Appunti di Lambda Calcolo dal corso di Fondamenti Logici dell'Informatica tenuto dal prof. Andrea Asperti. Anno Accademico 2004-2005.

Wilmer Ricciotti

18 luglio 2005

Indice

I Il λ-calcolo puro **7**

1 Implementazione **9**
1.1 Implementazione per sostituzione testuale 9
1.2 Notazione di De Bruijn 10
1.2.1 Codifica *name-free* dei λ-termini 10
1.2.2 Riduzione 11
1.2.3 Ricerca dei redex 12
1.2.4 Macchina di Krivine 13

2 Confluenza **15**
2.1 Sistemi di riscrittura e confluenza 16
2.2 Dimostrazione 18
2.2.1 λ-calcolo marcato 19
2.2.2 Preparativi 20
2.2.3 Dimostrazione dello strip lemma 21
2.2.4 Dimostrazione di Church-Rösser 25
2.3 Implicazioni 25

3 Teorie equazionali **27**
3.1 Le teorie $\lambda\beta$ e $\lambda\beta\eta$ 27
3.2 Il teorema di separabilità 30
3.2.1 Preparativi 31

II Calcoli tipizzati **35**

4 Il λ-calcolo tipizzato semplice **37**
4.1 Sintassi 37
4.2 Sistemi di tipi 38
4.2.1 Giudizi 39
4.2.2 Regole di tipo 39
4.2.3 Applicazioni 40
4.3 Programmi in $\lambda_{\rightarrow}$ 41
4.3.1 Potere espressivo 42

4.4 Corrispondenza di Curry-Howard 43
4.4.1 Il frammento implicativo 43
4.4.2 Il connettivo $\wedge$ 44
4.4.3 Il connettivo $\vee$ 46
4.5 Teoremi di normalizzazione 48
4.5.1 Teorema di normalizzazione debole 48
4.5.2 Teorema di normalizzazione forte 51

5 Il Calcolo delle Costruzioni **57**
5.1 Sintassi 58
5.2 Sistemi di tipi 58
5.2.1 Giudizi 59
5.2.2 Regole di tipo 59
5.2.3 Applicazioni 60
5.3 Programmi in $\lambda_{\rightarrow}$ 61
5.3.1 Potere espressivo 62
5.4 Corrispondenza di Curry-Howard 62
5.4.1 Il frammento implicativo 62
5.4.2 Il connettivo $\wedge$ 63
5.4.3 Il connettivo $\vee$ 65
5.5 Teoremi di normalizzazione 67
5.5.1 Teorema di normalizzazione debole 68

Elenco delle figure

2.1 Teorema di Church-Rösser 15
2.2 Strip lemma 19
2.3 Dimostrazione dello strip lemma. 25

4.1 Espressioni di tipo nel λ-calcolo tipizzato semplice 38
4.2 Termini del λ-calcolo tipizzato semplice 38
4.3 Esempio generico di regola di tipo 39
4.4 Regole di tipo per il λ-calcolo tipizzato semplice 40
4.5 Regole di inferenza per il connettivo $\Rightarrow$ 43
4.6 Albero di prova con etichette 44
4.7 Regole di inferenza per il connettivo $\wedge$ 44
4.8 Regole di tipo per i tipi coppia 45
4.9 Regole di inferenza per il connettivo $\vee$ 46
4.10 Regole di tipo per i tipi somma 47

5.1 Espressioni di tipo nel λ-calcolo tipizzato semplice 58
5.2 Termini del λ-calcolo tipizzato semplice 58
5.3 Esempio generico di regola di tipo 59
5.4 Regole di tipo per il λ-calcolo tipizzato semplice 60
5.5 Regole di inferenza per il connettivo $\Rightarrow$ 62
5.6 Albero di prova con etichette 63
5.7 Regole di inferenza per il connettivo $\wedge$ 64
5.8 Regole di tipo per i tipi coppia 64
5.9 Regole di inferenza per il connettivo $\vee$ 65
5.10 Regole di tipo per i tipi somma 66

Parte I

Il λ-calcolo puro

Capitolo 1

Implementazione

Dopo aver mostrato come nel λ-calcolo sia possibile esprimere qualunque funzione calcolabile, è giunto il momento di chiedersi come esso possa essere implementato nei calcolatori. L'argomento è interessante in quanto il lambda-calcolo rappresenta il modello formale su cui svariati linguaggi di programmazione sono fondati: l'esistenza di algoritmi più o meno efficienti per la riduzione di λ-termini si ripercuote dunque su molti linguaggi funzionali.

1.1 Implementazione per sostituzione testuale

La tecnica più semplice per l'implementazione del λ-calcolo e della riduzione di λ-termini, deriva dalla definizione stessa del linguaggio e della β-riduzione. Banalmente, si individua un qualunque *redex* (da *reducible expression*, espressione riducibile) nella forma $(\lambda x.MN)$ e lo si sostituisce con $M[N/x]$ (secondo la regola della β-riduzione).

Queste operazioni, a prima vista, possono essere compiute agevolmente tanto sulla stringa che rappresenta il λ-termine, quanto sull'albero di sintassi astratta; tuttavia uno sguardo più attento ci rivela la necessità di risolvere problemi non banali, sacrificando l'efficienza. In particolare, notiamo che la riduzione richiede una sostituzione che può essere applicata soltanto previo *occur-check*. Ogni controllo di occorrenza richiede tempo lineare e, forse, la ridenominazione di una variabile con una fresca. Non siamo disposti a pagare un tempo lineare per ogni riduzione (specialmente se si considera che nella gran parte dei casi il controllo di occorrenza non rivela collisioni, ed è dunque inutile), pertanto dovremo cambiare approccio.

Nella pratica, solamente le prime versioni di Lisp hanno utilizzato una variante di questo approccio, mantenendo liste di associazioni nome-valore. Ricordiamo però che Lisp non effettuava alcun controllo di occorrenza, implementando dunque la semantica dello scope dinamico, che a noi non interessa.

1.2 Notazione di De Bruijn

Dal momento che tutti i nostri problemi derivano dai nomi delle variabili, non ci meraviglia che l'idea che fornisce la soluzione del problema sia quella di passare a una notazione *name-free*, ossia priva di nomi.

1.2.1 Codifica *name-free* dei λ-termini

Osserviamo un λ-termine qualunque: come già mostrato, esso può essere rappresentato con un albero, i cui nodi interni sono astrazioni e applicazioni e le cui foglie sono le variabili. Per la proprietà degli alberi, ogni nodo è collegato alla radice da un unico cammino: non c'è dunque nessuna ambiguità nel definire ogni variabile non con un nome, ma con un indice numerico (appartenente ai numeri naturali) che indica di quante astrazioni risalire, lungo il cammino dalla foglia alla radice, per raggiungere l'astrazione cui la variabile è legata. Parallelamente, la notazione delle λ-astrazioni può essere modificata in modo da non indicare nomi di variabili, non più necessari. Pertanto un λ-termine

$$\lambda x.(x\ \lambda y.(y\ x))$$

viene trasformato in

$$\lambda.(0\ \lambda.(0\ 1))$$

È opportuno notare alcune particolarità di questa notazione, conosciuta con il nome di *notazione di De Bruijn*[1]. Due occorrenze della stessa variabile (ossia occorrenze di variabile legate allo stesso binder) possono assumere a seconda della loro posizione dentro l'albero sintattico indici diversi; allo stesso modo, non è assolutamente detto che due indici uguali si riferiscano alla stessa variabile: l'unico modo per capire a quale binder si riferisca un indice numerico è quello di risalire l'albero sintattico di un numero di livelli di astrazione pari al valore dell'indice. Per questo la notazione di De Bruijn è considerata abbastanza scomoda per la comprensione dei λ-termini e viene utilizzata soltanto per gli algoritmi di riduzione automatica.

Definiamo ora un algoritmo per riscrivere λ-termini nella codifica di De Bruijn. L'algoritmo è definito per induzione strutturale e si suppone che operi solamente su λ-termini chiusi.

$$[M]_{vars} = \begin{cases} \text{posizione di } y \text{ in } vars & \text{se } M \text{ è della forma } y \\ \lambda.[M']_{x::vars} & \text{se } M \text{ è della forma } \lambda x.M' \\ ([P]_{vars}\ [Q]_{vars}) & \text{se } M \text{ è della forma } (P\ Q) \end{cases}$$

L'idea è quella di decomporre ogni termine nei suoi sottotermini, tenendo traccia delle variabili legate ai livelli superiori. Per fare questo si utilizza

[1] Si pronuncia *de brain*.

una lista $vars$: inizialmente la lista è vuota, poi ad ogni astrazione la si amplia premettendo la variabile legata e si propaga la nuova lista ai livelli inferiori; nelle applicazioni ci si limita a propagare la lista ricevuta dal livello superiore; infine le variabili vengono sostituite con l'indice numerico corrispondente alla posizione del nome della variabile nella lista $vars$.

1.2.2 Riduzione

La codifica di De Bruijn sarebbe perfettamente inutile se non esistesse un metodo efficace ed efficiente per ridurre le espressioni codificate. Il metodo esiste ma sfortunatamente non è intuitivo quanto la sostituzione testuale; pertanto cercheremo di renderlo più comprensibile per mezzo di alcuni esempi.

Si supponga di voler ridurre il redex più esterno di

$$\lambda z.(\lambda x.((z\ x)\ \lambda y.(y\ x))\ \lambda w.(w\ z))$$

La parte destra del redex andrà sostituita a tutte le occorrenze di x nella parte sinistra. Nella codifica di De Bruijn, l'operazione diventa meno ovvia, dato che le variabili non sono più identificate per nome. Ad esempio in

$$\lambda.(\lambda.((1\ 0)\ \lambda.(0\ 1))\ \lambda.(0\ 1))$$

la sostituzione dovrà avvenire in corrispondenza di indici diversi (cerchiati in fig.); come se non bastasse, l'indice corrispondente a z nella parte sinistra del redex dovrà essere decrementato (per via dell'eliminazione della λ che legava x), mentre l'indice corrispondente a z nella parte destra del redex, essendo copiato due volte e a livelli di annidamento diversi in seguito alla riduzione, rimarrà pari a 1 nel primo caso, mentre sarà incrementato a 2 nel secondo.

Per formulare un primo algoritmo in grado di ridurre termini codificati secondo De Bruijn, notiamo che in seguito a una sostituzione:

- gli indici di variabili libere (non locali) del sottotermine sinistro del redex considerato devono essere decrementati di 1
- gli indici di variabili legate del sottotermine sinistro, eccetto l'indice della variabile da sostituire, non sono modificati
- gli indici della variabile da sostituire nel sottotermine sinistro sono sostituiti con il sottotermine destro del redex, in cui le variabili libere vengono incrementate di un valore pari all'annidamento di ciascuna occorrenza della variabile da sostituire.

Definiamo ora un algoritmo[2] per ridurre termini codificati secondo De Bruijn. Per prima cosa definiamo una funzione `lift` per incrementare le variabili libere di un λ-termine di un dato valore.

```
let lift n M =
    let rec liftaux k M =
        match M with
            "λ.P" -> "λ" ^ liftaux (k + 1) P
            | "(P Q)" -> "(" ^ (liftaux k P) ^ (liftaux k Q) ^ ")"
            | m -> (* indice di variabile *)
                if m < k (* locale *) then m
                else m + n
    in liftaux 0 M;;
```

A questo punto è possibile definire la funzione che sostituisce N in M secondo la semantica della riduzione:

```
let subst N M =
    let rec substaux k M =
        match M with
            "λ.P" -> "λ" ^ substaux (k + 1) P
            | "(P Q)" -> "(" ^ (substaux k P) ^ (substaux k Q) ^ ")"
            | m -> (* indice di variabile *)
                if m < k (* locale *) then m
                else if m = k (* da sostituire *)
                    then lift k N
                else (* libera in M *) m - 1
    in substaux 0 M;;
```

1.2.3 Ricerca dei redex

Abbiamo definito un algoritmo per calcolare β-riduzioni nella codifica di De Bruijn; tuttavia per interpretare il λ-calcolo è necessario definire anche una *strategia* per la ricerca dei redex: infatti in un termine possono essere presenti molteplici redex e la scelta del redex da ridurre a ogni passo influenza sia l'efficienza dell'interpretazione (ossia il numero di riduzioni da compiere prima di raggiungere la forma normale) sia la terminazione (non ogni strategia di riduzione consente di giungere alla forma normale - se esiste - in un numero finito di passi).

Due opposte strategie per la riduzione sono la *chiamata per valore* (in cui vengono ridotte soltanto espressioni il cui termine destro è privo di redex) e la *chiamata per nome* (che al contrario riduce sempre il redex più esterno, qualunque sia la forma del parametro del redex).

La strategia **leftmost-outermost** implementa la chiamata per nome. Essa esiste in due varianti:

[2] Questo algoritmo e i successivi sono descritto con una notazione pseudo-ML. Si ricorda che in ML l'operatore `^` esprime la concatenazione di stringhe e che l'istruzione `match` corrisponde all'incirca a uno `switch-case`.

- variante *weak*: non riduce mai all'interno di λ-astrazioni.
- variante estesa: riduce anche all'interno di λ-astrazioni; si può dimostrare che è una strategia *safe*, nel senso che se un termine ha una forma normale, è sempre possibile raggiungerla seguendo questa strategia.

L'implementazione della strategia leftmost-outermost estesa utilizza una pila e una ricerca in profondità, finché non viene individuata una λ-astrazione.

codice	**pila**	$\rightarrow$	**codice**	**pila**
$(M\ N)$	P		M	N::P
$\lambda.M$	N::P		(subst $M\ N$)	P
$\lambda.M$	$\emptyset$		dipende ...	

1.2.4 Macchina di Krivine

Sorgenti di inefficienza

Nella strategia implementata fin qui, purtroppo, ogni riduzione richiede una scansione completa del ramo sinistro del redex. Può capitare che due riduzioni consecutive scandiscano due volte essenzialmente lo stesso albero, mentre sarebbe opportuno scandirlo una sola volta e poi sostituire contemporaneamente tutte le variabili coinvolte nelle riduzioni.

Per effettuare un'operazione del genere, in pratica, occorre tenere traccia dei termini che andranno sostituiti. È per questo che per migliorare l'efficienza della nostra strategia introduciamo la nozione di *ambiente*. L'ambiente (concettualmente simile alla pila dei record di attivazione utilizzata dai linguaggi di programmazione) ci consente di differire ogni sostituzione fino a quando non è strettamente necessaria, utilizzando una politica *lazy*. L'implementazione di questa strategia lazy necessita di una procedura `msubst` che valuta le variabili libere di un termine `M` all'interno dell'ambiente `e`.

```
let msubst M e =
    let n = length e in
        let rec msubstaux M k = match M with
            "(P Q)" -> "(" ^ (msubstaux P k) ^ (msubstaux Q k) ^ ")"
            "λ.P" -> "λ." ^ (msubstaux P (k + 1))
            m -> if m < k (* locale *) then m
                else if k <= m < k + n (* da sostituire *)
                    then (lift k (nth (m - k) e))
                else (* libera *) m - n
        in msubstaux M 0;;
```

La procedura `msubst` opera una sostituzione multipla e sfrutta la funzione `nth m e` che restituisce l'`m`-esimo elemento della lista `e`. L'algoritmo può essere utilizzato secondo la seguente tabella:

ambiente	**codice**	**pila**	$\longrightarrow$	**ambiente**	**codice**	**pila**
e	$(M\ N)$	P		e	M	(msubst N e)::P
e	$\lambda.M$	N::P		N::e	M	(msubst N e)::P
e	m	P		$\emptyset$	(nth m e)	P

Sfortunatamente `msubst` è un'operazione complessa, che richiede un tempo lineare, per via della scansione della lista dovuta alla funzione `nth`. Si può fare di meglio riprogettando la struttura dell'ambiente ed estendendo ulteriormente l'approccio lazy, come nella macchina che discuteremo nel prossimo paragrafo.

La macchina di Krivine

La seguente tabella descrive la *macchina di Krivine*, proposta dall'omonimo matematico francese, tuttora vivente.

ambiente	**codice**	**pila**	$\longrightarrow$	**ambiente**	**codice**	**pila**
e	$(M\ N)$	P		e	M	$\langle N, e\rangle$::P
e	$\lambda.M$	**c**::P		**c**::e	M	P
$\langle N, e\rangle$::e′	0	P		e	N	P
c::e	n + 1	P		e	n	P
e	$\lambda.M$	$\emptyset$		risultato = $\langle \lambda M, e\rangle$		

Capitolo 2

Confluenza

L'obiettivo di questo capitolo sarà dimostrare un importante teorema sul λ-calcolo, dovuto a Church e Rösser, che qui enunciamo.

Teorema 2.1 (Church-Rösser)**.** *Sia M un termine. Se esistono due catene di riduzioni tali che $M \xrightarrow{*}_\beta P$ e $M \xrightarrow{*}_\beta Q$, allora esistono un termine N e altre due catene di riduzioni $P \xrightarrow{*}_\beta N$ e $Q \xrightarrow{*}_\beta N$.*

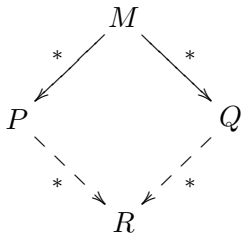


Figura 2.1: Teorema di Church-Rösser

Una rappresentazione grafica del teorema è data in fig. 2.1. Ricordiamo che le frecce continue esprimono una quantificazione universale, mentre quelle tratteggiate esprimono una quantificazione esistenziale.

Prima di passare alla dimostrazione, abbastanza complessa, di questa proprietà del λ-calcolo (denominata *confluenza*), vogliamo spiegarne intuitivamente la portata.

In primo luogo, essa **non dice nulla sulla terminazione del calcolo**. Sappiamo già che il λ-calcolo ammette computazioni non terminanti, ossia che non riducono a forma normale: precisiamo però che anche qualora un termine possieda una forma normale, da esso possono avere origine catene di riduzioni infinite. Ad esempio, nel termine

$$(\lambda xy.y\ (\delta\ \delta))$$

ogni riduzione sul redex interno riporta al termine iniziale, generando una catena infinita; d'altra parte è sempre possibile decidere di ridurre il redex più esterno, ottenendo

$$\lambda y.y$$

che è un termine in forma normale. In pratica, anche se una strategia di riduzione non conduce alla forma normale, in ogni momento posso decidere di cambiare strategia, e quindi con la strategia giusta calcolare la forma normale.

2.1 Sistemi di riscrittura e confluenza

È complicato dimostrare la confluenza generale nel λ-calcolo per via delle due premesse di lunghezza arbitraria. È evidente che per raggiungere qualche risultato dovremo cominciare dimostrando una versione semplificata del problema e poi dimostrare che questa implica la confluenza generale.

Definizione 2.2 (proprietà del diamante)**.** Un sistema di riscrittura gode della *proprietà del diamante* (o *Church-Rösser in un passo*) se, dati un termine M e due riduzioni (in un passo) $M \to P$ e $M \to Q$, esistono un termine N e due riduzioni (in un passo) $P \to N$ e $Q \to N$.

Questa proprietà è una condizione sufficiente per avere confluenza generale: intuitivamente, con la confluenza in un passo potremmo riempire il "diamante grande" della confluenza generale (la dimostrazione formale è lasciata come esercizio). Purtroppo questa proprietà non vale nel λ-calcolo. Si consideri ad esempio il termine

$$(\delta\ (I\ I))$$

contenente due redex. Un passo di riduzione sul redex esterno produce $((I\ I)\ (I\ I))$, mentre riducendo il redex interno otteniamo $(\delta\ I)$. L'unica riduzione successiva possibile in quest'ultimo caso produce $(I\ I)$, ma è evidente che per richiudersi su questo termine avendo seguito l'altro lato del diamante occorreranno due riduzioni. Questo è dovuto al fatto che ogni riduzione può produrre più copie del sottotermine di destra nel sottotermine di sinistra[1].

Un'altra proprietà simile alla confluenza, ma più semplice, è la *confluenza locale*.

Definizione 2.3 (confluenza locale)**.** Un sistema di riscrittura soddisfa la *confluenza locale* se, dati un termine M e due riduzioni (in un passo) $M \to$

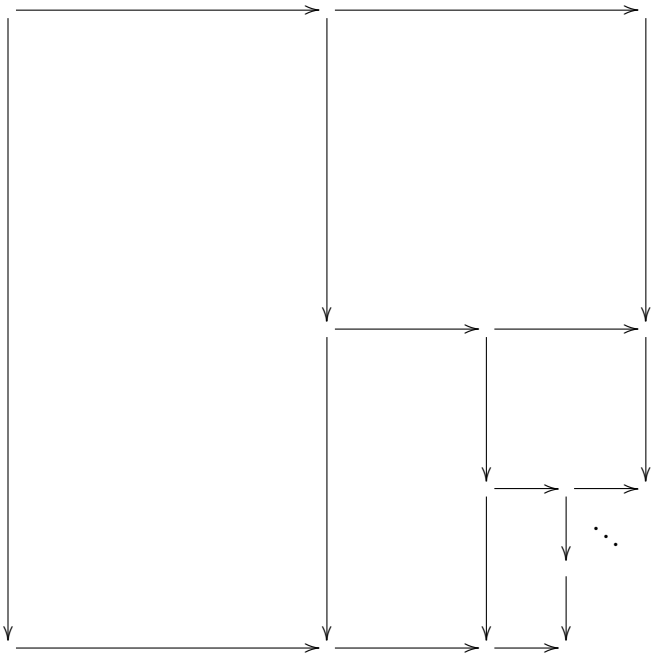
[1] Esistono sistemi di riscrittura *lineari*, in cui i termini non vengono duplicati, per i quali vale la proprietà del diamante.

P e $M \to Q$, esistono un termine N e due catene di riduzioni $P \xrightarrow{*} N$ e $Q \xrightarrow{*} N$.

La confluenza locale, però, non implica la confluenza generale. Un sistema di riscrittura per cui vale la confluenza locale ma non la confluenza generale è schematizzato nel seguente diagramma:

È facile vedere che queste regole di riscrittura rispettano la confluenza locale: nell'unico caso interessante, da A è possibile derivare in un passo B e C, e infine confluire da B in C in due passi. Tuttavia la confluenza generale non vale perché, sebbene sia possibile derivare da A in più passi sia C sia D, poi non è possibile chiudere il diamante.

Un altro esempio di sistema in cui vale solo la confluenza locale è mostrato nel seguente diagramma.



Dal momento che entrambi i sistemi di riscrittura mostrati in cui vale la confluenza locale ma non la confluenza generale sono non terminanti, ci chiediamo se confluenza locale e terminazione siano una condizione sufficiente per garantire la confluenza generica. La risposta è affermativa.

Teorema 2.4. *Se in un sistema di riscrittura sempre terminante vale la confluenza locale, allora vale anche la confluenza generale.*

Dimostrazione. Sia $\nu(M)$ la funzione che restituisce la lunghezza massima delle catene di riduzione aventi origine M. Indicando con $\xrightarrow{+}$ la riduzione

in almeno un passo, senz'altro vale l'implicazione:

$$M \xrightarrow{+} P \Rightarrow \nu(M) > \nu(P)$$

(si noti che questa proposizione è vera soltanto se P non ammette catene di derivazioni infinite).

Vogliamo ora dimostrare che se da M si derivano in più passi P e Q, esiste un termine N tale che da P e da Q si deriva in più passi N. La dimostrazione è per induzione su $\nu(M)$.

Nel caso base $\nu(M) = 0$, deve essere $M = P = Q = N$, pertanto non abbiamo nulla da dimostrare.

Nel caso induttivo, vale la confluenza generale per termini M' tali che $\nu(M') < \nu(M)$ (per ipotesi induttiva). Supponiamo che le catene di riduzioni da M a P e Q siano come le seguenti:

$$M \to M' \xrightarrow{*} P$$

$$M \to M'' \xrightarrow{*} Q$$

Allora per l'ipotesi di confluenza locale esiste un termine R tale che $M' \xrightarrow{*} R$ e $M'' \xrightarrow{*} R$.

D'altra parte $\nu(M') < \nu(M)$; quindi per ipotesi induttiva esiste un termine P' tale che $P \xrightarrow{*} P'$ e $R \xrightarrow{*} P'$.

Infine anche $\nu(M'') < \nu(M)$; quindi sempre per ipotesi induttiva esiste un termine N tale che $P' \xrightarrow{*} N$ e $Q \xrightarrow{*} N$. □

Si dimostra che nel λ-calcolo vale la confluenza locale. Tuttavia il λ-calcolo puro non è un formalismo terminante, pertanto ai nostri scopi anche la confluenza locale è inutile.

2.2 Dimostrazione

La proprietà che dovremo dimostrare per derivare la confluenza generale è la seguente.

Lemma 2.5 (strip lemma)**.** *Sia M un termine. Dati due termini P e Q tali che $M \to_\beta P$ (in un passo) e $M \xrightarrow{*}_\beta Q$ (in qualsiasi numero di passi), esiste un termine N tale che $P \xrightarrow{*}_\beta N$ e $Q \xrightarrow{*}_\beta N$ (in qualsiasi numero di passi).*

Il nome di questo lemma è dovuto alla sua rappresentazione grafica (fig. 2.2): una striscia (*strip*) del diagramma che rappresenta la confluenza generale. Intuitivamente, riducendo il diagramma in tante strisce quante sono necessarie, saremo in grado di dimostrare la confluenza.

Tuttavia, la dimostrazione dello strip lemma non è banale: essa richiede di tener traccia dei residui del redex $\mathscr{R} : M \to_\beta P$, che potrebbero essere moltiplicati dalla catena di riduzioni $\sigma : M \xrightarrow{*}_\beta Q$. Per riuscire nel nostro intento, necessitiamo di una sintassi che ci consenta di "marcare" i residui.

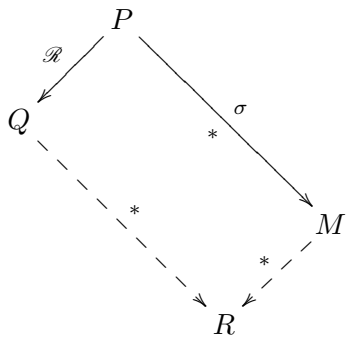


Figura 2.2: Strip lemma

2.2.1 λ-calcolo marcato

Per tracciare i residui del redex $\mathscr{R}$, l'idea è quella di etichettare $\mathscr{R}$ in modo tale che l'etichetta sia mantenuta (eventualmente duplicata) dalla catena di riduzioni σ. A questo scopo, dobbiamo estendere la sintassi del λ-calcolo.

Definizione 2.6 (λ-calcolo marcato)**.** L'insieme dei termini del λ-calcolo marcato, indicato con $\mathbf{\Lambda}^*$, è il più piccolo insieme definito come segue:

- se x è una variabile, allora $x \in \mathbf{\Lambda}^*$
- se $M \in \mathbf{\Lambda}^*$ e $N \in \mathbf{\Lambda}^*$, allora $(M\ N) \in \mathbf{\Lambda}^*$
- se $M \in \mathbf{\Lambda}^*$ e x è una variabile, allora $(\lambda x.M) \in \mathbf{\Lambda}^*$
- se $M \in \mathbf{\Lambda}^*$, $N \in \mathbf{\Lambda}^*$ e x è una variabile, allora $(\underline{\lambda x}.M\ N) \in \mathbf{\Lambda}^*$

Le regole di conversione e riduzione sono analoghe a quelle per il calcolo non marcato. In particolare la β-riduzione è definita dalle regole seguenti:

$$\begin{array}{lll} (\lambda x.M\ N) & \to_\beta & M[^N/_x] \\ (\underline{\lambda x}.M\ N) & \to_\beta & M[^N/_x] \end{array}$$

A rigore, bisognerebbe definire anche la sostituzione nel calcolo marcato; noi daremo per scontata la definizione di sostituzione, limitandoci a evidenziare che la sostituzione mantiene i redex marcati: in particolare una sostituzione $M[^N/_x]$ può duplicare i redex marcati eventualmente presenti in N, se x occorre in M più di una volta.

Definiamo anche due funzioni di trasformazione da termini marcati a termini non marcati: la funzione di cancellazione delle marche:

$$|\cdot| : \mathbf{\Lambda}^* \to \mathbf{\Lambda} = \begin{cases} |x| & = & x \\ |(M\ N)| & = & (|M|\ |N|) \\ |\lambda x.M| & = & \lambda x.|M| \\ |(\underline{\lambda x}.M\ N)| & = & (\lambda x.M\ N) \end{cases}$$

e la funzione di riduzione dei redex etichettati:

$$\varphi : \mathbf{\Lambda}^* \to \mathbf{\Lambda} = \begin{cases} \varphi(x) &= x \\ \varphi((M\ N)) &= (\varphi(M)\ \varphi(N)) \\ \varphi(\lambda x.M) &= \lambda x.\varphi(M) \\ \varphi((\underline{\lambda x}.M\ N)) &= \varphi(M)[\varphi(N)/x] \end{cases}$$

2.2.2 Preparativi

Prima di dimostrare lo strip lemma, introduciamo le seguenti proprietà della sostituzione e del calcolo etichettato.

Lemma 2.7. *Siano P, Q e R termini del λ-calcolo etichettato e x, y variabili: se y non è libera in R, allora $P[Q/y][R/x] = P[R/x][Q[R/x]/y]$.*

(per il momento non lo dimostro: non fatto a lezione).

Lemma 2.8. *Siano M e N termini del λ-calcolo etichettato e x una variabile: allora $\varphi(M)[\varphi(N)/x] = \varphi(M[N/x])$.*

Dimostrazione. Procediamo per induzione sulla struttura di M.

Se $M = x$:

$$\begin{array}{rcll} \varphi(x)[\varphi(N)/x] &=& x[\varphi(N)/x] & \text{(def. di } \varphi\text{)} \\ &=& \varphi(N) & \text{(def. di sostituzione)} \\ &=& \varphi(x[N/x]) & \text{(def. di sostituz., all'indietro)} \end{array}$$

Si noti il trucco, che sarà utilizzato in tutti i passi della dimostrazione, in cui la parte finale dell'equivalenza è ottenuta procedendo "all'indietro".

Se $M = y \neq x$:

$$\begin{array}{rcll} \varphi(y)[\varphi(N)/x] &=& y[\varphi(N)/x] & \text{(def. di } \varphi\text{)} \\ &=& y & \text{(def. di sostituzione)} \\ &=& \varphi(y[N/x]) & \text{(def. di sostituz. e di } \varphi\text{, all'indietro)} \end{array}$$

Se $M = (P\ Q)$: in questo passo induttivo e nei successivi, dovendo dimostrare un'equivalenza del tipo $a = b$, procederemo semplificando prima a e poi b e mostrando infine che l'equivalenza semplificata è banalmente implicata dall'ipotesi induttiva. Si ha:

$$\begin{array}{rcll} \varphi((P\ Q))[\varphi(N)/x] &=& (\varphi(P)\ \varphi(Q))[\varphi(N)/x] & \text{(def. di } \varphi\text{)} \\ &=& (\varphi(P)[\varphi(N)/x]\ \varphi(Q)[\varphi(N)/x]) & \text{(def. di sostituzione)} \end{array}$$

Parallelamente:

$$\begin{array}{rcll} \varphi((P\ Q)[N/x]) &=& (\varphi(P[N/x]\ Q[N/x]) & \text{(def. di sostituzione)} \\ &=& (\varphi(P[N/x])\ \varphi(Q[N/x])) & \text{(def. di } \varphi\text{)} \end{array}$$

Infine, per ipotesi induttiva:

$$(\varphi(P)[\varphi(N)/x]\ \varphi(Q)[\varphi(N)/x]) \quad = \quad \varphi(P[N/x]\ Q[N/x])$$

Se $M = \lambda y.P$**:** possiamo supporre, senza perdita di generalità, che y non sia una variabile libera in N (in caso contrario possiamo α-convertire M sostituendo y con una variabile fresca). Si ha:

$$\begin{array}{rcll} \varphi(\lambda y.P)[\varphi(N)/x] & = & (\lambda y.\varphi(P))[\varphi(N)/x] & \text{(def. di } \varphi\text{)} \\ & = & \lambda y.(\varphi(P)[\varphi(N)/x]) & \text{(def. di sostituzione)} \end{array}$$

Parallelamente:

$$\begin{array}{rcll} \varphi((\lambda y.P)[N/x]) & = & \varphi(\lambda y.(P[N/x])) & \text{(def. di sostituzione)} \\ & = & \lambda y.\varphi(P[N/x]) & \text{(def. di } \varphi\text{)} \end{array}$$

Infine, per ipotesi induttiva:

$$\lambda y.(\varphi(P)[\varphi(N)/x]) \quad = \quad \lambda y.\varphi(P[N/x])$$

Se $M = (\underline{\lambda y}.P\ Q)$**:** supponiamo, ancora senza perdita di generalità, che y non sia una variabile libera in N. Si ha:

$$\begin{array}{rcll} \varphi((\underline{\lambda y}.P\ Q))[\varphi(N)/x] & = & \varphi(P)[\varphi(Q)/y][\varphi(N)/x] & \text{(def. di } \varphi\text{)} \\ & = & \varphi(P)[\varphi(N)/x][\varphi(Q)[\varphi(N)/x]/y]] & \text{(lemma 2.7)} \end{array}$$

Parallelamente:

$$\begin{array}{rcll} \varphi((\underline{\lambda y}.P\ Q)[N/x]) & = & \varphi(\underline{\lambda y}.P[N/x]\ Q[N/x]] & \text{(def. di sostituzione)} \\ & = & \varphi(\overline{P[N/x]})[\varphi(Q[N/x])/y]] & \text{(def. di } \varphi\text{)} \end{array}$$

Infine, per ipotesi induttiva:

$$\varphi(P)[\varphi(N)/x][\varphi(Q)[\varphi(N)/x]/y]] \quad = \quad \varphi(P[N/x])[\varphi(Q[N/x])/y]]$$

(si noti che qui l'ipotesi induttiva è usata due volte, per $\varphi(P)[\varphi(N)/x] = \varphi(P[N/x])$ e $\varphi(Q)[\varphi(N)/x] = \varphi(Q[N/x])$.

□

2.2.3 Dimostrazione dello strip lemma

La dimostrazione è composta da tre sotto-lemmi, che enunceremo e dimostreremo singolarmente. Al termine, mostreremo come dai tre sotto-lemmi derivi lo strip lemma.

Lemma 2.9. *Siano M e N termini del λ-calcolo standard. Per ogni catena di riduzioni $\sigma : M \xrightarrow{*}_\beta N$ e per ogni termine M' del calcolo etichettato tale che $|M'| = M$, esistono un termine N' del calcolo etichettato e una catena di riduzioni $\tau : M' \xrightarrow{*}_\beta N'$ tali che $|N'| = N$. In altre parole il diagramma*

$$\begin{array}{ll} (\mathbf{\Lambda}) & M \xrightarrow[*]{\sigma} N \\ & |\cdot| \uparrow \qquad\qquad \uparrow |\cdot| \\ (\mathbf{\Lambda}^*) & M' \dashrightarrow[*]{\tau} N' \end{array}$$

commuta.

Il lemma ci dice che ogni riduzione del calcolo standard può essere simulata nel calcolo etichettato; dato che il calcolo etichettato è identico al calcolo standard eccetto che per i redex etichettati e che i redex etichettati vengono ridotti allo stesso modo dei redex normali, non dimostreremo questa proprietà, considerandola ovvia.

Lemma 2.10. *Siano M' e N' termini del λ-calcolo etichettato. Per ogni catena di riduzioni $\tau : M' \xrightarrow{*}_\beta N'$ esiste un'altra catena di riduzioni $\sigma : \varphi(M') \xrightarrow{*}_\beta \varphi(N')$. In altre parole il diagramma*

$$\begin{array}{ll} (\mathbf{\Lambda}^*) & M' \xrightarrow[*]{\tau} N' \\ & \varphi \downarrow \qquad\qquad \downarrow \varphi \\ (\mathbf{\Lambda}) & \varphi(M') \dashrightarrow[*]{\sigma} \varphi(N') \end{array}$$

commuta.

Dimostrazione. La dimostrazione è per induzione sulla lunghezza della catena τ. Ci limitiamo a dimostrare il caso base, ossia che τ sia una riduzione in un passo: procediamo per induzione sulla struttura dei termini che partecipano alla riduzione.

$(\lambda x.P\ Q) \longrightarrow_\beta P[Q/x]$**:** si ha:

$$\begin{array}{rcll} \varphi((\lambda x.P\ Q)) & = & (\lambda x.\varphi(P)\ \varphi(Q)) & \text{(def. di } \varphi) \\ & \longrightarrow_\beta & \varphi(P)[\varphi(Q)/x] & \\ & = & \varphi(P[Q/x]) & \text{(lemma 2.8)} \end{array}$$

$(\underline{\lambda x}.P\ Q) \longrightarrow_\beta P[Q/x]$**:** si ha:

$$\begin{array}{rcll} \varphi((\underline{\lambda x}.P\ Q)) & = & \varphi(P)[\varphi(Q)/x] & \text{(def. di } \varphi) \\ & = & \varphi(P[Q/x]) & \text{(lemma 2.8)} \end{array}$$

$(P\ Q) \longrightarrow_\beta (P'\ Q)$**:** si hanno:

$$\begin{aligned} \varphi((P\ Q)) &= (\varphi(P)\ \varphi(Q)) \\ \varphi((P'\ Q)) &= (\varphi(P')\ \varphi(Q)) \end{aligned}$$

D'altra parte, per ipotesi induttiva, se $P \longrightarrow_\beta P'$ allora $\varphi(P) \xrightarrow{*}_\beta \varphi(P')$; pertanto $(\varphi(P)\ \varphi(Q)) \xrightarrow{*}_\beta (\varphi(P')\ \varphi(Q))$.

$(P\ Q) \longrightarrow_\beta (P\ Q')$**:** analogo al caso precedente.

$\lambda x.P \longrightarrow_\beta \lambda x.P'$**:** si hanno:

$$\begin{aligned} \varphi(\lambda x.P) &= \lambda x.\varphi(P) \\ \varphi(\lambda x.P') &= \lambda x.\varphi(P') \end{aligned}$$

D'altra parte, per ipotesi induttiva, se $P \longrightarrow_\beta P'$ allora $\varphi(P) \xrightarrow{*}_\beta \varphi(P')$; pertanto $\lambda x.\varphi(P) \xrightarrow{*}_\beta \lambda x.\varphi(P')$.

□

Lemma 2.11. *Per ogni termine M' del λ-calcolo etichettato, esiste una catena di riduzioni $\sigma : |M'| \xrightarrow{*}_\beta \varphi(M')$. In altre parole il diagramma*

$$\begin{array}{ll} (\mathbf{\Lambda}^*) & M' \\ & \quad {\scriptstyle |\cdot|} \swarrow \quad \searrow {\scriptstyle \varphi} \\ (\mathbf{\Lambda}) & |M'| \overset{\sigma}{\underset{*}{- - \succ}} \varphi(M') \end{array}$$

commuta.

Dimostrazione. La dimostrazione è per induzione strutturale su M.

$M = x$**:** $|x| = x = \varphi(x)$, pertanto il lemma è banalmente verificato.

$M = (P\ Q)$**:** si ha:

$$\begin{aligned} |(P\ Q)| &= (|P|\ |Q|) \\ \varphi((P\ Q)) &= (\varphi(P)\ \varphi(Q)) \end{aligned}$$

D'altra parte, per ipotesi induttiva:

$$\begin{aligned} \exists \sigma' : |P| &\xrightarrow{*}_\beta \varphi(P) \\ \exists \sigma'' : |Q| &\xrightarrow{*}_\beta \varphi(Q) \end{aligned}$$

Pertanto $\exists \sigma : (|P|\ |Q|) \xrightarrow{*}_\beta (\varphi(P)\ \varphi(Q))$.

$M = \lambda x.P$**:** si ha:

$$\begin{aligned} |\lambda x.P| &= \lambda x.|P| \\ \varphi(\lambda x.P) &= \lambda x.\varphi(P) \end{aligned}$$

D'altra parte, per ipotesi induttiva:

$$\exists \sigma' : |P| \quad \stackrel{*}{\longrightarrow}_\beta \quad \varphi(P)$$

Pertanto $\exists \sigma : (\lambda x.|P|) \stackrel{*}{\longrightarrow}_\beta \lambda x.\varphi(P)$.

$M = (\underline{\lambda x}.P\ Q)$**:** si ha:

$$\begin{aligned} |(\underline{\lambda x}.P\ Q)| &= (\lambda x.|P|\ |Q|) \\ \varphi((\underline{\lambda x}.P\ Q)) &= \varphi(P)[^{\varphi(Q)}/_x] \end{aligned}$$

A questo punto possiamo ridurre in un passo:

$$(\lambda x.|P|\ Q) \longrightarrow_\beta |P|[^{|Q|}/_x]$$

Per ipotesi induttiva:

$$\begin{aligned} |P| &\stackrel{*}{\longrightarrow}_\beta \varphi(P) \\ |Q| &\stackrel{*}{\longrightarrow}_\beta \varphi(Q) \end{aligned}$$

Per dimostrare il teorema, quindi, è ora sufficiente dimostrare la seguente proprietà:

$$\frac{M \stackrel{*}{\longrightarrow}_\beta M' \qquad N \stackrel{*}{\longrightarrow}_\beta N'}{M[^N/_x] \stackrel{*}{\longrightarrow}_\beta M'[^{N'}/_x]}$$

Per comodità, scomponiamo questa proprietà in due parti:

$$\frac{M \stackrel{*}{\longrightarrow}_\beta M'}{M[^N/_x] \stackrel{*}{\longrightarrow}_\beta M'[^N/_x]}$$

$$\frac{N \stackrel{*}{\longrightarrow}_\beta N'}{M[^N/_x] \stackrel{*}{\longrightarrow}_\beta M[^{N'}/_x]}$$

La prima può essere dimostrata agevolmente per induzione; per la seconda occorre dimostrare che la riduzione $M \stackrel{*}{\longrightarrow}_\beta M'$ può essere compiuta sulle varie copie di M presenti in N in modo indipendente (la dimostrazione completa è per induzione strutturale). □

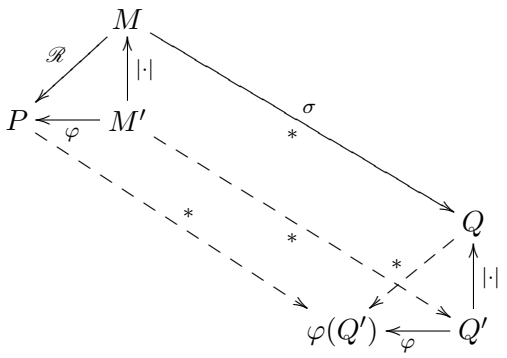


Figura 2.3: Dimostrazione dello strip lemma.

Siamo ora in grado di dimostrare lo strip lemma. Dati un termine M e due riduzioni $\mathscr{R} : M \longrightarrow_\beta P$ e $\sigma : M \xrightarrow{*}_\beta Q$, costruiamo il termine M' del λ-calcolomarcato, identico a M tranne per il fatto che il redex $\mathscr{R}$ è stato marcato. Per costruzione:

$$\begin{aligned} |M'| &= M \\ \varphi(M') &= P \end{aligned}$$

Per il lemma 2.9, la catena di riduzioni $M \xrightarrow{*}_\beta Q$ può essere simulata nel calcolo etichettato: esistono cioè un termine Q' tale che $|Q'| = Q$ e una catena di riduzioni per mezzo della quale $M' \xrightarrow{*}_\beta Q'$. Dal lemma 2.11 segue invece l'esistenza di una catena di riduzioni da $Q = |Q'|$ a $\varphi(Q')$. Infine, per il lemma 2.10, esiste una catenza di riduzioni da P a $\varphi(Q')$.

Pertanto $\varphi(Q')$ è il termine in cui *confluiscono* P e Q e lo strip lemma è dimostrato. □

2.2.4 Dimostrazione di Church-Rösser

Per induzione sulla lunghezza n della catena $M \xrightarrow{*}_\beta P$.

$n = 1$**:** segue direttamente dallo strip lemma.

$n > 1$**:** sia P' un termine tale che $M \longrightarrow_\beta P' \xrightarrow{*}_\beta P$. Per lo strip lemma, esiste un termine Q' tale che $P' \xrightarrow{*}_\beta Q'$ e $Q \xrightarrow{*}_\beta Q'$. Per ipotesi induttiva, essendo la catena $P' \xrightarrow{*}_\beta P$ lunga $n - 1$, esiste infine un termine R tale che $P \xrightarrow{*}_\beta R$ e $Q' \xrightarrow{*}_\beta R$. □

2.3 Implicazioni

Discutiamo brevemente alcune delle conseguenze immediate del teorema di Church-Rösser.

Corollario 2.12. *Se la forma normale di un λ-termine esiste, questa è unica.*

Dimostrazione. Supponiamo che M abbia due forme normali distinte P e Q. Dal teorema di Church-Rösser segue che esiste un termine R tale che $P \xrightarrow{*}_\beta R$ e $Q \xrightarrow{*}_\beta R$. Tuttavia, poiché P e Q sono in forma normale, queste derivazioni devono essere di lunghezza zero, ossia $P = R = Q$, contro l'ipotesi che voleva P e Q distinti: assurdo. □

Corollario 2.13. *Se due termini M e N sono β-equivalenti, esiste un termine P tale che $M \xrightarrow{*}_\beta P$ e $N \xrightarrow{*}_\beta P$.*

Dimostrazione. Dalla definizione di β-equivalenza, segue che esistono due sequenze $Q_1, Q_2, \ldots Q_{n-1}$ e $P_1, P_2, \ldots P_n$ di termini tali che:

$$\begin{array}{lll} M & \xrightarrow{*}_\beta & P_1 \\ Q_{i-1} & \xrightarrow{*}_\beta & P_i \quad i = 2, 3, \ldots n \\ Q_i & \xrightarrow{*}_\beta & P_i \quad i = 1, 2, \ldots n-1 \\ N & \xrightarrow{*}_\beta & P_n \end{array}$$

La dimostrazione è per induzione su n.

$n = 1$**:** la dimostrazione è ovvia, perché M e N riducono allo stesso termine P_1.

$n > 1$**:** per ipotesi induttiva, esiste un termine R tale che

$$\begin{array}{rcl} M & \xrightarrow{*}_\beta & R \\ Q_{n-1} & \xrightarrow{*}_\beta & R \end{array}$$

D'altra parte, per la definizione di β-equivalenza

$$\begin{array}{rcl} Q_{n-1} & \xrightarrow{*}_\beta & P_n \\ N & \xrightarrow{*}_\beta & P_n \end{array}$$

Poiché Q_{n-1} riduce ai due termini R e P_n, per il teorema di Church-Rösser esiste un termine P tale che

$$\begin{array}{rcl} R & \xrightarrow{*}_\beta & P \\ P_n & \xrightarrow{*}_\beta & P \end{array}$$

ma allora anche M ed N riducono in più passi a P ed il corollario è dimostrato. □

È evidente che quest'ultimo corollario ci offre un modo operativo per verificare la β-equivalenza di due termini dotati di forma normale: se la forma normale è la stessa, i due termini sono β-equivalenti, mentre se le forme normali sono differenti, i due termini non sono β-equivalenti. Se invece i due termini non hanno forma normale, questo procedimento non funziona; questo non ci sorprende: si dimostra infatti che la β-equivalenza non è decidibile nel caso generale.

Capitolo 3

Teorie equazionali

La relazione di β-riduzione formalizza la nozione di calcolo nel λ-calcolo: in altre parole due termini β-equivalenti sono due costrutti sintatticamente diversi che però *denotano* (dimostrabilmente) lo stesso oggetto. Come abbiamo visto (?), la β-equivalenza è una congruenza, ossia una relazione di equivalenza preservata da ogni contesto. Le espressioni del tipo

$$M =_\beta N$$

dove M e N sono termini, costituiscono un insieme di formule logiche che desideriamo rendere provabili sotto un'opportuna *teoria*. Ricordiamo che una teoria è un insieme di formule chiuso sotto una nozione di dimostrabilità o derivabilità.

3.1 Le teorie $\lambda\beta$ e $\lambda\beta\eta$

La teoria $\lambda\beta$ formalizza la β-equivalenza. Essa prevede regole per definire $=$ come una relazione di equivalenza:

$$s = s \qquad \text{(RIFLESSIVITÀ)}$$

$$\frac{s = t}{t = s} \qquad \text{(SIMMETRIA)}$$

$$\frac{s = t \qquad t = u}{s = u} \qquad \text{(TRANSITIVITÀ)}$$

regole che assicurano che $=$ sia una congruenza:

$$\frac{s = s' \qquad t = t'}{(s\ t) = (s'\ t')} \qquad \text{(APPLICAZIONE)}$$

$$\frac{s = t}{\lambda x.s = \lambda x.t} \qquad \text{(ASTRAZIONE)}$$

e la β-equivalenza:

$$(\lambda x.s\ t) = s[t/x] \qquad (\beta)$$

L'obiettivo, ovviamente, è quello di catturare nella nostra teoria il maggior numero possibile di equivalenze semantiche tra termini, pur sapendo (dal primo?? teorema di incompletezza di Gödel) che non sarà possibile renderla completa, dal momento che è possibile esprimere in essa l'aritmetica di Peano.

Ci chiediamo dunque se esistono equivalenze interessanti che la teoria $\lambda\beta$ non cattura e se è possibile aggiungerle come assiomi, senza che la teoria diventi inconsistente. Per verificare se l'equivalenza di due termini P e Q può essere aggiunta alla teoria senza renderla inconsistente, basta verificare se esiste una sequenza di termini $N_1, ..., N_k$ tale che

$$\begin{aligned} (P\ N_1\ ...\ N_k) &= fst \\ (Q\ N_1\ ...\ N_k) &= snd \end{aligned}$$

È evidente che in un caso simile non potremmo aggiungere l'equivalenza alla teoria: infatti ne seguirebbe l'equivalenza tra fst e snd, e quindi l'equivalenza tra ogni coppia di termini M, N, come segue:

$$\frac{\dfrac{fst = snd}{(fst\ M\ N) = (snd\ M\ N)}\ \text{(APPLICAZIONE)}}{M = N}\ (\beta)$$

In una situazione simile, i due termini P e Q sono quindi detti *discriminabili*.

In pratica, potrebbe essere interessante mostrare l'equivalenza di vari programmi diversi, ad esempio le due forme di funzione successore per gli interi di Church:

$$\begin{aligned} P &: \lambda nxy.(x\ (n\ x\ y)) \\ Q &: \lambda nxy.(n\ x\ (x\ y)) \end{aligned}$$

Sfortunatamente P e Q sono termini discriminabili, come mostra il seguente esempio:

$$\begin{array}{lll}(P\ \lambda xy.snd\ \lambda z.fst\ N) & \longrightarrow_\beta & (\lambda z.fst\ (\lambda xy.snd\ \lambda z.fst\ N)) \\ & \longrightarrow_\beta & fst \\ (Q\ \lambda xy.snd\ \lambda z.fst\ N) & \longrightarrow_\beta & (\lambda xy.snd\ \lambda z.fst\ (N\ \lambda z.fst)) \\ & \longrightarrow_\beta & snd\end{array}$$

da cui segue che in una teoria consistente i due termini non possono essere uguagliati. Questo risultato potrebbe essere considerato sorprendente, dato che ci aspetteremmo che il comportamento dei due termini che calcolano il successore fosse coincidente; la spiegazione è però molto semplice: i due termini sono equivalenti soltanto se applicati agli interi di Church, e con altri parametri possono manifestare comportamenti differenti.

Si considerino invece i due termini seguenti:

$$\begin{array}{c}M \\ \lambda x.(M\ x)\end{array}$$

Supponendo che M sia in forma normale, anche $\lambda x.(M\ x)$ è in forma normale; d'altra parte, come sappiamo, due forme normali diverse non sono mai β-equivalenti; tuttavia, è abbastanza semplice convincersi che, qualora x non compaia libera in M, i due termini si comporteranno nello stesso modo una volta applicati a un altro termine. Infatti, qualunque sia P:

$$(\lambda x.(M\ x)\ P) \quad \longrightarrow_\beta \quad (M\ P)$$

Introduciamo allora la seguente regola

$$M = \lambda x.(M\ x) \qquad (\eta)$$

detta η-equivalenza, tenendo presente che essa è valida solamente se x non compare libera in M. L'η-equivalenza introduce il principio dell'equivalenza estensionale[1].

Proposizione 3.1 (equivalenza estensionale)**.** *Due λ-termini M e N sono estensionalmente equivalenti se e solo se sono $\beta\eta$-equivalenti, ossia per ogni termine P vale*

$$(M\ P) =_{\beta\eta} (N\ P) \Leftrightarrow M =_{\beta\eta} N$$

[1]Si ricorda che due programmi sono estensionalmente equivalenti se per ogni input restituiscono lo stesso output

Dimostrazione. $\Leftarrow$**)** Per ipotesi, vale $(M\ P) =_{\beta\eta} (N\ P)$ per ogni P. Scegliamo $P = z \notin FV(M)$. Allora per la regola (ASTRAZIONE) si ha

$$(M\ z) =_{\beta\eta} (N\ z) \Rightarrow \lambda z.(M\ z) =_{\beta\eta} (N\ z)\lambda z.(N\ z)$$

Poiché, dalla regola (η)

$$M =_{\beta\eta} \lambda z.(M\ z)$$
$$N =_{\beta\eta} \lambda z.(N\ z)$$

per la proprietà (TRANSITIVA) si ha

$$M =_{\beta\eta} N$$

$\Rightarrow$**)** Ovvia, dalla regola (APPLICAZIONE).

□

Osservazione 1. La η-equivalenza è una forma di estensionalità relativamente debole, poiché in genere siamo interessati a funzioni equivalenti non su ogni parametro possibile, ma su ogni parametro appartenente a una certa classe, come gli interi di Church relativamente alle due forme di funzione successore.

Osservazione 2. Accanto alla η-equivalenza, si può considerare una nozione di η-riduzione

$$\lambda x.(M\ x) \longrightarrow_{\eta} M$$

soggetta alla stessa restrizione sulla variabile x, che non può quindi comparire libera in M. Utilizzando anche questa regola, il calcolo rimane confluente, pertanto ha senso parlare di forme $\beta\eta$-normali.

3.2 Il teorema di separabilità

o di separazione?

Dimostreremo ora il seguente risultato, dovuto al matematico italiano Corrado Böhm.

Teorema 3.2 (separabilità)**.** *Siano M e N due termini chiusi in forma $\beta\eta$-normale e differenti. Allora esiste una sequenza di termini chiusi $P_1, \ldots, P_k$ tali che*

$$\begin{cases} (M\ P_1 \ldots P_k) &= fst \\ (N\ P_1 \ldots P_k) &= snd \end{cases}$$

3.2.1 Preparativi

Diamo ora alcune definizioni che ci saranno utili nella dimostrazione del teorema.

Definizione 3.3 (forma normale di testa)**.** Un λ-termine si dice in *forma normale di testa* (hnf) se è della forma $\lambda x_1 \dots x_n.(y\ M_1 \dots M_k)$, con $M_1 \dots M_k$ qualunque; y è detta "variabile di testa".
N è una forma normale di testa di M se N è in hnf e vale $M =_{\beta\eta} N$.

Le hnf si ottengono comunemente come passaggi intermedi di un processo di riduzione *leftmost-outermost*. Ovviamente non tutti i termini possiedono una hnf (si pensi al solito termine divergente $(\delta\ \delta)$). Ogni forma normale è anche una hnf.

Definizione 3.4 (permutatore)**.** Un *permutatore di ordine* n è un termine della forma

$$\alpha_n \triangleq \lambda x_1 \dots x_n x.(x\ x_1 \dots x_n)$$

Definizione 3.5 (albero di Böhm)**.** L'*albero di Böhm* di un λ-termine M, indicato con $\mathrm{BT}(M)$ è definito come segue:

$$\mathrm{BT}(M) = \begin{cases} \bot & \text{se } M \text{ non ha una forma normale di testa} \\ \lambda\bar{x}.y \quad \text{(BT}(M_1) \cdots \text{BT}(M_k)\text{)} & \text{se } M =_\beta \lambda\bar{x}.(y\ M_1 \cdots M_k) \end{cases}$$

Due alberi di Böhm differiscono al primo livello se differiscono o nella radice o nel numero dei figli della radice.

Definizione 3.6 (trasformazioni di Böhm)**.** Si definisce *trasformazione di Böhm* ogni composizione delle seguenti trasformazioni da λ-termini in λ-termini:

1. applicazioni $\mathbf{B}_N : M \mapsto (M\ N)$
2. sostituzioni $\mathbf{B}_{N,y} : M \mapsto M[N/y]$

L'applicazione di una o più trasformazioni a un termine M sarà indicata posponendo le trasformazioni al termine, ad esempio:

$$M\ \mathbf{B}_P\ \mathbf{B}_Q\ \mathbf{B}_{N,y} \equiv (M\ P\ Q)[N/y]$$

Le trasformazioni di Böhm consentono di istanziare le variabili libere di un termine (utilizzando sostituzioni) e quelle legate in un'astrazione posta al primo livello dell'albero di Böhm corrispondente (utilizzando applicazioni).

Dimostrazione. La dimostrazione ragiona sul livello dell'albero di Böhm in cui i due termini si differenziano: se si differenziano al primo livello, si costruiscono delle liste argomenti che consentono di ridurre i due termini rispettivamente a fst e a snd; in caso contrario, viene mostrata una tecnica, detta *Böhm-out* che consente di far emergere la differenza, spostandola fino al primo livello, utilizzando solamente trasformazioni di Böhm.

$$\begin{aligned} M &= \lambda\bar{x}.(y\ P_1\ \ldots\ P_r) \\ N &= \lambda\bar{y}.(z\ R_1\ \ldots\ R_s) \end{aligned}$$

si differenziano al primo livello se:

1. $|\bar{x}| \neq |\bar{y}|$
2. $y \neq z$
3. $r \neq s$

Nel caso 1 è sempre possibile trasformare il termine avente la lista di argomenti di lunghezza minore con η-equivalenze, fino a rendere le liste di argomenti di M e N uguale lunghezza. Inoltre, se $|x| = |y|$, è sempre possibile α-convertire i due termini in modo che $\bar{x} = \bar{y}$.

Nel caso 2 i due termini sono della forma

$$\begin{aligned} M &= \lambda\bar{x}.(y\ P_1\ \ldots\ P_r) \\ N &= \lambda\bar{x}.(z\ R_1\ \ldots\ R_s) \end{aligned}$$

In questo caso sfrutteremo la presenza di due variabili di testa differenti. Istanzieremo y con:

$$\lambda w_1 \ldots w_r.fst$$

e z con:

$$\lambda w_1 \ldots w_s.snd$$

Tutte le variabili legate in $\lambda\bar{x}$ e diverse da y e z, potranno essere istanziate dove necessario con un termine qualunque.

Nel caso 3 i due termini sono della forma

$$\begin{aligned} M &= \lambda\bar{x}.(y\ P_1\ \ldots\ P_r) \\ N &= \lambda\bar{x}.(y\ R_1\ \ldots\ R_s) \end{aligned}$$

Possiamo raggiungere il nostro scopo sfruttando la variabile y e la differenza nel numero di parametri che le vengono passati. Questa volta occorrerà istanziare y con:

$$\lambda z_1 \ldots z_{s+1}.z_{s+1}$$

Di seguito, istanzieremo tutte le variabili legate in $\lambda\bar{x}$ e diverse da y con termini qualunque. Infine applicheremo la seguente lista di parametri:

$$\lambda w_1 \dots w_{s-r}.snd \underbrace{* \cdots *}_{s-r-1} fst$$

dove gli asterischi indicano termini qualunque.

Nel caso in cui i due termini M e N differiscano a un livello inferiore, utilizziamo la seguente tecnica (*Böhm-out*) per far risalire la differenza di un livello. Supponiamo di avere i due termini

$$\begin{aligned} M &= \lambda\bar{x}.(y\ P_1\ \dots\ P_r) \\ N &= \lambda\bar{x}.(y\ R_1\ \dots\ R_r) \end{aligned}$$

e di voler far risalire la differenza presente nei sottotermini P_k e R_k. Considerando gli alberi di Böhm $\mathrm{BT}(M)$ e $\mathrm{BT}(N)$, si distinguono tre casi:

1. se la variabile di testa y non occorre altre volte nel percorso che conduce ai sottoalberi che differiscono al primo livello, può essere istanziata con l'opportuna proiezione, ovvero $\lambda x_1 \cdots x_r.x_k$ per far risalire il sottoalbero differente;

2. se la variabile di testa y occorre più volte nel percorso che conduce alla differenza, occorre effettuare un'operazione di linearizzazione per ricondursi al caso precedente; sia h il massimo numero di figli di y nel percorso suddetto: y verrà istanziata con α_h (il permutatore di ordine h;

3. ogni altra variabile, se necessario, andrà istanziata con una variabile libera fresca. □

Osservazione 3. Se ci limitiamo a equazioni tra termini in forma normale, non è possibile estendere ulteriormente la nostra teoria: dal teorema di separabilità segue che tutti i termini che applicati a un parametro qualunque danno risultati equivalenti, sono a loro volta equivalenti. Pertanto la teoria $\lambda\beta\eta$ è incompleta soltanto per quanto riguarda i termini che non hanno forma normale. In particolare equivalenze come

$$(\delta\ \delta) = ((\delta\ \delta)\ N)$$

non sono derivabili né renderebbero la teoria inconsistente; ovviamente la loro utilità è pressoché nulla.

Parte II

Calcoli tipizzati

Capitolo 4

Il λ-calcolo tipizzato semplice

Il concetto di tipo ci consente di trattare in maniera sintattica certe caratteristiche semantiche dei programmi che si possono scrivere in un linguaggio. Nei linguaggi tipizzati, è necessario associare alle variabili, alle espressioni e più in generale ai termini dei programmi delle annotazioni di tipo. Le annotazioni di tipo legali costituiscono il sottolinguaggio delle espressioni di tipo *ben formate*. A seconda del linguaggio, le annotazioni di tipo devono essere specificate esplicitamente dal programmatore, oppure possono essere generate in modo automatico dall'interprete o dal compilatore.

Semanticamente, per tipo intendiamo l'insieme di valori che una variabile, un'espressione o un frammento di programma possono assumere durante l'esecuzione. Nei linguaggi tipizzati, l'intervallo di valori che una variabile può assumere è limitato e i programmi devono soddisfare la proprietà di buona tipizzazione (*well-typedness*); nei linguaggi non tipizzati, è possibile effettuare operazioni prive di senso, e queste non saranno rilevate, proprio perché non è richiesta una buona tipizzazione: introducendo dei vincoli per il programmatore, la buona tipizzazione previene numerosi errori di esecuzione.

In questo capitolo introdurremo il λ-calcolo tipizzato semplice ($\lambda_{\rightarrow}$), analizzandone sintassi e semantica e verificandone il potere espressivo.

4.1 Sintassi

La sintassi del λ-calcolo tipizzato semplice si fonda su quella del λ-calcolo puro, discostandosene ben poco. L'estensione di maggior rilievo riguarda ovviamente l'insieme dei tipi legali: esso comprende un insieme finito di tipi atomici (non strutturati), che indicheremo con le lettere maiuscole $A, B, C \ldots$ e sui quali non faremo assunzioni, e un insieme numerabile di tipi induttivi (strutturati), costruiti a partire da un numero finito di *costruttori di tipo*. Il principale costruttore a cui siamo interessati è per ovvi motivi

il costruttore dei tipi funzione ($\rightarrow$), che prende due tipi T_1 e T_2 e costruisce il tipo delle funzioni da T_1 in T_2.

L'insieme dei tipi legali costituisce il sottolinguaggio delle espressioni di tipo, descritto in forma di grammatica in fig. 5.1.

Perché non allinea i ::= ?

$\langle T \rangle$ ::= $\langle AT \rangle$ | $\langle T \rangle \rightarrow \langle T \rangle$

$\langle AT \rangle$ ::= A | B | C | ...

Figura 4.1: Espressioni di tipo nel λ-calcolo tipizzato semplice

La sintassi dei termini è modificata soltanto nel caso dell'astrazione: è infatti necessario specificare un tipo per ogni variabile legata. La grammatica è presentata in fig. 5.2.

$\langle term \rangle$ ::= $\langle var \rangle$ | $\lambda \langle var \rangle : \langle T \rangle . \langle term \rangle$ | $(\langle term \rangle \ \langle term \rangle)$

$\langle var \rangle$ ::= x | y | z | x_1 | ...

Figura 4.2: Termini del λ-calcolo tipizzato semplice

4.2 Sistemi di tipi

Dopo aver definito la sintassi del calcolo tipizzato, ci chiediamo come sia possibile assegnare un tipo ai termini del linguaggio. Allo stato attuale, infatti, i tipi sono soltanto un'aggiunta cosmetica che non pone alcun vincolo. Ad esempio il termine

$$(\lambda x : A.x \ \lambda x : A.x)$$

dove A è un tipo atomico, non dovrebbe essere considerato corretto perché il termine passato come parametro non è di tipo A, bensì (intuitivamente) di tipo $A \rightarrow A$. Termini come questo sono detti *mal tipizzati* e non ci interessa esprimere computazioni su di essi (un compilatore li rigetterebbe segnalando un errore di semantica statica).

Ci interessa tuttavia trovare un modo per definire la nozione di buona tipizzazione e per distinguere i termini tipizzati bene da quelli tipizzati male. I *sistemi di tipi* sono collezioni di regole che ci consentiranno di raggiungere questo scopo.

Un buon sistema di tipi (utile in pratica per un linguaggio di programce lo mettiamo? mazione) deve godere delle seguenti proprietà [**?**]:

- deve essere decidibilmente verificabile: deve esistere un algoritmo di type-checking che verifichi se un programma è ben tipato.

- deve essere trasparente: il programmatore deve essere in grado di prevedere se un programma è ben tipato o meno e perché.
- deve essere *enforceable*: le dichiarazioni di tipo dovrebbero essere il più possibile verificate staticamente, altrimenti dinamicamente. La consistenza tra le dichiarazioni di tipo e il codice associato ad esse dev'essere verificabile da un programma.

Introduciamo ora il formalismo dei sistemi di tipi.

4.2.1 Giudizi

L'entità atomica trattata dai sistemi di tipi è un particolare tipo di espressione formale denominato *giudizio*. Un giudizio ha la forma:

$$\Gamma \vdash \mathfrak{J}$$

che si legge "Γ verifica $\mathfrak{J}$". $\mathfrak{J}$ è un'asserzione; Γ è detto contesto e definisce i tipi delle variabili che compaiono libere in $\mathfrak{J}$, nella forma $x_1 : A_1, \ldots, x_n : A_n$ (qui il simbolo ':' rappresenta la relazione "appartiene al tipo"). È necessario che tutte le variabili libere nell'asserzione siano dichiarate in Γ. Per il calcolo tipizzato semplice, siamo interessati a una sola forma di asserzione, vale a dire

$$\Gamma \vdash M : T$$

dove M è un termine e T è un tipo qualunque, che si legge "M ha tipo T in Γ"

Talvolta indicheremo un contesto in maniera esplicita elencandone le componenti (ad esempio: $x : S, y : T$); indicheremo inoltre la concatenazione di ambienti con la virgola (ad esempio: Γ, Γ').

4.2.2 Regole di tipo

Un giudizio può essere *valido* o *non valido*. La validità di un giudizio è definita sulla base di certe regole di inferenza che costituiscono il sistema di tipi formale.

Una regola (fig. 5.3) è composta da un certo numero di giudizi (*premesse*) posti sopra una linea orizzontale e da un unico giudizio (*conclusione*) posto sotto la linea: se sono valide le premesse, è valida la conclusione.

$$\frac{\Gamma_1 \vdash \mathfrak{J}_1 \ldots \Gamma_n \vdash \mathfrak{J}_n}{\Gamma \vdash \mathfrak{J}} \qquad \text{(REGOLA GENERICA)}$$

Figura 4.3: Esempio generico di regola di tipo

Se una regola non prevede premesse, si intende che la conclusione è intrinsecamente valida, cioè è un assioma.

Le regole possono essere composte in un albero di derivazione: la conseguenza di una qualsiasi regola può essere utilizzata come premessa di un'altra regola. Per verificare la validità di un giudizio occorre dare un albero di derivazione corretto, le cui foglie siano assiomi e la cui radice sia il giudizio da verificare.

Le regole di tipo (fig. 5.4) per il calcolo tipizzato semplice sono tre, una per ciascun tipo di termine. La regola per le variabili è un assioma che assegna a x il tipo T, posto che $x : T$ compaia nel contesto. Un'applicazione ha tipo T_2 se la parte funzionale ha tipo $T_1 \to T_2$ e l'argomento ha tipo T_1. Un'astrazione $\lambda x : T_1.M$ ha tipo $T_1 \to T_2$ se, supponendo che x abbia tipo T_1, M ha tipo T_2.

$$\Gamma \vdash x : T \qquad \text{se } x : T \in \Gamma \qquad \text{(T-AX)}$$

$$\frac{\Gamma \vdash M : T_1 \to T_2 \qquad \Gamma \vdash N : T_1}{\Gamma \vdash (M\ N) : T_2} \qquad \text{(T-APP)}$$

$$\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x : T_1.M : T_1 \to T_2} \qquad \text{(T-ABS)}$$

Figura 4.4: Regole di tipo per il λ-calcolo tipizzato semplice

4.2.3 Applicazioni

Dato un sistema di tipi, un termine M è ben tipizzato in un ambiente Γ se esiste un tipo A per cui il giudizio $\Gamma \vdash M : A$ è valido. Il problema di controllare l'esistenza di un tipo ben formato per i termini di un programma è detto *typechecking*. Il problema di identificare, se esiste, il tipo di un termine è detto invece *type inference* o inferenza dei tipi.

Si noti che non per tutti i sistemi di tipi esistono algoritmi di typechecking o type inference (se esiste un algoritmo di type inference, ovviamente esiste anche l'algoritmo di typechecking, ma non è vero il contrario): per alcuni sistemi, questi problemi sono indecidibili (non è possibile determinare per via algoritmica se il problema ha soluzione o meno) o semidecidibili (esiste un semialgoritmo che termina soltanto se il problema ha soluzione). Ovviamente in un sistema di tipi per un linguaggio di uso pratico il typechecking, e all'occorrenza la type inference, devono essere decidibili.

Una proprietà importante che i sistemi di tipi devono rispettare è connessa alla semantica operazionale dei programmi tipizzati. Affinché il sistema di tipi sia corretto è necessario garantire che se $\Gamma \vdash M : A$ è un giudizio

valido e il termine M riduce a un termine M', allora $\Gamma \vdash M' : A$. Questa proprietà è detta proprietà di "riduzione del soggetto".

4.3 Programmi in $\lambda_{\rightarrow}$

Per mostrare il potere espressivo del λ-calcolo tipizzato semplice, procederemo come al solito mostrando codifiche per booleani e naturali e per le operazioni che li manipolano. Per quanto riguarda i booleani, potremmo aggiungere in modo banale annotazioni di tipo alle codifiche usate nel λ-calcolo puro:

$$\begin{aligned}\mathrm{true} &\triangleq \lambda x^{T_1}.\lambda y^{T_2}.x : T_1 \to T_2 \to T_1\\ \mathrm{false} &\triangleq \lambda x^{T_1}.\lambda y^{T_2}.y : T_1 \to T_2 \to T_2\end{aligned}$$

Notiamo però che in questo modo il principale costrutto che utilizza booleani – l'if-then-else – sarebbe praticamente impossibile da tipizzare, sia perché true e false hanno tipo diverso, sia perché non sarebbe possibile prevedere il tipo del risultato (T_1 o T_2?).

La soluzione è quindi quella di utilizzare lo stesso tipo per entrambe le continuazioni dell'if, e modificare quindi il tipo dei booleani di conseguenza.

$$\begin{aligned}\mathrm{true}_T &\triangleq \lambda x^{T}.\lambda y^{T}.x : T \to T \to T\\ \mathrm{false}_T &\triangleq \lambda x^{T}.\lambda y^{T}.y : T \to T \to T\end{aligned}$$

Si possono quindi definire anche il tipo booleano e l'if-then-else (si noti che booleani e if-then-else sono parametrizzati rispetto al tipo delle continuazioni).

$$\begin{aligned}\mathrm{Bool}_T &\triangleq T \to T \to T\\ \mathrm{ITE}_T &\triangleq \lambda b^{\mathrm{Bool}_T}.\lambda x^{T}.\lambda y^{T}.(b\ x\ y) : \mathrm{Bool}_T \to T \to T \to T\end{aligned}$$

Definiamo la codifica di Church per gli interi in modo del tutto analogo al λ-calcolo puro. In primo luogo lo zero:

$$0 \triangleq \lambda x : T_1.\lambda y : T_2.y : T_1 \to T_2 \to T_2$$

La differenza sostanziale rispetto al caso non tipizzato sta nella necessità di assegnare dei tipi ai parametri della funzione che rappresenta lo zero. Questi tipi sono stati indicati genericamente con T_1 e T_2, perché idealmente vorremmo che lo zero fosse un valore *polimorfo* in cui T_1 e T_2 fossero istanziabili con qualunque tipo. Vediamo cosa accade cercando di tipizzare la definizione dell'1.

$$1 \triangleq \lambda x : T_1 \to T_2.\lambda y : T_1.(x\ y) : (T_1 \to T_2) \to T_1 \to T_2$$

Per l'1 otteniamo un termine di tipo diverso. Anche in questo caso T_1 e T_2 stanno ad indicare due tipi qualunque. Vediamo infine il caso del numero 2.

$$2 \triangleq \lambda x : T \to T.\lambda y : T.(x\ (x\ y)) : (T \to T) \to T \to T$$

Anche in questo caso il tipo del termine è differente; T denota un tipo qualunque. Per i numeri interi maggiori di 2 si può invece verificare che il tipo risulta identico a quello del 2.

Questa situazione anomala si risolve verificando che è possibile specializzare i tipi dei numeri 0 e 1, in modo che *unifichino* con il tipo di 2. In particolare in 0 occorre sostituire T_1 con $T \to T$ e T_2 con T; in 1 si sostituiscono sia T_1 sia T_2 con T.

Per concludere, il tipo dei numeri interi è definito in modo parametrico rispetto a un qualunque tipo T, come segue:

$$Nat_T \triangleq (T \to T) \to T \to T$$

e anche gli interi di Church e la funzione successore sono definiti in modo parametrico rispetto a T:

$$\begin{aligned}
&0_T \triangleq \lambda x : T \to T.\lambda y : T.y : Nat_T \\
&1_T \triangleq \lambda x : T \to T.\lambda y : T.(x\ y) : Nat_T \\
&2_T \triangleq \lambda x : T \to T.\lambda y : T.(x\ (x\ y)) : Nat_T \\
&\vdots \\
&succ_T \triangleq \lambda n : Nat_T.\lambda x : T \to T.\lambda y : T.(x\ (n\ x\ y)) : Nat_T \to Nat_T
\end{aligned}$$

4.3.1 Potere espressivo

La possibilità di definire booleani e interi non è sufficiente a garantire un'elevata espressività al nostro sistema di calcolo: sarebbe opportuno mostrare almeno una codifica per le funzioni primitive ricorsive. Tuttavia, una tale codifica non esiste: per la precisione, il λ-calcolo tipizzato semplice non consente di tipizzare nemmeno funzioni "semplici" come il predecessore. In sostanza, le uniche funzioni esprimibili sono somma, prodotto e composizioni di queste.

È possibile estendere il potere espressivo di questo calcolo introducendo operatori *ad hoc* per la ricorsione, o equivalentemente un operatore punto fisso

$$\begin{aligned}
&FIX_T : (T \to T) \to T \\
&(FIX_T\ M) \longrightarrow_\beta (M\ (FIX_T\ M)) \quad \text{se } M : T
\end{aligned}$$

verificare il tipo di M...

Neanche in questo modo, tuttavia, il calcolo diventa Turing-completo.

4.4 Corrispondenza di Curry-Howard

In questa sezione presenteremo un'interessante analogia tra calcoli tipizzati e sistemi logici deduttivi. La *corrispondenza di Curry-Howard* fa corrispondere alle formule logiche i tipi di un linguaggio tipizzato, alle prove i termini (e in particolare a una prova della formula A un termine di tipo A).

La corrispondenza funziona particolarmente bene per la logica intuizionista (e specialmente per certi frammenti di essa), ma può essere estesa, nello spirito di trasporre i progressi della logica nei linguaggi di programmazione e viceversa.

4.4.1 Il frammento implicativo

Il frammento implicativo della logica intuizionista include le formule logiche espresse per mezzo di proposizioni atomiche (A, B, ...) e del connettivo $\Rightarrow$.

Le dimostrazioni sono date, in deduzione naturale, per mezzo delle regole (fig. 5.5) di introduzione dell'implicazione ($\Rightarrow i$) e di eliminazione dell'implicazione (*modus ponens* o $\Rightarrow e$).

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \Rightarrow B} \quad (\Rightarrow i) \qquad\qquad \frac{A \Rightarrow B \qquad A}{B} \quad (\Rightarrow e)$$

Figura 4.5: Regole di inferenza per il connettivo $\Rightarrow$

Secondo la regola di introduzione, possiamo derivare una *dimostrazione* di $A \Rightarrow B$ (A implica B) se abbiamo un procedimento costruttivo (un *algoritmo*) che trasforma dimostrazioni di A in dimostrazioni di B; la regola di eliminazione ci consente invece di comporre una dimostrazione di $A \Rightarrow B$ con una dimostrazione di A per ottenere una dimostrazione di B.

La corrispondenza di Curry-Howard ci consente di associare alle dimostrazioni ottenute componendo queste regole opportuni termini del λ-calcolo tipizzato semplice. Al fine di rendere la corrispondenza più evidente, supporremo che alle foglie scaricate di un albero di prova e alle regole di introduzione dell'implicazione sia sempre associata un'etichetta (appartenente all'insieme $\{x, y, z, x_1, \ldots\}$) in modo da identificare la specifica regola di introduzione in cui la foglia è scaricata[1] (fig. 5.6).

La corrispondenza è data dalle seguenti regole induttive che associano a dimostrazioni termini di $\lambda_{\rightarrow}$:

[1] Questo accorgimento rende la corrispondenza di Curry-Howard un vero e proprio isomorfismo.

1. Data una dimostrazione costituita dalla singola ipotesi A non scaricata, si fa corrispondere una variabile fresca $x : A$.

2. Data una dimostrazione costituita dalla singola ipotesi A, scaricata ed etichettata y, si fa corrispondere la variabile $y : A$.

3. Data una dimostrazione che termina con $(\Rightarrow i)$, siano $S \Rightarrow T$ la conclusione della dimostrazione, y l'etichetta dell'ultimo passo di derivazione e u il termine associato alla sottodimostrazione immediatamente superiore. Si fa corrispondere il termine $\lambda y : S.u : S \rightarrow T$.

4. Data una dimostrazione che termina con $(\Rightarrow e)$, siano $u : S \rightarrow T$ e $v : S$ i termini associati alle due sottodimostrazioni immediatamente superiori all'ultimo passo di derivazione. Si fa corrispondere il termine $(u\ v) : T$.

4.4.2 Il connettivo $\wedge$

In fig. 5.7 sono riportate le regole per la congiunzione logica in deduzione naturale. È possibile dimostrare la congiunzione di due formule A e B disponendo di due dimostrazioni, rispettivamente per A e per B; inversamente, avendo una dimostrazione di $A \wedge B$, è possibile dimostrare A, oppure dimostrare B.

Il costruttore di tipo che corrisponde (nel senso della corrispondenza di Curry-Howard) al connettivo $\wedge$ in $\lambda_{\rightarrow}$ è il costruttore dei tipi coppia, detto anche *prodotto cartesiano*, che indicheremo con $\times$. A differenza che nel calcolo non tipizzato, in $\lambda_{\rightarrow}$ non è possibile esprimere coppie di termini in modo generale, pertanto introducendo nuovi costrutti sintattici per supportare le coppie aumenta il potere espressivo del calcolo.

$$\dfrac{\dfrac{[A]^x}{B \Rightarrow A}\ (\Rightarrow i : y)}{A \Rightarrow (B \Rightarrow A)}\ (\Rightarrow i : x)$$

Figura 4.6: Albero di prova con etichette

$$\dfrac{A \quad B}{A \wedge B}\ (\wedge i) \qquad \dfrac{A \wedge B}{A}\ (\wedge e.1) \qquad \dfrac{A \wedge B}{B}\ (\wedge e.2)$$

Figura 4.7: Regole di inferenza per il connettivo $\wedge$

Vogliamo quindi aumentare la sintassi del calcolo, aggiungendo alla sintassi dei tipi il caso dei tipi coppia e alla sintassi dei termini il costrutto coppia e le due proiezioni fst e snd:

$\langle T \rangle$ ::= ... | $\langle T \rangle \times \langle T \rangle$

$\langle term \rangle$ ::= ... | $\langle \langle term \rangle, \langle term \rangle \rangle$ | $(fst\ \langle term \rangle)$ | $(snd\ \langle term \rangle)$

differenziare le parentesi angolari nel metalinguaggio

$$\frac{\Gamma \vdash M : S \qquad \Gamma \vdash N : T}{\Gamma \vdash \langle M, N \rangle : S \times T} \quad \text{(T-PAIR)}$$

$$\frac{\Gamma \vdash P : S \times T}{\Gamma \vdash (fst\ P) : S} \quad \text{(T-PROJ.1)}$$

$$\frac{\Gamma \vdash P : S \times T}{\Gamma \vdash (snd\ P) : T} \quad \text{(T-PROJ.2)}$$

Figura 4.8: Regole di tipo per i tipi coppia

In fig. 5.8 sono date le regole di tipizzazione per i tipi coppia. Si noti che considereremo *redex* anche l'applicazione delle proiezioni a un costrutto coppia, pertanto vanno aggiunte le seguenti nuove forme di β-riduzione:

$$\begin{aligned} (fst\ \langle M, N \rangle) &\longrightarrow_\beta M \\ (snd\ \langle M, N \rangle) &\longrightarrow_\beta N \end{aligned}$$

La corrispondenza di Curry-Howard per il λ-calcolo tipizzato semplice esteso con coppie è espressa dalle seguenti regole che, aggiunte a quelle per il solo frammento implicativo, associano a dimostrazioni in deduzione naturale termini del calcolo.

5. Data una dimostrazione che termina con ($\wedge i$), siano $a : S$ e $b : T$ i termini associati alle due sottodimostrazioni immediatamente superiori all'ultimo passo di derivazione. Si fa corrispondere il termine $\langle a, b \rangle : S \times T$.

6. Data una dimostrazione che termina con ($\wedge e.1$), sia $p : S \times T$ il termine associato alla sottodimostrazione immediatamente superiore. Si fa corrispondere il termine $(fst\ p) : S$.

7. Data una dimostrazione che termina con ($\wedge e.2$), sia $p : S \times T$ il termine associato alla sottodimostrazione immediatamente superiore. Si fa corrispondere il termine $(snd\ p) : T$.

4.4.3 Il connettivo $\vee$

L'ultimo connettivo tipico della logica intuizionista che ci interessa è la disgiunzione. In fig. 5.9 sono riportate le regole della deduzione naturale relative ad essa. Si può ottenere una dimostrazione di $A \vee B$ a partire da una dimostrazione di A, oppure a partire da una dimostrazione di B; la regola di eliminazione è invece più complessa: combinando una dimostrazione di $A \vee B$ e due dimostrazioni di $A \Rightarrow C$ e $B \Rightarrow C$, otteniamo una dimostrazione di C (l'idea è che si può concludere C ragionando per casi: se è vero A, posso concludere C da $A \Rightarrow C$; se invece è vero B, posso concludere C da $B \Rightarrow C$).

$$\frac{A}{A \vee B}\ (\vee i.1) \qquad \frac{B}{A \vee B}\ (\vee i.2)$$

$$\frac{A \vee B \qquad A \Rightarrow C \qquad B \Rightarrow C}{C}\ (\vee e)$$

Figura 4.9: Regole di inferenza per il connettivo $\vee$

Alle disgiunzioni logiche, corrispondono i *tipi somma* (noti anche come *unioni disgiunte* o *varianti*). Oggigiorno i tipi somma sono in disuso, ma in passato ebbero una certa diffusione grazie alla possibilità di memorizzare tipi di dati diversi nella stessa cella di memoria, minimizzando gli sprechi.

Come già avevamo fatto per le coppie, adattiamo il λ-calcolo tipizzato semplice aggiungendo alla sintassi dei tipi il caso dei tipi somma e alla sintassi dei termini le due iniezioni in_1 e in_2 (che incapsulano un'oggetto di qualunque tipo in un tipo somma) e il costrutto $case$ (che ci consentirà di ragionare sul tipo contenuto in una somma):

$\langle T \rangle$ ::= ... | $\langle T \rangle + \langle T \rangle$

$\langle term \rangle$ ::= ... | $(\mathrm{in}_1\ \langle term \rangle)$ | $(\mathrm{in}_2\ \langle term \rangle)$ | $(\mathrm{case}\ \langle term \rangle\ \mathrm{of}\ (\mathrm{in}_1 \langle var \rangle) \rightarrow \langle term \rangle, (\mathrm{in}_2 \langle var \rangle) \rightarrow \langle term \rangle)$

Per non appesantire ulteriormente la notazione, abbiamo utilizzato una grammatica imprecisa: da una parte le due iniezioni dovrebbero specificare il tipo somma destinazione (altrimenti il termine $(\mathrm{in}_1\, true)$ potrebbe avere tipo $Bool + X$ per ogni X, e viceversa il termine $(\mathrm{in}_2\, 0)$ potrebbe avere tipo $Y + Nat$ per ogni Y). Per motivi puramente tipografici assumeremo quindi che le iniezioni restituiscano sempre il tipo somma opportuno. In fig. 5.10 mostriamo le regole di tipo per i tipi somma.

Considereremo un *redex* anche l'applicazione del costrutto $case$ a un'i-

$$\frac{\Gamma \vdash M : S}{\Gamma \vdash (\mathrm{in}_1\, M) : S + T} \quad \text{(T-INJ.1)}$$

$$\frac{\Gamma \vdash N : T}{\Gamma \vdash (\mathrm{in}_2\, N) : S + T} \quad \text{(T-INJ.2)}$$

$$\frac{\Gamma \vdash M : S + T \qquad \Gamma, x : S \vdash P : U \qquad \Gamma, y : T \vdash Q : U}{\Gamma \vdash (\mathrm{case}\, M\, \mathrm{of}\ (\mathrm{in}_1\, x) \to P, (\mathrm{in}_2\, y) \to Q) : U} \quad \text{(T-CASE)}$$

Figura 4.10: Regole di tipo per i tipi somma

niezione, pertanto vanno aggiunte le seguenti nuove forme di β-riduzione:

$$\begin{aligned} (\mathrm{case}\ (\mathrm{in}_1\, M)\, \mathrm{of}\ (\mathrm{in}_1\, x) \to P, (\mathrm{in}_2\, y) \to Q) &\longrightarrow_\beta P[^M/_x] \\ (\mathrm{case}\ (\mathrm{in}_2\, N)\, \mathrm{of}\ (\mathrm{in}_1\, x) \to P, (\mathrm{in}_2\, y) \to Q) &\longrightarrow_\beta Q[^N/_y] \end{aligned}$$

La corrispondenza di Curry-Howard viene estesa con le seguenti regole che, aggiunte a quelle per il solo frammento implicativo, associano a dimostrazioni in deduzione naturale termini del calcolo.

8. Data una dimostrazione che termina con $(\vee i.1)$, sia $S \vee T$ la conclusione della dimostrazione e sia $a : S$ il termine associato alla sottodimostrazione immediatamente superiore all'ultimo passo di derivazione. Si fa corrispondere il termine $(in_1\ a) : S + T$.

9. Data una dimostrazione che termina con $(\vee i.2)$, sia $S \vee T$ la conclusione della dimostrazione e sia $b : T$ il termine associato alla sottodimostrazione immediatamente superiore all'ultimo passo di derivazione. Si fa corrispondere il termine $(in_2\ b) : S + T$.

10. Data una dimostrazione che termina con $(\vee e)$, siano

$$\begin{aligned} &u : S + T \\ &\lambda x : S.P : S \to U \\ &\lambda y : T.Q : T \to U \end{aligned}$$

i termini associati alle sottodimostrazioni immediatamente superiori all'ultimo passo di derivazione. Si fa corrispondere il termine

$$(\mathrm{case}\, u\, \mathrm{of}\ (\in_1 x) \to P, (\in_2 y) \to Q) : U$$

4.5 Teoremi di normalizzazione

Abbiamo già avuto modo di precisare come il λ-calcolo tipizzato semplice non consenta di esprimere numerose funzioni ricorsive, rivelandosi quindi un calcolo non Turing-completo. Ha dunque senso chiedersi se il calcolo tipizzato semplice è non terminante come quello non tipizzato. In effetti, come mostreremo in questa sezione, in $\lambda_{\rightarrow}$ è possibile esprimere solamente funzioni totali; equivalentemente, ogni termine di $\lambda_{\rightarrow}$ possiede una forma normale. Il procedimento di riduzione a forma normale di un termine è detto *normalizzazione*; con la normalizzazione sono connesse le seguenti due proprietà.

Definizione 4.1 (normalizzazione debole)**.** Un termine si dice *debolmente normalizzante* se esiste una sequenza di riduzioni che lo trasforma in forma normale. Un calcolo gode della proprietà di *normalizzazione debole* se tutti i termini in esso sono debolmente normalizzanti.

Definizione 4.2 (normalizzazione forte)**.** Un termine si dice *fortemente normalizzante* se ogni sequenza di riduzioni lo trasforma in forma normale. Un calcolo gode della proprietà di *normalizzazione forte* se tutti i termini in esso sono fortemente normalizzanti.

Chiaramente, la normalizzazione forte implica quella debole; il viceversa non è vero, come mostra il seguente termine del λ-calcolo puro

$$(\lambda x.y\ (\delta\ \delta))$$

che riduce a se stesso se si opera sul redex interno, mentre giunge immediatamente alla forma normale y se si opera sul redex esterno.

Entrambe le proprietà valgono per il λ-calcolo tipizzato semplice: le dimostreremo separatamente nei prossimi paragrafi.

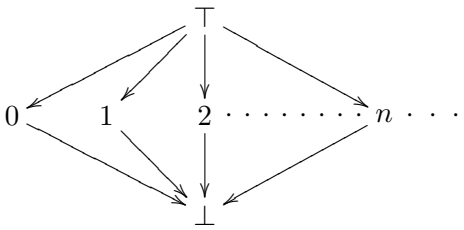
4.5.1 Teorema di normalizzazione debole

Teorema 4.3 (normalizzazione debole)**.** *Per ogni termine del λ-calcolo tipizzato semplice esiste una sequenza di riduzioni che lo trasforma in forma normale.*

Dimostrazione. Nella dimostrazione di questo teorema useremo la seguente nozione di ordinamento *ben fondato*.

Definizione 4.4. Un ordinamento parziale stretto si dice *ben fondato* se non ammette catene discendenti infinite.

Si noti che, affinché un ordinamento sia ben fondato, non occorre che l'insieme dei minoranti di ogni termine sia finito. Un esempio è dato dall'ordinamento su $\mathbb{N} \cup \{\top, \bot\}$ definito come nella seguente figura:



Le frecce mettono in relazione $\top$ con ogni numero naturale, pertanto esistono infiniti minoranti di $\top$; d'altra parte, le catene massimali (ossia tutte le catene che congiungono $\top$ con $\bot$) hanno lunghezza 2.

Definire un ordinamento ben fondato sui termini ci consentirà di scegliere sempre un redex che faccia "decrescere" il termine fino a raggiungere la forma normale. L'ordinamento cui siamo interessati è quello lessicografico tra coppie di naturali, definito come segue:

$$\langle n, m\rangle \lessdot \langle n', m'\rangle \text{ sse } n < n' \text{ o } n = n' \text{ e } m < m'$$

Proposizione 4.5. *L'ordinamento lessicografico tra coppie di naturali è ben fondato.*

Possiamo convincerci (informalmente) di questo fatto visualizzando le coppie di naturali in una tabella con infinite righe e infinite colonne:

$$\begin{bmatrix} \vdots & \cdots & \cdots & \langle x, y\rangle \\ \langle 2,0\rangle & & & \vdots \\ \langle 1,0\rangle & \langle 1,1\rangle & & \vdots \\ \langle 0,0\rangle & \langle 0,1\rangle & \langle 0,2\rangle & \cdots \end{bmatrix}$$

Evidentemente l'insieme delle coppie minori di una coppia fissata è costituito dalle coppie alla sua sinistra e dalle coppie appartenenti alle righe sottostanti. Allora una qualsiasi catena avente origine in $\langle x, y\rangle$ è una successione di coppie costituita da:

- l'elemento $\langle x, y\rangle$
- al più y coppie della forma $\langle x, i\rangle$ con i che varia tra $y-1$ e 0
- al più k_1 coppie della forma $\langle x-1, i\rangle$ con i compreso tra $k_1 - 1$ e 0 (dove k_1 è un numero finito)
- al più k_2 coppie della forma $\langle x-2, i\rangle$ con i compreso tra $k_1 - 1$ e 0 (dove k_2 è un numero finito)
- ...

- al più k_x coppie della forma $\langle 0, i\rangle$ con i compreso tra $k_x - 1$ e 0 (dove k_x è un numero finito)

La lunghezza della catena sarà dunque al più $\sum_{i=1}^{x} k_i + y + 1$, che è un numero finito. Pertanto l'ordinamento lessicografico è ben fondato.

Tornando alla dimostrazione della normalizzazione debole, essa deve risolvere due problemi: in primo luogo la riduzione di $(\lambda x : T.M\ N)$ può duplicare i redex presenti nel parametro N; in secondo luogo è possibile che vengano creati nuovi redex (o tra il termine ridotto $M[N/x]$ e il suo contesto, o all'interno del termine ridotto, in corrispondenza delle sostituzioni di N dentro M). Chiaramente, non si può sperare che le riduzioni diminuiscano la taglia del termine iniziale (essa potrebbe anzi crescere, ad esempio in termini che calcolano esponenziali). Dobbiamo definire un altro tipo di grandezza.

Definizione 4.6 (tipo di un redex)**.** Il tipo di un redex $(\lambda x^T.M\ N)$ è il tipo della sua parte funzionale, ossia $T \to U$, dove U è il tipo di M.

Definizione 4.7 (norma)**.** La norma di un tipo T è la grandezza $|T|$ definita induttivamente come segue:

1. se T è un tipo atomico, $|T| = 1$
2. se $T = U \to V$ o $T = U \times V$, $|T| = |U| + |V| + 1$.

La norma di un redex è pari alla norma del suo tipo.

Definizione 4.8 (peso di un termine)**.** Il peso di un termine M è la grandezza

$$w(M) = \langle m, n\rangle$$

dove m è la norma massima dei redex contenuti in M e n è pari al numero di redex aventi norma massima.

Notiamo ora che in seguito a una riduzione, se il redex originale era $(\lambda x^T.M\ N)$ di tipo $T \to U$:

1. i redex creati tra il termine ridotto e il contesto sono di tipo U
2. i redex creati sostituendo N in M sono di tipo T
3. i redex residui di N mantengono il loro tipo.

Poiché i nuovi redex sono sottotipi[2] del redex originale, la loro norma è strettamente minore di quella del redex originale. Sui redex residui, invece, non possiamo dire nulla: in particolare, è possibile che siano aumentati di numero e che la loro norma sia massima.

[2] In senso sintattico: nulla a che vedere con la nozione di sottotipo dei linguaggi orientati agli oggetti

Tuttavia, è possibile ridurre sempre il redex più interno, tra quelli di norma massima: in questo modo, i residui duplicati avranno necessariamente norma minore. Dimostriamo che utilizzando una strategia leftmost-innermost sui redex di norma massima, se $M \longrightarrow_\beta N$, allora $w(N) \lessdot w(M)$.

Sia $w(M) = \langle m, n \rangle$. Poiché abbiamo scelto il redex più interno tra quelli di norma massima, i redex di norma massima diminuiscono di uno. Per come abbiamo definito il peso, sono possibili due scenari:

1. se $n > 1$, allora $w(N) = \langle m, n-1 \rangle$
2. se $n = 1$, allora $w(N) = \langle m', p \rangle$, con $m' < m$ e p finito.

In entrambi i casi $w(N) \lessdot w(M)$, pertanto le catene di riduzioni generano catene discendenti di pesi secondo l'ordinamento lessicografico, che essendo ben fondato ammette solo catene di lunghezza finita. Di conseguenza anche le catene di riduzioni, secondo la predetta strategia, dovranno condurre a forma normale con un numero finito di passi. □

Osservazione 4. Si noti che la dimostrazione descritta non dice nulla sulla complessità della riduzione a forma normale. È possibile dare un limite alla complessità della riduzione in funzione del tipo del termine, ma la dimostrazione diventa più complessa.

4.5.2 Teorema di normalizzazione forte

Teorema 4.9 (normalizzazione forte)**.** *Dato un termine del λ-calcolo tipizzato semplice, qualsiasi sequenza di riduzioni lo trasforma in forma normale.*

La dimostrazione di questo teorema è più intricata di quella della normalizzazione debole e passa attraverso la definizione della seguente proprietà.

Definizione 4.10 (riducibilità)**.** Sia t un termine di tipo T; allora:

- se $T = A$, dove A è un tipo atomico, t è riducibile di tipo A ($t \in \mathbf{Red}_A$) se e solo se è fortemente normalizzante;
- se $T = U \times V$, t è riducibile di tipo $U \times V$ ($t \in \mathbf{Red}_{U \times V}$) se e solo se $(\mathrm{fst}\, t) \in \mathbf{Red}_U$ e $(\mathrm{snd}\, t) \in \mathbf{Red}_V$;
- se $T = U \to V$, t è riducibile di tipo $U \to V$ ($t \in \mathbf{Red}_{U \to V}$) se e solo se per ogni $u \in Red_U$, $(t\ u) \in Red_V$.

La riducibilità riassume tutte le proprietà necessarie affinché un termine sia fortemente normalizzante. Il nostro obiettivo è dimostrare che ogni termine riducibile è fortemente normalizzante e in seguito che ogni termine è riducibile.

Condizioni di riducibilità

Per la dimostrazione, è necessario provare prima le seguenti *Condizioni di riducibilità (CR)*.

Lemma 4.11 (Condizioni di riducibilità)**.** *Per i termini del λ-calcolo tipizzato semplice, valgono le seguenti condizioni di riducibilità:*

CR1 *Se t è riducibile di tipo T, allora è fortemente normalizzante.*

CR2 *Se t è riducibile di tipo T e t riduce a t', allora t' è riducibile di tipo T.*

CR3 *Se t è un termine neutro (ossia non è della forma $\lambda x^T.M$ o $\langle t_1, t_2 \rangle$) e t' è riducibile di tipo T per ogni ridotto t' di t, allora anche t è riducibile di tipo T.*

CR4 *Ogni termine neutro in forma normale è riducibile del tipo opportuno.*

Dimostrazione. La dimostrazione delle quattro proprietà è per induzione strutturale sul tipo del termine. Ometteremo la dimostrazione della CR4, che è un corollario della CR3.

Caso base Come caso base, supponiamo che t sia di tipo atomico A.

CR1 La proprietà è banalmente vera (dalla definizione di riducibilità).

CR2 La proprietà ci chiede di dimostrare che se t è riducibile, ogni suo ridotto lo è. Dalla definizione di riducibilità, segue che t è fortemente normalizzante, quindi ogni suo ridotto lo è. Ma allora, sempre per la definizione di riducibilità, i ridotti di t sono anch'essi riducibili.

CR3 Per questa proprietà, notiamo che se tutti i ridotti di t sono riducibili, sono anche fortemente normalizzanti. Ma allora anche t stesso lo è.

Tipi coppia Come primo caso induttivo, dimostriamo che se le proprietà CR valgono per i tipi U e V, allora valgono anche per $U \times V$.

CR1 Per ipotesi, $t \in Red_{U \times V}$. Dalla definizione di riducibilità, $(fst\ t) \in Red_U$. Ma allora, per ipotesi induttiva, CR1 vale per il tipo U, quindi $(fst\ t)$ è fortemente normalizzante. Dato che t è un sottotermine di $(fst\ t)$, anche t deve essere fortemente normalizzante, come richiesto.

CR2 Per ipotesi, $t \in Red_{U \times V}$ e $t \longrightarrow t'$. Dalla definizione di riducibilità, $(fst\ t) \in Red_U$ e $(snd\ t) \in Red_V$. Per ipotesi induttiva, vale anche che $(fst\ t') \in Red_U$ e $(snd\ t') \in Red_V$. Allora dalla definizione di riducibilità segue che $t' \in Red_{U \times V}$.

CR3 Per ipotesi, t è neutro e quindi non è della forma $\langle u, v\rangle$. Sempre per ipotesi, $t'_1, \ldots, t'_n \in Red_{U\times V}$, dove $t'_1, \ldots, t'_n$ sono tutti i ridotti di t. Dato che t è neutro, gli unici ridotti di $(fst\ t)$ e $(snd\ t)$ sono:

$$(fst\ t'_1) \ldots (fst\ t'_n)$$
$$(snd\ t'_1) \ldots (snd\ t'_n)$$

Per la definizione di riducibilità, essi sono tutti riducibili del tipo opportuno. Ma allora, per ipotesi induttiva, anche $(fst\ t) \in Red_U$ e $(snd\ t) \in Red_V$. Dalla definizione di riducibilità, segue quindi che anche $t \in Red_{U\times V}$.

Tipi freccia Dimostriamo che se le proprietà CR valgono per U e V, allora valgono anche per $U \to V$.

CR1 Per ipotesi, $t \in Red_{U\to V}$. Sia $x : U$ una variabile: per ipotesi induttiva, dalla proprietà CR4 segue che $x \in Red_U$. Allora, per definizione di riducibilità, $(t\ x) \in Red_V$ e quindi, per ipotesi induttiva, dalla proprietà CR1 segue che $(t\ x)$ è fortemente normalizzante. Pertanto anche t è fortemente normalizzante, in quanto sottotermine di $(t\ x)$.

non si poteva prendere direttamente $u \in Red_U$ come nel CR2?

CR2 Per ipotesi, $t \in Red_{U\to V}$ e $t \longrightarrow t'$. Sia $u \in Red_U$: allora, per definizione di riducibilità, vale $(t\ u) \in Red_V$. Dato che $(t\ u) \longrightarrow (t'\ u)$, per ipotesi induttiva (CR2) anche $(t'\ u) \in Red_V$. Pertanto, dalla definizione di riducibilità, segue che anche $t' \in Red_{U\to V}$.

CR3 Per ipotesi, sia $t : U \to V$ un termine neutro (in particolare, diverso da una λ-astrazione) tale che, per tutti i ridotti $t'_1, \ldots, t'_n$ di t, vale $t'_1, \ldots, t'_n \in Red_{U\to V}$. Dalla definizione di riducibilità, segue che

$$(t'_1\ u), \ldots, (t'_n\ u) \in Red_V$$

Vorremmo ora applicare l'ipotesi induttiva per derivare che anche $(t\ u) \in Red_V$. Purtroppo però i termini di cui sopra non sono tutti i ridotti di $(t\ u)$: dovremmo infatti includere anche tutti i termini della forma $(t\ u')$ dove u' è un ridotto di u. Su questi termini non sappiamo nulla, perciò si rende necessaria la seguente *sottoinduzione*.

Sia $\nu(u)$ la funzione che restituisce la lunghezza massima delle catene di riduzione aventi origine in u. Dimostriamo che qualunque sia $\nu(u)$, $(t\ u) \in Red_V$. Per induzione su $\nu(u)$:

- Se $\nu(u) = 0$, u è in forma normale, pertanto i ridotti di $(t\ u)$ sono i soli termini $(t'_1\ u), \ldots, (t'_n\ u)$; quindi per ipotesi induttiva (CR3) anche $(t\ u) \in Red_V$.

- Se $\nu(u) > 0$, per ipotesi induttiva (CR2) ogni ridotto u' di u è riducibile. Inoltre per ipotesi induttiva (essendo $\nu(u') < \nu(u)$) vale anche $(t\ u') \in Red_V$. Pertanto, tutti i ridotti di $(t\ u)$ sono riducibili e quindi per ipotesi induttiva (CR3) anche $(t\ u) \in Red_V$

Concludiamo quindi che, essendo $(t\ u)$ riducibile per qualunque $u \in Red_U$, anche $t \in Red_{U \to V}$. □

Osservazione 5. La dimostrazione delle proprietà CR sfrutta il fatto che i termini di un tipo T composto di U e V possono essere sintatticamente più semplici di termini dei tipi U e V. L'induzione sul tipo consente quindi di dedurre proprietà forti per tipi semplici e termini complessi, e di trasmetterle a termini più semplici, ma di tipo più complesso.

Dimostrazione del teorema

Nella dimostrazione del teorema, utilizzaremo i seguenti due lemmi.

Lemma 4.12. *Se $u \in Red_U$ e $v \in Red_V$, allora anche $\langle u, v\rangle \in Red_{U \times V}$.*

Dimostrazione. La dimostrazione non è ovvia, come saremmo tentati di dire, in quanto da un punto di vista sintattico $(fst\ \langle u, v\rangle) \not\equiv u$. Dimostriamo quindi la tesi per induzione su $\nu(u) + \nu(v)$:

- Se $\nu(u) + \nu(v) = 0$, allora l'unico ridotto di $(fst\ \langle u, v\rangle)$ è u. Dato che per ipotesi $u \in Red_U$, per la CR3 anche $(fst\ \langle u, v\rangle) \in Red_U$.
- Se $\nu(u) + \nu(v) > 0$, siano $u'_1, \ldots, u'_m$ e $v'_1, \ldots, v'_n$ i ridotti in un passo rispettivamente di u e v. Allora i ridotti di $(fst\ \langle u, v\rangle)$ sono nella forma

$$\begin{array}{ll} & u \\ \text{oppure} & (fst\ \langle u'_i, v\rangle) \quad (\text{per } i = 1, \ldots, m) \\ \text{oppure} & (fst\ \langle u, v'_j\rangle) \quad (\text{per } j = 1, \ldots, n) \end{array}$$

  D'altra parte $u \in Red_U$ per ipotesi, mentre per ipotesi induttiva (essendo $\nu(u'_i) + \nu(v) < \nu(u) + \nu(v)$ e $\nu(u) + \nu(v'_j) < \nu(u) + \nu(v)$) vale

$$(fst\ \langle u'_i, v\rangle), (snd\ \langle u, v'_j\rangle) \in Red_U$$

  Dalla CR3 segue quindi che anche $(fst\ \langle u, v\rangle) \in Red_U$.

In modo simile, si dimostra che $(snd\ \langle u, v\rangle) \in Red_V$: il lemma segue quindi dalla definizione di reducibilità. □

Lemma 4.13. *Se per ogni $u \in Red_U$ vale $t[u/x] \in Red_V$ allora $\lambda x^U.t \in Red_{U \to V}$.*

Dimostrazione. Notiamo che le ipotesi del lemma implicano che anche $t \in Red_V$ (basta scegliere $u = x$, dove x è una variabile di tipo U e come tale è riducibile); in particolare, t è anche fortemente normalizzante (CR1), pertanto il valore $\nu(t)$ è ben definito.

Sia $u \in Red_U$. Per induzione su $\nu(t) + \nu(u)$:

- Se $\nu(t) + \nu(u) = 0$, l'unico ridotto di $(\lambda x^U.t\ u)$ è $t[u/x]$, che è riducibile per ipotesi. Pertanto, dalla CR3 segue $(\lambda x^U.t\ u) \in Red_V$.
- Se $\nu(t) + \nu(u) > 0$, siano $t'_1, \ldots, t'_m$ e $u'_1, \ldots, u'_n$ i ridotti in un passo rispettivamente di t e u. Allora i ridotti di $(\lambda x^U.t\ u)$ sono della forma

$$\begin{array}{ll} & t[u/x] \\ \text{oppure} & (\lambda x^U.t'_i\ u) \quad (\text{per } i = 1, \ldots, m) \\ \text{oppure} & (\lambda x^U.t\ u'_j) \quad (\text{per } i = j, \ldots, n) \end{array}$$

  D'altra parte $t[u/x] \in Red_V$ per ipotesi, mentre per ipotesi induttiva (essendo $\nu(t'_i) + \nu(u) < \nu(t) + \nu(u)$ e $\nu(t) + \nu(u'_j) < \nu(t) + \nu(u)$) vale

$$(\lambda x^U.t'_i\ u), (\lambda x^U.t\ u'_j) \in Red_V$$

  Dalla CR3 segue quindi che anche $(\lambda x^U.t\ u) \in Red_V$.

Il lemma segue quindi dalla definizione di riducibilità. □

Ora siamo in grado di dimostrare che ogni termine del λ-calcolo tipizzato semplice è riducibile - e come tale fortemente normalizzante. Per farlo dimostreremo in realtà un enunciato più forte (un "trucco" che ci consentirà di disporre di ipotesi induttive più forti).

Proposizione 4.14. *Sia $t : T$ un termine e siano le sue variabili libere comprese in $x_1 : T_1$,...,$x_n : T_n$. Allora per ogni scelta di $u_1 \in Red_{T_1}$, ..., $u_n \in Red_{T_n}$, vale $t[u_1,\ldots,u_n/x_1,\ldots,x_n]$.*

Dimostrazione. Per induzione strutturale sul termine t. Scriveremo $t[\bar{u}/\bar{x}]$ in luogo di $t[u_1,\ldots,u_n/x_1,\ldots,x_n]$.

- Se $t = x_i$, si ha $x_i[\bar{u}/\bar{x}] = u_i \in Red_T$ per ipotesi.
- Se $t = (fst\ t')$, per ipotesi induttiva si ha $t'[\bar{u}/\bar{x}] \in Red_{T \times U}$ e quindi, per la definizione di riducibilità, anche $(fst\ t'[\bar{u}/\bar{x}]) \in Red_T$. Ma dalla definizione di sostituzione si ha $(fst\ t')[\bar{u}/\bar{x}] = (fst\ t'[\bar{u}/\bar{x}])$, pertanto $(fst\ t')[\bar{u}/\bar{x}] = t[\bar{u}/\bar{x}] \in Red_T$.
- Se $t = (snd\ t')$, il caso è analogo al precedente.

- Se $t = \langle t', t'' \rangle$, si applica quindi l'ipotesi induttiva in modo simile ai casi precedenti e si ricava $\langle t'[\bar{u}/\bar{x}], t''[\bar{u}/\bar{x}] \rangle \in Red_{T' \times T''}$ dal lemma 4.12 (dove T' e T'' sono i tipi di t' e t''). Ma dalla definizione di sostituzione si ha $\langle t', t'' \rangle[\bar{u}/\bar{x}] = \langle t'[\bar{u}/\bar{x}], t''[\bar{u}/\bar{x}] \rangle$, pertanto $\langle t', t'' \rangle[\bar{u}/\bar{x}] = t[\bar{u}/\bar{x}] \in Red_{T' \times T''}$.

- Se $t = (t'\ t'')$, si applica quindi l'ipotesi induttiva in modo simile ai casi precedenti e si ricava $(t'[\bar{u}/\bar{x}]\ t''[\bar{u}/\bar{x}]) \in Red_T$ dalla definizione di riducibilità. Ma dalla definizione di sostituzione si ha $(t'\ t'')[\bar{u}/\bar{x}] = (t'[\bar{u}/\bar{x}]\ t''[\bar{u}/\bar{x}])$, pertanto $(t'\ t'')[\bar{u}/\bar{x}] = t[\bar{u}/\bar{x}] \in Red_T$.

- Se $t = \lambda y^U.t'$ (e $T = U \to V$) per ipotesi induttiva $t'[\bar{u}/\bar{x}] \in Red_V$; si noti che y o non compare in t' oppure è compresa in $x_1, \ldots, x_n$ pertanto la precedente scrittura è equivalente a $t'[\bar{u}/\bar{x}, u/y]$, ossia $t'[\bar{u}/\bar{x}][u/y]$ dove u è un qualunque termine riducibile di tipo U. Allora per il lemma 4.13 si ha $\lambda y^U.(t'[\bar{u}/\bar{x}]) \in Red_{U \to V}$. Ma dalla definizione di sostituzione si ha $(\lambda y^U.t')[\bar{u}/\bar{x}] = \lambda y^U.(t'[\bar{u}/\bar{x}])$, pertanto $(\lambda y^U.t')[\bar{u}/\bar{x}] = t[\bar{u}/\bar{x}] \in Red_{U \to V}$.

□

Corollario 4.15. *Tutti i termini del λ-calcolo tipizzato semplice sono riducibili.*

Dimostrazione. Dal teorema precedente, ponendo $\bar{u} = \bar{x}$. □

Il teorema di normalizzazione forte segue immediatamente dal precedente corollario, per mezzo della proprietà CR1, secondo cui ogni termine riducibile è fortemente normalizzante. □

Capitolo 5

Il Calcolo delle Costruzioni

Al termine della nostra descrizione dei vari modelli di λ-calcolo tipizzato, è opportuno guardarci indietro per comprenderne meglio le differenze e i limiti.

Il λ-calcolo tipizzato semplice si differenziava dal λ-calcolo puro per l'introduzione delle annotazioni di tipo nelle λ-astrazioni; il sistema di tipi ne limitava molto il potere espressivo, poiché la λ-astrazione permetteva come prima di astrarre termini in termini, ma contemporaneamente fissava il tipo del risultato.

Dopo la parentesi del Sistema T, che estendeva il calcolo tipizzato semplice con un insieme di tipi induttivi e ricorsori predefiniti, nel precedente capitolo abbiamo presentato il Sistema F, che aumentava il potere espressivo dei calcoli precedenti introducendo un nuovo modello di astrazione, da tipi in termini.

Che cosa non possiamo fare nel Sistema F? A ben vedere esistono almeno due categorie di termini che non possono essere espresse direttamente nel calcolo:

- operatori che costruiscono tipi in funzione di altri tipi (questi operatori, quali $(List\ T)$, sono stati già utilizzati solamente come abbreviazioni sintattiche);
- termini il cui tipo dipende da altri termini (ad esempio, il tipo delle $List$ di lunghezza 4).

Il *Calcolo delle Costruzioni* nasce da questa intuizione, e quindi dalla generalizzazione dell'astrazione per consentire di esprimere:

- funzioni da termini in termini (come nel λ-calcolo tipizzato semplice)
- funzioni da tipi in termini (i *termini polimorfi* del Sistema F)
- funzioni da tipi in tipi (*operatori di tipo*)
- funzioni da termini in tipi (*tipi dipendenti*).

5.1 Sintassi

$\langle T \rangle$ ::= $\langle AT \rangle$ | $\langle T \rangle \rightarrow \langle T \rangle$

$\langle AT \rangle$ ::= A | B | C | ...

Figura 5.1: Espressioni di tipo nel λ-calcolo tipizzato semplice

La sintassi dei termini è modificata soltanto nel caso dell'astrazione: è infatti necessario specificare un tipo per ogni variabile legata. La grammatica è presentata in fig. 5.2.

$\langle term \rangle$ ::= $\langle var \rangle$ | $\lambda \langle var \rangle : \langle T \rangle . \langle term \rangle$ | $(\langle term \rangle \ \langle term \rangle)$

$\langle var \rangle$ ::= x | y | z | x_1 | ...

Figura 5.2: Termini del λ-calcolo tipizzato semplice

5.2 Sistemi di tipi

Dopo aver definito la sintassi del calcolo tipizzato, ci chiediamo come sia possibile assegnare un tipo ai termini del linguaggio. Allo stato attuale, infatti, i tipi sono soltanto un'aggiunta cosmetica che non pone alcun vincolo. Ad esempio il termine

$$(\lambda x : A.x \ \lambda x : A.x)$$

dove A è un tipo atomico, non dovrebbe essere considerato corretto perché il termine passato come parametro non è di tipo A, bensì (intuitivamente) di tipo $A \rightarrow A$. Termini come questo sono detti *mal tipizzati* e non ci interessa esprimere computazioni su di essi (un compilatore li rigetterebbe segnalando un errore di semantica statica).

Ci interessa tuttavia trovare un modo per definire la nozione di buona tipizzazione e per distinguere i termini tipizzati bene da quelli tipizzati male. I *sistemi di tipi* sono collezioni di regole che ci consentiranno di raggiungere questo scopo.

ce lo mettiamo?

Un buon sistema di tipi (utile in pratica per un linguaggio di programmazione) deve godere delle seguenti proprietà [**?**]:

- deve essere decidibilmente verificabile: deve esistere un algoritmo di type-checking che verifichi se un programma è ben tipato.
- deve essere trasparente: il programmatore deve essere in grado di prevedere se un programma è ben tipato o meno e perché.

- deve essere *enforceable*: le dichiarazioni di tipo dovrebbero essere il più possibile verificate staticamente, altrimenti dinamicamente. La consistenza tra le dichiarazioni di tipo e il codice associato ad esse dev'essere verificabile da un programma.

Introduciamo ora il formalismo dei sistemi di tipi.

5.2.1 Giudizi

L'entità atomica trattata dai sistemi di tipi è un particolare tipo di espressione formale denominato *giudizio*. Un giudizio ha la forma:

$$\Gamma \vdash \mathfrak{J}$$

che si legge "Γ verifica $\mathfrak{J}$". $\mathfrak{J}$ è un'asserzione; Γ è detto contesto e definisce i tipi delle variabili che compaiono libere in $\mathfrak{J}$, nella forma $x_1 : A_1, \ldots, x_n : A_n$ (qui il simbolo ':' rappresenta la relazione "appartiene al tipo"). È necessario che tutte le variabili libere nell'asserzione siano dichiarate in Γ. Per il calcolo tipizzato semplice, siamo interessati a una sola forma di asserzione, vale a dire

$$\Gamma \vdash M : T$$

dove M è un termine e T è un tipo qualunque, che si legge "M ha tipo T in Γ"

Talvolta indicheremo un contesto in maniera esplicita elencandone le componenti (ad esempio: $x : S, y : T$); indicheremo inoltre la concatenazione di ambienti con la virgola (ad esempio: Γ, Γ').

5.2.2 Regole di tipo

Un giudizio può essere *valido* o *non valido*. La validità di un giudizio è definita sulla base di certe regole di inferenza che costituiscono il sistema di tipi formale.

Una regola (fig. 5.3) è composta da un certo numero di giudizi (*premesse*) posti sopra una linea orizzontale e da un unico giudizio (*conclusione*) posto sotto la linea: se sono valide le premesse, è valida la conclusione.

$$\frac{\Gamma_1 \vdash \mathfrak{J}_1 \ldots \Gamma_n \vdash \mathfrak{J}_n}{\Gamma \vdash \mathfrak{J}} \qquad \text{(REGOLA GENERICA)}$$

Figura 5.3: Esempio generico di regola di tipo

Se una regola non prevede premesse, si intende che la conclusione è intrinsecamente valida, cioè è un assioma.

Le regole possono essere composte in un albero di derivazione: la conseguenza di una qualsiasi regola può essere utilizzata come premessa di un'altra regola. Per verificare la validità di un giudizio occorre dare un albero di derivazione corretto, le cui foglie siano assiomi e la cui radice sia il giudizio da verificare.

Le regole di tipo (fig. 5.4) per il calcolo tipizzato semplice sono tre, una per ciascun tipo di termine. La regola per le variabili è un assioma che assegna a x il tipo T, posto che $x : T$ compaia nel contesto. Un'applicazione ha tipo T_2 se la parte funzionale ha tipo $T_1 \to T_2$ e l'argomento ha tipo T_1. Un'astrazione $\lambda x : T_1.M$ ha tipo $T_1 \to T_2$ se, supponendo che x abbia tipo T_1, M ha tipo T_2.

$$\Gamma \vdash x : T \qquad \text{se } x : T \in \Gamma \qquad \text{(T-AX)}$$

$$\frac{\Gamma \vdash M : T_1 \to T_2 \qquad \Gamma \vdash N : T_1}{\Gamma \vdash (M\ N) : T_2} \qquad \text{(T-APP)}$$

$$\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x : T_1.M : T_1 \to T_2} \qquad \text{(T-ABS)}$$

Figura 5.4: Regole di tipo per il λ-calcolo tipizzato semplice

5.2.3 Applicazioni

Dato un sistema di tipi, un termine M è ben tipizzato in un ambiente Γ se esiste un tipo A per cui il giudizio $\Gamma \vdash M : A$ è valido. Il problema di controllare l'esistenza di un tipo ben formato per i termini di un programma è detto *typechecking*. Il problema di identificare, se esiste, il tipo di un termine è detto invece *type inference* o inferenza dei tipi.

Si noti che non per tutti i sistemi di tipi esistono algoritmi di typechecking o type inference (se esiste un algoritmo di type inference, ovviamente esiste anche l'algoritmo di typechecking, ma non è vero il contrario): per alcuni sistemi, questi problemi sono indecidibili (non è possibile determinare per via algoritmica se il problema ha soluzione o meno) o semidecidibili (esiste un semialgoritmo che termina soltanto se il problema ha soluzione). Ovviamente in un sistema di tipi per un linguaggio di uso pratico il typechecking, e all'occorrenza la type inference, devono essere decidibili.

Una proprietà importante che i sistemi di tipi devono rispettare è connessa alla semantica operazionale dei programmi tipizzati. Affinché il sistema di tipi sia corretto è necessario garantire che se $\Gamma \vdash M : A$ è un giudizio valido e il termine M riduce a un termine M', allora $\Gamma \vdash M' : A$. Questa proprietà è detta proprietà di "riduzione del soggetto".

5.3 Programmi in $\lambda_{\rightarrow}$

Per mostrare il potere espressivo del λ-calcolo tipizzato semplice, definiamo la codifica di Church per gli interi, in modo del tutto analogo al λ-calcolo puro. In primo luogo lo zero:

$$0 \triangleq \lambda x : T_1.\lambda y : T_2.y : T_1 \rightarrow T_2 \rightarrow T_2$$

La differenza sostanziale rispetto al caso non tipizzato sta nella necessità di assegnare dei tipi ai parametri della funzione che rappresenta lo zero. Questi tipi sono stati indicati genericamente con T_1 e T_2, perché idealmente vorremmo che lo zero fosse un valore *polimorfo* in cui T_1 e T_2 fossero istanziabili con qualunque tipo. Vediamo cosa accade cercando di tipizzare la definizione dell'1.

$$1 \triangleq \lambda x : T_1 \rightarrow T_2.\lambda y : T_1.(x\ y) : (T_1 \rightarrow T_2) \rightarrow T_1 \rightarrow T_2$$

Per l'1 otteniamo un termine di tipo diverso. Anche in questo caso T_1 e T_2 stanno ad indicare due tipi qualunque. Vediamo infine il caso del numero 2.

$$2 \triangleq \lambda x : T \rightarrow T.\lambda y : T.(x\ (x\ y)) : (T \rightarrow T) \rightarrow T \rightarrow T$$

Anche in questo caso il tipo del termine è differente; T denota un tipo qualunque. Per i numeri interi maggiori di 2 si può invece verificare che il tipo risulta identico a quello del 2.

Questa situazione anomala si risolve verificando che è possibile specializzare i tipi dei numeri 0 e 1, in modo che *unifichino* con il tipo di 2. In particolare in 0 occorre sostituire T_1 con $T \rightarrow T$ e T_2 con T; in 1 si sostituiscono sia T_1 sia T_2 con T.

Per concludere, il tipo dei numeri interi è definito in modo parametrico rispetto a un qualunque tipo T, come segue:

$$Nat_T \triangleq (T \rightarrow T) \rightarrow T \rightarrow T$$

e anche gli interi di Church e la funzione successore sono definiti in modo parametrico rispetto a T:

$$\begin{aligned}
&0_T \triangleq \lambda x : T \rightarrow T.\lambda y : T.y : Nat_T \\
&1_T \triangleq \lambda x : T \rightarrow T.\lambda y : T.(x\ y) : Nat_T \\
&2_T \triangleq \lambda x : T \rightarrow T.\lambda y : T.(x\ (x\ y)) : Nat_T \\
&\vdots \\
&succ_T \triangleq \lambda n : Nat_T.\lambda x : T \rightarrow T.\lambda y : T.(x\ (n\ x\ y)) : Nat_T \rightarrow Nat_T
\end{aligned}$$

5.3.1 Potere espressivo

La possibilità di definire booleani e interi non è sufficiente a garantire un'elevata espressività al nostro sistema di calcolo: sarebbe opportuno mostrare almeno una codifica per le funzioni primitive ricorsive. Tuttavia, una tale codifica non esiste: per la precisione, il λ-calcolo tipizzato semplice non consente di tipizzare nemmeno funzioni "semplici" come il predecessore. In sostanza, le uniche funzioni esprimibili sono somma, prodotto e composizioni di queste.

È possibile estendere il potere espressivo di questo calcolo introducendo operatori *ad hoc* per la ricorsione, o equivalentemente un operatore punto fisso

$$FIX_T : (T \to T) \to T$$
$$(FIX_T\ M) \longrightarrow_\beta (M\ (FIX_T\ M)) \quad \text{se } M : T$$

verificare... Neanche in questo modo, tuttavia, il calcolo diventa Turing-completo.

5.4 Corrispondenza di Curry-Howard

In questa sezione presenteremo un'interessante analogia tra calcoli tipizzati e sistemi logici deduttivi. La *corrispondenza di Curry-Howard* fa corrispondere alle formule logiche i tipi di un linguaggio tipizzato, alle prove i termini (e in particolare a una prova della formula A un termine di tipo A).

La corrispondenza funziona particolarmente bene per la logica intuizionista (e specialmente per certi frammenti di essa), ma può essere estesa, nello spirito di trasporre i progressi della logica nei linguaggi di programmazione e viceversa.

5.4.1 Il frammento implicativo

Il frammento implicativo della logica intuizionista include le formule logiche espresse per mezzo di proposizioni atomiche (A, B, ...) e del connettivo $\Rightarrow$.

Le dimostrazioni sono date, in deduzione naturale, per mezzo delle regole (fig. 5.5) di introduzione dell'implicazione ($\Rightarrow i$) e di eliminazione dell'implicazione (*modus ponens* o $\Rightarrow e$).

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \Rightarrow B} \quad (\Rightarrow i) \qquad\qquad \frac{A \Rightarrow B \qquad A}{B} \quad (\Rightarrow e)$$

Figura 5.5: Regole di inferenza per il connettivo $\Rightarrow$

Secondo la regola di introduzione, possiamo derivare una *dimostrazione* di $A \Rightarrow B$ (A implica B) se abbiamo un procedimento costruttivo (un *algoritmo*) che trasforma dimostrazioni di A in dimostrazioni di B; la regola di eliminazione ci consente invece di comporre una dimostrazione di $A \Rightarrow B$ con una dimostrazione di A per ottenere una dimostrazione di B.

La corrispondenza di Curry-Howard ci consente di associare alle dimostrazioni ottenute componendo queste regole opportuni termini del λ-calcolo tipizzato semplice. Al fine di rendere la corrispondenza più evidente, supporremo che alle foglie scaricate di un albero di prova e alle regole di introduzione dell'implicazione sia sempre associata un'etichetta (appartenente all'insieme $\{x, y, z, x_1, \ldots\}$) in modo da identificare la specifica regola di introduzione in cui la foglia è scaricata[1] (fig. 5.6).

$$\dfrac{\dfrac{[A]^x}{B \Rightarrow A}\ (\Rightarrow i : y)}{A \Rightarrow (B \Rightarrow A)}\ (\Rightarrow i : x)$$

Figura 5.6: Albero di prova con etichette

La corrispondenza è data dalle seguenti regole induttive che associano a dimostrazioni termini di $\lambda_{\rightarrow}$:

1. Data una dimostrazione costituita dalla singola ipotesi A non scaricata, si fa corrispondere una variabile fresca $x : A$.

2. Data una dimostrazione costituita dalla singola ipotesi A, scaricata ed etichettata y, si fa corrispondere la variabile $y : A$.

3. Data una dimostrazione che termina con $(\Rightarrow i)$, siano $S \Rightarrow T$ la conclusione della dimostrazione, y l'etichetta dell'ultimo passo di derivazione e u il termine associato alla sottodimostrazione immediatamente superiore. Si fa corrispondere il termine $\lambda y : S.u : S \rightarrow T$.

4. Data una dimostrazione che termina con $(\Rightarrow e)$, siano $u : S \rightarrow T$ e $v : S$ i termini associati alle due sottodimostrazioni immediatamente superiori all'ultimo passo di derivazione. Si fa corrispondere il termine $(u\ v) : T$.

5.4.2 Il connettivo $\wedge$

In fig. 5.7 sono riportate le regole per la congiunzione logica in deduzione naturale. È possibile dimostrare la congiunzione di due formule A e B

[1] Questo accorgimento rende la corrispondenza di Curry-Howard un vero e proprio isomorfismo.

disponendo di due dimostrazioni, rispettivamente per A e per B; inversamente, avendo una dimostrazione di $A \wedge B$, è possibile dimostrare A, oppure dimostrare B.

$$\frac{A \qquad B}{A \wedge B} \quad (\wedge i)$$

$$\frac{A \wedge B}{A} \quad (\wedge e.1) \qquad\qquad \frac{A \wedge B}{B} \quad (\wedge e.2)$$

Figura 5.7: Regole di inferenza per il connettivo $\wedge$

Il costruttore di tipo che corrisponde (nel senso della corrispondenza di Curry-Howard) al connettivo $\wedge$ in $\lambda_{\rightarrow}$ è il costruttore dei tipi coppia, detto anche *prodotto cartesiano*, che indicheremo con $\times$. A differenza che nel calcolo non tipizzato, in $\lambda_{\rightarrow}$ non è possibile esprimere coppie di termini in modo generale, pertanto introducendo nuovi costrutti sintattici per supportare le coppie aumenta il potere espressivo del calcolo.

Vogliamo quindi aumentare la sintassi del calcolo, aggiungendo alla sintassi dei tipi il caso dei tipi coppia e alla sintassi dei termini il costrutto coppia e le due proiezioni fst e snd:

$\langle T \rangle$ ::= ... | $\langle T \rangle \times \langle T \rangle$

$\langle term \rangle$ ::= ... | $\langle \langle term \rangle, \langle term \rangle \rangle$ | $(fst\ \langle term \rangle)$ | $(snd\ \langle term \rangle)$

differenziare le parentesi angolari nel metalinguaggio

$$\frac{\Gamma \vdash M : S \qquad \Gamma \vdash N : T}{\Gamma \vdash \langle M, N \rangle : S \times T} \quad \text{(T-PAIR)}$$

$$\frac{\Gamma \vdash P : S \times T}{\Gamma \vdash (fst\ P) : S} \quad \text{(T-PROJ.1)}$$

$$\frac{\Gamma \vdash P : S \times T}{\Gamma \vdash (snd\ P) : T} \quad \text{(T-PROJ.2)}$$

Figura 5.8: Regole di tipo per i tipi coppia

In fig. 5.8 sono date le regole di tipizzazione per i tipi coppia. Si noti che considereremo *redex* anche l'applicazione delle proiezioni a un costrutto coppia, pertanto vanno aggiunte le seguenti nuove forme di β-riduzione:

$$\begin{aligned} (fst\ \langle M, N \rangle) &\longrightarrow_{\beta} M \\ (snd\ \langle M, N \rangle) &\longrightarrow_{\beta} N \end{aligned}$$

La corrispondenza di Curry-Howard per il λ-calcolo tipizzato semplice esteso con coppie è espressa dalle seguenti regole che, aggiunte a quelle per il solo frammento implicativo, associano a dimostrazioni in deduzione naturale termini del calcolo.

5. Data una dimostrazione che termina con ($\wedge i$), siano $a : S$ e $b : T$ i termini associati alle due sottodimostrazioni immediatamente superiori all'ultimo passo di derivazione. Si fa corrispondere il termine $\langle a, b\rangle : S \times T$.

6. Data una dimostrazione che termina con ($\wedge e.1$), sia $p : S \times T$ il termine associato alla sottodimostrazione immediatamente superiore. Si fa corrispondere il termine $(fst\ p) : S$.

7. Data una dimostrazione che termina con ($\wedge e.2$), sia $p : S \times T$ il termine associato alla sottodimostrazione immediatamente superiore. Si fa corrispondere il termine $(snd\ p) : T$.

5.4.3 Il connettivo $\vee$

L'ultimo connettivo tipico della logica intuizionista che ci interessa è la disgiunzione. In fig. 5.9 sono riportate le regole della deduzione naturale relative ad essa. Si può ottenere una dimostrazione di $A \vee B$ a partire da una dimostrazione di A, oppure a partire da una dimostrazione di B; la regola di eliminazione è invece più complessa: combinando una dimostrazione di $A \vee B$ e due dimostrazioni di $A \Rightarrow C$ e $B \Rightarrow C$, otteniamo una dimostrazione di C (l'idea è che si può concludere C ragionando per casi: se è vero A, posso concludere C da $A \Rightarrow C$; se invece è vero B, posso concludere C da $B \Rightarrow C$).

$$\frac{A}{A \vee B}\ (\vee i.1) \qquad \frac{B}{A \vee B}\ (\vee i.2)$$

$$\frac{A \vee B \qquad A \Rightarrow C \qquad B \Rightarrow C}{C}\ (\vee e)$$

Figura 5.9: Regole di inferenza per il connettivo $\vee$

Alle disgiunzioni logiche, corrispondono i *tipi somma* (noti anche come *unioni disgiunte* o *varianti*). Oggigiorno i tipi somma sono in disuso, ma in passato ebbero una certa diffusione grazie alla possibilità di memorizzare tipi di dati diversi nella stessa cella di memoria, minimizzando gli sprechi.

Come già avevamo fatto per le coppie, adattiamo il λ-calcolo tipizzato semplice aggiungendo alla sintassi dei tipi il caso dei tipi somma e alla sintassi dei termini le due iniezioni in_1 e in_2 (che incapsulano un'oggetto

di qualunque tipo in un tipo somma) e il costrutto $case$ (che ci consentirà di ragionare sul tipo contenuto in una somma):

⟨*T*⟩ ::= ... | ⟨*T*⟩ + ⟨*T*⟩

⟨*term*⟩ ::= ... | $(in_1$ ⟨*term*⟩$)$ | $(in_2$ ⟨*term*⟩$)$ | $(case$ ⟨*term*⟩ of $in_1(\langle var\rangle) \rightarrow \langle term\rangle, in_2(\langle var\rangle) \rightarrow \langle term\rangle)$

Per non appesantire ulteriormente la notazione, abbiamo utilizzato una grammatica imprecisa: da una parte le due iniezioni dovrebbero specificare il tipo somma destinazione (altrimenti il termine $(in_1\ true)$ potrebbe avere tipo $Bool + X$ per ogni X, e viceversa il termine $(in_2\ 0)$ potrebbe avere tipo $Y + Nat$ per ogni Y). Per motivi puramente tipografici assumeremo quindi che le iniezioni restituiscano sempre il tipo somma opportuno. In fig. 5.10 mostriamo le regole di tipo per i tipi somma.

$$\frac{\Gamma \vdash M : S}{\Gamma \vdash (in_1\ M) : S + T} \quad \text{(T-INJ.1)}$$

$$\frac{\Gamma \vdash N : T}{\Gamma \vdash (in_2\ N) : S + T} \quad \text{(T-INJ.2)}$$

$$\frac{\Gamma \vdash M : S + T \qquad \Gamma, x : S \vdash P : U \qquad \Gamma, y : T \vdash Q : U}{\Gamma \vdash (case\ M\ of\ (in_1\ x) \rightarrow P, (in_2\ y) \rightarrow Q) : U} \quad \text{(T-CASE)}$$

Figura 5.10: Regole di tipo per i tipi somma

Considereremo un *redex* anche l'applicazione del costrutto $case$ a un'iniezione, pertanto vanno aggiunte le seguenti nuove forme di β-riduzione:

$$\begin{aligned} (case\ (in_1\ M)\ of\ x : S \rightarrow P, y : T \rightarrow Q) &\longrightarrow_\beta P[M/x] \\ (case\ (in_2\ N)\ of\ x : S \rightarrow P, y : T \rightarrow Q) &\longrightarrow_\beta Q[N/y] \end{aligned}$$

La corrispondenza di Curry-Howard viene estesa con le seguenti regole che, aggiunte a quelle per il solo frammento implicativo, associano a dimostrazioni in deduzione naturale termini del calcolo.

8. Data una dimostrazione che termina con $(\vee i.1)$, sia $S \vee T$ la conclusione della dimostrazione e sia $a : S$ il termine associato alla sottodimostrazione immediatamente superiore all'ultimo passo di derivazione. Si fa corrispondere il termine $(in_1\ a) : S + T$.

9. Data una dimostrazione che termina con $(\vee i.2)$, sia $S \vee T$ la conclusione della dimostrazione e sia $b : T$ il termine associato alla sottodimostrazione immediatamente superiore all'ultimo passo di derivazione. Si fa corrispondere il termine $(in_2\ b) : S + T$.

10. Data una dimostrazione che termina con $(\vee e)$, siano

$$u : S + T$$
$$\lambda x : S.P : S \to U$$
$$\lambda y : T.Q : T \to U$$

i termini associati alle sottodimostrazioni immediatamente superiori all'ultimo passo di derivazione. Si fa corrispondere il termine

$$(case\ u\ of\ (in_1\ x) \to P, (in_2\ y) \to Q) : U$$

5.5 Teoremi di normalizzazione

Abbiamo già avuto modo di precisare come il λ-calcolo tipizzato semplice non consenta di esprimere numerose funzioni ricorsive, rivelandosi quindi un calcolo non Turing-completo. Ha dunque senso chiedersi se il calcolo tipizzato semplice è non terminante come quello non tipizzato. In effetti, come mostreremo in questa sezione, in $\lambda_\to$ è possibile esprimere solamente funzioni totali; equivalentemente, ogni termine di $\lambda_\to$ possiede una forma normale. Il procedimento di riduzione a forma normale di un termine è detto *normalizzazione*; con la normalizzazione sono connesse le seguenti due proprietà.

Definizione 5.1 (normalizzazione debole)**.** Un termine si dice *debolmente normalizzante* se esiste una sequenza di riduzioni che lo trasforma in forma normale. Un calcolo gode della proprietà di *normalizzazione debole* se tutti i termini in esso sono debolmente normalizzanti.

Definizione 5.2 (normalizzazione forte)**.** Un termine si dice *fortemente normalizzante* se ogni sequenza di riduzioni lo trasforma in forma normale. Un calcolo gode della proprietà di *normalizzazione forte* se tutti i termini in esso sono fortemente normalizzanti.

Chiaramente, la normalizzazione forte implica quella debole; il viceversa non è vero, come mostra il seguente termine del λ-calcolo puro

$$(\lambda x.y\ (\delta\ \delta))$$

che riduce a se stesso se si opera sul redex interno, mentre giunge immediatamente alla forma normale y se si opera sul redex esterno.

Entrambe le proprietà valgono per il λ-calcolo tipizzato semplice: le dimostreremo separatamente nei prossimi paragrafi.

5.5.1 Teorema di normalizzazione debole

Teorema 5.3 (normalizzazione debole)**.** *Per ogni termine del λ-calcolo tipizzato semplice esiste una sequenza di riduzioni che lo trasforma in forma normale.*

Nella dimostrazione di questo teorema useremo la seguente nozione di ordinamento *ben fondato*.

Definizione 5.4. Un ordinamento parziale stretto si dice *ben fondato* se non esistono catene discendenti infinite.

Definire un ordinamento ben fondato sui termini ci consentirà di scegliere sempre un redex che faccia "decrescere" il termine fino a raggiungere la forma normale.

L'ordinamento cui siamo interessati è quello lessicografico tra coppie di naturali:

$$\langle n, m\rangle \lessdot \langle n', m'\rangle \text{ sse } n < n' \text{ o } n = n' \text{ e } m < m'$$

La dimostrazione della normalizzazione debole deve risolvere due problemi: in primo luogo la riduzione può duplicare redex...