "d contiene meno informazzione di e".
- Considerare (in $[D \to D]$) solo le funzioni f "continue", cioe' monotone (rispetto a $\sqsubseteq$) e tali che $f(\bigsqcup\langle x_i\rangle_{i\in\omega}) = \bigsqcup\langle f(x_i)\rangle_{i\in\omega}$ , dove $\langle x_i\rangle_{i\in\omega}$ e' una catena crescente in D.

Ricordiamo che, in un c.p.o. D , ogni funzione continua $f:D \to D$ ammette un minimo punto fisso d dato da $d = \bigsqcup\langle f^n(\bot)\rangle_{n\in\omega}$. Inoltre $fix = \lambda f \in [D\to D].\bigsqcup\langle f^n(\bot)\rangle_{n\in\omega}$ e' continua e, quindi, $fix \in D$ (nota: qui "$\lambda$" e' metateorico).

Se D e D' sono c.p.o. anche $D\times D'$ , $D+D'$ e $[D \to D']$ (dove $f \sqsubseteq g$ sse $\forall d \in D$ $f(d) \sqsubseteq g(d)$ ) sono c.p.o. . In particolare la categoria **c.p.o.** e' *cartesiana chiusa*.

Per maggiri dettagli sulla costruzione e sulle proprieta' di D si veda, ad esempio, Plotkin(1976).

Una altra caratteristica che viene spesso richiesta ai domini per l'interpretazione del λ-calcolo e, in generale, dei linguaggi di programmazione e' che siano ω-algebrici, cioe' che abbiano una infinita' numerabile di elementi finiti tali che $\forall x \in D\ x = \bigsqcup\{d \mid d \text{ e' finito e } d \sqsubseteq x\}$ , dove d e' finito se per ogni catena crescente $\langle x_i \rangle_{i\in\omega}$ $d \sqsubseteq \bigsqcup \langle x_i \rangle_{i\in\omega}$ implica $d \sqsubseteq x_i$ per qualche $i\in\omega$ . In questo senso sono usualmente intesi i "domini di Scott".

Interpretazione

Sia Env=( $\mathcal{V} \to D$) l'insieme degli ambienti (assegnazioni di valori in D alle variabili). Supponiamo che il c.p.o. A dei valori di base sia della forma $A = A_1 + \ldots + A_h + W$ dove $A_1,\ldots,A_h$ sono i c.p.o. dei valori di base e W e' un c.p.o. che contiene un solo elemento ? che rappresenta una condizione di "errore" (si veda piu' avanti). Negli esempi supporremo h=2 con $A_1 = N_\perp$, il c.p.o. piatto degli interi (ottenuto aggiungendo $\perp$ a N e assumendo $\perp \sqsubseteq n$ per ogni $n\in N$) e $A_2 = T_\perp$ ,il c.p.o. piatto dei booleani.

Per brevita' identificheremo gli elementi di A e [D→D] con le loro proiezioni in D e scriveremo, ad esempio, $d\in A$. Si noti che , in questo senso, $\perp_D$ appartiene a tutte le componenti. Quindi, ad esempio, $\perp_D \in N_\perp$.
L'interpretazione di un termine M e' una funzione $[\![\,]\!]: \Lambda \to Env \to D$ cosi' definita:

$[\![c]\!]\rho = \mathcal{K}(c)$ $(c\in\mathcal{C})$ dove $\mathcal{K}: \mathcal{C} \to D$ interpreta le costanti
$[\![x]\!]\rho = \rho(x)$ $(x\in \mathcal{V})$
$[\![(MN)]\!]\rho = [\![M]\!]\rho([\![N]\!]\rho)$ se $[\![M]\!]\rho \in [D\to D]$
? altrimenti ( cioe' se $[\![M]\!]\rho$ non e' una funzione)
$[\![\lambda x.M]\!]\rho = \lambda v\in D.\ [\![M]\!]\rho[v/x]$

Si puo' verificare che se M=N (nella teoria formale) allora $[\![M]\!]\rho = [\![N]\!]\rho$ (per ogni ρ). Inoltre si ha che $[\![Y]\!]\rho$ = fix , e questo da' una giustificazione semantica delle proprieta' di Y.

Notiamo che, a causa della presenza di costanti D non e' un modello di (λη) ma solo di (λ). Infatti ,per esempio, $[\![3]\!]\rho = 3$ mentre $[\![\lambda x.3x]\!]\rho = ?$ .

Se non si assume la presenza di costanti si puo' richiedere che D soddisfi $D \cong [D\to D]$ . Allora si puo' definire $[\![(MN)]\!]\rho = [\![M]\!]\rho([\![N]\!]\rho)$ e D diventa un modello di (λη). Per maggiori dettagli si veda Barendregt(1981/4).



## 2.4 (λ) come linguaggio di programmazione

Si puo' vedere (λ) come un linguaggio di programmazione puramente funzionale. In questi linguaggi, contrariarmente a quanto accade nel caso dei linguaggi di uso piu' comune (imperativi) non esiste la nozione di computazione come trasformazione di uno "stato" rappresentato dalla memoria (e, di conseguanza, la nozione di variabile come intesa, per esempio, in PASCAL). In questi linguaggi i "programmi" sono definiti usando unicamente la nozione di funzione e quella di applicazione di una funzione ai suoi argomenti. Esempi di tali linguaggi, che sono derivati, piu' o meno direttamente, dal λ-calcolo, sono la famiglia del LISP e, piu' recenti, linguaggi come ML (Gordon ed al.(1979)) e HOPE (Burstall ed al.(1980) ). Una caratteristica importante dei linguaggi funzionali e' che essi consentono la definizione di funzioni di ordine superiore (in grado quindi di accettare funzioni come argomenti e di restituire funzioni come risultati). Questo consente una notevole capacita' espressiva e permette di esprimere uno stile di programmazione elegante e chiaro. Per maggiori dettagli si vedano, per esempio, Burge(1978) o Henderson(1980).

Nel caso del λ-calcolo un programma e' rappresentato da un termine di Λ , l'applicazione e' rappresentata dall'applicazione in Λ e le regole di riduzione ( (β) piu' le eventuali regole per le costanti), corrispondono alle "regole di calcolo". La forma normale di un termine (se esiste) si puo' allora considerare come il risultato del programma rappresentato da quel termine. Per esempio

$$SQUARE \equiv \lambda n.\ n*n$$

definisce la funzione che esegue il quadrato di un numero (infatti $(SQUARE\ \underline{n}) \to_\beta \underline{n}*\underline{n} \to \underline{n^2}$ ). Nel seguito useremo semplicemente = per definire la relazione tra un "programma" ed il suo "risultato".

Nota. Ovviamente la definizione di un meccanismo pratico efficiente per la valutazione dei programmi pone tutta una serie di problemi teorici e pratici . Si veda, per esempio Arvind e al.(1984).

La disponibilita' di un operatore (interno) di punto fisso consente di definire funzioni mediante ricorsione. La funzione fattoriale, per esempio, puo' essere definita cosi':

$$FACT \equiv Y(\lambda f.\lambda n.\ \underline{if}\ (n=\underline{0})\ \underline{then}\ \underline{1}\ \underline{else}\ n*f(n-\underline{1}))$$

dove If-then-else puo' essere interpretata come una nuova costante (con una ovvia regola di riduzione associata).

Un altro esempio e' dato da

$$TWICE \equiv \lambda f.\lambda x.\ f(fx)\ .$$

TWICE rappresenta la funzione che itera due volte una funzione (f) su un suo argomento (x). Per esmpio (TWICE SQUARE $\underline{2}$ ) = $\underline{16}$ .

TWICE ha un significato anche quando viene applicata a se stessa. Infatti abbiamo che (TWICE TWICE) $\rightarrow_\beta$ $\lambda f.\lambda x.\ f^4(x)$ ( si ottiene, cioe' il funzionale che itera quattro volte una funzione). Se definiamo inoltre AUTO $\equiv \lambda x.xx$ abbiamo che (AUTO TWICE) $\rightarrow_\beta$ (TWICE TWICE). Quindi anche AUTO , che rappresenta l'auto-applicazione, puo' essere di utilita', in questo contesto. Ovviamente AUTO va usata con molta cautela perche' abbiamo visto che (AUTO AUTO) non ha forma normale (quindi ha valore "indefinito" nel modello).

E' inoltre possibile, data la natura senza tipi del λ-calcolo, scrivere programmi che portano,quando se ne tenta una valutazione, ad un uso non corretto delle funzioni. Per esempio (TWICE AUTO $\underline{2}$) porta a tentare una valutazione di ($\underline{2}$ $\underline{2}$). Si ha, infatti, $[\![$ (TWICE AUTO $\underline{2}$)$]\!]\rho$ = ?.

Notiamo, infine, che pur essendo il λ-calcolo un linguaggio puramente funzionale esso costituisce anche uno strumento utilissimo per lo studio dei principali concetti di linguaggi imperativi.

Anche in questo caso , per esempio, il λ-calcolo modella gli aspetti essenziali della nozione di applicazione di una funzione ai suoi argomenti. La β-riduzione, inoltre, e' essenzialmente la "copy rule" dell'ALGOL. E' noto, da un punto di vista piu' generale, che i linguaggi imperativi possono essere "tradotti" nel λ-calcolo (si veda, ad esempio, Landin(1965)).

Una nozione che si presta ad essere studiata nel contesto del λ-calcolo e' sicuramente quella di tipo.



## 3.Linguaggi con tipi.

### 3.1 Motivazioni

Sistemi basati sul λ-calcolo con tipi sono stati introdotti sia in logica matematica che in informatica. La motivazione essenziale e' quella di sviluppare sistemi che consentano di introdurre funzionali (costruttivi) di ordine superiore eliminando le patologie e l'eccessiva generalita' (per alcuni aspetti) del sistema senza tipi. E' utile dare una rapida panoramicha di queste motivazioni. Data la natura di queste note, porremo l'accento piu' sulle motivazioni informatiche.

#### Motivazioni logiche

In logica matematica lo sviluppo di sistemi basati sul λ-calcolo con tipi e' stata originato dalla esigenza di sviluppare sistemi logici molto generali che contenessero esplicitamente la nozione di funzione e non consentissero lo sviluppo di paradossi. La nozione di λ-calcolo con tipi e' stata introdotta da Church(1940) nella riformulazione del suo sistema fondazionale (che conteneva il λ-calcolo) che si era rivelato inconsistente (Church(1932/33) ). Sistemi con tipi erano stati anche studiati, nel contesto della Logica Combinatoria, da Curry(1934).
In questo filone possiamo anche inserire, piu' recentemente, i sistemi di Martin-Lof(1975).

Un altra motivazione per l'introduzione del λ-calcolo tipato e' stata quella di sviluppare una nozione costruttiva di funzionale di ordine superiore . Il primo approccio, in questo senso, e' stato quello di Godel(1958) con la definizione dei funzionali primitivi ricorsivi di tipo finito . Lo scopo principale di questo lavoro era la dimostrazione della consistenza dell'aritmetica, mediante una traduzione delle formule nel linguaggio dei funzionali . Questa dimostrazione e' stata estesa all'analisi da Spector (1962) e Girard(1972). Numerosi lavori si sono sviluppati in questa direzione: si veda, per esempio, Troelstra(1973).

#### Motivazioni informatiche

Una motivazone importante, dal punto di vista dell'informatica e' che i

linguaggi tipati consentono, in genere, uno stile di programmazione piu' chiaro e leggibile . Questo e' dovuto sia al fatto che molti termini di significato non molto evidente sono eliminati nei linguaggi tipati, sia al fatto che il tipo di un termine e' spesso molto utile per chiarire il significato del termine stesso. Su puo' osservare, per esempio, che il significato di un termine come λx.xx (che non ha tipo, in molti sistemi) non e' immediatamente comprensibile (cosa vuol dire applicare un termine a se stesso? a quali termini e' ragionevole applicarlo?).

Si e' verificato, inoltre, che i programmi scritti in linguaggi con una forte struttura di tipi contengono, in genere, una quantita' minore di errori . La verifica dei tipi di un programma, infatti, corrisponde ad una parziale verifica di correttezza, analoga al controllo dimensionale delle formule in fisica.

La presenza di una struttura di tipi, inoltre, puo' avere anche una positiva influnza sull'efficienza degli interpreti o compilatori dei relativi linguaggi. I programmi ben tipati, infatti, garantiscono l'impossibilta' che si verifichino, in fase di esecuzione del programma, errori determinati da una non corretta applicazione di una funzione ai suoi argomenti, come, per esempio, nel caso di (TWICE AUTO 2 ) in cui 2 viene applicato a se stesso. In questo modo e' possibile evitare tutta una serie di controlli in fase di esecuzione che sono invece indispensabili nel caso di linguaggi senza tipi (come in LISP).

## 3.2 Proprieta' di sistemi di tipi.

Introduciamo alcune proprieta' di sistemi di tipi che hanno significato, come vedremo, sia da un punto di vista logico sia da un punto di vista informatico.

### Capacita' espressiva

Indica la misura in cui la struttura di tipi permette di esprimere funzionalita' complesse e di sviluppare programmi di carattere generale, in grado cioe' di operare, in modo omogeneo, su dati di tipi differenti (polimorfismo). E' desiderabile che un sistema di tipi consenta la maggior espressivita' possibile, sia pure in modo controllato dai tipi.



Per esempio, l'obiettivo non deve essere tanto quello di escludere operatori come AUTO quanto piuttosto quello di fornire dei tipi sufficientemente espressivi da rendere comprensibile la funzionalita' di tale operatore ed indicare quindi i contesti in cui e' corretto usarlo. E' inoltre desiderabile che funzioni come TWICE, che non presuppongono un tipo particolare di argomenti ma solo una compatibilita' tra il primo ed il secondo argomento, siano usabili sul maggior numeri di argomenti possibile.

Una proprieta' che puo' dare un'indicazione della capacita' espressiva di un sistema di tipi puo' essere la classe delle funzioni rappresentabili (vedi 2.3) nel calcolo puro (cioe' senza un uso essenziale delle costanti di base, ma sfruttando solo l'applicazione funzionale). Come vedremo, pero', questa non puo' essere considerata una misura assoluta.

Nel λ-calcolo puro, per esempio, la capacita' espressiva e' grandissima, anche se incontrollata per assenza dei tipi. Infatti la classe delle funzioni rappresentabili e' la massima possibile.

### Proprieta' di terminazione

Un linguaggio di programmazione ha la proprieta' di terminazione se ogni programma che contiene solo operatori totali, assegnamenti (nel caso di linguaggi imperativi) , definizioni e applicazioni di funzioni senza ricorsione esplicita e senza effetti laterali (sempre nel caso di linguaggi imperativi) termina sempre. (Fortune et al. (1983)).

Un linguaggio ha quindi la proprieta' di terminazione se la possibilita' di non terminazione per i programmi puo' essere introdotta dall'uso esplicito di strumenti quali la ricorsione o l'iterazione (nei linguaggi imperativi). Un linguaggio con la proprieta' di terminazione garantisce che non e' possibile, con la sola struttura delle chiamate funzionali, introdurre la possibilita' di non terminazione (come puo' avvenire, per esempio, nel caso di (AUTO AUTO) ). Questo tipo di fenomeno puo' essere a volte difficile da controllare e rivelare. Inoltre questa proprieta' e' un buon indicatore (ovviamente tutt'altro che assoluto) della buona strutturazione del linguaggio.

Molti linguaggi fortemente tipati, come il PASCAL hanno la proprieta' di terminazione. Il λ-calcolo puro, ovviamente, non ha tale proprieta' (si

pensi a termini come Y o (AUTO AUTO) ).

## 4. Il λ-calcolo tipato semplice.

Il λ-calcolo tipato semplice formalizza l'idea di insiemi di funzioni organizzati gerarchicamente a partire da insiemi di base. Esso e' contenuto come sottosistema in Church(1940) e in Godel(1958).

### 4.1 Linguaggio

Sia K un insieme di tipi di base. Supporreme che K contenga almeno int, il tipo degli interi e bool, il tipo dei booleani. L'insieme T dei tipi semplici e' definito da

$K \subseteq T$

$\sigma\to\tau \in T$ se $\sigma,\tau \in T$.

L'operatore → e' il costruttore di tipi funzionali. Si potrebbero considerare anche altri costruttori di tipi come × (prodotto) + (somma disgiunta). Le proprieta' fondamentali del sistema, tuttavia, sono determinate soprattutto dai tipi funzionali, per cui ci limiteremo a considerare solo questi ultimi.

Notazione: $\sigma_1\to\ldots\to\sigma_n\to\tau$ e' una abbreviazione per $(\sigma_1\to(\ldots(\sigma_n\to\tau)\ldots))$.

Ad ogni variabile e costante viene associato un (unico) tipo (notazione $x^\sigma$, $c^\sigma$) . Siano $\mathcal{V}^\sigma$ e $\mathcal{C}^\sigma$, rispettivamente, l'insieme delle variabili e delle costanti di tipo σ (assumeremo le stesse costanti di 2.1, ovviamente con i tipi opportuni). L'insieme $\Lambda^\sigma$ dei temini di tipo σ e' definito da:

$\mathcal{C}^\sigma \subseteq \Lambda^\sigma$

$\mathcal{V}^\sigma \subseteq \Lambda^\sigma$

$(MN) \in \Lambda^\tau$ se $M \in \Lambda^{\sigma\to\tau}$ e $N \in \Lambda^\sigma$

$\lambda x^\sigma.M \in \Lambda^{\sigma\to\tau}$ se $M \in \Lambda^\tau$

Notazione: scriveremo $M^\sigma$, se necessario, per rendere esplicito il fatto che $M \in \Lambda^\sigma$. Inoltre ometteremo di specificare il tipo delle variabili quando non strettamente necessario.



Alcuni esempi:

$\lambda x^{int}.x \in \Lambda^{int\to int}$ (identita' su int)

$\lambda x^{int\to int}.\lambda x^{int}.f(fx) \in \Lambda^{(int\to int)\to int\to int}$ ( versione tipata di TWICE al tipo (int→int)→int→int )

Note. (i) Se vogliamo la versione di TWICE ad un altro tipo, come (bool→bool)→bool→bool, dobbiamo definire un altro termine $\lambda x^{bool\to bool}.\lambda f^{bool}.f(fx)$).

(ii)Non esiste una versione tipata di λx.xx (come di molti altri termini). Infatti x dovrebbe avere un tipo della forma σ e $\sigma\to\tau$ contemporaneamente, e questo e' impossibile. Notiamo che λx.xx e' in forma normale.

Si definisce una nozione di riduzione e di eguaglianza formale come in (λ) e (λη). Indicheremo con (λt), (ληt) i rispettivi sistemi. E' immediato verificare che se $M \in \Lambda^\sigma$ e $M\to_\beta N$ ( o $M\to_{\beta\eta} N$ ) anche $N \in \Lambda^\sigma$. Anche le regole per costanti,ovviamente, dovranno essere definite in modo da conservare i tipi. Il teorema di Church-Rosser si estende immediatamente a (λt), (ληt).

Il λ-calcolo tipato semplice corrisponde, sostanzialmente al modo di intendere i tipi del PASCAL. La differenza piu' importante e' che in PASCAL non si possono definire funzioni di ordine superiore ( e' solo permesso il passaggio di funzioni del primo ordine come parametri ). Questo e' anche l'approccio della maggir parte dei linguaggi derivati dal PASCAL come ADA.

### 4.2 Alcune proprieta'

Normalizzazione.

*Teorema.* Ogni termine di (λt) (o di ληt) ) e' fortemente normalizzabile. Questo teorema puo' essere visto come un corollario del teorema di normalizzazione forte per il λ-calcolo del II ordine la cui dimostrazione e' data in 6.2 . La prima dimostrazione e' dovuta a Tait(1967). Una versione piu' debole (con "normalizzabile" al posto di "fortemente normalizzabile") era stata dimostrata negli anni '40 da Turing (si veda Gandy(1980 I). Si veda anche Gandy (1980 II) per uno studio sulla relazione tra prove di normalizzazione e prove di normalizzazione forte.

In entrambi i casi la prova e' formalizzabile nell'aritmetica di Peano (si confronti con 6.2).

Una conseguenza di questo teorema e' il fatto che la $\beta(\eta)$-eguaglianza e' decidibile (contrariamente a quanto accade nel $\lambda$-calcolo puro). E' stato tuttavia dimostrato da Statman(1979) che il problema di decidere l'eguaglianza di due termini non e' ricorsivo elementare.

Rappresentabilita' di funzioni.

Sia $\sigma$ un tipo arbitrario. Definiamo $i \equiv (\sigma\to\sigma)\to\sigma\to\sigma$ : i rappresenta il tipo delle rappresentazioni (interne) degli interi. Definiamo

$$\mathbf{n}^i \equiv \lambda f^{\sigma\to\sigma}.\lambda x^{\sigma}.f^n(x) \quad \in \Lambda^i$$

($\mathbf{n}^i$ e' il corrispondente tipato di $\mathbf{n}$ definito in 2.2).
E' immediato verificare (lo lasciamo per esercizio), che e' possibile definire le versioni tipate di $+$, $*$, $\mathbf{c}$ introdotte in 2.3 . In particolare avremo $+ \in \Lambda^{i\to i\to i}$,...etc. .

Introduciamo la nozione di funzione rappresentabile nel $\lambda$-calcolo con tipi semplice come in 2.3, solo che il termine M che rappresenta una funzione $f:N^k\to N$ deve appartenere ora a $\Lambda^{i\to\ldots\to i}$.

Abbiamo allora il seguente risultato dovuto a Schwichtenberg(1976).

*Teorema*. Le funzioni rappresentabili in $(\lambda t)$ e $(\lambda\eta t)$ sono esattamente le funzioni generate da **0** e **1** usando le operazioni di addizione, moltiplicazione e condizionale.

Per esempio l'operazione di "predecessore" non e' rappresentabile, e nemmeno l'esponenziazione. Tali operazioni, tuttavia, sono rappresentabili in senso piu' debole (si veda Fortune et al.(1983). Altre semplici operazioni pero', come la sottrazione, non sono rappresentabili nemmeno in senso debole.

Da questi risultati, e dalle osservazioni precedenti, si puo' constatare che il $\lambda$-calcolo con tipi semplice possiede la proprieta' di terminazione, ma non ha una particolare potenza espressiva. Cio' e' dovuto sia alla semplicita' del linguaggio dei tipi sia alla rigida partizione dei termini in classi che fa si che non vi possano essere termini che posseggono contemporaneamente piu' tipi. Cio' impedisce la possibilita' di usare uno stesso termine ( per esempio l'identita' $\lambda x.x$ ) per rappresentare la stessa operazione su tipi differenti.



## 4.3 Formulae-as-types

Vi e' un'interessante analogia tra il $\lambda$-calcolo puro con tipi semplice ed il calcolo proposizionale intuizionista col solo connettivo $\Rightarrow$ (CPI) , che e' stata notata, indipendentemente, da Curry (Curry e Feys(1958)) e Howard(1980), e quindi sviluppata da vari altri autori (si veda Hindley e Seldin(1986) per riferimenti piu' ampi).

L'analogia si puo' assumere in questi termini. Assumiamo una fomulazione di (CPI) nello stile di deduzione natutale (Prawitz(1965)).
I tipi di $(\lambda t)$ corrispondono alle formule di (CPI), dove $\to$ si interpreta come $\Rightarrow$ e i tipi di base come proposizioni.
I termini di $(\lambda t)$ di tipo $\sigma$ corrispondono alle prove in (CPI) della proposizione corrispondente a $\sigma$ dove, in particolare l'applicazione $(M^{\sigma\to\tau}N^{\sigma})^{\tau}$ corrisponde alla regola di $\Rightarrow$-eliminazione (modus ponens) e l'astrazione $(\lambda x^{\sigma}.M^{\tau})^{\sigma\to\tau}$ corrisponde alla regola di $\Rightarrow$-introduzione.
Per esempio il termine $\lambda x^{\sigma}.x^{\sigma}\in\Lambda^{\sigma\to\sigma}$ corrisponde alla prova

$$\begin{array}{c}[\,\sigma\,]\\ \text{-------}\ (\Rightarrow\text{I})\\ \sigma\Rightarrow\sigma\end{array}$$

In particolare vale la seguente proprieta'.

*Proprieta'*. Una formula $\sigma$ e' provabile in CPI partendo da un insieme di premesse $\tau_1,\ldots,\tau_n$ se e solo se esiste un termine $M \in \Lambda^{\sigma}$ avente $x_1^{\tau}1,\ldots x_n^{\tau}n$ come uniche variabili libere.

Notiamo, in particolare, che non tutti i tipi contengono dei termini chiusi (cioe' senza variabili libere). Infatti un tipo che contiene un termine chiuso rappresenta una proposizione dimostrabile da un insieme vuoto di premesse e, quindi, una tautologia. Solo i tipi che corrispondono a tautologie contengono termini chiusi.
Per esempio $(\sigma\to\sigma)\to\sigma$ non contiene alcun termine chiuso.

Una $\beta$-riduzione del tipo $((\lambda x^{\sigma}.M^{\tau})^{\sigma\to\tau}N^{\sigma})^{\tau}$ $\to\beta$ $(M[N^{\sigma}/x])^{\tau}$ corrisponde ad una $\Rightarrow$-riduzione di una prova del tipo:

$$\dfrac{\begin{array}{c}[\sigma]\\ \mathfrak{D}_1\\ \tau\end{array}}{\sigma\Rightarrow\tau}\,(\Rightarrow I) \qquad \mathfrak{D}_2 \qquad \sigma \qquad \text{si } \Rightarrow\text{-riduce a} \qquad \begin{array}{c}\mathfrak{D}_2\\ \sigma\end{array} \qquad \begin{array}{c}\mathfrak{D}_1\\ \tau\end{array}$$

$$\dfrac{\sigma\Rightarrow\tau \qquad \sigma}{\tau}\,(\Rightarrow E)$$

Il teorema di normalizzazione per ($\lambda t$) e' quindi equivalente al teorema di normalizzazione delle deduzioni in CPI. Analogamente, tutte le proprieta' di ognuno dei due sistemi valgono anche nell'altro (si veda, ad esempio, Troelstra(1973) per un uso sistematico di questa proprieta').
Tale analogia si estende a varianti di ($\lambda t$) corrispondenti a sistemi logici piu' potenti, come ad esempio l'aritmetica (vedi Howard(1980)).

## 4.4 Semantica.

La natura stratificata dei tipi semplici consente che questi possano venire interpretati come insiemi.
In particolare, se in terpretiamo ogni tipo di base $k_i$ come un insieme $A_i$ possiamo definire, per ogni tipo $\sigma$ un insieme $A^\sigma$ nel seguente modo

$$A^{k_i} = A_i$$

$$A^{\sigma\to\tau} = A^\sigma \to A^\tau$$

dove $A^\sigma \to A^\tau$ e' l'insieme di <u>tutte</u> le funzioni da $A^\sigma$ in $A^\tau$ ( full type structure ).

Sia ora $\rho : \mathbf{U}_{\sigma\in T} \mathcal{V}^\sigma \to \mathbf{U}_{\sigma\in T} A^\sigma$ un ambiente che assegna ad ogni variabile un valore di tipo corrispondente. L'interpretazione dei termini di ($\lambda t$) si definisce nel modo seguente:

$[\![c]\!]\rho = \mathcal{K}(c)$ $\quad (c \in \mathbf{U}_{\sigma\in T} \mathcal{C}^\sigma)$ dove $\mathcal{K}$ interpreta le costanti

$[\![x]\!]\rho = \rho(x)$ $\quad (x \in \mathbf{U}_{\sigma\in T} \mathcal{V}^\sigma)$

$[\![M^{\sigma\to\tau} N^\sigma]\!]\rho = [\![M^{\sigma\to\tau}]\!]\rho([\![N^\sigma]\!]\rho)$ $\quad (\in A^\tau)$

$[\![\lambda x^\sigma . M^\tau]\!]\rho = \lambda v \in A^\sigma . [\![M^\tau]\!]\rho[v/x]$ $\quad (\in A^{\sigma\to\tau})$

Si puo' facilmente provare che se $M = N$ allora, per ogni ambiente $\rho$, $[\![M]\!]\rho = [\![N]\!]\rho$ . Quindi questa semantica e' adeguata.



Essendo la nozione di insieme estensionale, inoltre, questa interpretazione fornisce anche un modello di ($\lambda\eta t$).
Osserviamo che, a causa dei vincoli di tipo, possiamo escludere la presenza di errori di applicazione (si confronti con 2.3) .

E' stato osservato da vari autori (i primi sono stati D. Scott(1980) e J. Lambek(1980)) che ogni categoria cartesiana chiusa fornisce un modello di ($\lambda\eta t$) (e, quindi, di ($\lambda t$)). Viceversa, se ($\lambda\eta t$)+ e' il sistema ottenuto da ($\lambda\eta t$) aggiungendo la nozione di prodotto cartesiano e le relative costanti, ogni modello di ($\lambda\eta t$)+ definisce una categoria cartesiana chiusa. Inoltre una categoria cartesiana chiusa (cioe' un modello di ($\lambda\eta t$)) con un oggetto U tale che esista una retrazione da $U^U$ in U fornisce un modello del $\lambda$-calcolo senza tipi (Scott(1980)). Questi risultati mettono in rilievo l'interesse della nozione di categoria cartesiana chiusa nello studio della nozione di tipo e, in generale, della semantica dei linguaggi di programmazione.

## 4.5 Estensioni

Il sistema definito in 4.1 non puo' ancora essere considerato un vero linguaggio di programmazione a causa delle ipotesi fatte sulla natura delle costanti (si veda 2.1) e per la (relativamente) scarsa potenza espressiva del linguaggio puro. Accenniamo brevemente a tre interessanti estensioni del linguaggio base. Le prime due hanno motivazioni prevalentemente informatiche e sono incorporate in alcuni linguaggi di programmazione come ML (Gordon e al.(1979)), mentre la terza prevalentemente logiche.

### Operatore di punto fisso

L'operatore Y di punto fisso, non essendo normalizzabile, non ha ovviamente corrispondente tipato. Si puo' pero' aggiungere, per ogni tipo $\sigma$, una costante $\underline{Y}_\sigma \in A^{(\sigma\to\sigma)\to\sigma}$ con una regola di riduzione (derivata da Y):

$$(\underline{Y}_\sigma M^{\sigma\to\sigma}) \dashrightarrow (M^{\sigma\to\sigma}(\underline{Y}_\sigma M^{\sigma\to\sigma}))\ .$$

In questo modo si introduce nel sistema la possibilita' di definire

funzioni mediante ricorsione. Si verifichi, per esempio, che esiste un corrispondente tipato (con tipo int→int ) di FACT definito in 2.4 (si supponga di disporre, per ogni tipo $\sigma$, di una costante $\text{If-then-else}_\sigma \in \Lambda^{bool\to\sigma\to\sigma\to\sigma}$ ).

Abbreviazioni spesso usate per $\underline{Y}_\sigma(\lambda f^\sigma.M)$ sono: $\text{rec } f^\sigma.M$ o $\mu f^\sigma.M$.

Introducendo $\underline{Y}_\sigma$ non vale piu', ovviamente, il teoreme di normalizzazione , ma il linguaggio continua ad avere la proprieta' di terminazione nel senso di 3.2. La classe delle funzioni rappresentabili, se non si fa uso di $\underline{Y}_\sigma$, resta ovviamente la stessa che in ($\lambda$t).

Rispetto all'analogia delle formulae-as types l'introduzione di $\underline{Y}_\sigma$ al tipo $(\sigma\to\sigma)\to\sigma$ significa assumere vera la proposizione $(\sigma\to\sigma)\to\sigma$ il che porta immediatamente all'inconsistenza del sistema poiche' , con questa assunzione, ogni formula puo' essere dimosrata. Infatti, per ogni tipo $\sigma$, $\underline{Y}_\sigma(\lambda x^\sigma.x) \in A^\sigma$ .

Per quanto riguarda la semantica, non e' piu possibile interpretare i tipi come insiemi. Una interpretazione soddisfaciente dei tipi e' pero' possibile nella categoria **c.p.o.** (che e' cartesiana chiusa). In questo caso si interpretano i tipi di base k come c.p.o. $A^k$ (ricordiamo che ogni insieme puo' essere trasformato in un c.p.o. "piatto" aggiungendo un nuovo elemento $\perp$) ed interpretando $\sigma\to\tau$ con $[A^\sigma\to A^\tau]$. In questo caso $\underline{Y}_\sigma$ si interpreta come l'operatore di minimo punto fisso su $[A^\sigma\to A^\sigma]$. Si pone cioe' $[\![\underline{Y}_\sigma]\!] = \lambda f\in[A^\sigma\to A^\sigma].\bigsqcup\langle f^n(\perp_{A^\sigma})\rangle_{n\in\omega}$ .

Si noti che questo modello sfrutta solo il fatto che **c.p.o.** e' cartesiana chiusa e non richiede alcuna costruzione particolare.

Per alcune interessanti risultati che collegano l'interpretazione di un termine alle sue proprieta' di riduzione si veda Plotkin(1977).

Tipi ricorsivi

L'introduzione di tipi definiti ricorsivamente e' molto interessante del punto di vista informatico. Essa consente, in modo molto naturale, la definizione di strutture di dati complesse a partire da tipi di dati semplici. Ad esempio, sopponendo che di aver aggiunto ai tipi di ($\lambda$t) i costruttori × e + (vedi 4.1) con le ovvie costante ai tipi opportuni per costruttori ed estrattori, il tipo delle liste di interi si puo' rappresentare come il tipo intlist che soddisfa a

intlist = int + int×intlist

mentre il tipo degli alberi binari labellati inttree potrebbe essere descritto da

inttree = int + int×inttree×inttree.

In un linguaggio con tipi ricorsivi, inoltre, non e' piu' indispensabile definire l'operatore di punto fisso come una primitiva. Si assuma, infatti di definire, dato un tipo arbitrario $\sigma$, un tipo $\xi$ che soddisfa $\xi = \xi\to\sigma$. Assunto $x^\xi$ si ha $(xx)^\sigma$ (poiche' $\xi = \xi\to\sigma$) e $(f^{\sigma\to\sigma}(xx)^\sigma)^\sigma$ , da cui si ricava $(\lambda x^\xi.f(xx))^{\xi\to\sigma}$ e, quindi, anche $(\lambda x^\xi.f(xx))^\xi$ . di qui si ottiene subito $Y^{(\sigma\to\sigma)\to\sigma} \equiv \lambda f^{\sigma\to\sigma}.(\lambda x^\xi.f(xx))(\lambda x^\xi.f(xx)) \in \Lambda^{(\sigma\to\sigma)\to\sigma}$.

Si noti ancora che postulando un tipo $\zeta = \zeta\to\zeta$ , ogni termine del $\lambda$-calcolo puro ha (una versione tipata di) tipo $\zeta$. Cio' non e' piu' vero, pero', considerando le costanti. Resta infatti valido il fatto che un termine con un tipo non puo' mai determinare applicazioni scorrette.

Il teorema di normalizzazione, ovviamente, non vale piu' in questo sistema e si perde anche la proprieta' di terminazione. Le funzioni rappresentabili, ovviamente, sono le ricorsive parziali. I tipi ricorsivi restano pero' uno strumento molto utile nella pratica della programmazione, anche se la perdita della proprieta' di normalizzazione dimostra che vanno trattati con cautela.

Per quanto riguarda la semantica, **c.p.o.** fornisce un modello anche per questa estensione poiche' vi si possono costruire dei domini che soddisfano equazioni ricorsive utilizzando la tecnica richiamata in 2.4 .

Funzionali ricorsivi primitivi

Come ultima estensione presentiamo il linguaggio dei funzionali ricorsivi primitivi introdotti da Godel(1958). Ne diamo qui una formulazione equivalente dovuta a Schutte(1977) (il quale usa, pero' il linguaggio dei combinatori).

Si supponga di aggiungere a ($\lambda$t) , per ogni tipo $\sigma$, una costante $\underline{J}_\sigma \in \Lambda^{int\to(\sigma\to\sigma)\to\sigma\to\sigma}$ con la seguente regola di riduzione:

$\underline{J}_6 \underline{0} M^{6\to 6} N^6 \dashrightarrow N^6$

$\underline{J}_6 \underline{n+1} M^{6\to 6} N^6 \dashrightarrow M^{6\to 6}(\underline{J}_6 \underline{n} M^{6\to 6} N^6)$

cioe' $\underline{J}_6 \underline{n} M^{6\to 6} N^6 \dashrightarrow\!\!> M^n(N)$ (omettendo i tipi) e $\underline{J}_6$ rappresenta, quindi, un iteratore al tipo $6\to 6$.

Il teorema di normalizzazione vale anche per questo sistema (vedi Tait(1967)). Tale prova richiede un'induzione fino ad $\epsilon_0$ (vedi Schutte(1977)) e non e' quindi formalizzabile nell'aritmetica di Peano. La classe delle funzioni "programmabili" nel sistema (usando cioe' le costanti, $\underline{J}_6$ in particolare) coincide con la classe delle funzioni dimostrabilmente ricorsive nell'aritmetica.

Le motivazioni logiche di questa estensione, come gia' detto, erano quelle di fornire uno strumento per la dimostrazione della consistenza dell'aritmetica mediante una traduzione nel linguaggio dei funzionali (si vedano le opere citate). Da un punto di vista piu' informatico questo sistema e' interessante perche' fornisce un esempio di linguaggio in cui possono essere programmate solo funzioni totali ma sufficientemente potente per essere adeguato ad un uso pratico.

## 5.Assegnamento di tipi e polimorfismo

### 5.1 introduzione

Una delle limitazioni del λ-calcolo con tipi semplice, visto come linguaggio di programmazione, e' che non consente la definizione di operatori che possano essere applicati a dati di tipi differenti. Per esempio, nel sistema senza tipi, l'operatore TWICE ≡ λf.λx.f(fx) puo' essere applicato indifferentemente a SQUARE , a λf.(not f) (dove not e' considerata una costante) oppure a se stesso. In tutti i casi il significato dell'applicazione e' chiaro. TWICE rappresenta l'operazione di iterare due volte una funzione, indipendentemente dal suo tipo purche' ovviamente, il range ed il dominio della funzione coincidano. Nel sistema tipato, invece, bisogna introdurre un operatore TWICE per ogni tipo della funzione a cui lo si vuol applicare. Per esempio avremo:



$TWICE_{int} \equiv \lambda f^{int\to int}.\lambda x^{int}.f(fx) \in \Lambda^{(int\to int)\to int\to int}$

$TWICE_{bool} \equiv \lambda f^{bool\to bool}.\lambda x^{bool}.f(fx) \in \Lambda^{(bool\to bool)\to bool\to bool}$

$TWICE_{int2} \equiv \lambda f^{int2\to int2}.\lambda x^{int2}.f(fx) \in \Lambda^{(int2\to int2)\to int2\to int2}$

dove int2 ≡ (int→int)→int→int .

Nota che ($TWICE_{int2}$ $TWICE_{int}$) rappresenta l'operatore che itera quattro volte una funzione di tipo int→int.

Sarebbe pero' desiderabile, visto che le varie versioni di TWICE differiscono solo per i tipi, poter avere un'unica espressione valida per tutti i tipi della forma $(6\to 6)\to 6\to 6$.

Vi sono, nella letteratura,essenzialmente due proposte per fare cio'. la prima, che si rifa' alla nozione di schema di tipo, viene dicussa in questa sezione mentre la seconda, che conduce al λ-calcolo del II-ordine, un sistema molto piu' potente (ma di piu' difficile realizzazione), nella sezione successiva.

Il primo approccio e' quello di partire dal sistema senza tipi e di considerare i tipi, anziche come una partizione in classi dell'universo de valori, come predicati che esprimono certe proprieta' funzionali dei termini senza tipi. I tipi vengono allora "assegnati" ai termini di cui rappresentano correttamente le proprieta' funzionali mediante un sistema di regole formali, la cui adeguatezza garantisce la correttezza del sistema.

Per esempio un tipo come int→int rappresenta la proprieta' di un termine di restituire un risltato intero se applicato ad un intero. Questa proprieta' e' valida per λx.x+1 (funzione che opera correttamente solo su dati interi) cosi' come per λx.x (che opera correttamante su qualunqe dato). Ma λx.x ha anche le proprieta' rappresentate dai tipi bool→bool , (int→int)→int→int e da infiniti altri. E' possibile, allora, assegnare allo stesso termine piu' tipi, che ne reppresentano differenti (anche se omogenee) proprieta' funzionali.

In questo contesto e' naturale introdurre la nozione di "schema di tipo", aggiungendo delle variabili al linguaggio dei tipi. Uno schema di tipo 6 rappresenta tutte le sue possibili istanze con tipi propri (definiamo proprio un tipo se non contiene variabili), nel senso che un termine possiede la proprieta' rappresentata da 6 se possiede le

proprieta' rappresentate da tutte le istanze proprie di $\sigma$. Cosi' $\lambda x.x$, come osservato prima, ha tipo $t \to t$ (dove t e' una variabile).

Questo tipo di approccio ha avuto origine con la teoria della funzionalita' di Curry(1934) (si veda anche Curry e Feys(1958)), anche se le motivazioni fondazionali di Curry erano alquanto diverse. Lo stesso tipo di approccio e' poi stato sviluppato, del tutto indipendentemente (almeno agli inizi) nella definizione del linguaggio ML (si veda Milner(1978)).

Questo sistema di tipi, realizzato praticamente nei linguaggi ML (Gordon e al.(1979)) e HOPE(Burstall al.(1980) si puo' considerare, secondo l'opinione dell'autore, il piu' sofisticato sistema di tipi realizzato in un linguaggio reale e effettivamente utilizzato.

## 5.2 Il sistema formale

### Schemi di tipo

Definiamo l'insieme $T_S$ degli schemi di tipo, che chiameremo semplicemente tipi per brevita', in modo analogo a come avevamo definito T in 4.1, aggiungendo solo

$$V_T \subseteq T_S$$

dove $V_T$ rappresenta un insieme di variabili di tipo, che indicheremo con t,u,... . Un tipo proprio e' un tipo senza occorrenze di variabili di tipo (quindi appartenente a T). Un tipo con occorrenze di variabili viene detto anche polimorfo. Esempi di tipi polimorfi sono $t \to t$, $((t \to t) \to u) \to u$, $t \to int$,... .

### Regole di assegnazione

Una formula di assegnazione di tipo e' un oggetto della forma $M:\sigma$, dove $M \in \Lambda$ e $\sigma \in T_S$, che rappresenta il fatto che $\sigma$ e' un tipo per M (cioe' che M ha la proprieta' specificata da $\sigma$). Una base B e' un insieme di assunzioni della forma $x:\sigma$ dove $x \in V$. Per ogni $x \in V$ non puo' occorrere in B piu' di una assunzione di tipo per x. Un seguente e' un oggetto della forma $B \vdash M:\sigma$ che esprime il fatto che da B si puo' inferire il tipo $\sigma$ per M. Assumiamo che ad ogni costante c sia assegnato un unico tipo $\tau(c)$.



Le regole di assegnazione sono le seguenti:

$$\frac{}{B \vdash c:\tau(c)}$$

$$\frac{}{B \vdash x:\sigma} \quad \text{se } x:\sigma \in B$$

$$\frac{B \vdash M:\sigma \to \tau \qquad B \vdash N:\sigma}{B \vdash (MN):\tau} \ (\to E)$$

$$\frac{B \cup \{x:\sigma\} \vdash M:\tau}{B \vdash (\lambda x.M):\sigma \to \tau} \ (\to I) \quad \text{(assumendo che x non occorra in B)}$$

Le regole $(\to I)$ e $(\to E)$ corrispondono alle due regole di formazione dei termini. Diamo un esempio di deduzione:

$$\dfrac{\dfrac{\{x:int \to t\} \vdash x:int \to t \qquad \{x:int \to t\} \vdash \underline{3}:int}{\{x:int \to t\} \vdash (x\ \underline{3}):t}\ (\to E)}{\vdash (\lambda x.\ x\ \underline{3}):(int \to t) \to t}\ (\to I)$$

Esercizio: provare $\vdash (\lambda f.\lambda x.f(fx)):(t \to t) \to t \to t$. Abbiamo quindi, ad esempio, $\vdash (TWICE\ SQARE):int \to int$ e $\vdash (TWICE\ (\lambda f.(\underline{not}\ f))):bool \to bool$ dove pero', ora, TWICE e' lo stesso termine.

Per maggiori dettagli e formulazioni equivalenti si veda Hindley e Seldin(1986).

I termini che posseggono un tipo in questo sistema sono gli stessi di cui esiste una versione tipata in $(\lambda t)$. La classe delle funzioni rappresentabili, quindi, e' la stessa che in $(\lambda t)$. Notiamo pero' che questo sistema ha piu' capacita' espressiva per via del fatto che uno stesso termine corrisponde a piu' termini tipati. Continua anche a valere, ovviamente, il teorema di normalizzazione forte per i termini che posseggono tipi.

Si verifica facilmente che i tipi si conservano per riduzione (cioe' se $B\vdash M:\sigma$ e $M \to_\beta N$ vale anche $B\vdash N:\sigma$. I tipi non si conservano, pero' per espansione (cioe' $B\vdash M:\sigma$ e $N \to\!\!> M$ non implica $B\vdash N:\sigma$) e, in generale, per $=_\beta$. Per esempio $\vdash(\lambda x.x):t\to t$ e $(\lambda x.xx)(\lambda y.y) \to_\beta \lambda x.x$ , ma $(\lambda x.xx)(\lambda y.y)$ non ha tipo, poiche' $\lambda x.xx$ non ha tipo (si veda 4.1).

## 5.3 Schema di tipo principale

Definiamo una sostituzione come una funzione $s: V_T \to T_S$ ; s si puo' estendere ai tipi in $T_S$ in modo ovvio. Diciamo che una sostituzione s e' di base se $s(t)$ e' un tipo proprio per ogni variabile di tipo t. Se B e' una base $s(B)$ e' la base ottenuta applicando s a tutti i tipi in B.

Una prima ,ovvia, proprieta' delle deduzioni e' la seguente:

*Proprieta*. Se $B\vdash M:\sigma$ allora, per ogni sostituzione s, $s(B)\vdash M:s(\sigma)$.

Una proprieta' molto importante per le sue applicazioni e' la seguente:

*Teorema*. Dato un termine M, se $B\vdash M:\sigma$ per qualche base B e tipo $\sigma$ , esistono una base $B_p$ ed un tipo $\sigma_p$ che sono principali per M nel senso che:

1. $B_p\vdash M:\sigma_p$
2. per ogni B' e $\sigma'$ tali che $B'\vdash M:\sigma'$ si ha $B'=s(B_p)$ e $\sigma'=s(\sigma_p)$ per qualche sostituzione s.

Per esempio, $(t\to t)\to t\to t$ e' il tipo principale di TWICE (la base e' vuota). Per dare tipo all'applicazione (TWICE TWICE) basta costruire l'istanza $((t\to t)\to t\to t)\to(t\to t)\to t\to t$ della prima occorrenza di TWICE ottenendo il tipo $(t\to t)\to t\to t$ per (TWICE TWICE), che ne e' anche il tipo principale.

Questa proprieta' e' stata scoperta indipendentemenete da Curry(1969) e Hindley(1969). Lo stesso risultato e' stato provato indipendentemente da Milner(1978). Queste dimostrazioni usano sostanzialmente l'algoritmo di unificazione di Robinson e implicano il seguente corollario.

*Corollario*. Dato un termine M e' decidibile se ad M puo' essere assegnato un tipo ed esiste un algoritmo per trovare il suo tipo e la sua base principali (se esistono).



Grazie alle precedenti proprieta' i linguaggi che adottano questo sistema di tipi (come ML) permettono al programmatore di scrivere i programmi senza specificarne i tipi i quali vengono inferiti automaticamente dal verificatore statico (type-checker) in fase di compilazione (prima, quindi, della fase di esecuzione.) La proprieta' del tipo principale assicura che il tipo inferito e' il piu' generale possibile.

## 5.4 Semantica.

E naturale, in questo tipo di approccio, interpretare indipendentemente i tipi e i termini in un modello del $\lambda$-calcolo senza tipi, facendo poi vedere che le loro interpretazioni sono correlate correttamente secondo le regole di assegnamento.

Per semplicita' supporremo che i tipi di base siano solo int e bool e che il dominio di interpretazione soddisfi l'equazione:

$$D \cong N_\perp + T_\perp + [D\to D] + W$$

Interpreteremo ora un tipo proprio $\sigma$ come un opportuno sottoinsieme $[\![\sigma]\!]$ di D che contiene quei valori che posseggono le proprieta' funzionali espresse dal tipo . In particolare definiamo;

$$[\![int]\!] = \{ n \mid n\in N_\perp \text{ e } n\neq\perp \}$$

$$[\![bool]\!] = \{ v \mid v\in T_\perp \text{ e } v\neq\perp \} = \{true,false\}$$

$$[\![\sigma\to\tau]\!] = \{ d\in[D\to D] \mid \forall e\in[\![\sigma]\!]\ d(e)\in[\![\rho]\!] \}$$

Questa interpretazione dei tipi viene chiamata semantica semplice .(Hindley(1983)). Altre interpratazioni sono state proposte (Hindley(1983), Coppo e Zacchi(1986) ).

Si noti che $?\notin[\![\sigma]\!]$ per ogni tipo $\sigma$.

Sia B una base propria (contenente cioe' solo tipi propri). Un ambiente $\rho$ rispetta B se $x:\sigma\in B$ implica $\rho(x)\in[\![\sigma]\!]$. Se $\sigma$ e' un tipo proprio definiamo $B\models M:\sigma$ se per ogni $\rho$ che rispetta B $[\![M]\!]\rho\in[\![\sigma]\!]$.

Siano ora B e $\sigma$ arbitrari. Definiamo $B\models M:\sigma$ se per ogni sostituzione di base s $s(B)\models M:s(\sigma)$ . Nota che questa interpretazione e' in linea con l'idea di vedere i tipi polimorfi come schemi di tipo.

E'immediato verificare, per induzione sulla struttura delle deduzioni,

l'adeguatezza del sistema.

*Teorema*. $B \vdash M:\sigma \Rightarrow B \models M:\sigma$.

Poiche' $? \notin [\![\sigma]\!]$ , per ogni tipo $\sigma$, una conseguenza immediata' e' la seguente.

*Corollario*. Se $B \vdash M:\sigma$ allora , per ogni ambiente $\rho$ che rispetta B, $[\![M]\!]\rho \neq ?$.

Questa proprieta' assicura che un programma cui sia stato assegnato un tipo non dara' mai, nel corso della sua esecuzione un errore dovuto ad una non corretta applicazione. Quindi l'interprete del linguaggio (che rimane sostanzialmente un interprete del linguaggio senza tipi) , non deve eseguire tali controlli in fase di esecuzione (cio' non e' evitabile, invece, nel caso di linguaggi che non assumono nessun controllo di tipi come il LISP).

Il sistema di assegnazione , pero', non e' completo ($\Leftarrow$ non vale) a causa del fatto che i tipi non si conservano per $=_\beta$ (mentre l'interpretazione si).

Si introduca, allora, la regola

$$\frac{B \vdash^{Eq} M:\sigma \quad M =_\beta N}{B \vdash^{Eq} N:\sigma} \quad (Eq)$$

dove $\vdash^{Eq}$ indica la derivabilita' nel nuovo sistema.

Sopponiamo inoltre che A non contenga la costante if-then-else Infatti, in questo caso, un termine come $M \equiv$ if <u>true</u> then <u>1</u> else <u>true</u> non avrebbe tipo mentre ovviamente $\models M:int$ . Con questa restrizione il sistema diventa completo.

*Teorema*. $B \models M:\sigma \Rightarrow B \vdash^{Eq} M:\sigma$ .

Questa versione del teorema e' provata, in una forma leggermente diversa, in Coppo(1983). La completezza del sistema puro e stata dimostrata, indipendentemante, da Hindley(1983) e Barendregt e al.(1983), usando pero' modelli differenti.

Notiamo infine che $\vdash^{Eq}$ , e quindi $\models$, sono $\Sigma^0_1$ .



## 5.5 Estensioni

Si possono introdurre anche per questo sistema le estensioni esaminate per ($\lambda$t). Esamineremo l'introduzione dell'operatore di punto fisso, mentre per le equazioni ricorsive di tipo rimandiamo a Coppo(1985), MacQueen e al.(198-). Discuteremo anche una estensione che rende possibile assegnare tipi a tutti i termini normalizzabili.

### Operatore di punto fisso.

Poiche solo i termini normalizzabili hanno un tipo dovremo anche in questo caso introdurre una (unica) costante <u>Y</u>, assumendo che <u>Y</u> abbia <u>tutti</u> i tipi della forma $(\sigma \to \sigma) \to \sigma$ per $\sigma$ arbitrario (includendo quindi anche $(t \to t) \to t$ che e' il tipo principale). Introduciamo inoltre la regola di riduzione <u>Y</u>M --> M(<u>Y</u>M) . Cio' produce, dal punto di vista delle proprieta' di riduzione, effetti analoghi a quelli visti per ($\lambda$t). Viene pero' preservata l'esistenza del tipo principale e le relative proprieta'.

Questo sistema, con l'aggiunta delle equazioni di tipo, e' quello che corrisponde effettivamente al sistema di tipi del linguaggio ML (Milner(1978) ).

Per quanto riguarda la semantica <u>Y</u> verra' interpretato come l'operatore (interno) di minimo punto fisso su D (che e' anche il valore del termine corrispondente).

La semantica dei tipi definita in 5.4 non e piu' adeguata. Infatti si ha ora $\vdash (\underline{Y}(\lambda x.x)):t$ , il che significa che $(\underline{Y}(\lambda x.x))$ ha tipo $\sigma$ per ogni $\sigma$. Ora $[\![(\underline{Y}(\lambda x.x))]\!]\rho = \perp$ , infatti $\perp$ e' il minimo punto fisso dell'identita'.

Si definisca allora, per ogni $\sigma$ proprio:

$[\![int]\!] = \{ n \mid n \in N_\perp \}$ (nota: con le nostre convenzioni $\perp \in [\![int]\!]$

$[\![bool]\!] = \{ v \mid v \in T_\perp \}$

$[\![\sigma \to \tau]\!] = \{ d \in [D \to D] \mid d = \perp \text{ o } \forall e \in [\![\sigma]\!]\ d(e) \in [\![\rho]\!] \}$

In questo modo abbiamo $\perp \in [\![\sigma]\!]$ per ogni $\sigma$ proprio. Inoltre si vede facilmente che $[\![\sigma]\!]$ e' chiuso per limiti di catene crescenti ($[\![\sigma]\!]$ e', in particolare, un <u>ideale</u> in D).

Se definiamo $\models^Y$ analogamente a $\models$ (usando pero' la presente interpretazione dei tipi). Anche questo sistema si dimostra facilmente

adeguato.

*Teorema*. $B\vdash^{\forall} M:\sigma \Rightarrow B\models^{\forall} M:\sigma$.

La completezza fallisce anche in questo caso. Per rendere completo il sistema, pero', non e' piu' sufficiente aggiungere la regola (Eq). Infatti, per esempio, si ha $[\![(\lambda x.xx)(\lambda x.xx)]\!]\rho=\perp$ e, quindi, $\models^{\forall} (\lambda x.xx)(\lambda x.xx):\sigma$ per ogni tipo $\sigma$, ma $(\lambda x.xx)(\lambda x.xx):\sigma$ non e' provabile nemmeno usando (Eq). Per restituire la completezza al sistema e' necessario aggiungere una regola infinitaria ( Coppo(1984)). In questo modo, pero' la relazione di assegnamento diventa $\Pi^0_1$.

Tipi con intersezione.

Questa estensione e' stata introdotta per studiare dei sistemi di tipi che permettessero di assegnare un tipo a tutti i termini o, almeno, a tutti i termini normalizzabili. Il sistema cui accenniamo qui e' tratto da Coppo(1980). Per altre e piu' potenti formulazioni si veda anche Coppo e al.(1981), Barendregt e al.(1983).

Intruduciamo un nuovo operatore $\wedge$ di formazione dei tipi ed aggiungiamo alla definizione dei tipi $T_S$ il seguente punto:

$$\sigma\wedge\tau \in T_S \quad \text{se} \quad \sigma,\tau \in T_S .$$

Un tipo della forma $\sigma\wedge\tau$ intende rappresentare le proprieta' funzionali di un termine che ha entrambi i tipi $\sigma$ e $\tau$. Notiamo che questo estensione non sarebbe possibile con la nozione di tipi come partizioni adottata nella sezione 4.

Il sistema di regole di assegnazione (indicato con $\vdash^{\wedge}$ ) si ottiene aggiungendo alle regole in 5.2 le seguenti:

$$\frac{B\vdash^{\wedge} M:\sigma \quad B\vdash^{\wedge} M:\tau}{B\vdash^{\wedge} M:\sigma\wedge\tau}\ (\wedge I) \qquad \frac{B\vdash^{\wedge} M:\sigma\wedge\tau}{B\vdash^{\wedge} M:\sigma}\ (\wedge E)\ \text{(e simmetrica)}$$

In questo modo, possiamo dare tipo a molti termini che non ne avevano nei sistemi precedenti . Abbiamo, per esempio $\vdash^{\wedge} (\lambda x.xx):(t\wedge(t\to u))\to u$ e, ancora, $\vdash^{\wedge}(\lambda x.xx)(\lambda f.\lambda x.f(fx)):(t\to t)\to t\to t$ (esercizio: verifcarlo).



Si confrontino questi tipi con gli esempi in 5.3 . Inoltre, $\vdash^{\wedge}$ e' conservativo rispetto al sistema base $\vdash$ (Barendregt e al.(1983) ).

La proprieta' piu' interessante di questo sistema e' che fornisce una caratterizzazione completa dei termini fortemente normalizzabili.

*Teorema*(Coppo e Dezani(1980)). $M\in\Lambda$ e' fortemente normalizzabile se e solo se esiste una base B ed un tipo $\sigma$ tali che $B\vdash^{\wedge} M:\sigma$.

Questo sistema, pero', non corrisponde a nessun sistema logico nell'analogia delle formulae-as-types (Hindley(1984) ).

L'assegnamento di tipi diventa, ovviamente, indecidibile (e' $\Sigma^0_1$ anche in questo caso) anche se si puo' ancora dare una nozione di tipo principale (Coppo e al.(1980), Ronchi e Venneri(1984) )

Una restrizione decidibile e' stata studiata in Leivant(1983a).

## 6. Il λ-calcolo tipato del II ordine.

6.1 introduzione.

Il λ-calcolo tipato del secondo ordine (detto anche λ-calcolo polimorfo da alcuni autori) e' basato sull'idea di introdurre un operatore di astrazione sui tipi, analogo a quello di astrazione sui termini. Poiche', secondo l' analogia delle formulae-as-types, i tipi corrispondono alle proposizioni, un operatore di astrazione sui tipi rappresenta un operatore di astrazione sulle proposizioni. La principale motivazione logica per l'introduzione di tale operatore e' quella di definire estensoni del λ-calcolo tipato in cui sia possibile interpretare sistemi logici del secondo ordine (come l'analisi). Si puo' allora sfruttare l'analogia delle formula-as-types per dimostrare proprieta' di tali sistemi, come interpretazioni funzionali e proprieta' di normalizzazione delle deduzioni. Il λ-calcolo tipato del II ordine e' una versione essenziale del sistema F introdotto da Girard(1972), che ne dimostro' le principali proprieta', allo scopo di sviluppare una interpretazione funzionale dell'analisi seguendo l'approccio di Godel(1958) e di dimostrarne la consistenza mediante un teorema di normalizzazione.

Il λ-calcolo tipato del II ordine e' stato anche introdotto in modo indipendente da Reynolds(1974), con motivazioni esclusivamente informatiche. Abbiamo visto, per esempio, come sia desiderabile che operatori di carattere generale come λx.x (identita') abbiano un tipo polimorfo come t→t. Un concetto analogo si puo' esprimere in un calcolo tipato introducendo delle variabili nel linguaggio dei tipi (come in $T_s$) e definendo un identita' $\lambda x^t.x^t$ parametrizzata su t. Se si astrae ora rispetto a t si ottiene $\Lambda t.\lambda x^t.x^t$ che rappresenta l'identita' polimorfa il cui tipo si denota $\Delta t.t\to t$, e rappresenta il tipo dei termini che, applicati ad un tipo σ restituiscono termini di tipo σ→σ. L'identita' ad un certo tipo (come int), si otterra' quindi applicando $\Lambda t.\lambda x^t.x^t$ al tipo in questione.

Una differenza cruciale con i sistemi presentati in 5 e' data dal fatto che, nel λ-calcolo del II ordine, $\Delta t.t\to t$ e' considerato un tipo a tutti gli effetti (e' quindi possibile, per esempio, applicare una funzione al suo stesso tipo). Questo rende il sistema estremamente potente anche se pone numerosi problemi per la natura non predicativa che il sistema viene ad avere.

## 6.2 Linguaggio.

Sia ancora K l'insieme dei tipi di base e $V_T$ un insieme di variabili di tipo. . Linsieme $T_2$ dei tipi del II ordine e' definito da:

$$K \subseteq T_2$$
$$V_T \subseteq T_2$$
$$\sigma\to\tau \in T_2 \quad \text{se } \sigma,\tau \in T_2$$
$$\Delta t.\sigma \in T_2 \quad \text{se } \sigma \in T_2 .$$

Denotiamo con $C^\sigma$ e $V^\sigma$, rispettivamente, le costanti e le variabili di tipo σ. Definiamo l'insieme $\Lambda_2{}^\sigma$ dei termini di tipo σ nel seguente modo:

$$C^\sigma \subseteq \Lambda_2{}^\sigma$$
$$V^\sigma \subseteq \Lambda_2{}^\sigma$$
$$(MN) \in \Lambda_2{}^\tau \quad \text{se } M \in \Lambda_2{}^{\sigma\to\tau} \text{ e } N \in \Lambda_2{}^\sigma$$



$$\lambda x^\sigma.M \in \Lambda_2{}^{\sigma\to\tau} \quad \text{se } M \in \Lambda_2{}^\tau$$
$$(M\{\tau\}) \in \Lambda_2{}^{\sigma[\tau/t]} \quad \text{se } M \in \Lambda_2{}^{\Delta t.\sigma} \text{ e } \tau \in T_2$$
$$\Lambda t.M \in \Lambda_2{}^{\Delta t.\sigma} \quad \text{se } M \in \Lambda_2{}^\sigma .$$

Nell'ultimo punto bisogna supporre che t non occorra libera nei tipi delle variabili libere di M. Questa restrizione e' necessaria per escludere termini come $\lambda x^t.\Lambda t.x^t$ in cui la variabile di tipo t e' legata in una occorrenza di $x^t$ ma libera nell'altra. Indicheremo con (λt2) questo sistema.

Esempi: $\Lambda t.\lambda x^t.x^t \in \Lambda_2{}^{\Delta t.t\to t}$.

$TWICE_2 \equiv \Lambda t.\lambda f^{t\to t}.\lambda x^t.f(fx) \in \Lambda_2{}^{\Delta t.(t\to t)\to t\to t}$

(Abbreviazione : σ' abbrevia (σ→σ)→σ→σ).

$AUTO_2 \equiv \lambda x^{\Delta t.t'}.\Lambda s.x\{s\to s\}(x\{s\}) \in \Lambda_2{}^{\Delta t.t'\to\Delta s.s'}$.

$TWICE_2$ e $AUTO_2$ sono versioni al II ordine di TWICE e AUTO. Si noti come, in $AUTO_2$, si usa in modo essenziale l'applicazione di tipo che consente di dare due tipi diversi alle due occorrenze di x. Ovviamente $AUTO_2$ e' ora applicabile a $TWICE_2$.

Alle usuali regole di riduzione bisogna ora aggiungere una regole per l'applicazione di tipo che sara' della forma

$$(\Lambda t.M^\sigma)\{\tau\} \dashrightarrow M^\sigma[\tau/t]$$

dove $M^\sigma[\tau/t]$ rappresenta il termine ottenuto da M rimpiazzando tutte le occorrenze di t (nelle applicazioni di tipo e nei tipi delle variabili) con τ. Si vede facilmente che $M^\sigma[\tau/t] \in \Lambda_2{}^{\sigma[\tau/t]}$, quindi anche questa riduzione conserva correttamente i tipi. Anche in questo sistema, quindi, un termine non puo' mai generare applicazioni scorrette durante la sua valutazione.

Una complicazione di questo sistema e' dovuta al fatto che un termine di tipo $\Delta t.\sigma$ puo' essere applicato a qualunque tipo compreso, per esempio, $\Delta t.\sigma$ . Si perde la struttura gerarchica dei tipi tipica dei sistemi visti in precedenza.

La nozione di funzione che ammette un tipo come parametro e' stata introdotta in alcuni linguaggi di programmazione come CLU (Liskov e al. (1977)) o ALPHARD (Wulf e al. 1976)) ma, fino ad ora, in una forma

molto ristretta in cui buona parte della potenza di ($\lambda$t2) e' perduta.

## 6.3 Proprieta'.

Il teorema di normalizzazione forte, come si e' detto, continua a valere.

*Teorema*. Ogni termine di ($\lambda$t2) e' fortemente normalizzabile.

Questo risultato e' stato dimostrato da Girard(1972) adattando la tecnica di Tait(1967). Il problema tecnico principale, rappresentato dalla natura non gerarchica dei tipi, e' stato superato da Girard definendo a priori per ogni tipo, degli insiemi di termini (candidati di riducibilita') dotati di certe proprieta tra cui quella di normalizzazione forte e dimostrando quindi che $\Lambda_2^{\sigma}$ (per ogni $\sigma$) e' un candidato di riducibilita' .

Diamo un cenno alla dimostrazione di questo importate risultato, seguendo (con piccole varianti) l'approccio di Girard(1972). Supporremo, per semplicita', che non vi siano costanti. La generalizzazione al caso delle costanti e' comunque semplice.

Supporremo i termini definiti a meno di una ridenominazione delle variabili legate, ignorando cosi' i problemi dovuti ad eventuali conflitti di nomi tra variabili libere e legate.

Introduciamo prima alcune definizioni. Abbrevieremo fortemente normalizzabile con f.n. .

Un insieme di termini $A^{\sigma} \subseteq \Lambda_2^{\sigma}$ e' un candidato di riducibilita' (c.r.) di tipo $\sigma$ se soddisfa i seguenti punti:

(P1) Se $M \in A^{\sigma}$ allora M e' f.n.

(P2) Se $M \in A^{\sigma}$ e $N \dashrightarrow M$ allora $N \in A^{\sigma}$

(P3) Se $xe_1 \ldots e_k \in \Lambda_2^{\sigma}$ dove $e_1,\ldots,e_k$ ($k \geq 0$) sono tipi o termini f.n. allora $xe_1 \ldots e_k \in A^{\sigma}$.

Nota che, per ogni tipo $\sigma$, vi possono essere piu' c.r. di tipo $\sigma$. Inoltre per ogni variabile $x^{\sigma}$ ed ogni c.r. $A^{\sigma}$ di tipo $\sigma$, $x^{\sigma} \in A^{\sigma}$. Sia $CR^{\sigma}$ l'insieme di tutti i c.r. di tipo $\sigma$ e CR l'unione di tutti i $CR^{\sigma}$ per ogni $\sigma \in T_2$.

Sia ora $CRenv = \mathcal{V}_T \to CR$ l'insieme di tutte le associazioni di c.r. (di un tipo arbitrario) ad ogni variabile di tipo. Definiamo una sostituzione



come in 5.3 (con $T_2$ al posto di $T_s$). Se $\theta \in CRenv$ sia $s_\theta$ la sostituzione definita da $s_\theta(t) = \sigma$ se $\theta(t) \in CR^{\sigma}$.

Sia ora $\sigma \in T_2$ e $\theta \in CRenv$ . Definiamo $|\sigma|_\theta \subseteq \Lambda_2^{s_\theta(\sigma)}$ per induzione su $\sigma$ nel seguente modo:

$|t|_\theta = \theta(t)$ se $t \in \mathcal{V}_T$

$|\sigma \to \tau|_\theta = \{M \in \Lambda_2^{s_\theta(\sigma\to\tau)} \mid \forall N \in |\sigma|_\theta \ MN \in |\tau|_\theta \}$

$|\Delta t.\sigma|_\theta = \{M \in \Lambda_2^{s_\theta(\Delta t.\sigma)} \mid \forall \tau \in T_2 \ \forall A^{\tau} \in CR^{\tau} \ M\{\tau\} \in |\sigma|_{\theta[A^{\tau}/t]} \}$ .

Dimostriamo ora due lemmi.

*Lemma 1*. Per ogni $\sigma \in T_2$ e per ogni $\theta \in CRenv$ $|\sigma|_\theta$ e' un c.r.di tipo $s_\theta(\sigma)$.

*Dimostrazione* . Per induzione su $\sigma$. Se $\sigma \in \mathcal{V}_T$ il lemma e' vero per definizione. Negli altri due casi basta verificare (P1) , (P2) e (P3) della definizione di c.r..

Caso $\sigma \equiv \rho \to \tau$. La verifica di (P2) e' banale .

(P1). Sia $x^{s_\theta(\rho)}$ una variabile di tipo $s_\theta(\rho)$. Per ipotesi induttiva $x^{s_\theta(\rho)} \in |\rho|_\theta$ e, quindi, $Mx^{s_\theta(\rho)} \in |\tau|_\theta$. Per ip. ind. $Mx^{s_\theta(\rho)}$ e' f.n. e cio' implica che anche M e' f.n. (esercizio: perche?).

(P3). Sia $xe_1 \ldots e_k \in \Lambda_2^{s_\theta(\rho\to\tau)}$ con $e_1,\ldots,e_k$ tipi o termini f.n. . Sia inoltre $M_{k+1} \in |\rho|_\theta$ . Per ip. ind. $M_{k+1} \in \Lambda_2^{s_\theta(\rho)}$ e' f.n. e quindi, ancora per ip. ind., $xe_1 \ldots e_k M_{k+1} \in |\tau|_\theta$ . Quindi $xe_1 \ldots e_k \in |\rho \to \tau|_\theta$ .

Caso $\sigma \equiv \Delta t.\rho$ . Anche in questo caso la verifica di (P2) e' banale.

(P1). Sia $M \in |\Delta t.\rho|_\theta$ . Allora, preso un arbitrario $\tau \in T_2$ e $A^{\tau} \in CR^{\tau}$ $M\{\tau\} \in |\rho|_{\theta[A^{\tau}/t]}$ e quindi, per ip. ind. , $M\{\tau\}$ e' f.n. . Allora anche M e' f.n. (esercizio: perche' ?) .

(P3). Sia $xe_1 \ldots e_k \in \Lambda_2^{s_\theta(\Delta t.\rho)}$ con $e_1,\ldots,e_k$ tipi o termini f.n. e sia $\tau \in T_2$. Allora $xe_1 \ldots e_k\{\tau\} \in \Lambda_2^{s_\theta[\tau/t](\rho)}$ e, per ip. ind. , per ogni $A^{\tau} \in CR^{\tau}$ $xe_1 \ldots e_k\{\tau\} \in |\rho|_{\theta[A^{\tau}/t]}$ . Quindi $xe_1 \ldots e_k \in |\Delta t.\rho|_\theta$ . □

Se $M \in \Lambda_2^{\sigma}$ ed s e' una sostituzione s(M) denota il termine ottenuto da M applicando s a tutti i tipi occorrenti in M. Si noti che $s(M) \in \Lambda_2^{s(\sigma)}$.

*Lemma 2*. Sia $M \in \Lambda_2^{\sigma}$, $\theta \in CRenv$ e $M_i \in |\tau_i|_\theta$ ($1 \leq i \leq k$) dove $x_1^{\tau_1},\ldots,x_k^{\tau_k}$

sono le variabili libere in M. Allora $s_\theta(M)[M_i/x_i] \in |\sigma|_\theta$ .

*Dimostrazione* . Induzione su M. Il caso $M \equiv x^\sigma$ e' immediato .

Sia $\underline{M \equiv \lambda x^\rho.M'} \in \Lambda_2{}^{\rho\to\tau}$ (con $\sigma \equiv \rho\to\tau$) , dove $M' \in \Lambda_2{}^{\tau}$ . Sia $N \in |\rho|_\theta$ . Si ha che $s(\lambda x^\rho.M')[M_i/x_i]N \to s(M')[M_i/x_i, N/x] \in |\tau|_\theta$ per ip. ind. . Quindi, per il lemma 1 e (P2), $s(\lambda x^\rho.M')[M_i/x_i] \in |\rho\to\tau|_\theta$ .

Sia $\underline{M \equiv \Lambda t.M'} \in \Lambda_2{}^{\Delta t.\rho}$ (con $\sigma \equiv \Delta t.\rho$), dove $M' \in \Lambda_2{}^{\rho}$ . Sia $\tau \in T_2$ e $A^\tau \in CR^\tau$ . Per ip. ind. $s_\theta[\tau/t](M')[M_i/x_i] \in |\rho|_{\theta[A^\tau/t]}$ (notiamo che t non puo' occorrere in $\tau_i$ , essendo $x_i$ libera in M'). Inoltre $s_\theta(\Lambda t.M')[M_i/x_i](\tau) \to s_\theta[\tau/t](M')[M_i/x_i]$ . Quindi ,per il lemma 1 e (P2), $s_\theta(\Lambda t.M')[M_i/x_i] \in |\Delta t.\rho|_\theta$ .

Gli ultimi due casi ($M \equiv M'M''$ e $M \equiv M'\{\tau\}$ ) sono lasciati per esercizio. □

*Dimostrazione del teorema* . Sia $\theta_o \in CRenv$ tale che $s_{\theta_o}$ sia la sostituzione identica (cioe' $\forall t \in \mathcal{V}_T\ \theta_o(t) \in CR^t$ ) . Allora, se $x^\sigma$ e' una variabile di tipo $\sigma$ libera in M , $x^\sigma \in |\sigma|_{\theta_o}$ . Sia ora $M \in \Lambda_2{}^{\sigma}$ . Per il Lemma 2 $s_{\theta_o}(M) \equiv M \in |\sigma|_{\theta_o}$ e quindi, per il Lemma 1, M e' f.n. . □

La classe delle funzioni rappresentabili e' ora molto grande. Si definisca:

$$\iota = \Delta t.(t\to t)\to t\to t \quad \text{(tipo delle rappresentazioni degli interi)}$$

Si vede facilmente che le funzioni definite in 4.2 sono definibili, al tipo opportuno, anche in (λt2).

Per esempio $+_2 \equiv \lambda p^\iota.\lambda q^\iota.\Lambda t.\lambda f^{t\to t}.\lambda x^t.\ p\{t\}f(q\{t\}fx) \in \Lambda_2{}^{\iota\to\iota\to\iota}$ .

Si puo' pero' fare di piu'. Un termine per rappresentare l'esponenziazione e', per esempio, il seguente $EXP \equiv \lambda p^\iota.\lambda q^\iota.\Lambda t.\ p\{t\to t\}(q\{t\}) \in \Lambda_2{}^{\iota\to\iota\to\iota}$ .

Anche il predecessore e la sottrazione risultano rappresentabili ( vedi, ad esempio, Fortune ed al.(1983)). Inoltre gli iteratori $J_\sigma$ definiti in 4.5 si possono rappresentare nel modo seguente:

$$J_\sigma \equiv \lambda n^\iota.\lambda x^{\sigma\to\sigma}.\lambda y^\sigma.\ n\{\sigma\}xy \in \Lambda_2{}^{\iota\to(\sigma\to\sigma)\to\sigma\to\sigma} .$$

Tutte le funzioni provabili ricorsive nell'aritmetica del I ordine ($\epsilon_0$-ricorsive) sono quindi reppresentabili in (λt2).

Si puo' fare pero' ancora di piu'.



*Teorema* . Le funzioni rappresentabili in (λt2) sono esattamente quelle provabili ricorsive nell'aritmetica del II ordine.

Questo risultato e' stato dimostrato indipendentemente da Girard(1972) e Statman(1981), anche se in contesti diversi. Notiamo che, in (λt2), la classe delle funzione definibili utilizzando le costanti rimane la stessa (infatti tutte la costanti elementari sono rappresentabili nel sistema puro).

Anche (λt2), quindi, rapresenta un linguaggio di programmazione di potenza piu' che accettabile in cui sono programmabili solo funzioni totali.

## 6.4 Formulae-as-types.

Il sistema corrispondente a (λt2) nell'analogia delle formulae-as-types e' il calcolo proposizionale implicativo intuizionista del II ordine ($CPI_2$).

Tale sistema si ottiene da CPI aggiungendo un quantificatore universale sui simboli proposizionali e le regole:

$$\frac{\sigma}{\forall t.\sigma}\ (\forall I) \qquad\qquad \frac{\forall t.\sigma}{\sigma[\tau/t]}\ (\forall E)$$

dove t rappresenta ora una variabile proposizionale. Nella regola (∀I) bisogna assumere (come in 6.2) che t non occorra libera in nessuna assunzione da cui la formula $\sigma$ dipende . La regole (∀I) corrisponde all'astrazione di tipo $\Lambda t.M^\sigma$ e la regole (∀E) all'applicazione di tipo $M^{\Delta t.\sigma}\{\tau\}$ . La riduzione di un'applicazione di tipo $(\Lambda t.M^\sigma)\{\tau\} \to M^\sigma[\tau/t]$ corrisponde allora ad una ∀-riduzione ( vedi Prawitz(1965)) del tipo:

$$\begin{array}{l} \mathcal{D} \\ \sigma \\ \hline \forall t.\sigma\ (\forall I) \\ \hline \sigma[\tau/t]\ (\forall E) \end{array} \quad \text{si ∀-riduce a} \quad \begin{array}{l} \mathcal{D}[\tau/t] \\ \sigma[\tau/t] \end{array}$$

Il teorema di normalizzazione di ($\lambda$t2) implica allora la proprieta' di normalizzazione delle deduzioni in $CPI_2$.

Per quanto molto essenziale $CPI_2$ e' un sistema molto potente. E' infatti possibile interpretare il calcolo predicativo del II ordine in $CPI_2$ . E' stato dimostrato che la normalizzazione di $CPI_2$ implica (nell'aritmetica) la consistenza dell'analisi. La prova e' dovuta a risultati di Statman e Prawitz, si veda Fortune ed al.(1983) per i riferimenti. Il teorema di normalizzazione per ($\lambda$t2) (e $CPI_2$) e' quindi indipendente dall'analisi.

Vi sono altre interessanti analogie tra ($\lambda$t2) e l'aritmetica (intuizionista) del II ordine. Ricordiamo che tale sistema e' definito sul calcolo predicativo intuizionista del II ordine con i connettivi $\Rightarrow$ , $\forall^1$ e $\forall^2$ (quantificatori al I e II ordine) ) , cui si aggiungono le costanti $\underline{0}$ e $\underline{succ}$ ed il seguante assioma di induzione (accanto agli assiomi relativi all'eguaglianza (del primo ordine) e al successore )

$$\forall^2 P.\ (\ (\forall^1 x.(P(x)\Rightarrow P(\underline{succ}(x)))) \Rightarrow P(\underline{0})\Rightarrow \forall^1 y.P(y)\ )$$

In questo sistema la proprieta' di essere un numero naturale e' espressa da :

$$N(y) = \forall^2 P.\ (\ (\forall^1 x.(P(x)\Rightarrow P(\underline{succ}(x))))\Rightarrow P(\underline{0})\Rightarrow P(y)\ )$$

Se cancelliamo allora le variabili e le quantificazioni al primo ordine, otteniamo:

$$\forall^2 P.((P\Rightarrow P)\Rightarrow P\Rightarrow P)$$

che corrisponde al tipo degli interi nella rappresentazione scelta in 6.3 . La dimostrazione che un certo oggetto e' un numero, allora, corrisponde (in questo senso debole) al termine che lo rappresenta. Riportiamo un esempio relativivo a $\underline{succ}(\underline{0})$ , che e' rappresentato in ($\lambda$t2) da $\Lambda p.\lambda f^{p\to p}\lambda x^{p}.f^{p\to p}x^{p}$. Accanto ad ogni formula riportiamo, tra parentesi {}, il termine corrispondente alla relativa sottodeduzione.



$$\dfrac{\dfrac{\dfrac{[\ \forall^1 x.P(x)\Rightarrow P(\underline{succ}(x))\ ]\ \{f^{p\to p}\}}{P(\underline{0})\Rightarrow P(\underline{succ}(\underline{0}))\ \{f^{p\to p}\}}(\forall^1 E) \qquad [\ P(\underline{0})\ ]\ \{x^{p}\}}{P(\underline{succ}(\underline{0})\ \{f^{p\to p}x^{p}\}}(\Rightarrow E)}{P(\underline{0})\Rightarrow P(\underline{succ}(\underline{0}))\ \{\lambda x^{p}.f^{p\to p}.x^{p}\}}(\Rightarrow I)$$

$$\dfrac{(\forall^1 x.P(x)\Rightarrow P(\underline{succ}(x)))\Rightarrow P(\underline{0})\Rightarrow P(\underline{succ}(\underline{0}))\ \{\lambda f^{p\to p}\lambda x^{p}.f^{p\to p}x^{p}\}}{\forall^2 P.((\forall^1 x.P(x)\Rightarrow P(\underline{succ}(x)))\Rightarrow P(\underline{0})\Rightarrow P(\underline{succ}(\underline{0})))\ \{\Lambda p.\lambda f^{p\to p}\lambda x^{p}.f^{p\to p}x^{p}\}}(\forall^2 I)$$

L'analogia si puo' portare ancora piu' avanti. Un $\lambda$-termine corrispondente alla prova che una certa funzione f e' totale (proprieta' rappresentabile dalla formula $\forall x.(N(x)\Rightarrow N(f(x)))$ ) rappresenta tale funzione in ($\lambda$t2). Per maggiori dettagli si vedano Leivant(1983b) e Bohm e Berarducci (1985).

## 6.5 Semantica.

La natura non-gerarchica dei tipi rende evidenti le difficolta' di una interpretazione analoga a quella vista per ($\lambda$t). Infatti intuitivamente un tipo della forma $\Delta t.\sigma$ dovrebbe contenere funzioni che, applicate ad un tipo $\tau$, restituiscono un valore di tipo $\sigma[\tau/t]$. ma $\Delta t.\sigma$, essendo un tipo, e' uno dei possibile argomenti. In certi casi, quindi, l'interpretazione di un tipo deve essere applicabile a se stessa.

Reynolds(1984) ha dimostrato che <u>non e' possibile</u> dare una interpretazione di ($\lambda$t2) in cui ogni tipo della forma $\sigma\to\tau$ viene interpretato come l'insieme di <u>tutte</u> le funzioni da (l'interpretazione di) $\sigma$ in (l'interpretazione di) $\tau$, sia pure lasciando la massima liberta' nell'interpretazione dei tipi della forma $\Delta t.\sigma$ (purche', ovviamenta, adeguate a dare un modello di ($\lambda$t2)). In questo caso, infatti, si potrebbe definire un insieme P tale che esiste un isomorfismo tra P e $(P\to B)\to B$ , dove B rappresenta l'interpretazione di $\Delta t.(t\to t)\to t$ ( che contiene certamente almeno due elemento distinti: le interpretazioni di

$\Lambda t.\lambda x^t.\lambda y^t.x$ e $\Lambda t.\lambda x^t.\lambda y^t.y$ ). Cio' e' chiaramente impossibile per ragioni di cardinalita'.

E' possibile fornire una interpretazione di ($\lambda t2$) partendo da (opportuni) modelli del $\lambda$-calcolo senza tipi. L'idea e', in linea generale, la seguente (si veda Scott(1976)). Sia D un dominio che soddisfi, per esempio, l'equazione vista in 2.3 . Definiamo un <u>retratto</u> di D come un elemento $a \in [D \to D]$ (quindi interno a D) tale che $a = a \circ a$ . Sia $range(a) = \{d \in D \mid \exists e \in D\ a(e) = d\}$ . Un retratto rappresenta l'identita' sul proprio range ed identifica, in questo modo, un sottoinsieme di D (che corrisponde, in un certo senso, a un sottodominio di D) . D'ora in poi identificheremo i retratti col proprio range. Per esempio una funzione

$$\lambda v \in D.\ \text{if } v \in N_\perp \text{ then } v \text{ else } \perp$$

rappresenta in retratto che definisce il sottodominio degli interi.

Inoltre se a e b sono due ritratti anche $a \circ\!\!\to b = \lambda v \in D.b \circ v \circ a$ e' un retratto di D che rappresenta lo spazio delle funzioni da a in b . Infatti, per ogni $d \in range(a)$ , $a \circ\!\!\to b(v)(d) \in range(b)$ . Molti altri costruttori di tipi, come $\times$ e $+$ hanno retrazioni corrispondenti.

In questo modo e' possibile identificare i tipi con elementi di D, rendendo omogenee le interpretazioni di termini e tipi.

In certi casi si ha che una classe di ritratti di D e' essa stessa un retratto di D, cioe e' il range di un elemento $t \in D$ appartenete alla stessa classe. Cio' avviene, per esempio, per le ritrazioni finitarie su domini $\omega$-algebrici (McCracken(1982)). In questo caso $t$ rappresenta la collezione di tutti i tipi (se $d \in D$, infatti, $t(d)$ e' un retratto, cioe' un tipo).

Sia ora h una funzione tale che $range(h) \subseteq range(t)$ . Definiamo

$$\Pi(h) = \lambda f.\lambda x.\ h(t\,x)(f(t\,x))$$

E' immediato verificare che $\Pi(h)$ e' un retratto. Notiamo inoltre che , se a interpreta un tipo della forma $\Delta t.\sigma$ , i valori nel range di $\Pi(a)$ sono funzioni che mappano un tipo "$tx$" in valori di tipo $h(tx)$ e, quindi, sono adatti a rappresentare un termine che mappa ogni tipo $\{\tau\}$ in termini di tipo $\sigma[\tau/t]$ .

Se interpretiamo i tipi di base con i corrispondenti retratti , e indichiamo con $\eta : Tenv = \mathcal{V}_T \to range(t)$ un ambiente per l'interpretazione



delle variabili di tipo possiamo definire l'interpretazione dei tipi $[\![\,]\!] : T_2 \to Tenv \to D$ nel seguente modo

$$[\![k]\!]\eta = k \quad \text{(dove } k \text{ e' un retratto corrispondente al tipo base } k)$$
$$[\![t]\!]\eta = \eta(t)$$
$$[\![\sigma \to \tau]\!]\eta = [\![\sigma]\!]\eta \circ\!\!\to [\![\tau]\!]\eta$$
$$[\![\Delta t.\sigma]\!]\eta = \Pi(\lambda v \in D.[\![\sigma]\!]\eta[v/t] \circ t\,)$$

Sia $\eta \in Tenv$ e $\rho : (\bigcup_{\sigma \in T_2} \mathcal{V}_T^{\sigma}) \to D$ un ambiente per l'interpretazione delle variabili che rispetta $\eta$, nel senso che $\rho(x^\sigma) \in range([\![\sigma]\!]\eta)$ . L'interpretazione $[\![\,]\!] : (\bigcup_{\sigma \in T_2} \Lambda^{\sigma}) \to Tenv \to Env \to D$ dei termini sara' ora data da:

$$[\![x^\sigma]\!]\eta\rho = \rho(x)$$
$$[\![MN]\!]\eta\rho = [\![M]\!]\eta\rho([\![N]\!]\eta\rho)$$
$$[\![\lambda x^\sigma.M]\!]\eta\rho = (\lambda v \in D.[\![M]\!]\eta(\rho[v/x])) \circ [\![\sigma]\!]\eta$$
$$[\![M\{\tau\}]\!]\eta\rho = [\![M]\!]\eta\rho([\![\tau]\!]\eta)$$
$$[\![\Lambda t.M]\!]\eta\rho = (\lambda v \in D.[\![M]\!](\eta[v/t])\rho) \circ t$$

E' facile verificare che tale interpretazione e' ben definita nel senso che se $M \in \Lambda_2^{\sigma}$ allora $[\![M]\!]\eta\rho \in [\![\sigma]\!]\eta$ (per ogni $\eta$ e $\rho$). Inoltre $M = N$ implica $[\![M]\!]\eta\rho = [\![N]\!]\eta\rho$ (per ogni $\eta$ e $\rho$) .

Notiamo che, anche se D e' un dominio senza tipi , la struttura di tipi e' comunque ricostruita all'interno di D poiche' ogni valore e' "proiettato" nel range del retratto corrispondente al suo tipo. Un aspetto non del tutto soddisfacente di questa interpretazione e' il fatto che, poiche' i retratti hanno la struttura dei domini, l'interpretazione di ogni tipo contiene un elemento minimo che e' naturale interpretare come indefinito. Cio' pero' non sembra strettamente necessario dal momento che ogni termine di ($\lambda t2$) e' normalizzabile.

Questo tipo di approccio, implicito in Scott(1976), e' stato sviluppato da McCraken(1979) (che considera come retrazioni le chiusure su $P_\omega$), McCracken(1982) (che considera le ritrazioni finitarie su un dominio algebrico)) e Amadio e al.(1986) (che considerano invece le proiezioni finitarie).

Recentemente Girard(1985) ha proposto una nuova interpretazione di ($\lambda t2$) basata sulla sua semantica qualitativa. Per una esposizione di

carattere generale sulla nozione di modello per ($\lambda$t2) si veda Bruce e al.(198-).

## 6.7 Restrizioni ed estensioni.

Data la grandissima potenza del sistema puo' essere interessante anche lo studio di sue opportune restrizioni, soprattutto riguardo alla possibilita' di renderne piu' agevole la realizzazione pratica.

Una proposta interessante (Fortune ed al.(1983)) e' quella di restringere i termini di ($\lambda$t2) ai soli termini stazionari, dove un termine e' stazionario se ogni tipo $\tau$ che occorre in una applicazione della forma $(M^{\Delta t.\sigma})\{\tau\}$ e' o una singola variabile di tipo o un tipo chiuso (cioe' senza occorrenze di variabili di tipo). Per esempio , riferendoci alla sezione 6.3, i termini $J_\sigma$ sono stazionari (se $\sigma$ e' chiuso) mentre EXP non lo e'.

Si vede facilmente che i termini stazionari sono chiusi per riduzione. Se un termine M e' stazionario e si riduce ad N allora ogni tipo che occorre come argomento di un'applicazione di tipo in N occorre anche in un'applicazione di tipo in M. Inoltre tutti i possibili tipi delle sottoespressioni di N devono appartenere ad un insieme finito facilmente calcolabile a partire da M. In questo modo e' possibile prevedere in anticipo quali tipi saranno eventualmente coinvolti nella riduzione di M, e questo puo' essere di aiuto per migliorare l'efficienza dell'interprete. Anche con questa restrizione il linguaggio resta molto potente. Per esempio , poiche' i termini $J_\sigma$ sono stazionari , almeno le funzioni $\epsilon_0$-ricorsive sono rappresentabili. Inoltre il teorema di normalizzazione per le espressioni stazionarie e' indipendente dalla $\Pi^1_1$-analisi (per referenze si veda Fortune e al.(1983)).

Altre restrizioni possono essere suggerite, mediante l'analogia delle formulae-as-types, da sottosistemi predicativi della logica del II ordine. (Leivant(1983b)).

Per quanto riguarda le estensioni si possono introdurre, senza particolari difficolta', sia l'operatore di punto fisso che i tipi ricorsivi. L'interpretazione proposta in 6.5 si adatta senza difficolta' a queste estensioni.



Un'altra estensione, che conduce ad un sistema ancora piu' potente e' quella di consentire astrazioni anche su funzionali che operano su tipi. Per assicurare la buona definizione di tali funzionali, si puo' dare alla collezione dei funzionali su tipi una struttura di tipi (che verranno detti ordini e indicati con $\zeta$ , $\xi$ ).

Il linguaggio degli ordini e' definito come i tipi semplici in 4.1, partendo da un'unico ordine di base T (useremo pero' $\Rightarrow$ al posto di $\rightarrow$). T rappresenta l'ordine dei tipi.

Si abbia ora, per ogni ordine $\zeta$, un insieme di variabili $t^\zeta$ di ordine $\zeta$ e un insieme (eventualmente vuoto) di costanti di ordine $\zeta$. L'insieme $T^\zeta$ dei funzionali di tipo di ordine $\zeta$ e' definito come $\Lambda^\sigma$ in 4.1 . Bisogna anche introdurre, ovviamente, una nozione di $\beta$-riduzione sui funzionali di tipo, come in ($\lambda$t).

Dobbiamo inoltre assumere l'esistensa delle seguenti costanti

$\rightarrow \in T^{T\Rightarrow T\Rightarrow T}$ (scriveremo $\sigma\rightarrow\tau$ per $\rightarrow\sigma\tau \in T^T$ )

$\Delta_\zeta \in T^{(\zeta\Rightarrow T)\Rightarrow T}$ (scriveremo $\Delta t^\zeta.\sigma$ per $\Delta_\zeta(\lambda t^\zeta.\sigma)$, dove $\sigma \in T^T$)

Per ogni tipo $\sigma$ di ordine T, infine, definiamo l'insieme $\Lambda_\oplus^\sigma$ dei termini di tipo $\sigma$ come in 6.2 ma con queste modifiche per quanto riguarda astrazione e applicazione di tipo:

$(M\{\tau\}) \in \Lambda_\oplus^{\sigma[\tau/t^\zeta]}$ se $M \in \Lambda_\oplus^{\Delta t^\zeta.\sigma}$ e $\tau \in T^\zeta$

$\Lambda t^\zeta.M \in \Lambda_\oplus^{\Delta t^\zeta.\sigma}$ se $M \in \Lambda_\oplus^\sigma$

Le regole di riduzione per i termini di ($\lambda$t2) si estendono immediatamente.

Anche questo sistema e' stato introdotto (in una forma leggermente diversa) da Girard(1972) che ne dimostra il teorema di normalizzazione forte. In Girard(1973) si dimostra anche che le funzioni rappresentabili in tale sistema sono quelle provabili ricorsive nella aritmetica di ordine finito (teoria degi ordini). Questo sistema, quindi, costituisce una estensione propria di ($\lambda$t2).

La nozione di ordine e' poi stata introdotta indipendentemente, nello sviluppo di modelli di ($\lambda$t2), dalla McCracken(1979).

L'interpretazione proposta in 6.5 si adatta senza problemi a questo sistema. E' anche possibile definire estensioni quali i funzionali di tipo

ricorsivi (perdendo ovviamente la proprieta' di normalizzazione) introducando un operatore $\mu_\zeta \in T^{(\zeta=>\zeta)=>\zeta}$ con la regola $\mu_\zeta(\Delta t^\zeta.\sigma)\text{-->}\sigma[\mu_\zeta(\Delta t^\zeta.\sigma)/t^\zeta]$ . La nozione di tipo ricosivo si ottiene come caso particolare per $\zeta$=T.

6.7 Inferenza di tipi del II ordine.

Anche l'approccio di assegnare tipi ai termini, studiato nella sezione 5, si puo' estendere a tipi del II ordine.

Si consideri lo stesso insieme di tipi $T_2$ , dove pero', per uniformita' con la letteratura, scriveremo $\forall t.\sigma$ al posto di $\Delta t.\sigma$ . Il sistema di assegnazione di tipi (che denoteremo con $\vdash^2$ ) si ottiene aggiungendo al sistema studiato in 5.2 le regole:

$$\frac{B\vdash^2 M:\sigma}{B\vdash^2 M:\forall t.\sigma}\ (\forall I) \qquad \frac{B\vdash^2 M:\forall t.\sigma}{B\vdash^2 M:\sigma[\tau/t]}\ (\forall E)$$

in ($\forall$I) si assume che t non occorra libera in B.

Il sistema cosi' ottenuto e' equivalente a ($\lambda$t2) nel senso che $M\in\Lambda_2{}^\sigma$ se e solo se $B\vdash^2 M:\sigma$ dove B = {x:$\tau$| $x^\tau$ e' libera in M} . Tutte le proprieta' sintattiche di ($\lambda$t2) , in particolare il teorema di normalizzazione forte e la caratterizzazione della classe delle funzioni rappresentabili, valgono anche per i termini che possoggono tipi in questo sistema.

Un vantaggio di questo approccio rispetto a ($\lambda$t2) e' che i termini non contengono esplicitamente la nozione di applicazione di tipo, che , a volte, appesantisce notevolmente la loro scrittura.

In questo sistema di assegnazione di tipi , pero', non esiste piu' la nozione di tipo principale. Per esempio $(\forall t.t)\to(\forall t.t)$ e $(\forall t.t\to t)\to(\forall t.t\to t)$ sono entrambi tipi di $\lambda x.xx$ senza essere istanze di nessun tipo assegnebile a $\lambda x.xx$ . Non e' noto, inoltre, se l'insieme dei tipi di un termine e' ricorsivo e se e' decidibile , dato un termine, se questo ammette almeno un tipo (la congettura piu' diffusa e' che non lo



sia, in entrambi i casi).

E' tuttavia interessante, da un punto di vista informatico, lo sviluppo di algoritmi di assegnazione e verifica di tipi che richiedano eventualmente qualche informazione da parte del programmatore, come la specifica dei tipi delle variabili . Per esempio come in $\lambda x{:}\forall t.t\to t.\ xx$ .

Una semantica di questo sistema si puo' fornire seguendo l'approccio proposto nelle sezioni 5.4 e 5.5 . Per i dettagli si vedano, per esempio, MacQueen ed al.(198-) e Coppo e Zacchi(1986).

## Riferimenti


Amadio R., Bruce K., Longo G. (1986) The finitary projection model for second order lambda calculus and solutions to higher order domain equations, in Proc. of the IEEE Symposium on Logic in Computer Science, Cambridge Mass. , 122-130.

Arvind, Kathail V., Pignali K. (1984) Sharing of computation in functional language implementation, rapporto interno MIT, Cambridge Mass..

Barendregt H. (1981/4) The lambda calculus: its syntax and semantics, North-Holland (prima edizione 1981, seconda 1984).

Barendregt H., Coppo M., Dezani M. (1983) A filter lambda-model and the completeness of type assignment, J. Symbolic Logic 48(4) , 931-940.

Bohm C., Berarducci A. (1985) Automathic syntesis of typed $\Lambda$-programs on term algebras, Theor. Comput. Sci. 39, 135-154.

Bruce K., Meyer R., Mitchell J. (198-) The semantics of second-order lambda calculus, di prossima pubblicazione su Information and Control.

Burge W. (1978) Recursive programming techniques, Addison-Wesley.

Burstall R., MacQueen D., Sannella D. (1980) HOPE: an experimental applicative language, in Proc. of the LISP conference, Stanford University .

Church A. (1932/33) A set of postulates for the foundation of logic, Annals of Math. 33(2), 346-366 e 34, 839-864 .

Church A. (1940) A formalization of the simple theory of types , J.

Symbolic Logic 5, 56-68.

Church A. (1941) The calculi of lambda-conversion, Princeton University Press.

Constable R., Zlatin D. (1984) The type theory of PL/CV3, ACM Trans. on Prog. Languages 6,94-117.

Coppo M. (1980) An extended polymorphic type system for applicative languages, in Proc. of Mathematical Foundations of Computer Science, Lecture Notes in Computer Science 88, Springer-Verlag.

Coppo M., Dezani-Ciancaglini M. (1980) An extension of basic functionality theory for lambda-calculus, Notre Dame J. Formal Logic 21(4), 685-693.

Coppo M., Dezani-Ciancaglini M., Venneri B. (1980) Principal type schemes and lambda calculus semantics, in To H.B.Curry: essays on combinatory logic, lambda-calculus and formalism, J. Seldin and R. Hindley eds., Academic Press, 536-560.

Coppo M., Dezani-Ciancaglini M., Venneri B. (1981) Functional characters of solvable terms, Zeit. Math. Logik und Grund. Math. 27, 45-58.

Coppo M. (1983) On the semantics of polymorphism, Acta Informatica 20, 159-170.

Coppo M. (1984) Completeness of type assignment in continuous lambda-models, Theor. Comput. Sci. 29, 309-324.

Coppo M. (1985) The completeness theorem for recursively defined types, in Proc. of ICALP '85, Lecture Notes in Computer Science 194, Springer-Verlag, 120-129.

Coppo e Zacchi (1986) Type inference and logical relations, in Proc. of the IEEE Symposium on Logic in Computer Science, Cambridge Mass., 218-226.

Coquand T., Huet G. (1985) Constructions: a higher order proof system for mechanizing mathematics, in EUROCAL85, Linz, Lecture Notes in Computer Science 203, Springer-Verlag .

Curry H. (1934) Functionality in combinatory logic, Proc. Nat. Acad. Sci. U.S.A. 20, 584-590.

Curry H., Feys R. (1958) Combinatory logic I, North-Holland.

Curry H. (1969) Modified basic functionality in combinatory logic, Dialectica 23, 83-92.



Fortune S., Leivant D., O'Donnel M. (1983) The expressiveness of simple and second order type structures, J. ACM 30, 151-185.

Gandy R. (1980 I) An early proof of normalization by A.M. Turing, in To H.B.Curry: essays on combinatory logic, lambda-calculus and formalism, J. Seldin and R. Hindley eds., Academic Press, 453-455.

Gandy (1980 II) Proofs of strong normalization, in To H.B.Curry: essays on combinatory logic, lambda-calculus and formalism, J. Seldin and R. Hindley eds., Academic Press, 457-477.

Girard J.Y. (1972) Interpretation functionelle et elimination des coupures dans l'aritmetique d'ordre superieur, These d'Etat, Paris VII.

Girard J.Y. (1973) Quelque resultats sur les interpretations fonctionelles, Cambridge summer scool in mathematical logic, Lecture Notes in Mathematics 337, Sringer-Verlag.

Girard J.Y. (1985) The system F of variable types fifteen years later, rapporto interno, Universita' Paris VII.

Godel K. (1958) Uber eine bisher noch benutzte erweinterung des finiten standpunktes, Dialectica 12, 280-287.

Gordon M., Milner R., Wadsworth C. (1979) Edinburgh LCF, Lecture Notes in Computer Science 78, Springer-Verlag.

Henderson P. (1980) Functional programming, Prentice-Hall.

Hindley R. (1969) The principal type-scheme of an object in combinatory logic, Trans. Amer. Math. Soc. 146, 29-40.

Hindley R. (1983) The completeness theorem for typing lambda-terms, Theor. Comput. Sci. 22, 1-18.

Hindley R. (1984) Coppo-Dezani types do not correspond to propositional logic, Theor. Comput. Sci. 28, 235-236.

Hindley R., Seldin J. (1986) Introduction to combinators and λ-calculus, Cambridge University Press.

Howard W. (1980) The formulas-as-types notion of construction, in To H.B.Curry: essays on combinatory logic, lambda-calculus and formalism, J. Seldin and R. Hindley eds., Academic Press, 479-490.

Kreisel G. (1959) Interpretation of analysis by means of constructive functionals of finite types, in "Constructivity of mathematics" , A. Heyting ed., North-Holland, 101-128.

Landin P. (1965) A correspondence between ALGOL-60 and Church's lambda notation, Comm. ACM 8, 89-101 e 158-165.

Leivant D. (1983a) Polymorphic type inference, in Proc. of the 10th ACM Symp. on principles of programming languages, 88-98.

Leivant D. (1983b) Reasoning about functional programs and complexity classes associted with type disciplines, in Proc. of the 24th Symp. on Foundation of computer science. 460-469.

Liskov B., Snyder A., Atkinson R., Schaffert C. (1977) Abstraction mechanisms in CLU, Comm. ACM 20, 564-576.

MacQueen, Plotkin, Sethi (198-) An ideal model for recursive polymorphic types, di prossima pubblicazione su Information e Control.

Martin-Lof P. (1975) An intuitionistic theory of types: predicative part, in Logic colloquium '83, North-Holland, 73-118.

McCracken N. (1979) An investigation of a programming language with a polymorphic type structure, tesi, University of Syracuse (U.S.A.).

McCracken N. (1982) A finitary retract model for the polymorphic lambda-calculus, rapporto interno, University of Syracuse.

Milner R. (1978) A theory of type polymorphism in programming, J. Comput. System Sci. 17, 348-375.

Mulmuley K. (198-) Fully abstract submodels ot typed λ-calculi, di prossima pubblicazione su J.C.S.S.

Plotkin G. (1977) LCF considered as a programming language, Theor. Comput. Sci. 5, 223-257.

Plotkin G. (1978) The category of complete partial orders: a tool for making meanings, dispense per la Scuola estiva di informatica teorica, Pisa, Dipartimento di Informatica.

Prawitz D. (1965) Natural deduction, Almqvist and Wiksell, Stockholm.

Reynolds J. (1974) Toward a theory of type structure, in proc. of Coll. sur la programmation, Lecture Notes in Computer Science 19, Springer-Verlag, 408-425.

Reynolds J. (1984) Polymorphism is not set-theoretic, in Proc of Semantics of Data Types, Lecture Notes in Computer Science 173, 145-156.

Ronchi della Rocca S., Venneri B. (1984) Principal type schemes for an extended type theory, Theor. Comput. Sci. 28, 151-171.



Schutte K. (1977) Proof thoery, Springer-Verlag.

Schwichtenberg H. (1976) Definierbare funktionen im λ-kalkul mit typen, Archiv Math. Logik 17, 113-114.

Scott D. (1972) Continuous Lattices, Lecture Notes in Mathematics 274, Springer-Verlag, 97-136.

Scott D. (1976) Data types as lattices, SIAM J. Computing 5, 522-587.

Spector C. (1962) Provably recursive functionals of analysis, in Recursive function theory, Proc. Symposia Pure Maths. 5, Amer. Math. Soc., 1-28.

Statman R. (1979) The typed λ-calculus is not elementary recursive, Theor. Comput. Sci. 9, 73-82.

Statman R. (1981) Number theoretic functions computable by polymorphic programs, in Proc. of the 22th IEEE Symp. on Found. of Comput. Sci., 279-283.

Tait W. (1967) Intensional interpretation of functionals of finite type, J. Symbolic Logic 32, 198-212.

Troelstra A. (1973) Mathematical investigations of intuitionistic arithmetic and analysis, Lecture Notes in Mathematics 344, Springer-Verlag.

Wulf W., London R., Shaw M. (1976) An introduction to the construction and verificarion of ALPHARD programs, IEEE Trans. Softw. Eng. SE-2, 253-264.