Università degli Studi di Roma Tre

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica

Logica Lineare e Complessità Computazionale

Relatore
Prof. Giuseppe Di Battista

Correlatore
Prof. Lorenzo Tortora de Falco

Candidato
Damiano Mazza

Anno Accademico 2001/2002

A moms e papo

Introduzione

Computer Science is no more about computers than astronomy is about telescopes.
Edsger Wybe Dijkstra

Sta scritto (in un qualche libro) che ogni libro è una risposta ad una qualche domanda. Una frase contenente un quantificatore universale va sempre presa con la dovuta cautela, specie se in essa c'è un vago sapore autoreferenziale; tuttavia, in questo caso, possiamo dire di soddisfare piuttosto bene l'asserto. In particolare, la domanda cui si tenta di rispondere qui è la seguente: cos'hanno in comune il Teorema di Pitagora ed il programma che utilizziamo tutti i giorni per leggere la posta elettronica?

Sebbene non venga mai fornita esplicitamente una risposta, l'intera trattazione che segue è imperniata proprio attorno ad una versione, certamente più generale, di tale questione. In effetti, la nostra discussione si collocherà all'interfaccia tra due grandi settori della ricerca contemporanea, quello della Logica e quello dell'Informatica. La prima ha una tradizione millenaria, ed è già di per sé una disciplina "di confine", nella quale si mescolano indissolubilmente matematica e filosofia; la seconda, al contrario, non ha neanche un secolo di vita, ma è al giorno d'oggi uno dei settori più prolifici della scienza umana, sia in campo teorico che applicativo.

La grandissima maggioranza dei nostri lettori avrà probabilmente sentito parlare della logica come, più o meno, di una scienza nella quale si cerchi di mostrare come alcune cose siano *necessariamente vere* a partire dalla verità di altre. In effetti, non c'è nulla di sbagliato in tutto ciò; parte dello scopo di questo lavoro è far vedere che però c'è anche dell'altro. L'"altro" di cui parliamo si può riassumere in quattro parole: *Corrispondenza di Curry-Howard.*

La corrispondenza di Curry-Howard, scoperta negli anni '60, è quella che dà un senso alla domanda con la quale abbiamo aperto quest'introduzione. Sebbene infatti, molto probabilmente, i teoremi della geometria euclidea non abbiano davvero nulla a che vedere con i client di posta elettronica[1], in senso

[1]Ebbene sì: la domanda iniziale serviva effettivamente solo a catturare l'attenzione del

più generale la corrispondenza di Curry-Howard ci garantisce che, vista in un certo modo, *ogni dimostrazione matematica è un programma*, al pari di qualunque altra applicazione che gira sul nostro PC. Da questa prospettiva, la logica perde il suo volto abituale, per diventare un vero e proprio linguaggio di programmazione. In altre parole, non si è più interessati alla *verità* di una certa proposizione o alla *dimostrabilità* di un qualche asserto a partire da altri, ma piuttosto alla *struttura* delle dimostrazioni stesse, ed al loro *significato* in quanto "oggetti in grado di interagire l'uno con l'altro", proprio come un client ed un server Web interagiscono scambiandosi informazioni. Si badi inoltre che la corrispondenza va nei due sensi: la logica può parlare di informatica (e sarà principalmente questa la direzione da noi esplorata) ma, allo stesso modo, l'informatica può parlare di logica; la ricchezza delle cose che una può dire dell'altra è limitata solo dalla nostra comprensione dei loro rispettivi linguaggi.

Venendo a noi, il ramo dell'informatica che tratteremo in modo specifico sarà la Teoria della Complessità Computazionale. Essendo una branca dell'informatica, la teoria della complessità è ancor più giovane di quest'ultima, ed è un campo la cui esistenza vera e propria risale solo agli ultimi trent'anni. Essa si occupa, sempre per porla in termini "interrogativi", di trovare soluzioni a questioni del seguente tipo: qual è la quantità minima di risorse (ad esempio in termini di tempo impiegato) per calcolare, diciamo, la somma di due numeri? In che modo essa è legata alla grandezza dei due numeri in gioco? La teoria della complessità è uno dei tanti settori della scienza nel quale si hanno di gran lunga più domande che risposte; il nostro lavoro si può vedere anche come un contributo, microscopico, al tentativo di "risanare" questo bilancio.

Nella prima parte dell'esposizione, tratteremo il legame tra Logica ed Informatica nei termini accennati sopra, mostrando come sia possibile utilizzare la logica per analizzare i problemi di complessità computazionale. Nella seconda parte, esamineremo un possibile approccio logico alla teoria della complessità: presenteremo i sistemi già esistenti che permettono di riformulare in termini puramente logici i problemi di complessità, e concluderemo introducendo una variante di tali sistemi, discutendone approfonditamente le proprietà.

Poiché, come abbiamo abbondantemente sottolineato, il nostro strumento di analisi dei problemi informatici sarà la logica, il primo capitolo della trattazione sarà dedicato proprio ad un'introduzione ai sistemi di derivazione logici. Qui si lavorerà ancora in ambito "tradizionale", vedendo la

lettore; se egli è arrivato fin qui, tanto da leggere anche questa nota inutile, lo incitiamo a proseguire almeno fino alla fine dell'introduzione...

logica nella sua veste più abituale. Enunceremo alcuni dei risultati classici, in particolare il Teorema di Cut-Elimination (sezione 1.3), che è la base della corrispondenza di Curry-Howard.

Nel capitolo 2 dimenticheremo per un momento la logica e ci volgeremo verso l'altro lato del confine su cui ci siamo posti, presentando tre modelli tradizionali del calcolo: le macchine di Turing, le funzioni ricorsive e il λ-calcolo. Di quest'ultimo introdurremo anche la versione *semplicemente tipata* (2.3.2), di fondamentale importanza in ambito logico.

Il capitolo 3 è il cuore della prima parte, e mette assieme le nozioni introdotte nei capitoli precedenti per giungere all'esposizione della già citata Corrispondenza di Curry-Howard, senza la quale nulla di quanto diremo nel seguito avrebbe senso. Si ritornerà dapprima in campo logico, introducendo la Logica Intuizionista (3.1); poi, si presenterà un sistema di derivazione alternativo a quello introdotto nel capitolo 1, all'interno del quale potremo finalmente, tornando a parlare di λ-calcolo, stabilire la nostra corrispondenza dimostrazioni/programmi (3.3).

Il capitolo 4 è volto ad approfondire quanto esposto nel capitolo 3. In particolare, una volta constatato che una dimostrazione può essere vista come un programma, ci occuperemo di stabilire quali programmi siano effettivamente rappresentabili mediante la logica. Baseremo i nostri risultati principalmente sul cosiddetto Sistema $\mathbb{F}$, il cui potere espressivo, come vedremo nella sezione 4.2, è talmente elevato da poter considerare tale sistema un vero e proprio linguaggio di programmazione. Faremo inoltre un accenno ad $\mathbf{AF}_2$ (Aritmetica Funzionale al secondo ordine), un sistema che "estende" $\mathbb{F}$ e nel quale è possibile dimostrare alcune proprietà di correttezza dei programmi davvero notevoli.

Il capitolo 5 introduce finalmente la teoria della complessità computazionale, esponendo la sua versione "standard" in termini di macchine di Turing. Nel seguito (sezione 5.2) sarà altresì considerato un altro possibile approccio alla teoria della complessità, basato questa volta sulle funzioni ricorsive introdotte nel capitolo 2. L'ultima sezione del capitolo è invece dedicata alla rivisitazione del teorema di cut-elimination, vecchia conoscenza del capitolo 1, cui verrà data una lettura in termini di complessità computazionale. In particolare, stabiliremo che, nell'ambito della logica classica, non è possibile fornire alcuna caratterizzazione delle classi di complessità computazionale (in termini di cut-elimination), lasciando però aperta la questione dell'esistenza di "frammenti" nei quali invece sia possibile affrontare i problemi di complessità.

La seconda parte della trattazione si apre con il capitolo 6, il quale introduce la Logica Lineare che, nata per tutti altri motivi, fornisce l'unica soluzione attualmente disponibile al problema posto nel capitolo 5.

Nel capitolo 7 introdurremo una classe di oggetti completamente nuovi nel mondo della logica, i quali costituiscono uno tra gli strumenti più interessanti forniti dalla logica lineare, che prendono il nome di *proof-net* (reti dimostrative). In sostanza, vedremo che le proof-net sono una presentazione "parallela" delle dimostrazioni della logica lineare, le quali si riducono essenzialmente a grafi aventi certe proprietà. Per mezzo delle proof-net, sarà possibile vedere il processo di calcolo presente in logica come un processo di *riscrittura di grafi.*

Nel capitolo 8 esporremo, in modo molto sintetico, lo "stato dell'arte" della rappresentazione delle classi di complessità computazionale in logica lineare. Introdurremo dunque alcuni frammenti della logica lineare che possono essere visti come linguaggi di programmazione in grado di rappresentare tutte e sole le funzioni di una certa classe di complessità. In particolare, vedremo una caratterizzazione di **ELEMENTARY** (la classe delle funzioni calcolabili in tempo elementare) e varie caratterizzazioni di **P** (la classe delle funzioni calcolabili in tempo polinomiale da una macchina di Turing deterministica).

Nell'ultimo capitolo, che è il frutto del nostro lavoro di ricerca, presenteremo una nuova caratterizzazione di **P** in termini di un sottoinsieme delle proof-net della logica lineare. La presentazione sarà molto dettagliata, ma le conoscenze acquisite nel corso della lettura dei capitoli precedenti dovrebbero, speriamo, quantomeno evitare al lettore nuovo a questi argomenti la sensazione che si stia parlando ostrogoto[2]. Il capitolo 9 sarà diviso in quattro sezioni: in 9.1 verrà definito il nostro sistema, che chiameremo $\overline{\mathbf{LLL}}$; in 9.2 verrà esposta la sua dinamica (la procedura di cut-elimination ad esso associata), investigando la possibilità di rappresentare le proof-net del sistema per mezzo di una sintassi più "idonea" a far funzionare la cut-elimination, nota con il nome di *nouvelle syntaxe*; in 9.3 verrà dimostrata la **PTIME**-correttezza di $\overline{\mathbf{LLL}}$, ovvero che l'insieme delle funzioni programmabili nel sistema è contenuto in **P**; la sezione 9.4 sarà invece dedicata alla dimostrazione dell'inclusione inversa, cioè che la classe **P** è contenuta nell'insieme delle funzioni programmabili in $\overline{\mathbf{LLL}}$. Ciò completerà il lavoro, dimostrando l'equivalenza fra il nostro sistema e la classe delle funzioni deterministiche polinomiali.

[2]O una lingua particolarmente incomprensibile per chi conosca l'ostrogoto. . .

Indice

Parte I

Logica e Informatica

Capitolo 1

I sistemi logici e la cut-elimination

In questo primo capitolo faremo una rapidissima introduzione alla logica, coprendo in qualche dettaglio gli aspetti che saranno più di rilievo per il seguito della discussione. In particolare, parleremo delle questioni basilari della logica classica e dei sistemi di deduzione basati su di essa, presentando il teorema di cut-elimination*, risultato che costituisce le fondamenta del legame logica/informatica.*

1.1 La logica classica

E' impossibile dire quanto sia antica nell'essere umano l'intuizione originale del legame *causa/effetto*, la comprensione che il *dopo* possa dipendere in qualche modo da quello che è venuto *prima*. E' possibile però dire che questa è una delle proprietà fondamentali di quella che noi chiamiamo intelligenza, e la gestione del concetto più generale di *legame logico* è probabilmente alla base del ragionamento umano. Implicitamente, dunque, la comparsa della logica risale agli albori della nostra stessa esistenza.

Col tempo, quello che era uno strumento utilizzato per lo più inconsapevolmente da parte di tutti ha cominciato a divenire oggetto centrale di studio, alla ricerca dei meccanismi ad esso sottostanti. A partire dai matematici e filosofi dell'antica Grecia, il ragionamento umano, in particolare quello matematico, è stato oggetto di una graduale *formalizzazione*; si è man mano indagato sull'esistenza di leggi che governano lo snodarsi dei nostri percorsi deduttivi, e si è tentato col tempo di fornire una definizione rigorosa delle regole del pensiero umano, estrapolandone il funzionamento fondamentale. Da questa ricerca nasce la logica matematica, il cui scopo originale è appunto quello di *ragionare sul ragionamento*, di investigare il concetto di *conseguenza logica* e di isolare dunque i legami logici ben formati da quelli "fasulli".

Dalla nascita delle "teorie assiomatiche" di Euclide, ai sillogismi di Aristotele, alle regole logiche degli Scolastici, si è così arrivati fino al *boom* dei formalismi logici d'inizio secolo scorso. La logica matematica moderna nasce proprio in quegli anni, sotto la spinta di una delle questioni scientifiche fondamentali dell'epoca, vale a dire la dimostrazione della *consistenza* della matematica. Quello che si tentava di fare, cosa che fu proposta in modo "ufficiale" da parte di un matematico del calibro di Hilbert nel suo famoso "secondo problema", era di definire un sistema formale nell'ambito del quale non solo tutta la matematica trovasse posto, ma il quale fosse anche in grado di dimostrare la propria coerenza, costituendo così un mondo chiuso, auto-contenuto, esente da contraddizioni.

Oggi sappiamo che il sogno di Hilbert non è realizzabile; il celeberrimo teorema di incompletezza di Gödel stabilisce che la chiusura dei sistemi formali è impossibile: per giustificare tutto ciò che sta dentro si ha sempre bisogno di qualcosa che sta fuori. Nonostante tutto, l'esplorazione dei sistemi deduttivi logici che è stata compiuta nella prima parte del secolo scorso costituisce ancora la base di tutti i risultati importanti della logica matematica contemporanea, non ultimi tra i quali quelli riguardanti l'Intelligenza Artificiale e l'Informatica Teorica, che è l'ambito della nostra dissertazione.

Procederemo dunque con il presentare la logica matematica nella versione definitasi proprio in quel periodo; in particolare, sceglieremo di operare inizialmente nel sistema conosciuto come Logica Classica, per il quale introdurremo la nozione di *linguaggio formale al primo ordine.*

Definizione 1.1 (Linguaggio formale al primo ordine) *Un linguaggio formale al primo ordine per la logica classica è un linguaggio il cui alfabeto base è composto dai seguenti simboli:*

- *un insieme numerabile di* simboli per variabili *(o, più semplicemente,* variabili*),* $x, y, z, \ldots$
- *un insieme numerabile di* simboli per funzioni *(o, più semplicemente,* funzioni*),* $f^{k_1}, g^{k_2}, h^{k_3}, \ldots$*, dove* $k_1, k_2, k_3, \ldots$ *indicano le arità delle funzioni. Se l'arità è zero, i simboli saranno detti* simboli per costanti *(o, più semplicemente,* costanti*); spesso l'arità verrà omessa, chiaramente a patto che ciò non causi problemi di ambiguità*
- *un insieme numerabile di coppie di* simboli di predicato *(o, più semplicemente,* predicati*),* $P^{k_1}, \neg P^{k_1}, Q^{k_2}, \neg Q^{k_2}, R^{k_3}, \neg P^{k_3}, \ldots$*, dove* $k_1, k_2, k_3, \ldots$ *indicano le arità dei predicati. I simboli la cui arità è zero saranno detti* simboli per variabili proposizionali *(o, più semplicemente,* varia-

bili proposizionali*); spesso l'arità verrà omessa, chiaramente a patto che ciò non causi problemi di ambiguità*

- *le* costanti logiche $\mathcal{T}$ *e* $\mathcal{F}$
- *i* simboli logici $\neg$, $\wedge$, $\vee$, $\Rightarrow$, $\forall$, $\exists$
- *i simboli di "punteggiatura" '.', ',', '(' e ')'*

Definiremo ora quelli che chiameremo i *termini* e le *formule* sul linguaggio:

Definizione 1.2 (Termine al primo ordine) *Se $\mathcal{L}$ è un linguaggio al primo ordine, i termini su $\mathcal{L}$ sono definiti induttivamente come segue:*

- *le costanti e le variabili sono termini*
- *se f è una funzione k-aria, e $t_1, \ldots, t_k$ sono k termini, allora $f(t_1, \ldots, t_k)$ è un termine*

Definizione 1.3 (Formula al primo ordine) *Se $\mathcal{L}$ è un linguaggio al primo ordine, le formule su $\mathcal{L}$ sono definite induttivamente come segue:*

- *le costanti logiche sono formule*
- *le variabili proposizionali sono formule*
- *se P e $\neg P$ sono due predicati di arità k, e se $t_1, \ldots, t_k$ sono k termini sul linguaggio $\mathcal{L}$, allora $P(t_1, \ldots, t_k)$ e $\neg P(t_1, \ldots, t_k)$ sono formule*
- *se A è una formula, allora $\neg A$ è una formula*
- *se A e B sono formule, allora $A \wedge B$, $A \vee B$ e $A \Rightarrow B$ sono formule*
- *se $A[x]$ è una formula che non contiene quantificatori ($\forall$ e $\exists$) e contiene la variabile x, diremo che tutte le occorrenze di x sono* libere. *In tal caso, $\forall x.A$ e $\exists x.A$ sono anch'esse formule, e in esse tutte le occorrenze della variabile x diventano* vincolate. *Nel caso in cui ci siano occorrenze di altre variabili nella formula A, tali occorrenze rimarranno naturalmente libere*
- *se $A[x]$ è una formula contenente una qualche occorrenza libera della variabile x, allora $\forall x.A$ e $\exists x.A$ sono formule, e anche in tal caso tutte le occorrenze di x non saranno più libere*

Le formule che sono costanti logiche o variabili proposizionali sono dette anche *formule atomiche*. Le formule che contengono variabili libere sono dette *aperte*; quelle che invece non ne contengono sono dette *chiuse*. Se $A[x_1, \ldots, x_n]$ è una formula aperta in cui le variabili libere sono $x_1, \ldots, x_n$, allora chiameremo *chiusura universale* e *chiusura esistenziale* di A le formule, rispettivamente, $\forall x_1 \ldots \forall x_n.A$ e $\exists x_1 \ldots \exists x_n.A$.

E' interessante notare come, nella definizione dei termini e delle formule, si è fatto ricorso ad un *meta-linguaggio*, più esteso del linguaggio al primo ordine di cui si sta parlando. I simboli $t_1, \ldots, t_k$ che denotano i termini sono esempi di meta-simboli; così lo sono anche i simboli per le formule come A e B, e la notazione $A[x]$. Nel seguito, sarà utilizzata un'altra meta-notazione simile, per indicare la sostituzione di termini a variabili: se $\mathcal{L}$ è un linguaggio al primo ordine, e se t è termine di $\mathcal{L}$ e $A[x]$ una formula di $\mathcal{L}$ contenente occorrenze libere di x, allora $A[t/x]$ denoterà la medesima formula all'interno della quale tutte le occorrenze di x sono state sostituite dal termine t. Inoltre, precisiamo fin d'ora un abuso di linguaggio che commetteremo spessissimo nel corso dell'esposizione, costituito dalla confusione arbitraria dei concetti di *formula* e *occorrenza di formula*.

Il calcolo logico che si può costruire sulle formule della logica classica al primo ordine (su un certo linguaggio $\mathcal{L}$) è spesso chiamato anche *calcolo dei predicati*. A tal proposito, esiste una versione più semplice, detta *calcolo proposizionale*, che si ottiene definendo i linguaggi proposizionali come sottoinsiemi di quelli al primo ordine:

Definizione 1.4 (Linguaggio formale proposizionale) *Un linguaggio formale proposizionale è un linguaggio basato sul sottoinsieme dell'alfabeto di un linguaggio al primo ordine, ottenuto togliendo dall'alfabeto di partenza tutti i simboli di variabile, i simboli per funzioni di qualsiasi arità, e i simboli di predicato di arità non nulla; inoltre, sono esclusi i simboli logici* $\forall$ *e* $\exists$*, e i simboli di punteggiatura '.' e ','.*

E' chiaro che per un linguaggio proposizionale non ha più senso parlare di termini; le formule su un linguaggio proposizionale sono invece definibili nel seguente modo:

Definizione 1.5 (Formule proposizionali) *Le formule su un linguaggio proposizionale* $\mathcal{L}$ *sono definibili induttivamente come segue:*

- *le costanti logiche sono formule*
- *le variabili proposizionali sono formule*
- *se* A *è una formula,* $\neg A$ *è una formula*
- *se* A *e* B *sono due formule,* $A \wedge B$, $A \vee B$ *e* $A \Rightarrow B$ *sono formule*

I linguaggi introdotti fin qui sono, da soli, perfettamente inutili. Essi infatti sostanzialmente *non parlano di nulla*; sorge la necessità fondamentale di attribuire ai simboli della logica classica (proposizionale o al primo ordine) un qualche *significato*, una qualche interpretazione che definisca quale sia l'oggetto dei "discorsi" che si possono costruire con tali linguaggi. Il compito di riempire questo vuoto esistenziale è quello della *semantica dei modelli*, che attribuisce un ambito ben preciso ai termini e un valore di verità (del tipo *vero* o *falso*) alle formule[1]. Definiamo allora il concetto di *interpretazione*[2] di un linguaggio al primo ordine $\mathcal{L}$:

Definizione 1.6 (Interpretazione) *Se $\mathcal{L}$ è un linguaggio al primo ordine, e $\mathcal{D}$ un insieme non vuoto che chiameremo* dominio, *un'interpretazione σ è un'applicazione che mette in corrispondenza i termini di $\mathcal{L}$ ai seguenti oggetti:*

- *se x è una variabile, $\sigma(x) \in \mathcal{D}$*
- *se c è una costante, $\sigma(c) \in \mathcal{D}$*
- *se f è un simbolo di funzione di arità k, ad f viene associata una funzione $|f| : \mathcal{D}^k \rightarrow \mathcal{D}$, e dunque, se $t_1, \ldots, t_k$ sono k termini di $\mathcal{L}$, $\sigma(f(t_1, \ldots, t_k)) = |f|(\sigma(t_1), \ldots, \sigma(t_k)) \in \mathcal{D}$*

L'interpretazione dunque stabilisce di *cosa* parla il linguaggio $\mathcal{L}$. La *realizzazione* invece stabilisce di cosa parlano le formule su $\mathcal{L}$, e in che modo esse vadano lette. Sia, nel seguito, $\underline{2} = \{0, 1\}$.

Definizione 1.7 (Realizzazione) *Se $\mathcal{L}$ è un linguaggio al primo ordine, σ un'interpretazione di dominio $\mathcal{D}$ su tale linguaggio, chiamiamo realizzazione su $\mathcal{L}$ un'applicazione $\rho(\sigma)$ dalle formule di $\mathcal{L}$ all'insieme $\underline{2}$ così fatta:*

- *alle formule atomiche viene associato un elemento di $\underline{2}$ arbitrario, ad eccezione delle costanti logiche, per le quali è*
  $\rho(\sigma)(\mathcal{T}) = 1$
  $\rho(\sigma)(\mathcal{F}) = 0$
- *se P e $\neg P$ sono due predicati k-ari, allora ad essi vengono associate due relazioni $|P|, |\neg P| \subseteq \mathcal{D}^k$ tali che $|P| \cap |\neg P| = \emptyset$ e $|P| \cup |\neg P| = \mathcal{D}^k$, così che, se $t_1, \ldots, t_k$ sono k termini di $\mathcal{L}$,*
  $\rho(\sigma)(P(t_1, \ldots, t_k)) = 1$ *sse* $\langle \sigma(t_1), \ldots, \sigma(t_k) \rangle \in |P|$
  $\rho(\sigma)(\neg P(t_1, \ldots, t_k)) = 1$ *sse* $\langle \sigma(t_1), \ldots, \sigma(t_k) \rangle \notin |P|$

[1]Vedremo nel seguito, anche se solo accennandolo, che questo non è l'unico tipo di semantica che può esistere in logica; in particolare, a livello di teoria della dimostrazione e in ambito informatico hanno enormemente più importanza le cosiddette *semantiche denotazionali*.

[2]In merito alla nomenclatura c'è da fare una precisazione: quella che noi chiamiamo *interpretazione* è da alcuni chiamata *assegnazione*, e chi fa questa scelta molto probabilmente si riferisce a quella che noi chiameremo *realizzazione* proprio con il termine di interpretazione, il ché può essere causa di fastidiosissime confusioni. Anche sulle cose più stupide, mettersi d'accordo non è mai banale...

- *se A è una formula,* $\rho(\sigma)(\neg A) = 1 - \rho(\sigma)(A)$
- *se A e B sono due formule,*
  $\rho(\sigma)(A \wedge B) = \rho(\sigma)(A) \cdot \rho(\sigma)(B)$
  $\rho(\sigma)(A \vee B) = 1 - \rho(\sigma)(\neg A) \cdot \rho(\sigma)(\neg B)$
  $\rho(\sigma)(A \Rightarrow B) = \rho(\sigma)(\neg A \vee B)$
- *se $A[x]$ è una formula contenente occorrenze libere della variabile x,*
  $\rho(\sigma)(\forall x.A) = 1$ *sse, per ogni termine t del linguaggio,* $\rho(\sigma)(A[t/x]) = 1$
  $\rho(\sigma)(\exists x.A) = 1 - \rho(\sigma)(\forall x.\neg A)$

E' chiaro che, se agli elementi di $\underline{2}$ associamo i significati di *vero* e *falso*, le realizzazioni non fanno altro che attribuire alle formule il valore di verità che ci aspetteremmo: $A \wedge B$ è vera se e solo se sono vere sia A che B, $A \vee B$ è vera se e solo se almeno una delle due è vera, e così via.

Possiamo a questo punto introdurre il concetto fondamentale di *modello* (al primo ordine):

Definizione 1.8 (Modello al primo ordine) *Sia $\mathcal{L}$ un linguaggio al primo ordine, σ un'interpretazione e A una formula su tale linguaggio, contenente le variabili libere $x_1, \ldots, x_n$. Allora, diremo che la realizzazione $\rho(\sigma)$ è un* modello *di A, e scriveremo*

$$\rho(\sigma) \models A$$

se e soltanto se, per qualunque scelta di n termini $t_1, \ldots, t_n$ di $\mathcal{L}$, si ha

$$\rho(\sigma)(A[t_1/x_1, \ldots, t_n/x_n]) = 1$$

Osserviamo che la validità di una formula aperta in un modello è equivalente a quella della sua chiusura universale. Per questo motivo, quando si parla di modelli si considerano in genere solamente formule chiuse, nel qual caso non ha più importanza specificare la particolare interpretazione utilizzata, contando solo il dominio di essa. Dunque, se A è una formula chiusa e $\mathcal{M}$ una realizzazione, "$\mathcal{M}$ è modello di A" (o anche "A è valida in $\mathcal{M}$") si scriverà semplicemente come $\mathcal{M} \models A$.

Dalla validità di una singola formula si può passare rapidamente alla validità di un insieme arbitrario di formule:

Definizione 1.9 *Sia $\mathbf{S}$ un insieme di formule chiuse del linguaggio al primo ordine $\mathcal{L}$ e $\mathcal{M}$ una realizzazione per tale linguaggio; allora, diremo che $\mathcal{M}$ è modello di $\mathbf{S}$, e scriveremo*

$$\mathcal{M} \models \mathbf{S}$$

se e solo se, per ogni formula A di $\mathbf{S}$, $\mathcal{M} \models A$.

Arriviamo così ai concetti chiave di *soddisfacibilità*, *conseguenza logica* e *verità logica*, che definiremo di seguito:

Definizione 1.10 (Soddisfacibilità) *Una formula chiusa A è soddisfacibile se e soltanto se esiste una realizzazione $\mathcal{M}$ tale che $\mathcal{M} \models A$. Allo stesso modo, un insieme di formule chiuse* **S** *è soddisfacibile se e soltanto se esiste una realizzazione $\mathcal{M}$ tale che $\mathcal{M} \models S$.*
Viceversa, diremo che una formula A [un insieme **S***] è insoddisfacibile se non ammette modelli.*

In merito alla soddisfacibilità, esiste il seguente teorema fondamentale:

Teorema 1.1 (Teorema di Compattezza) *Sia* **S** *un insieme infinito di formule.* **S** *è soddisfacibile se e soltanto se ogni sottoinsieme finito $\mathbf{S}_{fin}$ di* **S** *è soddisfacibile.*

La dimostrazione del teorema di compattezza viene omessa in questa sede; il lettore interessato la potrà trovare, ad esempio, in [7].

Definizione 1.11 (Conseguenza logica e verità logica) *Sia A una formula e* **S** *un'insieme di formule. Denotiamo con $\mathsf{Mod}(A)$ e $\mathsf{Mod}(\mathbf{S})$ rispettivamente l'insieme dei modelli di A e di* **S***. Diremo che A è* conseguenza logica *di* **S***, e scriveremo*

$$\mathbf{S} \models A$$

se e soltanto se

$$\mathsf{Mod}(\mathbf{S}) \subseteq \mathsf{Mod}(A)$$

Nel caso in cui **S** *sia l'insieme vuoto, scriveremo*

$$\models A$$

e, poiché ovviamente tutte le realizzazioni sono modelli dell'insieme vuoto, questa notazione significherà che A è soddisfatta anch'essa da tutti i modelli; diremo allora che A è una verità logica.

Il significato intuitivo di $\mathbf{S} \models A$ è che, ogni volta che sono vere tutte le formule di **S**, dev'essere necessariamente vera anche A; una verità logica invece è appunto una formula che è "sempre vera".

Anche nel caso della logica classica proposizionale si possono fornire restrizioni opportune delle definizioni introdotte fin qui, e dare validità anche in tale ambito ai concetti di modello, soddisfacibilità, conseguenza logica e verità logica; non ci interessa però entrare nel dettaglio della cosa.

Quello che è interessante accennare è che esiste invece un'estensione della logica proposizionale che include i quantificatori, detta *logica proposizionale al secondo ordine*. I linguaggi formali alla base di tale logica sono definiti in modo analogo a quanto fatto per il caso proposizionale e del primo ordine; in particolare, si prende la definizione di un linguaggio proposizionale e si aggiunge la possibilità di quantificare (esistenzialmente o universalmente) sulle variabili proposizionali. Ad esempio, se P è una variabile proposizionale, la formula $\forall P.((P \Rightarrow P) \Rightarrow (P \Rightarrow P))$ è una formula proposizionale al secondo ordine. Il significato intuitivo della quantificazione universale al secondo ordine è il seguente: la formula $\forall P.A$ è vera se è soltanto se qualunque *formula* B si scelga per rimpiazzare le occorrenze di P, $A[B/P]$ è vera. Per l'esistenziale ovviamente vale il discorso esattamente duale. L'introduzione dei quantificatori aumenta l'espressività del sistema; ad esempio, è possibile eliminare dalla sintassi le costanti logiche $\mathcal{T}$ e $\mathcal{F}$, giacché quest'ultima può essere sostituita dalla formula $\forall P.P$, che è logicamente equivalente al falso (cioè $\mathcal{M}$ è un modello di $\forall P.P$ se e soltanto se $\mathcal{M}$ è un modello di $\mathcal{F}$, ovvero $\forall P.P$ non ha modelli — lasciamo al lettore curioso la soddisfazione di controllare che sia effettivamente così!). L'eliminazione delle costanti logiche è un guadagno risibile rispetto al guadagno che si ottiene dal punto di vista dell'espressività computazionale; ma questo sarà oggetto dei capitoli a venire.

Per contro, l'utilizzo contemporaneo dei quantificatori al primo e al secondo ordine (definendo quindi quella che è la logica classica al secondo ordine) fornisce un aumento enorme del potere espressivo, consentendo di poter parlare non solo di oggetti di un certo dominio ma anche, allo stesso tempo, di sottoinsiemi qualunque del medesimo dominio. Ad ogni modo, non ci avventureremo minimamente nella presentazione dei modelli al secondo ordine, neanche nel caso proposizionale; basti sapere che esistono, e che per essi si possono definire gli stessi concetti che per i modelli al primo ordine.

1.2 Il calcolo dei sequenti

Fin qui abbiamo parlato di linguaggi formali (proposizionali, al primo ordine, al secondo ordine e così via), definendo le formule su tali linguaggi a attribuendo loro un significato tramite i modelli. Sappiamo ad esempio cosa significa che una formula è conseguenza logica di altre due formule, e cosa significhi che un'altra formula è una verità logica o che è insoddisfacibile. Tuttavia, non abbiamo ancora mai accennato a *come* si può mostrare la validità di tali proprietà per talune formule, e la non validità per talaltre. Non abbiamo cioè introdotto un concetto cruciale della logica, che è quello di *dimostrazione*.

Quando si parla di dimostrazioni, se ne vorrebbe poter avere a disposizione una rappresentazione concreta, che consenta di prendere un qualche oggetto e dire, ad esempio, "questa è una dimostrazione della formula A". Il compito di ridurre il concetto astratto di dimostrazione a qualcosa di manipolabile a livello simbolico spetta alla *sintassi* della logica, e in particolare a tutta una branca che si è sviluppata a partire dall'inizio del secolo scorso e che ha preso il nome di Teoria della Dimostrazione.

In questa sede adotteremo un approccio molto semplicistico alla questione, riducendoci a presentare solamente quegli aspetti della teoria della dimostrazione che sono d'interesse per l'informatica teorica. In particolare, non parleremo inizialmente di *dimostrazioni di formule* ma piuttosto di *derivazioni di sequenti*. Anzitutto dunque occorre dare la definizione di *sequente*:

Definizione 1.12 *Siano* Γ *e* Δ *due liste ordinate di occorrenze*[3] *di formule (eventualmente vuote), pari rispettivamente a* $C_1, \ldots, C_m$ *e* $D_1, \ldots, D_n$*. Se* Λ *è una lista di formule, denoteremo d'ora in poi con* $|\Lambda|$ *il numero di formule che la compongono; in questo caso ad esempio si ha* $|\Gamma| = m$ *e* $|\Delta| = n$*. L'oggetto sintattico che denoteremo con*

$$\Gamma \vdash \Delta$$

sarà chiamato sequente*, e si leggerà "Gamma* tesi *Delta". Il significato di tale notazione è che* esiste una dimostrazione del fatto che, se tutte le formule di Γ sono vere, allora è vera almeno una delle formule di Δ. *Le formule di* Γ *saranno dunque dette* premesse *del sequente e le formule di* Δ conclusioni *del sequente.*

Il sequente asserisce dunque l'esistenza di una dimostrazione della verità della *disgiunzione* delle sue conclusioni a partire dalla verità della *congiunzione* delle sue premesse[4]. Inoltre, è chiaro che, a priori, alcuni sequenti asseriranno legami logici validi, altri no.

Vale la pena analizzare tre casi particolari di sequente, in modo da chiarire ancor più il significato intuitivo di tale notazione:

- il sequente $\vdash \Delta$, ottenuto quando $|\Gamma| = 0$, significa che esiste una dimostrazione del fatto che almeno una formula di Δ è sempre vera;
- il sequente $\Gamma \vdash$, ottenuto quando $|\Delta| = 0$, significa che esiste una dimostrazione della contraddittorietà delle formule di Γ, ovverosia che non possono essere tutte quante vere allo stesso tempo: supporre che lo siano porta ad un assurdo;

[3]In questo caso è importante precisare che si tratta di *occorrenze di formule* e non semplicemente di formule; infatti, una lista Γ può contenere più occorrenze della medesima formula, come nel caso della lista $\Gamma = A, A, B$. Avendo chiarito questo punto, d'ora in avanti proseguiremo ad adottare la già menzionata confusione tra i due concetti.

[4]Si badi attentamente al fatto che il sequente non dice nulla riguardo a come sia fatta tale dimostrazione; questo punto sarà sviluppato più avanti (vedi sezione 3.2)

- il sequente vuoto, $\vdash$, ottenuto quando $|\Gamma| = |\Delta| = 0$, è il disastro totale della logica: significa che esiste una dimostrazione dell'assurdo, asserendo dunque la contraddittorietà fondamentale del sistema logico all'interno del quale viene derivato.

Sappiamo ora cos'è un sequente e qual è il suo significato intuitivo, che consiste nell'esprimere sintatticamente la dimostrabilità di un legame logico semantico. Ma come si può prendere un sequente e stabilire se quanto asserisce è valido o no? La risposta sta in quello che viene chiamato *calcolo dei sequenti*, introdotto originariamente da Gentzen all'inizio degli anni '30. Il calcolo dei sequenti fornisce delle regole per poter derivare un sequente a partire da altri sequenti. Esso fornisce un modo per costruire una nuova dimostrazione conoscendo la validità di altre: poiché infatti un sequente asserisce l'esistenza di una dimostrazione, il passare da uno o più sequente/i ad un altro significa asserire l'esistenza di una dimostrazione a partire da altre.

Quello che serve anzitutto è un punto di partenza, un sequente che non dev'essere costruito a partire da altri. Bisogna cioè trovare dei sequenti che, intuitivamente, proclamino di rappresentare dimostrazioni talmente banali e ovvie la cui esistenza è praticamente incontestabile. Questo sarà il punto di partenza di tutta la logica, l'unico *assioma* che dovrà essere preso così com'è senza possibilità di spiegazioni. I sequenti in questione sono quelli ottenuti mediante la prima regola che introduciamo per il calcolo dei sequenti della logica classica, e che è la *regola dell'identità* (talvolta detta appunto, come nel nostro caso, *assioma*):

$$\frac{}{A \vdash A}\ \mathsf{ax}$$

dove A è una formula qualsiasi del linguaggio scelto, sia esso proposizionale, del primo ordine o di qualsiasi altro genere. La riga orizzontale posizionata sopra il sequente è detta *linea di derivazione*; essa serve appunto ad indicare che, in base alla regola annotata sulla destra della riga stessa, è possibile derivare il sequente in basso a partire dai sequenti che sono al di sopra della riga. In questo caso al di sopra non c'è nulla; la regola dell'assioma dice infatti che, a partire dal vuoto, è possibile asserire la validità del sequente $A \vdash A$, qualunque sia A. In altre parole, l'"atto di fede" che il calcolo dei sequenti ci propone di fare è di credere che *esiste sempre una dimostrazione di A a partire da A stessa*, vale a dire che ci dobbiamo convincere senza ulteriori spiegazioni del fatto che "se A è vera, allora A è vera". Tutto sommato non è un grande sforzo...

La seconda regola del calcolo dei sequenti che introdurremo, che costituisce assieme all'assioma il gruppo delle *regole dell'identità*, è la cosiddetta *regola del taglio*, detta più semplicemente *cut*:

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma', A \vdash \Delta'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\,\mathsf{cut}$$

La regola del taglio permette di comporre due dimostrazioni in qualche modo "complementari", nel senso che una ha A come conclusione e l'altra come premessa, in modo da costruire una nuova dimostrazione in cui A non compare più. La funzione di A dunque è quella di fare da "ponte" tra due dimostrazioni; il cut quindi non è null'altro che una formalizzazione dell'uso abituale dei *lemmi* nel corso delle dimostrazioni: se non so dimostrare direttamente B, posso cercare di dimostrare che da A segue B e dimostrare poi A, cosa che magari è più semplice da fare. Questa versione particolare della regola del taglio esprime anche il principio logico universalmente noto come *modus ponens*:

$$\frac{\vdash A \qquad A \vdash B}{\vdash B}\,\mathsf{cut}$$

Il *modus ponens* è considerato da molti come una delle basi di tutti processi cognitivi e deduttivi umani, e sembrerebbe dunque una regola assolutamente fondamentale del calcolo dei sequenti. Tuttavia, la vita è piena di sorprese, e nella prossima sezione vedremo come quest'idea apparentemente valida possa essere contraddetta in modo radicale.

Un'osservazione: se proviamo a controllare, troviamo che il sequente a conclusione della regola del taglio è ancora valido qualora lo siano quelli di partenza, ovvero continua a rispettare il suo significato intuitivo qualora gli altri lo rispettino: se tutte le premesse del sequente sono vere, allora almeno una sua conclusione è vera. Infatti, supponendo che i due sequenti di partenza siano "buoni", Γ e Γ' contengono tutte formule vere, e anche A è vera perché compare a sinistra; dunque Δ potrebbe non contenere alcuna formula vera, ma Δ' ne contiene senz'altro una. Allora la conclusione della regola ha tutte formule vere a sinistra e almeno una formula vera a destra, e quindi le cose funzionano. Questo fatto non è una coincidenza, ma una proprietà fondamentale del calcolo dei sequenti, detta *proprietà di correttezza*, o di *validità*. La morale della proprietà di correttezza è che *il calcolo dei sequenti preserva la verità delle formule dall'alto verso il basso*. Dal basso verso l'alto vale la proprietà duale: ciò che si preserva è la falsità. Noi verificheremo esplicitamente la correttezza solo nel caso già esaminato del cut, lasciando al lettore curioso la facoltà di constatare che tutte le altre regole che introdurremo sono anch'esse corrette.

Presentiamo ora il gruppo delle *regole logiche*, cioè le regole che introducono costanti logiche, connettivi logici e quantificatori. Tali regole saranno sempre a coppie: una regola per introdurre il connettivo a sinistra e una regola per introdurre il connettivo a destra del sequente. Inoltre, le regole per le costanti logiche e quelle per i connettivi $\wedge$ e $\vee$ avranno due versioni, che chiameremo formulazione *additiva* e formulazione *moltiplicativa*. Tale nomenclatura non esisteva nella definizione originale di Gentzen; è un utile anacronismo introdotto alla luce della scoperta della Logica Lineare (avvenuta a più di cinquant'anni di distanza), della quale parleremo nel capitolo 6.

Introduciamo anzitutto le regole per la negazione e l'implicazione, che hanno un'unica formulazione[5]:

$$\frac{\Gamma \vdash A, \Delta}{\Gamma, \neg A \vdash \Delta}\,\neg\mathsf{L} \qquad\qquad \frac{\Gamma, A, \vdash \Delta}{\Gamma \vdash \neg A, \Delta}\,\neg\mathsf{R}$$

$$\frac{\Gamma \vdash A, \Delta \qquad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \Rightarrow B \vdash \Delta, \Delta'}\,\Rightarrow\mathsf{L} \qquad\qquad \frac{\Gamma, A, \vdash B, \Delta}{\Gamma \vdash A \Rightarrow B, \Delta}\,\Rightarrow\mathsf{R}$$

Come si vede, la negazione consente di trattare il simbolo $\vdash$ un po' come un'uguaglianza in un'equazione: le formule possono spostarsi da un lato all'altro ma "cambiano di segno". La giustificazione di tale comportamento è chiara se si considera il significato intuitivo dei sequenti: da $\vdash A$, che significa che esiste una dimostrazione del fatto che A è sempre vera, si potrà certamente ottenere $\neg A \vdash$, il cui significato è esattamente lo stesso, cioè $\neg A$ non può mai essere vera.

Le regole dell'implicazione sono abbastanza semplici da comprendere. In particolare, l'introduzione dell'implicazione a destra conferma l'impressione, che il lettore attento avrà senz'altro avuto, del fatto che il simbolo $\vdash$ è assimilabile ad un'implicazione logica: a livello intuitivo $A \vdash B$ è molto simile, anche se completamente diverso a livello formale, a $A \Rightarrow B$.

Passiamo ora alla formulazione additiva delle regole per le costanti e per gli altri connettivi logici:

$$\text{(nessuna regola sinistra)} \qquad\qquad \frac{}{\Gamma \vdash \mathcal{T}, \Delta}\,\mathcal{T}\mathsf{aR}$$

$$\frac{\Gamma, A, \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,\wedge\mathsf{aL1} \qquad \frac{\Gamma, B, \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,\wedge\mathsf{aL2} \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma \vdash B, \Delta}{\Gamma \vdash A \wedge B, \Delta}\,\wedge\mathsf{aR}$$

$$\frac{}{\Gamma, \mathcal{F} \vdash \Delta}\,\mathcal{F}\mathsf{aL} \qquad\qquad \text{(nessuna regola destra)}$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma, B \vdash \Delta}{\Gamma, A \vee B \vdash \Delta}\,\vee\mathsf{aL} \qquad \frac{\Gamma \vdash A, \Delta}{\Gamma \vdash A \vee B, \Delta}\,\vee\mathsf{aR1} \qquad \frac{\Gamma \vdash B, \Delta}{\Gamma \vdash A \vee B, \Delta}\,\vee\mathsf{aR2}$$

[5] In realtà anche le regole dell'implicazione avrebbero due versioni, e quella che noi presentiamo sarebbe classificata come moltiplicativa. Si dà il caso però che l'implicazione additiva sia una sorta di mostriciattolo, il cui utilizzo è più che raro; di conseguenza, per non complicare troppo le cose, tralasceremo questo particolare.

E' evidente la profonda simmetria che c'è tra ciascuna regola e la sua duale; questa simmetria è una delle caratteristiche più eleganti del calcolo dei sequenti, e sarà presente anche nella formulazione moltiplicativa che introdurremo fra breve.

Sul gruppo di regole additive ci sono un paio di osservazioni da fare. Le liste Γ e Δ che compaiono nelle regole $\wedge\mathsf{aR}$ e $\vee\mathsf{aL}$, che in generale (non solo nel caso additivo) chiameremo il *contesto* della formula principale, sono esattamente le stesse sia nella premessa destra che nella premessa sinistra; i contesti dunque vengono per così dire "unificati" dalle due regole. Inoltre, nelle regole sinistre per la $\wedge$ e destre per la $\vee$, viene aggiunta una formula arbitraria, senza alcun legame con l'altra formula o con il contesto. Queste caratteristiche sono alla base della distinzione con la formulazione moltiplicativa, che è invece la seguente:

$$\frac{\Gamma \vdash \Delta}{\Gamma, \mathcal{T} \vdash \Delta}\,\mathcal{T}\mathsf{mL} \qquad \frac{}{\vdash \mathcal{T}}\,\mathcal{T}\mathsf{mR}$$

$$\frac{\Gamma, A, B \vdash \Delta}{\Gamma, A \wedge B \vdash \Delta}\,\wedge\mathsf{mL} \qquad \frac{\Gamma \vdash A, \Delta \qquad \Gamma' \vdash B, \Delta'}{\Gamma, \Gamma', \vdash A \wedge B, \Delta, \Delta'}\,\wedge\mathsf{mR}$$

$$\frac{}{\mathcal{F} \vdash}\,\mathcal{F}\mathsf{mL} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \mathcal{F}, \Delta}\,\mathcal{F}\mathsf{mR}$$

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma', B \vdash \Delta'}{\Gamma, \Gamma', A \vee B \vdash \Delta, \Delta'}\,\vee\mathsf{mL} \qquad \frac{\Gamma \vdash A, B, \Delta}{\Gamma \vdash A \vee B, \Delta}\,\vee\mathsf{mR}$$

Come già accennato, la differenza tra la formulazione additiva e quella moltiplicativa è chiara: nel primo caso le formule derivate nelle premesse dipendono dallo stesso contesto, e tale contesto viene unificato; nel secondo caso i contesti possono essere differenti, e dunque vengono mantenuti separati nella conclusione. Allo stesso modo, la $\wedge$ a sinistra e la $\vee$ a destra, nel caso moltiplicativo, non aggiungono nessuna nuova formula al sequente; queste due regole in pratica formalizzano sintatticamente il significato intuitivo delle virgole che separano le formule nei sequenti: congiunzioni a sinistra e disgiunzioni a destra.

Con le regole che abbiamo fin qui a disposizione possiamo cominciare a costruire alcune brevi ma interessanti derivazioni, le quali mostrano come la regola dell'assioma racchiuda tre principi fondamentali della logica. Eccone intanto una:

$$\frac{\dfrac{}{A \vdash A}\,\mathsf{ax}}{\vdash A \Rightarrow A}\,\Rightarrow\mathsf{L}$$

Questa derivazione spiega l'altro nome che abbiamo dato all'assioma; essa infatti consente di dire che vale il cosiddetto *principio dell'identità*, che si riassume appunto nella formula $A \Rightarrow A$: dalla verità di A segue la verità di A stessa. Queste altre due, perfettamente simmetriche, mostrano invece come l'assioma racchiuda in sé anche, rispettivamente, il *principio del terzo escluso* (*tertium non datur*) e il *principio di non contraddizione*:

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\,\mathsf{ax}}{\vdash \neg A, A}\,\neg\mathsf{R}}{\vdash \neg A \vee A}\,\vee\mathsf{mR} \qquad \dfrac{\dfrac{\dfrac{}{A \vdash A}\,\mathsf{ax}}{A, \neg A \vdash}\,\neg\mathsf{L}}{A \wedge \neg A \vdash}\,\wedge\mathsf{mL}$$

Il principio del terzo escluso significa intuitivamente che o A è vera, oppure è vera la sua negazione; i valori di verità che le formule possono assumere sono solo due, non esiste una terza possibilità. Il principio di non contraddizione asserisce invece che non è possibile che una formula e la sua negazione siano vere allo stesso tempo; supporre tale situazione porta automaticamente ad un assurdo. Questi due principi, assieme con il già citato principio d'identità, sono i fondamenti della logica classica e sono alla base della nostra concezione intuitiva del significato che una formula della logica matematica deve avere. E' dunque ragionevole che siano inclusi tutti e tre nell'unico assioma del calcolo dei sequenti.

Se ci fermassimo qui, avremmo effettivamente completato l'esposizione delle regole logiche per il calcolo dei seguenti nel caso proposizionale. Nel caso del primo ordine, occorre invece stabilire come funzioni l'introduzione dei quantificatori:

$$\dfrac{\Gamma, A[t] \vdash \Delta}{\Gamma, \forall x.A \vdash \Delta}\,\forall\mathsf{L} \qquad \dfrac{\Gamma \vdash A[x], \Delta}{\Gamma \vdash \forall x.A, \Delta}\,\forall\mathsf{R}(\star)$$

$$\dfrac{\Gamma, A[x] \vdash \Delta}{\Gamma, \exists x.A \vdash \Delta}\,\exists\mathsf{L}(\star) \qquad \dfrac{\Gamma \vdash A[t], \Delta}{\Gamma \vdash \exists x.A, \Delta}\,\exists\mathsf{R}$$

La condizione aggiuntiva $(\star)$ per le regole $\forall\mathsf{R}$ e $\exists\mathsf{L}$ è che la variabile x non abbia occorrenze libere nel contesto, cioè in nessuna formula di Γ o Δ. Il significato di tale condizione diventa chiaro osservando la regola destra per il quantificatore universale, che corrisponde ad una dimostrazione di una proprietà di validità generale. Tali dimostrazioni di solito sono caratterizzate dal fatto che, preso un oggetto qualsiasi del dominio di interesse, si mostra la validità di una certa proprietà per quell'oggetto ma *senza fare ipotesi particolari su di esso*; ciò autorizza, alla fine della dimostrazione, a dire che quello che si è mostrato valido per quell'oggetto vale anche per tutti gli altri, visto che esso potrebbe essere rimpiazzato nella dimostrazione da qualsiasi altro oggetto senza che ciò faccia la minima differenza. In termini di linguaggio logico al primo ordine, l'oggetto generico è rappresentato proprio dalla variabile x. Di conseguenza, se questa comparisse libera dentro altre formule nel sequente, significherebbe che la validità di A dipende dalla validità (o non validità) di altre formule che parlano sempre di x, e dunque tale oggetto non sarebbe più veramente *generico.*

Questo completa l'esposizione di tutte le regole logiche possibili. Nel caso di logiche di ordine superiore, le regole per i quantificatori si possono modificare in modo ovvio per introdurre $\forall$ e $\exists$ a qualsiasi ordine, basta naturalmente considerare non più simboli di variabile come x ma, ad esempio nel caso del secondo ordine, simboli di variabile proposizionale come P, e così via. La condizione $(\star)$ rimane sempre la stessa.

Per completare il calcolo, mancano però ancora tre regole, le cosiddette *regole strutturali*, che introduciamo qui di seguito:

$$\frac{\Gamma, A, B, \Gamma' \vdash \Delta}{\Gamma, B, A, \Gamma' \vdash \Delta}\,\mathsf{XL} \qquad \frac{\Gamma \vdash \Delta, A, B, \Delta'}{\Gamma \vdash \Delta, B, A, \Delta'}\,\mathsf{XR}$$

$$\frac{\Gamma, A, A \vdash \Delta}{\Gamma, A \vdash \Delta}\,\mathsf{CL} \qquad \frac{\Gamma \vdash A, A, \Delta}{\Gamma \vdash A, \Delta}\,\mathsf{CR}$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, A \vdash \Delta}\,\mathsf{WL} \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash A, \Delta}\,\mathsf{WR}$$

Le regole strutturali si chiamano così perché non hanno nulla a che vedere con la semantica dei connettivi logici e delle formule in generale; al contrario, esse agiscono sulla struttura delle dimostrazioni che i sequenti rappresentano, determinandone alcune proprietà che in logica classica sono fondamentali:

X La *regola di scambio*, o *exchange*, serve a dire che l'ordine con il quale compaiono le formule nella premessa e nella conclusione del sequente non ha assolutamente importanza. Effettivamente, scrivendo $A, B \vdash C$ non si ha la minima intenzione di dire che C può essere ottenuta solo se A è vera prima di B, poiché ciò in logica classica non ha alcun senso. Tra le altre cose, l'exchange permette di dimostrare sintatticamente la commutatività dei connettivi $\wedge$ e $\vee$.

C La *regola della contrazione*, o semplicemente *contrazione*, asserisce che la validità di un legame logico fra premessa e conclusione di un sequente non dipende dal numero di copie della stessa formula che si presentano nel sequente stesso. Ciò corrisponde a dire ad esempio che, riferendoci alla parte sinistra dei sequenti, se una formula è vera, essa lo è quante volte si vuole: se per dimostrare B ho dovuto supporre la verità di A due volte ($A, A \vdash B$), in realtà mi basta che quest'ultima formula sia vera, giacché se lo è lo sarà una volta per tutte ($A \vdash B$); in logica classica tutto ciò ha perfettamente senso.

W La *regola dell'indebolimento*, o *weakening*, esprime invece la possibilità di aggiungere ad una dimostrazione ipotesi o conclusioni arbitrarie: se ho dimostrato che C segue da A, non c'è nulla di male a dire che C segue sia da A che da B; la dimostrazione di quest'ultima asserzione si ottiene dall'altra semplicemente ignorando l'ipotesi aggiuntiva. Di nuovo, nell'ambito della logica classica questo ragionamento è perfettamente ammissibile.

Come abbiamo detto, l'exchange serve a dimostrare la commutatività dei connettivi logici, ma è in realtà indispensabile per poter manipolare in modo flessibile i sequenti. Le regole della contrazione e del weakening giocano invece un ruolo fondamentale rispetto alla struttura delle altre regole del calcolo dei sequenti. Grazie a tali regole, la distinzione tra formulazione additiva e formulazione moltiplicativa diventa solo una questione di nomenclatura. Utilizzando le regole strutturali si può infatti mostrare come le due formulazioni siano perfettamente equivalenti: si può scegliere di buttarne via una delle due, o di utilizzarle entrambe a piacimento; ai fini della derivabilità nel calcolo dei sequenti classico non ha alcuna importanza. Vedremo che questa non è l'unica diavoleria dovuta alle regole strutturali; in particolare, il weakening è il principale responsabile della non-costruttività della logica classica (vedi 3.1), mentre la contrazione è la causa dell'esplosione incontrollabile della complessità dei sistemi logici (vedi 5.3). Per una discussione più approfondita sulle regole strutturali, e su come la loro "ristrutturazione" porta alla definizione di una nuova logica (la logica lineare appunto), rimandiamo al capitolo 6.

Il calcolo dei sequenti al primo ordine che abbiamo così introdotto prende il nome di sistema **LK**. Con esso, abbiamo finalmente a disposizione un calcolo sintattico che ci permette di *dimostrare* enunciati logici. In particolare, per creare definitivamente il ponte tra sintassi e semantica, diamo la seguente definizione:

Definizione 1.13 *Sia $\mathcal{L}$ un linguaggio al primo ordine, $C_1, \ldots, C_m$ e $D_1, \ldots, D_m$ formule chiuse su tale linguaggio, e $\mathcal{M}$ una realizzazione per $\mathcal{L}$. Diremo che $\mathcal{M}$ è modello del sequente $C_1, \ldots, C_m \vdash D_1, \ldots, D_n$, e scriveremo*

$$\mathcal{M} \models (C_1, \ldots, C_m \vdash D_1, \ldots, D_n)$$

se e soltanto se

$$\mathcal{M} \models (C_1 \wedge \ldots \wedge C_m) \Rightarrow (D_1 \vee \ldots \vee D_n)$$

In questo modo abbiamo formalizzato la semantica intuitiva che avevamo fin qui attribuito ai sequenti.

Sulla base della definizione appena introdotta, è possibile dimostrare il seguente teorema:

Teorema 1.2 (Correttezza di LK) *Se* $\Gamma \vdash \Delta$ *è derivabile in* **LK**, *allora*

$$\models (\Gamma \vdash \Delta)$$

Dimostrazione. La proprietà di correttezza delle regole del calcolo dei sequenti garantisce la validità del teorema. La dimostrazione è una semplice induzione strutturale sulla lunghezza delle derivazioni, che omettiamo. □

Il teorema di correttezza dunque asserisce la validità del passaggio dalla sintassi alla semantica: un risultato trovato a livello sintattico ha senz'altro un corrispondente a livello semantico. Ciò, pur essendo confortante, non è per nulla sorprendente. Molto meno banale è invece il seguente risultato, che garantisce la validità del passaggio inverso:

Teorema 1.3 (Completezza di LK, Gödel) *Se* $\models (\Gamma \vdash \Delta)$ *allora* $\Gamma \vdash \Delta$ *è derivabile in* **LK**.

Dimostrazione. Omessa; si veda, ad esempio, [7]. □

La versione (equivalente) che normalmente si dà per il teorema di completezza di Gödel, che chiarisce meglio l'importanza del risultato, è la seguente: *data una formula* A*, se essa è vera, allora è derivabile; altrimenti, esiste un modello di* $\neg A$.

1.2.1 Il calcolo dei sequenti "one-sided"

Il lettore attento avrà certamente notato che, sfruttando la regola per la negazione a destra di **LK**, si può dimostrare banalmente la validità della seguente proposizione:

Proposizione 1.1 *Se in* **LK** *è derivabile il sequente* $\Gamma \vdash \Delta$*, allora è anche derivabile il sequente* $\vdash \neg\Gamma, \Delta$*, dove* $\neg\Gamma$ *indica la sequenza di formule contenente tutte le formule di* Γ *negate.*

A partire da questa osservazione, diventa chiara la possibilità di fornire una versione cosiddetta "one-sided" di **LK**, in cui viene utilizzato solamente il lato destro dei sequenti. La comodità di una rappresentazione del genere è evidente: il numero di regole necessarie a descrivere il calcolo praticamente si dimezza. Inoltre, si può rimuovere dalla sintassi il connettivo $\Rightarrow$, ponendo la formula $A \Rightarrow B$ come semplice abbreviazione della formula $\neg A \vee B$.

Riportiamo dunque qui sotto la versione "one-sided" di **LK**, che sarà utile nel seguito (capitolo 6) quando introdurremo la logica lineare:

- **Regole dell'identità**

$$\frac{}{\vdash \neg A, A}\,\mathsf{ax} \qquad\qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, \neg A}{\vdash \Gamma, \Delta}\,\mathsf{cut}$$

▶ **Regole strutturali**

$$\frac{\vdash \Gamma, A, B, \Delta}{\vdash \Gamma, B, A, \Delta}\ \mathsf{X} \qquad \frac{\vdash \Gamma, A, A}{\vdash \Gamma, A}\ \mathsf{C} \qquad \frac{\vdash \Gamma}{\vdash \Gamma, A}\ \mathsf{W}$$

▶ **Regole logiche additive**

$$\frac{}{\vdash \Gamma, \mathcal{T}}\ \mathcal{T}\mathsf{a} \qquad \frac{\vdash \Gamma, A \qquad \vdash \Gamma, B}{\vdash \Gamma, A \wedge B}\ \wedge\mathsf{a}$$

(nessuna regola per $\mathcal{F}$) $$\frac{\vdash \Gamma, A}{\vdash \Gamma, A \vee B}\ \vee\mathsf{a1} \qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \vee B}\ \vee\mathsf{a2}$$

▶ **Regole logiche moltiplicative**

$$\frac{}{\vdash \mathcal{T}}\ \mathcal{T}\mathsf{m} \qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \wedge B}\ \wedge\mathsf{m}$$

$$\frac{\vdash \Gamma}{\vdash \Gamma, \mathcal{F}}\ \mathcal{F}\mathsf{m} \qquad \frac{\vdash \Gamma, A, B}{\vdash \Gamma, A \vee B}\ \vee\mathsf{m}$$

▶ **Regole logiche per i quantificatori**

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, \forall x.A}\ \forall\ (\star) \qquad \frac{\vdash \Gamma, A[t]}{\vdash \Gamma, \exists x.A}\ \exists$$

Come al solito, la condizione $(\star)$ significa che la variabile x (al primo ordine in questo caso, ma a qualsiasi altro ordine in generale) non deve essere libera in Γ.

1.3 Il teorema di cut-elimination

I teoremi di correttezza e completezza che abbiamo enunciato nella sezione precedente stabiliscono assieme l'equivalenza tra l'approccio semantico e quello sintattico. Il calcolo dei sequenti è *completo* rispetto alla semantica: in esso si possono derivare tutte le verità logiche. Se ci si pensa un momento, la completezza di **LK** è una proprietà sorprendente: essa stabilisce che l'insieme delle formule vere in tutte i modelli corrisponde esattamente all'insieme delle formule derivabili nel sistema. Il fatto davvero singolare è che i due insiemi di cui si stabilisce l'equivalenza sono definiti in modo completamente diverso; uno parla di formule come di oggetti che assumono un certo valore in un certo ambito semantico, l'altro considera le formule come puri oggetti sintattici. Il calcolo dei sequenti infatti è in sé privo di qualsiasi riferimento al significato dei simboli che esso manipola. In virtù di ciò è stato possibile definire alcune varianti più "meccanizzabili" di **LK** le quali costituiscono la base della Ricerca Automatica di Dimostrazioni[6], un campo piuttosto vasto con applicazioni importanti anche (e soprattutto) all'Intelligenza Artificiale.

Il risultato che ci accingiamo ad introdurre in questa sezione non è meno sorprendente. Sempre riferendoci all'Intelligenza Artificiale, abbiamo già analizzato in precedenza la regola del taglio di **LK**, concludendo che essa dovrebbe essere essenziale per il buon funzionamento del sistema. Nel *cut* infatti è codificato un principio, quello del *modus ponens*, che è considerato basilare per i processi deduttivi umani; sembrerebbe dunque che tale regola sia indispensabile a qualunque sistema puramente sintattico il quale voglia imitare il nostro modo di ragionare.

I condizionali utilizzati nel paragrafo precedente sono ben motivati; infatti, in contrasto con le intuizioni ivi discusse, vale per **LK** il seguente teorema:

Teorema 1.4 (Hauptsatz, Gentzen) *Se il sequente $\Gamma \vdash \Delta$ è derivabile in **LK**, allora esiste una derivazione del medesimo sequente nella quale non vengono utilizzate regole del taglio.*

L'*Hauptsatz*[7] asserisce che, nel sistema **LK**, la regola del taglio è *superflua*: qualunque cosa si voglia dimostrare, se ne può benissimo fare a meno. Quella che è forse l'unica regola "intelligente" del calcolo dei sequenti si rivela essere, ai fini della derivabilità di formule, completamente inutile. Dietro a questo fatto scandaloso c'è in realtà, come vedremo nel capitolo 3, qualcosa di molto profondo, che è il fondamento stesso del legame tra logica e informatica.

Il punto cruciale della dimostrazione di Gentzen, datata 1934, è che essa è costruttiva; l'Hauptsatz si può cioè riformulare nel seguente modo:

Teorema 1.5 *Se π è una derivazione del sequente $\Gamma \vdash \Delta$ in **LK**, esiste una procedura effettiva che consente di trasformare π in una derivazione π' del medesimo sequente, tale che π' non fa uso di regole del taglio.*

Per questo motivo, l'Hauptsatz viene anche detta *Teorema di Cut-Elimination*, in quanto fornisce appunto una procedura per l'eliminazione dei tagli da una derivazione.

Vediamo allora com'è fatta questa procedura, quali sono cioè le trasformazioni elementari che consentono di partire da una derivazione che fa uso

[6]La meccanizzazione delle dimostrazioni, di cui il celeberrimo Teorema dei Quattro Colori è un illustre esempio, è il vero miracolo della completezza: un computer, l'essere più demente mai creato a questo mondo, diventa in grado di dimostrare teoremi che nessun essere umano è attualmente in grado di verificare...

[7]Nella grammatica tedesca, l'*hauptsatz* è la "frase principale" di un periodo.

di tagli a una derivazione cosiddetta *cut-free*, cioè senza regole del taglio. Nel seguito presenteremo in realtà solamente alcune regole di eliminazione, in modo da fornire l'idea generale di come funzionino le cose; l'analisi completa di tutte le regole sarà fatta più avanti (capitolo 3) per $\mathbf{LJ}_2$, il calcolo dei sequenti per la Logica Intuizionista al secondo ordine.

Facciamo anzitutto un'osservazione a carattere generale. Gli oggetti dei quali ci stiamo per occupare sono sotto-derivazioni (dunque alberi) di questo tipo:

$$\dfrac{\dfrac{\dots \quad \begin{array}{c}\vdots\,\pi\\ \Gamma_i \vdash \Delta_i\end{array} \quad \dots}{\Gamma \vdash A, \Delta}\,\mathsf{R} \qquad \dfrac{\dots \quad \begin{array}{c}\vdots\,\pi'\\ \Gamma'_j \vdash \Delta'_j\end{array} \quad \dots}{\Gamma', A \vdash \Delta'}\,\mathsf{R}'}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\,\mathsf{cut}$$

dove π e π' sono due derivazioni, R e R' due regole qualunque di $\mathbf{LK}$ e $0 \leq i, j \leq 2$. Le regole che stiamo per introdurre, che chiameremo *regole di riduzione*, consisteranno in riscritture dell'albero generico in un altro albero, il quale o non contiene più cut, oppure contiene altri cut *più semplici* di quello di partenza; il concetto di "semplicità" di un cut può naturalmente essere formalizzato, ma ciò non ci interessa in quest'introduzione iniziale. Basti sapere che la semplicità di un cut è legata alla semplicità della formula tagliata, la quale intuitivamente tiene conto della presenza di connettivi logici all'interno della formula stessa: più connettivi ci saranno in una formula, più alta sarà la sua complessità, e dunque più alta sarà la complessità di un eventuale cut su quella formula. Le regole di riduzione dunque creeranno nuovi cut che, in sostanza, agiranno sulle sotto-formule della formula tagliata, le quali sono necessariamente più semplici.

Ecco dunque la prima regola di riduzione, che coinvolge la regola dell'assioma:

$$\dfrac{\begin{array}{c}\vdots\,\pi\\ \Gamma \vdash A, \Delta\end{array} \qquad \dfrac{}{A \vdash A}\,\mathsf{ax}}{\Gamma \vdash A, \Delta}\,\mathsf{cut} \qquad \longrightarrow \qquad \begin{array}{c}\vdots\,\pi\\ \Gamma \vdash A, \Delta\end{array}$$

Sebbene assolutamente banale, questa regola di riduzione è il pilastro di tutta la cut-elimination. Grazie ad essa infatti i tagli spariscono definitivamente, generando derivazioni cut-free. Tutto il processo di eliminazione del taglio può essere in realtà visto come una "migrazione" dei cut verso gli assiomi; quando un cut raggiunge un assioma, si applica la regola di riduzione appena introdotta ed esso viene finalmente eliminato.

La prossima regola mostra come comportarsi nel caso in cui la formula tagliata contenga come connettivo principale $\wedge$, e tale connettivo sia stato introdotto mediante regole additive:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma\vdash A,\Delta\end{array}\quad \begin{array}{c}\vdots\,\pi_2\\ \Gamma\vdash B,\Delta\end{array}}{\Gamma\vdash A\wedge B,\Delta}\,\wedge\mathsf{aR} \qquad \dfrac{\begin{array}{c}\vdots\,\pi_3\\ \Gamma',A\vdash\Delta'\end{array}}{\Gamma',A\wedge B\vdash\Delta'}\,\wedge\mathsf{aL1}}{\Gamma,\Gamma'\vdash\Delta,\Delta'}\,\mathsf{cut} \quad\longrightarrow\quad \dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma\vdash A,\Delta\end{array}\quad \begin{array}{c}\vdots\,\pi_3\\ \Gamma',A\vdash\Delta'\end{array}}{\Gamma,\Gamma'\vdash\Delta,\Delta'}\,\mathsf{cut}$$

Si noti come la nuova derivazione non contenga più alcuna traccia della sotto-derivazione π_2; nel caso in cui la regola nel sotto-albero destro fosse stata una $\wedge\mathsf{aL2}$, sarebbe stata π_1 ad essere "scartata". Per contro, la seguente è la regola di riduzione per la situazione analoga, ma nel caso in cui le regole che hanno introdotto $A \wedge B$ siano moltiplicative:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma\vdash A,\Delta\end{array}\quad \begin{array}{c}\vdots\,\pi_2\\ \Gamma'\vdash B,\Delta'\end{array}}{\Gamma,\Gamma'\vdash A\wedge B,\Delta,\Delta'}\,\wedge\mathsf{mR} \qquad \dfrac{\begin{array}{c}\vdots\,\pi_3\\ \Gamma'',A,B\vdash\Delta''\end{array}}{\Gamma'',A\wedge B\vdash\Delta''}\,\wedge\mathsf{mL}}{\Gamma,\Gamma',\Gamma''\vdash\Delta,\Delta',\Delta''}\,\mathsf{cut} \quad\longrightarrow\quad \dfrac{\begin{array}{c}\vdots\,\pi_2\\ \Gamma'\vdash B,\Delta'\end{array}\quad \dfrac{\begin{array}{c}\vdots\,\pi_1\\ \Gamma\vdash A,\Delta\end{array}\quad \begin{array}{c}\vdots\,\pi_3\\ \Gamma'',A,B\vdash\Delta''\end{array}}{\Gamma,\Gamma'',B\vdash\Delta,\Delta''}\,\mathsf{cut}}{\Gamma,\Gamma',\Gamma''\vdash\Delta,\Delta',\Delta''}\,\mathsf{cut}$$

Questa regola di riduzione crea due nuovi cut, utilizzando tutte le sotto-derivazioni presenti nella derivazione di partenza.

Un'altra regola di riduzione interessante è quella per i tagli su formule il cui connettivo principale sia un quantificatore universale:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi\\ \Gamma\vdash A[x],\Delta\end{array}}{\Gamma\vdash\forall x.A,\Delta}\,\forall\mathsf{R} \qquad \dfrac{\begin{array}{c}\vdots\,\pi'\\ \Gamma',A[t]\vdash\Delta'\end{array}}{\Gamma',\forall x.A\vdash\Delta'}\,\forall\mathsf{L}}{\Gamma,\Gamma'\vdash\Delta,\Delta'}\,\mathsf{cut} \quad\longrightarrow\quad \dfrac{\begin{array}{c}\vdots\,\pi[t/x]\\ \Gamma\vdash A[t/x],\Delta\end{array}\quad \begin{array}{c}\vdots\,\pi'\\ \Gamma',A[t]\vdash\Delta'\end{array}}{\Gamma,\Gamma'\vdash\Delta,\Delta'}\,\mathsf{cut}$$

In questa regola si effettua, nell'albero di π, una sostituzione generale di tutte le occorrenze libere della variabile x con il termine t. Ciò è possibile perché la regola $\forall\mathsf{R}$ nel sotto-albero destro soddisfa necessariamente la condizione $(\star)$, e dunque le sole occorrenze libere di x in π riguardano la formula A e le sue sottoformule; tutto il resto sarà lasciato inalterato.

In modo analogo si possono definire le regole per tutte le coppie di regole che introducono i vari connettivi; in particolare, la gestione di un taglio su una formula introdotta da regole la cui formulazione non è omogenea (ad esempio additiva per l'introduzione a sinistra e moltiplicativa per l'introduzione a destra) può essere effettuata senza problemi grazie all'intervento delle regole strutturali. Per quanto riguarda queste ultime, proponiamo qui di seguito le regole di riduzione che determinano la riscrittura della derivazione nel caso in cui una delle due occorrenze della formula tagliata provenga proprio da un weakening o da una contrazione:

$$\dfrac{\dfrac{\vdots\ \pi}{\Gamma \vdash A, \Delta} \quad \dfrac{\dfrac{\vdots\ \pi'}{\Gamma' \vdash \Delta'}}{\Gamma', A \vdash \Delta'}\ \mathsf{WL}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ \mathsf{cut} \quad \longrightarrow \quad \dfrac{\dfrac{\vdots\ \pi'}{\Gamma' \vdash \Delta'}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ \mathsf{W}$$

$$\dfrac{\dfrac{\vdots\ \pi}{\Gamma \vdash A, \Delta} \quad \dfrac{\dfrac{\vdots\ \pi'}{\Gamma', A, A \vdash \Delta'}}{\Gamma', A \vdash \Delta'}\ \mathsf{CL}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ \mathsf{cut} \quad \longrightarrow \quad \dfrac{\dfrac{\dfrac{\vdots\ \pi}{\Gamma \vdash A, \Delta} \quad \dfrac{\dfrac{\vdots\ \pi}{\Gamma \vdash A, \Delta} \quad \dfrac{\vdots\ \pi'}{\Gamma', A, A \vdash \Delta'}}{\Gamma, \Gamma', A \vdash \Delta, \Delta'}\ \mathsf{cut}}{\Gamma, \Gamma, \Gamma' \vdash \Delta, \Delta, \Delta'}\ \mathsf{cut}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\ \mathsf{C}$$

La linea di derivazione più spessa indica, nel caso della regola per il weakening, l'applicazione di più regole WL e WR, mentre nel caso della contrazione l'applicazione di più regole CL e CR. La peculiarità dei passi di riduzione che coinvolgono regole strutturali è chiara: nel caso del weakening viene cancellato un intero ramo della derivazione; nel caso della contrazione invece un ramo viene duplicato. In quest'ultima situazione c'è da osservare che i due nuovi cut sono ancora sulla medesima formula A, e dunque la loro complessità non è diminuita. Tuttavia, intuitivamente, prima o poi la trasformazione creerà un cut la cui formula tagliata è finalmente introdotta da una regola logica, e allora ci si ritroverà in uno dei casi già discussi. Formalmente la cosa è alquanto delicata, e lo diviene ancora di più in considerazione del fatto che in realtà weakening e contrazioni potrebbero introdurre la formula in entrambe i rami della derivazione, ma alla fine tutto è risolvibile senza grandissimi problemi concettuali.

Resta ancora un problema; tutti i casi analizzati fin'ora (ad eccezione di quelli che coinvolgono regole strutturali), considerano la formula tagliata come *appena introdotta* sia nel ramo destro che in quello sinistro. Ciò potrebbe non essere affatto vero; il cut potrebbe operare su una formula che non è minimamente coinvolta nella regola precedente. Il seguente è un possibile esempio di una situazione del genere:

$$\dfrac{\dfrac{\dfrac{\vdots\ \pi_A}{\Gamma \vdash A, \Delta} \quad \dfrac{\vdots\ \pi_B}{\Gamma \vdash B, \Delta}}{\Gamma \vdash A \wedge B, \Delta}\ \wedge\mathsf{aR} \qquad \dfrac{\dfrac{\dfrac{\vdots\ \pi'}{\Gamma', A \vdash C, D, \Delta'}}{\Gamma', A \wedge B \vdash C, D, \Delta'}\ \wedge\mathsf{aL1}}{\Gamma', A \wedge B \vdash C \vee D, \Delta'}\ \vee\mathsf{mR}}{\Gamma, \Gamma' \vdash C \vee D, \Delta, \Delta'}\ \mathsf{cut}$$

Come si vede, il cut opera su $A \wedge B$ ma, mentre quest'ultima nel ramo sinistro proviene direttamente da una regola $\wedge\mathsf{aR}$, nel ramo destro l'ultima regola è una $\vee$mR, che non coinvolge la formula tagliata. Nell'eventualità di casi come questo, si può dimostrare che il cut commuta con qualsiasi altra regola del calcolo dei sequenti, a patto naturalmente che la formula tagliata non sia coinvolta nella regola in questione. Si possono dunque fornire una serie di *regole di commutazione*, grazie alle quali i cut possono "salire" lungo l'albero della derivazione, verso le foglie. Nel caso in questione, la trasformazione sarà la seguente:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\,\pi_A\\ \Gamma \vdash A, \Delta\end{array} \quad \begin{array}{c}\vdots\,\pi_B\\ \Gamma \vdash B, \Delta\end{array}}{\Gamma \vdash A \wedge B, \Delta}\wedge\mathsf{aR} \qquad \dfrac{\begin{array}{c}\vdots\,\pi'\\ \Gamma', A \vdash C, D, \Delta'\end{array}}{\Gamma', A \wedge B \vdash C, D, \Delta'}\wedge\mathsf{aL1}}{\Gamma, \Gamma' \vdash C, D, \Delta, \Delta'}\mathsf{cut}}{\Gamma, \Gamma' \vdash C \vee D, \Delta, \Delta'}\vee\mathsf{mR}$$

Seguendo la "traccia" fornita dagli esempi che abbiamo introdotto fin qui, si può dimostrare l'Hauptsatz in modo formale, a partire dalle seguenti definizioni preliminari:

Definizione 1.14 *Definiamo il* grado *di una formula A, che indichiamo con $d(A)$, nel seguente modo:*

- *Se A è atomica, $d(A) = 1$*
- $d(A \wedge B) = d(A \vee B) = d(A \Rightarrow B) = max(d(A), d(B)) + 1$
- $d(\neg A) = d(\forall x.A) = d(\exists x.A) = d(A) + 1$. *In particolare $d(A[t/x]) = d(A)$ per ogni termine t.*

Il grado di un cut è semplicemente il grado della formula tagliata.

Definizione 1.15 *Il* rango *di una certa occorrenza di una formula nell'albero di derivazione è il numero di regole che si trovano al di sopra di essa; ad esempio, il rango di entrambe le occorrenze di A nel sequente $A \vdash A$ ottenuto mediante la regola dell'assioma è pari a 1, e tutte le altre regole aumentano di un'unità il rango delle occorrenze coinvolte.*
Il rango di un cut è definito come il massimo tra i ranghi delle due occorrenze della formula tagliata, più 1.

Definizione 1.16 *Se σ è una derivazione la cui ultima regola è un cut, del tipo*

$$\dfrac{\begin{array}{c}\vdots\,\pi\\ \Gamma \vdash A, \Delta\end{array} \qquad \begin{array}{c}\vdots\,\pi'\\ \Gamma', A \vdash \Delta'\end{array}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\mathsf{cut}$$

tale che π e π' sono cut-free, allora diremo che σ è quasi-cut-free.

Si noti come il *grado* formalizzi in qualche modo il concetto di "complessità" di un cut di cui si è parlato in precedenza. La proprietà fondamentale delle regole di riduzione, di cui abbiamo visto qualche esempio, è la seguente: le regole di riduzione logiche eliminano un cut di grado d in modo completo, oppure creano uno o più cut di grado $d-1$; le riduzioni strutturali e logiche o eliminano un cut (è il caso del weakening) oppure, cosa più importante, rimpiazzano un cut di rango ρ con almeno un cut di rango $\rho-1$. C'è dunque almeno uno dei due parametri che diminuisce sempre sotto l'applicazione di qualsiasi regola; questo fatto sarà la chiave della dimostrazione dell'Hauptsatz. Infatti, il primo passo verso l'eliminazione del taglio sarà il seguente lemma:

Lemma 1.1 *Se σ è una derivazione quasi-cut-free, è possibile trasformarla in una derivazione σ' del medesimo sequente tale che σ' è cut-free.*

Dimostrazione. Procederemo per doppia induzione sul grado d e sul rango ρ del cut che conclude la derivazione σ. Con π e π' chiameremo le due sotto-derivazioni le cui conclusioni costituiscono rispettivamente la premessa sinistra e destra del cut, esattamente come nella definizione 1.16.

i. $d = 1$. Procediamo per sotto-induzione sul rango:

   (a) Il rango è minimo, cioè $\rho = 2$. Le due derivazioni π e π' non possono essere che assiomi, e dunque la forma normale si raggiunge applicando una semplice regola di riduzione.

   (b) Il rango ρ è generico. Poichè la formula tagliata è atomica, entrambe le occorrenze non possono che essere state introdotte da due assiomi. Tra tali assiomi e il cut ci sono dunque un certo numero di regole (al più $\rho - 1$) che non operano sulla formula in questione. In particolare, anche le ultime regole prima del cut sono di questo tipo; effettuando uno o due passi di riduzione commutativi otteniamo una derivazione che contiene una o due sotto-derivazioni quasi-cut-free in cui il rango del cut conclusivo è minore di ρ. Per ipotesi di sotto-induzione, tali derivazioni sono normalizzabili.

ii. Il grado d è generico. Anche qui effettuiamo una sotto-induzione sul rango:

   (a) Il rango è minimo, vale a dire $\rho = 2$. Anche in questo caso, pur non essendo atomica, la formula è introdotta direttamente per mezzo di due assiomi; basta un passo di riduzione per normalizzare la derivazione.

   (b) Il rango ρ è generico. Se una tra π e π' non si conclude con una regola logica che introduce la formula tagliata, mediante le riduzioni commutative otteniamo una derivazione che contiene una sotto-derivazione quasi-cut-free in cui il rango del cut conclusivo è minore di ρ; per ipotesi di sotto-induzione, tale derivazione è normalizzabile. Se, al contrario, π e π' si concludono con una regola che introduce la formula tagliata, applichiamo il passo di riduzione appropriato e otteniamo una derivazione che contiene almeno una sotto-derivazione quasi-cut-free in cui il grado dell'unico taglio è minore di d; per ipotesi d'induzione, abbiamo la tesi.

□

Il teorema 1.5 si ottiene mediante una semplice induzione sul numero di cut presenti in una derivazione; siamo così arrivati all'Hauptsatz, e alla sorprendente eliminazione del *modus ponens* dalle regole fondamentali della logica.

Il Teorema di Cut-Elimination ha, a livello puramente logico, diversi corollari di grande importanza. Uno di essi è relativo alla cosiddetta *proprietà della sotto-formula*:

Definizione 1.17 *Si dice che una derivazione π di **LK** del sequente $\Gamma \vdash \Delta$ ha la* proprietà della sotto-formula *se tutti gli assiomi che sono le foglie di π introducono esclusivamente sotto-formule delle formule di Γ e Δ.*

Esaminando le regole di **LK**, ci accorgiamo che l'unica regola che vìola la proprietà della sotto-formula è proprio il cut; poiché però sappiamo che il cut è ridondante, sappiamo anche che per qualsiasi sequente $\vdash A$ può essere trovata una derivazione per cui valga il principio della sotto-formula. In virtù del Teorema di Completezza di Gödel, ciò significa che tutte le formule vere sono dimostrabili a partire esclusivamente dalle loro sotto-formule[8].

Diretta conseguenza della proprietà della sotto-formula è la *consistenza* di **LK**, che può essere formalizzata nel seguente teorema:

Teorema 1.6 *Il sequente vuoto $\vdash$ non è derivabile in **LK**.*

[8]Una conseguenza strabiliante di ciò è che, ad esempio, tutti i teoremi della Teoria Analitica dei Numeri ammettono versioni "puramente aritmetiche", in cui non viene utilizzato alcuno strumento dell'Analisi delle Funzioni nel Campo Complesso. Tuttavia, la formalizzazione di tali dimostrazioni genererebbe derivazioni di **LK** che non partono "dal vuoto", cioè le cui foglie non sono solo regole dell'identità, ma anche sequenti del tipo $\vdash B$, dove B è un assioma della teoria nella quale viene dimostrato il risultato (in questo caso, formule che fanno parte degli Assiomi di Peano, degli Assiomi del Campo Reale, eccetera). Il Teorema di Cut-Elimination naturalmente non è estendibile alle derivazioni che non partono dal vuoto, e dunque non è applicabile a questo caso. Inoltre, come vedremo in 5.3 le versioni cut-free delle dimostrazioni possono facilmente diventare di dimensioni superiori a qualsiasi misura concepibile dell'Universo...

Dimostrazione. Per il Teorema di Cut-Elimination, se esistesse una derivazione del sequente vuoto, sarebbe possibile trovarne una cut-free, dunque per la quale varrebbe la proprietà della sotto-formula; ma poiché l'unica sotto-formula del vuoto è la formula vuota, la derivazione in questione consisterebbe del solo sequente vuoto, e dunque non sarebbe una derivazione di **LK**. □

A livello informatico, il Teorema di Cut-Elimination ha invece conseguenze travolgenti, che discuteremo nel capitolo 3.

Capitolo 2

Modelli di computazione

Dedicheremo il presente capitolo ad una breve introduzione ai modelli di computazione classici: le macchine di Turing, le funzioni ricorsive e il λ-calcolo. Esporremo così i tre sistemi fondamentali dell'Informatica Teorica ai quali faremo riferimento nel resto della trattazione.

2.1 Le macchine di Turing

Le macchine di Turing sono state introdotte dal matematico inglese Alan Turing negli anni '40. Anche se all'epoca i calcolatori automatici erano praticamente inesistenti, tale modello ideale non è molto lontano da quelli che sono i calcolatori elettronici reali sviluppatisi dalla seconda metà del secolo scorso fino ai giorni nostri. In particolare, vedremo come l'approccio seguito dalle macchine di Turing sia estremamente di basso livello, simile, in ambito reale, ad una programmazione di tipo imperativo, o addirittura direttamente a livello di linguaggio assembly.

Diamo allora la definizione della nostra macchina di Turing:

Definizione 2.1 (Macchina di Turing) *Una* macchina di Turing $\mathbf{M}$ *è una tripla*

$$\langle \Sigma \cup \{\Box\}, S \cup \{s_0, s_\infty\}, \delta \rangle$$

dove

- Σ*, che è detto l'*alfabeto *di* $\mathbf{M}$*, è un insieme finito di simboli, cui viene aggiunto il simbolo speciale* $\Box$ *("blank"); per brevità, porremo* $\overline{\Sigma} := \Sigma \cup \{\Box\}$.
- S *è un insieme finito di* stati *in cui* $\mathbf{M}$ *si può trovare, cui vengono aggiunti gli stati* s_0 *e* s_∞*, detti rispettivamente* stato iniziale *e* stato finale *di* $\mathbf{M}$*; per brevità, porremo* $\overline{S} := S \cup \{s_0, s_\infty\}$.

- *δ, che è detta la* funzione di transizione *di* **M**, *è una funzione parziale di tipo*
$$\delta : \overline{S} \times \overline{\Sigma} \rightarrow \overline{S} \times \overline{\Sigma} \times \{\leftarrow, \rightarrow\}$$
tale che nel dominio non ci sia alcuna coppia di $\overline{S} \times \overline{\Sigma}$ la cui prima componente è s_∞.

Ad **M** *è inoltre associato un* nastro*, costituito da una quantità infinita numerabile di celle, ciascuna delle quali contiene un simbolo di $\overline{\Sigma}$.* **M** *è poi dotata di una* testina*, la quale è ad ogni dato istante posizionata su esattamente una cella del nastro. Tramite la testina, la macchina può leggere il contenuto della cella e modificarlo.*

Il funzionamento di una macchina di Turing è molto semplice. Ad ogni dato istante, la nostra macchina si trova in uno stato ben preciso, supponiamo s, e la sua testina è posizionata su una ben determinata cella del nastro, la quale contiene, ad esempio, il simbolo σ. A questo punto, la macchina di Turing fornisce in input alla sua funzione di transizione δ la coppia $\langle s, \sigma \rangle$; se questa coppia non è nel dominio di δ, la macchina si ferma. Altrimenti, δ restituisce una tripla costituita da uno stato s', un simbolo σ' e un elemento di $\{\leftarrow, \rightarrow\}$; la nostra macchina allora aggiornerà il suo stato a s', modificherà il contenuto della cella su cui si trova la testina in modo che essa contenga il simbolo σ', e sposterà la testina stessa secondo la seguente regola: se il terzo elemento della tripla restituita da δ è $\leftarrow$, la testina sarà spostata sulla cella immediatamente "a sinistra" di quella corrente, mentre se tale elemento è $\rightarrow$, la testina sarà spostata sulla cella immediatamente "a destra". I termini "sinistra" e "destra" si riferiscono naturalmente ad un qualche orientamento arbitrario del nastro, fissato però una volta per tutte per ogni macchina di Turing.

A questo punto, la nostra macchina si trova nuovamente in un qualche stato, con la testina posizionata su una cella in cui sarà contenuto un qualche carattere; il procedimento può dunque essere applicato in modo identico, finchè non succede una delle seguenti due cose:

- ▶ la macchina si trova in uno stato q e la testina è posizionata su una cella contenente il carattere κ tali che $\langle q, \kappa \rangle$ non è nel dominio di δ. In questo caso, diremo che la macchina ha terminato il suo calcolo con una configurazione *non accettante*;

- ▶ la macchina arriva allo stato finale s_∞, sul quale, per definizione, si fermerà sicuramente. In tal caso, diremo che la macchina ha terminato il suo calcolo con una configurazione *accettante*.

Naturalmente, è senz'altro possibile che la nostra macchina di Turing non arrivi mai a una di queste due situazioni, e che questa continui dunque a calcolare all'infinito.

La dinamica descritta sopra può essere formalizzata nel seguente modo:

Definizione 2.2 (Configurazione) *Sia* $\mathbf{M}$ *una macchina di Turing, la cui funzione di transizione è* δ*. Una* configurazione *di* $\mathbf{M}$ *è una tripla*

$$\langle s, l, r\rangle$$

dove $s \in \overline{S}$*, e* $l = (\lambda_i)_{i\in\mathbb{N}}$ *e* $r = (\rho_i)_{i\in\mathbb{N}}$ *sono due successioni di simboli dell'alfabeto* $\overline{\Sigma}$*, tali che* λ_0 *è il simbolo contenuto nella cella immediatamente a sinistra della cella corrente,* λ_1 *il simbolo contenuto nella cella ancora più a sinistra, e così via, mentre* ρ_0 *è il simbolo contenuto nella cella corrente,* ρ_1 *quello contenuto nella cella immediatamente a destra, e così via.*
Qualunque siano l *e* r*, definiamo le seguenti tre classi di configurazioni notevoli di* $\mathbf{M}$*:*

- $\langle s_0, l, r\rangle$ *è una* configurazione iniziale
- $\langle s, l, r\rangle$*, in cui* $\langle s, \rho_0\rangle$ *non appartiene al dominio di* δ*, è una* configurazione finale non accettante
- $\langle s_\infty, l, r\rangle$ *è una* configurazione finale accettante

In genere, se $\langle s, l, r\rangle$ è una configurazione di una macchina di Turing di alfabeto Σ, considereremo l ed r non come successioni di simboli ma come stringhe (anche infinite) sull'alfabeto $\overline{\Sigma}$, con le seguenti convenzioni:

· A $(\sigma_i)_{i\in\mathbb{N}}$ associamo la stringa $\sigma_0\sigma_1\sigma_2\ldots$, e viceversa

· Se $(\sigma_i)_{i\in\mathbb{N}}$ è una successione per la quale esiste un certo $m \geq 0$ tale che, per ogni $i \geq m$, $\sigma_i = \Box$, allora a tale successione associamo la stringa finita $\sigma_0\sigma_1\ldots\sigma_{m-1}$; se $m = 0$, la stringa associata sarà ε, cioè la stringa vuota. Viceversa, alla stringa vuota associamo la successione $(\Box)_{i\in\mathbb{N}}$, mentre alla generica stringa finita, non vuota, $\alpha_0\alpha_1\ldots\alpha_{m-1}$ associamo la successione $(\sigma_i)_{i\in\mathbb{N}}$ tale che, per $i < m$, $\sigma_i = \alpha_i$, mentre per $i \geq m$, $\sigma_i = \Box$

Naturalmente, tra le stringhe finite di un alfabeto qualsiasi è definita la concatenazione, per la quale utilizzeremo la notazione usuale: se u e v sono due stringhe finite, tali che $u = a_1\ldots a_m$ e $v = b_1\ldots b_n$, la loro concatenazione uv sarà la stringa $a_1\ldots a_m b_1\ldots b_n$.

Definizione 2.3 (Transizione) *Sia* $\mathbf{M}$ *una macchina di Turing, la cui funzione di transizione è* δ*, e siano* $c = \langle s, (\lambda_i)_{i\in\mathbb{N}}, (\rho_i)_{i\in\mathbb{N}}\rangle$ *e* $c' = \langle s', l', r'\rangle$ *due configurazioni di* $\mathbf{M}$*. Diremo che* c' *è la configurazione successiva di* c *in* $\mathbf{M}$*, e scriveremo*

$$c \overset{\mathbf{M}}{\to} c'$$

se e solo se

$$\delta(s, \rho_0) = \langle s', \rho, \leftarrow \rangle$$

e

$$l' = (\lambda'_i)_{i \in \mathbb{N}}, \textit{con } \lambda'_i = \lambda_{i+1} \ \forall i \in \mathbb{N}$$
$$r' = (\rho'_i)_{i \in \mathbb{N}}, \textit{con } \rho'_0 = \lambda_0,\ \rho'_1 = \rho,\ \textit{e } \forall i \geq 2,\ \rho'_i = \rho_{i-1}.$$

oppure

$$\delta(s, \rho_0) = \langle s', \lambda, \rightarrow \rangle$$

e

$$l' = (\lambda'_i)_{i \in \mathbb{N}}, \textit{con } \lambda_0 = \lambda \ \textit{e } \forall i \geq 1,\ \lambda'_i = \lambda_{i-1}$$
$$r' = (\rho'_i)_{i \in \mathbb{N}}, \textit{con } \rho'_i = \rho_{i+1} \ \forall i \in \mathbb{N}.$$

In questo caso, diremo anche che $\mathbf{M}$ *passa da* c *a* c' *per mezzo di una* transizione elementare.
Denoteremo con $\overset{\mathbf{M}}{\twoheadrightarrow}$ *la chiusura riflessiva e transitiva di* $\overset{\mathbf{M}}{\rightarrow}$*, e scriveremo dunque*

$$c_0 \overset{\mathbf{M}}{\twoheadrightarrow} c_{n+1}$$

se esistono n *configurazioni* $c_1, \ldots, c_n$ *di* $\mathbf{M}$ *tali che*

$$c_0 \overset{\mathbf{M}}{\rightarrow} c_1 \overset{\mathbf{M}}{\rightarrow} \ldots \overset{\mathbf{M}}{\rightarrow} c_n \overset{\mathbf{M}}{\rightarrow} c_{n+1}$$

Ogni transizione elementare di una macchina di Turing è dunque un passo atomico di computazione, e ogni calcolo svolto da una macchina di Turing può essere visto come una sequenza di tali passi atomici.

Vediamo ora come possiamo utilizzare la dinamica delle macchine per effettuare veri e propri calcoli. Anzitutto, possiamo considerare il contenuto iniziale del nastro come l'input della computazione; poiché siamo in generale interessati ad effettuare calcoli su input finiti, supporremo d'ora in poi che le configurazioni iniziali siano tutte della forma

$$\langle s_0, \varepsilon, r_0 \rangle$$

dove r_0 è una stringa finita (naturalmente sull'alfabeto della macchina di Turing in questione). Diremo allora che la macchina di Turing $\mathbf{M}$ accetta la stringa r_0 se

$$\langle s_0, \varepsilon, r_0 \rangle \overset{\mathbf{M}}{\twoheadrightarrow} \langle s_\infty, l_\infty, r_\infty \rangle$$

dove l_∞ e r_∞ sono due stringhe qualunque. Al contrario, diremo che $\mathbf{M}$ rifiuta la stringa r_0 se

$$\langle s_0, \varepsilon, r_0 \rangle \overset{\mathbf{M}}{\twoheadrightarrow} c$$

dove c è una qualsiasi configurazione finale non accettante di $\mathbf{M}$. In questo modo, una macchina di Turing può *decidere* un linguaggio, cioè può essere costruita in modo tale da accettare solo le stringhe che fanno parte di un certo sottoinsieme di Σ^*, dove con Σ^* denotiamo l'insieme di tutte le stringhe finite composte da simboli di Σ. Chiaramente, possono esistere configurazioni iniziali per le quali la nostra macchina di Turing non arriva mai a fermarsi; ciò significa che alcune stringhe resteranno *indecidibili*, nel senso che la macchina non determinerà mai se una stringa di questo genere appartiene o no al linguaggio. Possiamo formalizzare quanto detto come segue:

Definizione 2.4 *Sia* $\mathbf{M}$ *una macchina di Turing di alfabeto* Σ*, e sia* $\mathcal{L} \subseteq \Sigma^*$ *un linguaggio su tale alfabeto. Diremo che* $\mathbf{M}$ decide $\mathcal{L}$ *se, per ogni* $u \in \mathcal{L}$*,* $\mathbf{M}$ *accetta* u*, mentre per ogni* $v \notin \mathcal{L}$*,* $\mathbf{M}$ *rifiuta* v*. Diremo invece che* $\mathbf{M}$ semi-decide $\mathcal{L}$ *se* $\mathbf{M}$ *accetta tutte le stringhe di* $\mathcal{L}$*, ma non termina se in input gli viene data una stringa che non appartiene a* $\mathcal{L}$*.*
Allo stesso modo, diremo che un linguaggio $\mathcal{L}$ *è* decidibile *se esiste una macchina di Turing in grado di deciderlo;* $\mathcal{L}$ *sarà invece* semi-decidibile *se esiste una macchina di Turing in grado di semi-deciderlo.*

Poiché un qualunque problema può essere ridotto ad un cosiddetto *problema di decisione* (un problema cioè per il quale la risposta è "sì" o "no"), e poiché ogni problema di decisione può essere ridotto al problema di determinare l'appartenenza o meno di una certa stringa ad un certo linguaggio, abbiamo in questo modo stabilito come una macchina di Turing possa essere utilizzata per risolvere problemi.

Tuttavia, il lettore attento avrà notato che, nella definizione di configurazione finale accettante o non accettante, è stato completamente ignorato il contenuto del nastro; ciò che importa è, una volta che la macchina si sia fermata, andare a vedere lo stato in cui si trova. Se esso è s_∞, concludiamo che la macchina ci ha dato una risposta positiva, se è diverso da s_∞, concludiamo invece che la risposta è negativa.

A tutto ciò esiste chiaramente un'alternativa: possiamo scegliere di ignorare lo stato con cui la macchina termina, e considerare esclusivamente il contenuto del nastro. In questo modo, si può programmare una macchina di Turing perché essa, piuttosto che risolvere un problema, calcoli una funzione. Ecco come fare:

Definizione 2.5 *Sia* f *una funzione parziale da* $(\Sigma^*)^n$ *a* Σ^**, cioè una funzione che, a partire da* n *stringhe sull'alfabeto* Σ*, ne restituisce un'altra; sia inoltre* $\mathbf{M}$ *una macchina di Turing di alfabeto* Σ*. Allora, diremo che* $\mathbf{M}$ calcola f *se e solo se:*

- *se* $f(u_1, \ldots, u_n)$ *è definita, allora*

$$\langle s_0, \varepsilon, u_1 \Box u_2 \Box \ldots \Box u_n \rangle \overset{\mathbf{M}}{\twoheadrightarrow} c$$

dove c è una configurazione finale (accettante o no) di $\mathbf{M}$ *tale che*

$$c = \langle s, l, f(u_1, \dots, u_n) \rangle$$

dove $s \in \overline{S}$ e l è una stringa qualsiasi

- *se $f(u_1, \dots, u_n)$ non è definita, allora la configurazione*

  $$\langle s_0, \varepsilon, u_1 \Box u_2 \Box \dots \Box u_n \rangle$$

  dà luogo ad una successione infinita di passi, cioè di transizioni elementari.

Un classe particolare di macchine di Turing di questo tipo sono quelle sull'alfabeto $\{0, 1\}$, le cui stringhe possono essere interpretate come numeri interi in notazione binaria; tali macchine di Turing assomigliano molto a programmi, che girano su un normale computer, i quali calcolano funzioni da interi a interi, utilizzando appunto la rappresentazione binaria.

Una conseguenza immediata della definizione 2.5 è che tutte le macchine di Turing che decidono problemi sono casi particolari di macchine di Turing che calcolano funzioni, in cui il risultato della funzione viene ignorato, e viene piuttosto preso in considerazione lo stato finale. Ad ogni problema è dunque ben associata una funzione, mentre ad una funzione può in generale essere impossibile associare un problema in modo "canonico".

Possiamo allora dare la nostra prima definizione di calcolabilità:

Definizione 2.6 (Turing-calcolabilità) *Sia f una funzione da $\mathbb{N}^n$ a $\mathbb{N}$. Diremo che f è* Turing-calcolabile *se esiste una macchina di Turing $\mathbf{M}_f$ che la calcola.*

La definizione data per le macchine di Turing non è l'unica possibile; in pratica, su ogni libro che tratti di Informatica Teorica se ne trova una diversa. Un'alternativa interessante che menzioniamo in questa sede è la macchina di Turing multinastro:

Definizione 2.7 (Macchina di Turing multinastro) *Una macchina di Turing a k nastri $\mathbf{M}^k$ è una tripla*

$$\langle \Sigma \cup \{\Box\}, S \cup \{s_0, s_\infty\}, \delta \rangle$$

definita esattamente come per le macchine di Turing mononastro, ad eccezione della funzione di transizione, per la quale è

$$\delta : \overline{S} \times \overline{\Sigma}^k \to \overline{S} \times (\overline{\Sigma} \times \{\leftarrow, \to\})^k$$

Ad $\mathbf{M}^k$ non è associato un solo nastro, ma k; ciascuno di essi ha una testina, in grado di leggere e modificare il contenuto della cella su cui è posizionata, e in grado di muoversi indipendentemente dalle altre testine.

Naturalmente, anche per le macchine di Turing multinastro sono definibili i concetti di configurazione, transizione e calcolabilità (che in questo caso chiameremo *k-Turing-calcolabilità*), in modo del tutto analogo a quanto fatto per le macchine di Turing mononastro. In particolare, le configurazioni non saranno più triple ma $(2k+1)$-ple, dove k è il numero di nastri della macchina.

Le macchine di Turing multinastro sono "più comode" da programmare perché forniscono evidentemente più spazio sul quale lavorare. Ad esempio, nel caso in cui si voglia calcolare una funzione n-aria, si può pensare di riservare n nastri di "sola lettura" agli input, prendere un altro nastro di "sola scrittura" per l'output, e utilizzare poi un numero arbitrario di altri nastri (sia in scrittura che in lettura) per "fare i calcoli". Sebbene ciò possa far sperare in un aumento del potere di calcolo delle macchine multinastro rispetto a quelle mononastro, vale la seguente proposizione:

Proposizione 2.1 *Una funzione da interi a interi f è k-Turing-calcolabile se e solo se è Turing-calcolabile.*

Dimostrazione. La dimostrazione si compone di due parti: l'implicazione "f Turing-calcolabile, allora f k-Turing-calcolabile" è banalmente vera ponendo $k = 1$. Per quanto riguarda l'implicazione inversa, si può fare la seguente considerazione. Se $\mathbf{M}^k$ è una macchina di Turing a k-nastri di alfabeto Σ, ad ogni istante il contenuto dei suoi nastri può essere codificato da un'unica stringa (rappresentabile dunque su un unico nastro) su un alfabeto di $(|\Sigma|+1)^k$ simboli, dove $|\Sigma|$ è il numero di simboli di Σ. Supponendo infatti di aver assegnato una numerazione alle celle di ciascun nastro, il contenuto delle k celle del generico indice i forma una stringa di lunghezza k sull'alfabeto $\overline{\Sigma}$, e di tali stringhe ne esistono proprio $(|\Sigma|+1)^k$.

Ora, ad ogni dato istante le testine dei k nastri saranno posizionate su una cella ben precisa; possiamo dunque immaginare che ci siano altri k nastri, uno per ogni nastro di $\mathbf{M}^k$, i quali siano interamente riempiti di simboli $\Box$ ad eccezione della cella corrispondente alla posizione della testina per il nastro che dobbiamo codificare. Possiamo immaginare che tale cella contenga un simbolo qualsiasi di Σ, fissato a priori.

In base alle considerazione fatte sopra, il contenuto dei $2k$ nastri (quelli di $\mathbf{M}^k$ più i k "ausiliari", che rappresentano le posizioni di ciascuna testina) può essere "compresso" in unico nastro, utilizzando un alfabeto di $(|\Sigma|+1)^{2k}$ simboli. Così facendo, abbiamo rappresentato la generica configurazione di $\mathbf{M}^k$ con una configurazione di una macchina ad un solo nastro $\mathbf{M}$.

La "simulazione" di $\mathbf{M}^k$ con $\mathbf{M}$ è ora abbastanza semplice. Basterà cercare la posizione di tutte le testine, registrando con una serie di stati opportuni il contenuto delle rispettive celle, ed effettuare la transizione di $\mathbf{M}^k$ apportando al nastro le modifiche necessarie. E' chiaro dunque che $\mathbf{M}$ avrà non solo più simboli, ma anche più stati di $\mathbf{M}^k$, e inoltre ogni passo di $\mathbf{M}^k$ si tradurrà in più passi di $\mathbf{M}$, ma ciò non ha importanza ai fini della calcolabilità. I dettagli della simulazione possono essere trovati in [19]. $\square$

Un'altra variante, indubbiamente di enorme interesse, è quella delle macchine di Turing non-deterministiche:

Definizione 2.8 (Macchina di Turing non deterministica) *Una macchina di Turing non deterministica* $\mathbf{N}$ *è una tripla*

$$\langle \Sigma \cup \{\square\}, S \cup \{s_0, s_\infty\}, \delta \rangle$$

definita esattamente come per le macchine di Turing deterministiche, ad eccezione della funzione di transizione, per la quale è

$$\delta : \overline{S} \times \overline{\Sigma} \to 2^{\overline{S} \times \overline{\Sigma} \times \{\leftarrow, \rightarrow\}}$$

dove, se A è un insieme, 2^A ne denota l'insieme delle parti, cioè l'insieme di tutti i sottoinsiemi di A.

Le macchine di Turing non deterministiche introducono una differenza radicale rispetto alle loro controparti deterministiche: ad ogni passo, la funzione di transizione determina una "scissione" nel comportamento della macchina; ogni configurazione ha quindi più di un successore. In pratica, è come se ad ogni passo si creassero più "mondi paralleli" in cui l'esecuzione della macchina prende una strada diversa. Naturalmente, la configurazione di una macchina non deterministica sarà semplicemente un insieme finito di configurazioni deterministiche, e il passo elementare genererà la configurazione successiva come unione di tutte le configurazioni generate dagli elementi della configurazione in esame. La calcolabilità viene definita di conseguenza: una macchina di Turing non deterministica termina quando una delle sue configurazioni contiene una configurazione deterministica finale (accettante o no, a seconda se si stia decidendo un linguaggio o calcolando una funzione).

Il mondo della computazione non deterministica è il paradiso degli indecisi: di fronte ad una scelta, ci si rifiuta sempre di prendere una decisione, e si opta così per intraprendere *tutte le strade possibili.* Possiamo descrivere la differenza tra computazione deterministica e non deterministica con un semplice esempio. Supponiamo di trovarci all'ingresso di un labirinto, del quale ovviamente si vuole trovare l'uscita. Come tutti i labirinti, quello che ci accingiamo ad attraversare contiene vicoli ciechi, vale a dire punti in cui non è più possibile proseguire in alcuna direzione, se non tornando indietro da dove si arrivati. Il nostro labirinto però contiene un'altra insidia, ben più temibile, costituita da un certo numero di *cammini infiniti*. Naturalmente, tali cammini sono indistinguibili: all'interno del labirinto non si è mai in grado di dire se alla prossima svolta ci sarà l'uscita, oppure se dovremo ancora proseguire per chissà quanto.

Al contrario di quanto avviene nella nostra realtà deterministica, in quanto esseri non deterministici avremmo la capacità di creare, davanti ad una scelta qualsiasi, un numero arbitrario di nostri *alter ego*. Questi *alter ego*, agendo in dimensioni parallele, sarebbero in grado di prendere ciascuno una strada diversa, scegliendo dunque una delle possibili opzioni che ci erano state offerte. Di fronte ad un bivio del nostro labirinto, saremmo quindi in grado di "sdoppiarci" in modo da prendere in contemporanea entrambe le direzioni. Alla fine, una delle miriadi di nostre copie in una delle miriadi di dimensioni parallele create nel corso dell'avventura arriverà all'uscita, e richiamerà a sé tutti i suoi *alter ego*, salvandoli possibilmente dall'essersi imbattuti in un cammino infinito. E' interessante notare che un osservatore posizionato nella dimensione cui appartiene il nostro *alter ego* "vincitore", quello cioè che ha trovato l'uscita, ha avuto l'impressione che noi conoscessimo alla perfezione la strada per attraversare il labirinto; infatti, in quella dimensione, non abbiamo compiuto un solo errore, e siamo arrivati all'uscita *percorrendo il cammino più corto possibile*.

Purtroppo, nella nostra misera condizione deterministica, l'unico metodo infallibile che conosciamo per uscire vivi dal labirinto è quello di *esplorare una ad una tutte le possibilità*. In particolare, sapendo dell'esistenza di cammini infiniti, opteremo in questo caso per un'esplorazione "in ampiezza" del labirinto, piuttosto che "in profondità". Ad ogni bivio, sceglieremmo una delle due strade, continuando però soltanto fino al bivio successivo, accertandoci dunque che non si tratti di un vicolo cieco. A questo punto, torneremmo indietro ed esploreremmo l'altra strada, sempre fino al bivio successivo. Osserviamo che, così facendo, saremo costretti a tornare indietro fino all'ingresso (per poi rientrare scegliendo un'altra strada) un numero anche altissimo di volte, tante quante ne servono per arrivare a trovare l'uscita.

L'esempio sopra mette in luce due aspetti interessanti: il primo è che, a quanto pare, il non determinismo non aumenta il potere computazionale delle macchine di Turing. L'intuizione è in effetti verificata dalla seguente proposizione:

Proposizione 2.2 *Una funzione f da interi a interi è Turing-calcolabile in senso non deterministico se e solo essa è Turing-calcolabile in senso deterministico.*

Dimostrazione. Anche qui, un'implicazione dell'enunciato è ovvia. Per l'altro verso dell'implicazione, vale un argomento che è sostanzialmente una versione formalizzata dell'esplorazione del labirinto esposta sopra; si veda [19] per i dettagli. □

Il secondo aspetto, per certi versi molto più interessante, è che la risoluzione di un problema con metodi non deterministici sembra offrire vantaggi enormi dal punto di vista del tempo richiesto per arrivare alla soluzione. Anche se non abbiamo ancora formalizzato il concetto di *complessità computazionale* (la cosa sarà oggetto del capitolo 5), e in particolare di cosa s'intenda per "tempo richiesto all'esecuzione di un programma", è chiaro da un punto di vista intuitivo che, nell'esempio del labirinto, il tempo richiesto per trovare l'uscita nel caso deterministico può essere mostruosamente più grande di quello richiesto per trovarla grazie ai "poteri magici" utilizzabili nel caso non deterministico. Questa considerazione informale è in realtà la base di una delle questioni fondamentali dell'Informatica Teorica, attorno alla quale roteano le più grandi questioni ancora irrisolte del campo.

Tornando alle proposizioni 2.1 e 2.2, queste sono solo "punte d'iceberg" di una serie di risultati analoghi riguardanti tutte le possibili varianti mai concepite di macchine di Turing. Tutto ciò ha indotto a pensare che le macchine di Turing, nella nostra definizione iniziale, catturino in modo completo il significato di *calcolabilità*. In altre parole, sembra non possa esistere una funzione che sia calcolabile in senso intuitivo ma per la quale non esista una macchina di Turing in grado di calcolarla. Abbiamo allora la celeberrima tesi di Church-Turing:

Proposizione 2.3 (Tesi di Church-Turing) *Una funzione è "calcolabile" se e solo se è Turing-calcolabile.*

La tesi di Church-Turing, chiaramente non dimostrabile formalmente, asserisce in sostanza che la nostra idea di calcolabilità è codificata a livello matematico in modo esatto dalle macchine di Turing. Vedremo nel seguito che esistono altri modelli di calcolo, equivalenti alle macchine di Turing, i quali possono vantare la stessa corrispondenza con la nostra intuizione, avallando ancora di più l'ipotesi che tutte le formalizzazioni del concetto di "calcolabilità" che abbiamo oggi a disposizione sono in effetti le migliori che potremo mai trovare.

2.2 Le funzioni ricorsive

Quello delle funzioni ricorsive, a differenza delle macchine di Turing introdotte nella sezione precedente, è un modello *puramente astratto*. Con la nozione di funzione ricorsiva si cerca cioè di catturare il significato intuitivo di funzione calcolabile in modo assiomatico, astraendosi da qualsiasi riferimento ad una particolare "realizzazione fisica". E' chiaro infatti che, come già osservato, le macchine di Turing si avvicinano parecchio ai calcolatori elettronici reali, i quali ne rappresentano, pur approssimativamente, una realizzazione concreta. Al contrario, le funzioni ricorsive non potranno essere rapportate a nessun oggetto concreto; il modello assiomatico è, utilizzando un'espressione dell'informatica moderna, *machine independent*.

Iniziamo con il definire una prima classe di funzioni ricorsive:

Definizione 2.9 (Funzioni ricorsive primitive) *L'insieme $\mathcal{PR}$ delle funzione ricorsive primitive è l'insieme generato induttivamente nel seguente modo:*

i (Costante). *La funzione costante 0 è una funzione ricorsiva primitiva di arità nulla*

ii (Proiezioni). *Le proiezioni π_i^n, tali che*

$$\pi_i^n(x_1, \ldots, x_n) = x_i \qquad 1 \leq i \leq n$$

sono funzioni ricorsive primitive di arità n

iii (Successore). *La funzione successore s, tale che*

$$s(x) = x + 1$$

è una funzione ricorsiva primitiva di arità 1.

iv (Composizione). *Se $g_1, \ldots, g_m$ sono m funzioni ricorsive primitive di arità n, e se h è una funzione ricorsiva primitiva di arità m, allora*

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

è una funzione ricorsiva primitiva di arità n

v (Ricorsione primitiva). *Se g è una funzione ricorsiva primitiva di arità n, e se h è una funzione ricorsiva primitiva di arità $n+2$, allora*

$$f(x_1, \ldots, x_n, y) = \begin{cases} g(x_1, \ldots, x_n) & \textit{se } y = 0 \\ h(x_1, \ldots, x_n, z, f(x_1, \ldots, x_n, z)) & \textit{se } y = z + 1 \end{cases}$$

è una funzione ricorsiva primitiva di arità $n+1$

La definizione di funzione ricorsiva primitiva cattura un grandissimo numero di funzioni intuitivamente calcolabili; tuttavia, grazie ad un argomento diagonale, è possibile dimostrare piuttosto rapidamente che esistono funzioni calcolabili non appartenenti alla classe $\mathcal{PR}$. In particolare, si può trovare un esempio concreto, costituito da quella nota come la *funzione di Ackermann*:

$$\begin{aligned} &A(0, y) = y + 1 \\ &A(x+1, 0) = A(x, 1) \\ &A(x+1, y+1) = A(x, A(x+1, y)) \end{aligned}$$

Non è difficile mostrare (per induzione sull'ordine lessicografico delle coppie (x, y) in input alla funzione A) che la funzione di Ackermann è ben definita, ed è dunque calcolabile in senso intuitivo. Tuttavia, si può dimostrare che la funzione di Ackermann non appartiene a $\mathcal{PR}$. Lo schema di ricorsione primitiva si rivela essere troppo debole per definire una tale funzione, la cui crescita spaventosa supera quella raggiungibile da qualsiasi funzione ricorsiva primitiva.

La classe $\mathcal{PR}$ non è dunque il modello che stiamo cercando; per estenderlo, occorre aggiungere uno schema più potente della ricorsione primitiva. Come vedremo, sarà inoltre necessario passare alla nozione di funzione parziale, che definiamo di seguito:

Definizione 2.10 (Funzione parziale) *Una* funzione parziale f *da* $\mathbb{N}^n$ *a* $\mathbb{N}$ *è una coppia* $\langle f_0, D\rangle$*, dove* f *è una funzione definita su tutto* $\mathbb{N}^k$ *e* D *è un sottoinsieme di* $\mathbb{N}^k$*, detto* dominio *di* f*. Per un generico* $\vec{x} \in \mathbb{N}^k$*, abbiamo*

$$f(\vec{x}) = \begin{cases} f_0(\vec{x}) & \textit{se } \vec{x} \in D \\ \perp & \textit{altrimenti} \end{cases}$$

Il simbolo $\perp$ *indica che* f *non è definita; in breve, per il generico* $\vec{x} \in \mathbb{N}^k$*, se* $\vec{x} \in D$*, scriveremo* $f(\vec{x}) \downarrow$*, se* $\vec{x} \notin D$*, scriveremo* $f(\vec{x}) \uparrow$*.*
Se f *è una funzione (anche non parziale), indicheremo con* $\mathsf{dom}f$ *il suo dominio.*

Definizione 2.11 (Funzione totale) *Una funzione* f *di arità* k *è detta* totale *se* $\mathsf{dom}f = \mathbb{N}^k$*.*

Evidentemente, tutte le funzioni ricorsive primitive sono totali. Introduciamo ora la notazione μ, la quale servirà a definire il cosiddetto *schema di minimizzazione*:

Definizione 2.12 *Sia* P *un* predicato unario*, cioè una funzione*

$$P : \mathbb{N} \to \{0, 1\}$$

Denoteremo con

$$\mu x.P(x)$$

il più piccolo intero x *tale che rende vero* P*, cioè tale* $P(x) = 0$[1]*. Se tale intero non esiste,* $\mu x.P(x) \uparrow$*.*

Passiamo ora finalmente alla definizione della classe, più ampia, delle funzioni ricorsive parziali o, più semplicemente, funzioni ricorsive:

[1]Per ragioni tecniche, conviene prendere 0 come "vero" e 1 come "falso".

Definizione 2.13 (Funzioni ricorsive) *L'insieme $\mathcal{R}$ delle funzioni ricorsive è l'insieme generato induttivamente nel seguente modo:*

i (Costante). *La funzione costante 0 è una funzione ricorsiva totale di arità nulla*

ii (Proiezioni). *Le proiezioni π_i^n, tali che*

$$\pi_i^n(x_1, \ldots, x_n) = x_i \qquad 1 \leq i \leq n$$

sono funzioni ricorsive totali di arità n

iii (Successore). *La funzione successore s, tale che*

$$s(x) = x + 1$$

è una funzione ricorsiva totale di arità 1.

iv (Composizione). *Se $g_1, \ldots, g_m$ sono m funzioni ricorsive di arità n, e se h è una funzione ricorsiva di arità m, allora*

$$f(x_1, \ldots, x_n) = h(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$$

è una funzione ricorsiva di arità n, il cui dominio è

$$\mathsf{dom} f = \mathsf{dom} h \cap \bigcap_{i=1}^{m} \mathsf{dom} g_i$$

v (Ricorsione primitiva). *Se g è una funzione ricorsiva di arità n, e se h è una funzione ricorsiva di arità $n+2$, allora*

$$f(x_1, \ldots, x_n, y) = \begin{cases} g(x_1, \ldots, x_n) & \textit{se } y = 0 \\ h(x_1, \ldots, x_n, z, f(x_1, \ldots, x_n, z)) & \textit{se } y = z + 1 \end{cases}$$

è una funzione ricorsiva di arità $n+1$, il cui dominio è dato da

$$\begin{array}{ll} f(\vec{x}, 0) \downarrow & \textit{sse } g(\vec{x}) \downarrow \\ f(\vec{x}, y+1) \downarrow & \textit{sse } f(\vec{x}, y) \downarrow \textit{ e } h(\vec{x}, y, f(\vec{x}, y)) \downarrow \end{array}$$

vi (Minimizzazione). *Se g è una funzione ricorsiva di arità $n+1$, allora*

$$f(x_1, \ldots, x_n) = \mu y.(g(x_1, \ldots, x_n, y) = 0)$$

è una funzione ricorsiva di arità n, il cui dominio è dato da

$$f(\vec{x}) \downarrow \textit{ sse } \forall z < y,\ g(\vec{x}, z) \downarrow \textit{ e } g(\vec{x}, z) > 0,\ \textit{e } g(\vec{x}, y) \downarrow \textit{ e } g(\vec{x}, y) = 0$$

Lo schema di minimizzazione, anche detto *schema-μ*, fornisce alle funzioni ricorsive il potere espressivo che mancava allo schema di ricorsione primitiva. Grazie ad esso, funzioni come quella di Ackermann diventano definibili. In particolare, lo schema-μ è talmente potente che, modificando in modo minimo le funzioni ricorsive di base, è possibile esprimere la ricorsione primitiva per mezzo di esso. Infatti, si dimostra che la seguente è una definizione alternativa dell'insieme $\mathcal{R}$:

Definizione 2.14 (Funzioni ricorsive, definizione alternativa) *L'insieme $\mathcal{R}$ delle funzioni ricorsive è l'insieme generato induttivamente nel seguente modo:*

i (Proiezioni). *Come 2.13*

ii (Addizione). *La funzione addizione $+$, tale che*

$$+(x, y) = x + y$$

è una funzione ricorsiva totale di arità 2.

iii (Moltiplicazione). *La funzione moltiplicazione $\cdot$, tale che*

$$\cdot(x, y) = x \cdot y$$

è una funzione ricorsiva totale di arità 2.

iv (Minore). *La funzione $\chi_<$, che è la funzione caratteristica della relazione $<$ tra interi non negativi, cioè tale che*

$$\chi_<(x, y) = \begin{cases} 0 & \textit{se } x < y \\ 1 & \textit{altrimenti} \end{cases}$$

è una funzione ricorsiva totale di arità 2.

v (Composizione). *Come 2.13*

vi (Minimizzazione). *Come 2.13*

Osserviamo che, grazie allo schema-μ, non è più necessario aggiungere alcuna costante tra le funzioni di base; la costante nulla è infatti definibile come

$$0 = \mu x.(\pi_1^1(x) = 0)$$

per la costante 1 abbiamo invece

$$1 = \mu x.(\chi_<(0, x) = 0)$$

e, in generale, una volta definita la costante n, $n + 1$ è definibile come

$$n + 1 = \mu x.(\chi_<(n, x) = 0)$$

A partire dalle funzioni ricorsive, è possibile definire il sottoinsieme di quelle totali:

Definizione 2.15 (Funzioni ricorsive totali) *L'insieme $\mathcal{TR}$ delle funzioni ricorsive totali è il sottoinsieme di $\mathcal{R}$ costituito dalle sole funzioni totali.*

E' opportuno notare inoltre che l'unica fonte di "parzialità" delle funzioni ricorsive è lo schema-μ; è solo grazie ad esso che è possibile costruire funzioni non totali, come ad esempio

$$\Omega(x) = \mu y. \left(\chi_<(\pi_2^2(x,y), \mu z. \left(\pi_3^3(x,y,z) = 0\right)) = 0\right)$$

che è una funzione chiaramente ricorsiva (è ottenuta applicando un numero finito di volte gli schemi di composizione e minimizzazione a partire dalle proiezioni e da $\chi_<$, che sono funzioni ricorsive di base), per la quale si ha $\Omega(x) \uparrow$ per ogni $x \in \mathbb{N}$. Infatti, in sostanza Ω cerca il più piccolo intero non negativo minore di zero, che chiaramente non esiste.

In termini di macchine di Turing, la parzialità di una funzione corrisponde al fatto che la macchina che la calcola cicla all'infinito per qualche valore dell'input; nel caso di Ω, la macchina corrispondente non termina mai qualunque sia l'input fornitogli. Le funzioni ricorsive totali corrispondono dunque a programmi ben costruiti, i quali terminano per tutti i possibili valori in ingresso.

Sempre in ambito di macchine di Turing, abbiamo accennato a come la risoluzione di un problema qualsiasi fosse riconducibile alla risoluzione di un problema di decisione, la quale è in ultima analisi il calcolo di una funzione. L'analogo del problema di decisione nel caso delle funzioni ricorsive è dunque costituito da una funzione particolare, il cui valore di uscita può essere solo 0 o 1. Questo tipo di funzioni, come abbiamo già visto nella definizione della notazione μ, prendono il nome di *predicati*. Un predicato può naturalmente rientrare o meno in una delle classi $\mathcal{PR}$ e $\mathcal{TR}$ definite sopra; in tal caso, parleremo rispettivamente di *predicato ricorsivo primitivo* e *predicato ricorsivo*. Chiaramente non ha senso definire predicati parziali; esiste però un'altra classe di predicati, che definiamo nel seguito:

Definizione 2.16 (Funzione caratteristica) *Sia $A \subseteq \mathbb{N}^k$. La* funzione caratteristica *di A, che denoteremo con χ_A, è il predicato k-ario tale che*

$$\chi_A(a) = \begin{cases} 0 & se\ a \in A \\ 1 & se\ a \notin A \end{cases}$$

Definizione 2.17 (Predicati ricorsivamente enumerabili) *Un predicato P si dice* ricorsivamente enumerabile *se esiste una funzione ricorsiva parziale f, con $D = \mathsf{dom} f$, tale che $\chi_D = P$.*

Grazie alla funzione caratteristica, è possibile parlare della ricorsività di *insiemi*:

Definizione 2.18 *Un insieme A è ricorsivo primitivo, ricorsivo, o ricorsivamente enumerabile se la sua funzione caratteristica χ_A è un predicato, rispettivamente, ricorsivo primitivo, ricorsivo o ricorsivamente enumerabile.*

Il concetto di ricorsivamente enumerabile è più chiaro alla luce della seguente proposizione, che non dimostreremo:

Proposizione 2.4 *Un insieme A è ricorsivamente enumerabile se e solo se esiste una funzione ricorsiva totale f tale che $\mathsf{Im} f = A$, cioè A è l'immagine di f.*

In altre parole, se un insieme A è ricorsivamente enumerabile, è possibile trovare una funzione ricorsiva totale *che ne enumera gli elementi*, cioè che mette ogni elemento di A in corrispondenza con, ad esempio, un numero naturale.

Insiemi (o predicati) ricorsivi e ricorsivamente enumerabili sono in diretta corrispondenza con linguaggi decidibili e semi-decidibili in ambito di macchine di Turing. Infatti, se prendiamo un insieme ricorsivo, essendo la sua funzione caratteristica totale è sempre possibile stabilire l'appartenenza o meno di un oggetto a tale insieme; al contrario, se un insieme è ricorsivamente enumerabile, l'unica possibilità per stabilire se un oggetto vi appartiene o no è quella di controllare tutti gli elementi uno ad uno tramite la funzione totale che li enumera. E' chiaro che, se l'oggetto in questione appartiene all'insieme, prima o poi comparirà nell'enumerazione, e dunque potremo concludere in senso affermativo; tuttavia, se l'oggetto non appartiene all'insieme, l'enumerazione non lo menzionerà mai, è noi rimarremo nell'impossibilità di deciderne l'appartenenza.

Per quanto riguarda $\mathcal{TR}$, è possibile dimostrare mediante un argomento diagonale che tale insieme, pur essendo chiaramente numerabile, non è ricorsivamente enumerabile. Ecco perché la definizione di funzione ricorsiva prende in considerazione le funzioni parziali: grazie ad esse si ottengono buone proprietà di chiusura che non sono ottenibili con le funzioni totali.

Possiamo infine enunciare il seguente teorema:

Teorema 2.1 *Una funzione è ricorsiva se e solo è Turing-calcolabile.*

Dimostrazione. Si veda [7]. □

Questa equivalenza fa sì che, in termini di potenza espressiva, i due modelli si rafforzino a vicenda; nessuno ha mai trovato una funzione intuitivamente calcolabile che non fosse ricorsiva o Turing-calcolabile, e il fatto che due modelli definiti in modo così diverso finiscano per coincidere aumenta senz'altro la credibilità della tesi di Church-Turing.

2.3 Il λ-calcolo

2.3.1 Il λ-calcolo puro

Il λ-calcolo è stato introdotto negli anni '30 da Alonzo Church, con l'obiettivo di fornire una caratterizzazione sintattica delle funzioni matematiche. In realtà, il λ-calcolo si rivela essere molto di più: al suo interno sono infatti rappresentabili tutte le funzioni ricorsive.

Il λ-calcolo è dunque un modello di computazione della stessa espressività delle macchine di Turing e dell'assiomatizzazione ricorsiva. Rispetto a questi due sistemi, il λ-calcolo costituisce una sorta "via di mezzo": in quanto costruito per rappresentare funzioni, è da vari punti di vista molto vicino al modello ricorsivo; tuttavia, poiché il calcolo è basato sulla riscrittura di certe sequenze di simboli, la quale segue regole puramente sintattiche, gli oggetti del λ-calcolo possono in effetti essere visti come programmi, più vicini dunque allo spirito algoritmico delle macchine di Turing.

Infatti, se le macchine di Turing costituiscono un approccio "imperativo" alla programmazione, il λ-calcolo è il modello per antonomasia della programmazione funzionale. Come per le macchine di Turing, anche nel λ-calcolo esiste un "passo elementare" di esecuzione, ma l'esecuzione di tali passi non è determinabile in modo univoco. Esisteranno dunque diverse *strategie di esecuzione*, proprio come un compilatore funzionale può scegliere soluzioni diverse per la traduzione di un programma.

Un'altro dei pregi esclusivi del λ-calcolo è quello di essere il modello più vicino (in certi casi perfettamente coincidente) ai modelli di calcolo di origine logica; la discussione di ciò sarà oggetto del prossimo capitolo.

Passiamo dunque alla definizione della sintassi del λ-calcolo. Gli oggetti di cui "parla" il λ-calcolo (nella sua versione "pura", che è quella di cui ci stiamo occupando ora) sono particolari sequenze di caratteri, dette λ-termini:

Definizione 2.19 (λ-termine) *L'insieme $\mathcal{L}$ dei termini del λ-calcolo è l'insieme generato induttivamente nel seguente modo:*

var *Sia $\mathcal{V} = \{x, y, z, \ldots\}$ un insieme infinito numerabile di simboli, detti* variabili. *Per ogni $x \in \mathcal{V}$,*

$$x$$

è un λ-termine.

λ *Se t un λ-termine e x una variabile,*

$$(\lambda x.t)$$

è un λ-termine, detto λ-astrazione *della variabile x nel termine t.*

@ *Se t e u sono due λ-termini,*

$$(tu)$$

è un λ-termine, detto applicazione *del termine t al termine u.*

Essendo il λ-calcolo un sistema di riscrittura puramente sintattico, la questione della "parentesizzazione" dei termini è in genere rilevante. In particolare, esistono più convenzioni sintattiche per i λ-termini, ciascuna con i suoi vantaggi e svantaggi. Quella che abbiamo scelto noi è la più vicina alla sintassi dei linguaggi funzionali più diffusi, come ML e CaML, e non dovrebbe essere di difficile lettura. Comunque, per alleggerire la notazione, e renderla ancor più prossima a quella dei linguaggi funzionali, nel seguito adotteremo le seguenti convenzioni:

- $(\lambda x_1.(\lambda x_2.(\ldots(\lambda x_n.t)\ldots)))$ sarà riscritto semplicemente come $(\lambda x_1 x_2 \ldots x_n.t)$
- $((((t_1 t_2)t_3)\ldots)t_n)$ sarà riscritto semplicemente come $(t_1 t_2 t_3 \ldots t_n)$. In altre parole, l'*applicazione è associativa a sinistra*; $((tu)v)$ potrà essere riscritto come (tuv), mentre $(t(uv))$ rimarrà tale.
- In generale, faremo più economia possibile sulle parentesi, in particolare omettendo sistematicamente quelle a livello più alto; $(\lambda x.t)$ sarà semplicemente $\lambda x.t$, (uv) sarà semplicemente uv, e così via. Le parentesi dunque saranno utilizzate esclusivamente come indicazione dell'ordine di applicazione (come nel caso dei termini tuv e $t(uv)$ dell'esempio sopra) e per distinguere l'applicazione di un termine che comincia per λ ad un altro termine, come nel caso di $(\lambda x.t)u$, dalla λ-astrazione di un termine che è l'applicazione di due termini, come $\lambda x.tu$.

Introduciamo ora il concetto di *variabile libera* e *variabile vincolata*, identico a quello già incontrato nelle formule logiche in presenza dei quantificatori:

Definizione 2.20 (Variabili libere e vincolate) *Sia $t \in \mathcal{L}$. Definiamo l'insieme delle* variabili libere *di t, che denotiamo con $\mathsf{FV}(t)$, induttivamente sulla struttura di t:*

var *Se $t = x$, con $x \in \mathcal{V}$, allora $\mathsf{FV}(t) = \{x\}$.*

λ *Se $t = \lambda x.u$, allora $\mathsf{FV}(t) = \mathsf{FV}(u) \setminus \{x\}$.*

@ *Se $t = uv$, allora $\mathsf{FV}(t) = \mathsf{FV}(u) \cup \mathsf{FV}(v)$.*

Se x è una variabile che compare in una λ-astrazione di t, allora le occorrenze di x contenute nel sotto-termine in cui è stata fatta l'astrazione sono dette vincolate.
Un termine t tale che $\mathsf{FV}(t) = \emptyset$ è detto chiuso.

La distinzione tra variabili libere e variabili vincolate è una questione che, pur essendo sotto certi aspetti secondaria, si rivela essere uno dei punti chiave del λ-calcolo. In sostanza, l'astrazione di una variabile comporta che questa sia vincolata *nel termine in cui essa viene astratta*; ai fini di tale termine, la variabile vincolata diventa come la variabile d'integrazione in un integrale: il suo nome non conta, ed essa può essere rinominata a piacimento. Se ad esempio consideriamo il termine

$$t = xy\lambda x.xy$$

è chiaro che, pur essendo $x \in \mathsf{FV}(t)$, c'è un'occorrenza di x (quella in $\lambda x.xy$) che è vincolata; il fatto che una variabile appartenga all'insieme delle variabili libere di un termine dunque non significa che questa non possa mai comparire in un'astrazione. Possiamo dunque sostituire x con qualsiasi variabile, ma *solo per quanto riguarda le sue occorrenze vincolate.*

Nel ridenominare una variabile occorre però fare estrema attenzione. Considerando ancora il termine t del nostro esempio, rinominare l'occorrenza vincolata di x con z è perfettamente lecito:

$$xy\lambda z.zy$$

Tuttavia, scegliere y come nuovo nome per x porta al seguente termine:

$$xy\lambda y.yy$$

che ha un significato completamente diverso rispetto a t; è come se, nell'integrale

$$I(x) = \int_a^b f(x)g(y)dy$$

sostituissimo la variabile d'integrazione y con x:

$$\int_a^b f(x)g(x)dx$$

I due oggetti sono radicalmente diversi; il primo è una funzione di x, il secondo è una costante.

La questione fondamentale è che le variabili vincolate, come le variabili d'integrazione, indicano oggetti *a cui dovrà essere sostituito qualcosa.* La sostituzione, che vedremo fra breve, è infatti una delle operazioni (se non *l'operazione*) fondamentali del λ-calcolo; ecco perché il discorso della ridenominazione delle variabili è alquanto delicato. Il problema è essenzialmente questo: una variabile, in generale, porta con sé due informazioni: il suo nome, e il *luogo* dove essa si trova, che indica il punto dove andrà posizionato ciò che ad essa sarà sostituito. Per una variabile libera sono importanti entrambe le informazioni; per le variabili vincolate invece ciò che importa è solo il luogo. La gestione corretta di queste due informazioni, se condotta puramente a livello sintattico, diventa una questione assolutamente non banale; ad essa sono legate, per certi aspetti, alcune ricerche estremamente recenti in campo logico, in particolare nell'ambito della cosiddetta *ludica*.

La formalizzazione del concetto di "equivalenza di due termini sotto la ridenominazione di variabili vincolate", che nel λ-calcolo si chiama α-equivalenza, è una faccenda che va oltre gli scopi di questa breve introduzione; al lettore curioso consigliamo di leggere il primo capitolo di [16], in cui la questione dell'α-equivalenza è trattata in modo più che rigoroso. In questa sede, ci accontenteremo della seguente definizione informale:

Definizione 2.21 (α-equivalenza) *Due termini t e u di $\mathcal{L}$ sono detti α-equivalenti, cosa che indicheremo con $t \equiv_\alpha u$, se si può ottenere l'uno a partire dall'altro rinominando un certo numero di variabili vincolate, facendo attenzione ai problemi di "cattura di variabili" menzionati sopra.*

Proposizione 2.5 *$\equiv_\alpha$ è una relazione d'equivalenza su $\mathcal{L}$.*

D'ora in poi, considereremo i λ-termini *modulo l'α-equivalenza*; in altre parole, al posto di $\mathcal{L}$ faremo sempre riferimento all'insieme $\Lambda = \mathcal{L}/\equiv_\alpha$, cioè l'insieme dei λ-termini quozientato rispetto all'α-equivalenza.

Definiamo allora l'operazione fondamentale di *sostituzione* nel modo seguente:

Definizione 2.22 (Sostituzione) *Siano s e v due termini di Λ, e sia x una variabile. Definiamo allora la* sostituzione del termine v in s al posto di x, *che denoteremo con $s[v/x]$, per induzione sulla struttura di v:*

var *Se $s = y$, allora*
$$s[v/x] = s$$
Al contrario, se $s = x$, allora
$$s[v/x] = v$$

λ *Se $s = \lambda y.t$, allora*
$$s[v/x] = \lambda z.t[v/x, z/y]$$
dove $z \notin \mathsf{FV}(v)$ è una variabile "nuova", che viene sostituita al posto di y. Se invece $s = \lambda x.t$, allora
$$s[v/x] = s$$

@ *Se* $s = tu$, *allora*

$$s[v/x] = t[v/x]u[v/x]$$

La sostituzione può naturalmente essere estesa ad un numero arbitrario di variabili; se s, $v_1, \ldots, v_n$ sono λ-termini, e se $x_1, \ldots, x_n$ sono variabili, indicheremo con

$$s[v_1/x_1, \ldots, v_n/x_n]$$

la sostituzione *contemporanea* dei termini $v_1, \ldots, v_n$ al posto delle rispettive variabili $x_1, \ldots, x_n$.

Definiamo ora il concetto di *contesto*, che sarà preliminare al seguito dell'esposizione:

Definizione 2.23 (Contesto) *Un* contesto *è un'espressione sintattica, che denoteremo con $C[\bullet]$, in cui c'è un "buco". Un contesto è tale che, per ogni $t \in \Lambda$, $C[t] \in \Lambda$, dove $C[t]$ indica il contesto a cui è stato rimpiazzato t al posto del "buco".*

Esempi di contesti sono $\lambda x. \bullet x$, $\bullet xyz$, nonchè semplicemente $\bullet$, che è il contesto vuoto.

Ora, se ρ è una relazione sui termini di Λ, diremo che ρ *passa al contesto* se e solo se, essendo t e u due λ-termini,

$$t \; \rho \; u \Rightarrow C[t] \; \rho \; C[u] \quad \text{per ogni contesto } C[\bullet]$$

Allo stesso modo, la *chiusura contestuale* $\tilde{\rho}$ di una relazione ρ su Λ è la più piccola relazione contenente ρ che passa al contesto.

Siamo ora pronti a trattare la dinamica del λ-calcolo, vale a dire il processo di riscrittura che lo rende un modello di computazione. Partiremo dalla regola di conversione di base, per poi definire la sua versione più generale:

Definizione 2.24 (β_0-conversione) *Definiamo sui termini di Λ la seguente relazione di riscrittura:*

$$(\lambda x.t)u \rightarrow_0 t[u/x]$$

che è detta β_0-conversione.
Un termine s della forma $(\lambda x.t)u$, cioè composto dall'applicazione di un'astrazione ad un altro termine, è detto redex. *Il termine che risulta dalla β_0-conversione di s è detto il* ridotto *di s.*

Definizione 2.25 (β-conversione) *Definiamo la β_1-conversione (β-conversione in un passo) come la chiusura contestuale della β_0-conversione. Se u risulta dalla β_1-conversione di t, scriveremo*

$$t \rightarrow u$$

La β-conversione sarà allora definita come la chiusura riflessiva e transitiva della β_1-conversione. Per la β-conversione adotteremo la scrittura

$$t_0 \twoheadrightarrow t_{n+1}$$

intendendo che o $t_0 = t_{n+1}$, oppure esistono n λ-termini $t_1, \ldots, t_n$ tali che

$$t_0 \to t_1 \to \ldots \to t_n \to t_{n+1}$$

Un nome alternativo per la β-conversione è quello di *β-riduzione*; se $t \twoheadrightarrow u$, diremo in genere che *t si riduce a u.*

In sostanza, la dinamica del λ-calcolo è costituita da un processo di riscrittura, scomponibile in passi atomici (la β_1-conversione), in cui ogni volta si cerca all'interno del termine che si vuole ridurre un sotto-termine di forma particolare (quello che abbiamo chiamato *redex*); una volta trovatone uno, lo si rimpiazza con il suo *ridotto*, secondo una precisa regola di sostituzione (la β_0-conversione). Quando non è più possibile trovare alcun redex all'interno del termine che si sta elaborando, il processo finisce, e si è arrivati alla cosiddetta *forma normale*:

Definizione 2.26 (Forma normale) *Un termine t è detto* normale *se non esiste alcun termine u tale che*

$$t \to u$$

In sostanza, t non contiene alcun redex.
Allo stesso modo, se t è un termine normale, e s è un termine tale che

$$s \twoheadrightarrow t$$

diremo che t è una forma normale *di s.*

Il lettore attento si sarà ricordato che, nell'introduzione della presente sezione, avevamo accennato al fatto che la dinamica del λ-calcolo fosse in grado di generare situazioni in cui più strategie di esecuzione di uno stesso programma sono possibili. Sapendo che i programmi sono i λ-termini, e sapendo che l'esecuzione di un λ-termine corrisponde al processo di β-riduzione, ora siamo in grado di verificare la correttezza di quanto era stato anticipato. E' chiaro, infatti, che all'interno di un singolo λ-termine ci può essere più di un redex e, poiché la β_1-conversione non dice nulla su quale vada scelto, lo stesso termine può essere ridotto in più modi diversi, conducendo a più termini diversi. Poiché la β-riduzione semplicemente itera il procedimento, viene da domandarsi se non sia possibile che il processo di riduzione diverga, cioè che esso porti a due risultati diversi a seconda di quale strategia di esecuzione si scelga.

In merito a ciò, vale il seguente teorema fondamentale:

Teorema 2.2 (Proprietà di Church-Rosser) *Se s, t e u sono tre termini tali che*

$$s \twoheadrightarrow t$$
$$s \twoheadrightarrow u$$

allora esiste un termine v tale che

$$t \twoheadrightarrow v$$
$$u \twoheadrightarrow v$$

Dimostrazione. Si veda [16]. □

Grazie alla proprietà di Church-Rosser, detta anche proprietà di confluenza, possiamo dimostrare la seguente proposizione:

Proposizione 2.6 (Unicità della forma normale) *Se un termine t ha una forma normale, allora essa è unica.*

Dimostrazione. Supponiamo che t abbia due forme normali u' e u''. Per definizione di forma normale, abbiamo

$$t \twoheadrightarrow u'$$

e

$$t \twoheadrightarrow u''$$

Ma allora, per la confluenza, esiste un altro termine v tale che

$$u' \twoheadrightarrow v$$

e

$$u'' \twoheadrightarrow v$$

Ora, poiché abbiamo supposto u' e u'' normali, non può essere che $v = u'$ e $v = u''$, e dunque $u' = u''$. □

Se vediamo la forma normale di un termine come il risultato dell'esecuzione di un programma, la proprietà di Church-Rosser ci garantisce dunque che tale risultato, se esiste, è unico; ciò è senz'altro confortante da un punto di vista informatico.

Se le varie strategie non contano ai fini del risultato, esse sono invece determinanti ai fini del numero di passi necessari ad arrivare a tale risultato. In altri termini, è possibile studiare la β-riduzione dal punto di vista della sua *ottimizzazione*. Un esempio che può chiarire le idee è il seguente. Supponiamo di avere un termine B tale che

$$B \twoheadrightarrow \lambda xy.x$$

e due termini u e v le cui rispettive forme normali sono u' e v'. Consideriamo ora il seguente temine:

$$(\lambda b.buv)B$$

Possiamo immaginare almeno due strategie diverse; la prima, effettua dapprima tutti i calcoli che riguardano B, e poi passa a u e v:

$$(\lambda b.buv)B \to Buv \twoheadrightarrow (\lambda xy.x)uv \to (\lambda y.u)v \to u \twoheadrightarrow u'$$

La seconda lavora invece prima su u e v, e poi riduce B:

$$(\lambda b.buv)B \to Buv \twoheadrightarrow Bu'v \twoheadrightarrow Bu'v' \twoheadrightarrow (\lambda xy.x)u'v' \to (\lambda y.u')v' \to u'$$

E' chiaro come la prima strategia sia in generale migliore della seconda; in particolare, quest'ultima effettua tutta una serie di calcoli su v, che potrebbero dal punto di vista della durata essere più lunghi di tutti i calcoli su B e su u messi assieme. Tali calcoli alla fine si rivelano essere completamente inutili, visto che v', la forma normale di v, viene semplicemente "buttata via".

L'ottimizzazione della β-riduzione è un soggetto di ricerca di altissimo interesse, all'interno del quale si sta tutt'oggi lavorando. In particolare, le applicazioni dei risultati in questo campo sono di grande aiuto alla costruzione di interpreti e compilatori funzionali ad alte prestazioni, i quali riducano al minimo possibile i tempi di esecuzione dei programmi che vengono loro sottoposti.

Tornando alla forma normale di un termine, nella proposizione 2.6 siamo stati ben attenti a specificare che essa è unica *a patto che esista*. Infatti, il λ-calcolo contiene oggetti alquanto bizzarri, per i quali il processo di β-riduzione non porta mai ad una forma normale, qualunque strategia si scelga per ridurli.

L'esempio tipico di un tale termine, nonché il più semplice, è il seguente. Definiamo anzitutto

$$\Delta = \lambda x.xx$$

e poniamo

$$\Omega = \Delta\Delta$$

Il termine Ω contiene chiaramente un redex; poiché è l'unico che c'è, se vogliamo ridurre Ω non abbiamo scelta:

$$\Omega = (\lambda x.xx)\Delta \to \Delta\Delta = \Omega$$

La situazione è un po' inattesa, ma ben chiara: abbiamo trovato un termine, non normale, che si riduce in sé stesso. Non c'è speranza che Ω arrivi mai ad una forma normale; questo rappresenta chiaramente un programma che cicla all'infinito, senza possibilità di fermarsi.

La chiave di volta della costruzione sta nel termine Δ. A ben guardarlo, questo signore ci sta offrendo di eseguire un'operazione un po' fuori dal comune: Δ ci dice infatti, "datemi un programma, e io vi restituirò il risultato di questo programma *applicato a lui stesso*". Qualunque cosa significhi "applicare un programma a sé stesso", nel λ-calcolo abbiamo pieno diritto di farlo. In particolare, poiché non ci viene posta alcuna restrizione su quale programma dare in input a Δ, nulla ci vieta di fare la cosa più perversa che ci viene in mente, che è quella di *applicarlo a lui stesso*... et voilà, abbiamo costruito un termine che cicla all'infinito.

Una volta scoperta la presenza di simili meraviglie nel λ-calcolo, possiamo giustamente pensare di classificare i termini di Λ in base al loro comportamento rispetto alla β-riduzione. Per *sequenza di riduzione* intenderemo nel seguito una sequenza di applicazioni della β_1-conversione, la quale termina solo quando si è arrivati ad una forma normale.

Definizione 2.27 *I termini del λ-calcolo possono essere di tre specie:*

- *un termine t è detto* fortemente normalizzabile *se tutte le sequenze di riduzione di t hanno lunghezza finita*

- *un termine t è detto* normalizzabile *se esiste una sequenza di riduzione di t di lunghezza finita*

- *un termine t è detto* non normalizzabile *se non esiste alcuna sequenza di riduzione di lunghezza finita*

Chiaramente, un termine fortemente normalizzabile è anche normalizzabile. Esemplari tipici delle tre classi di "normalizzazione" sono i seguenti:

· $(\lambda fx.f(f(fx)))(\lambda x.x)\lambda xy.y$ è fortemente normalizzabile, e la sua forma normale è $\lambda xy.y$

· $(\lambda xy.y)\ \Omega\ \lambda fx.f(fx)$ è normalizzabile, ma non fortemente normalizzabile; la sua forma normale è $\lambda fx.f(fx)$, ma è chiaro che, essendoci un redex in Ω, si può continuare all'infinito a scegliere di ridurre tale redex, producendo una sequenza di riduzione che non termina mai

· abbiamo già visto che Ω non è normalizzabile; un altro esempio di termine non normalizzabile, ma non ciclico (cioè che non si riduce in sé stesso) è $H = (\lambda x.y(xx))\lambda x.y(xx)$, per il quale chiaramente si ha $H \to yH \to y(yH) \to \ldots$

I termini fortemente normalizzabili corrispondono a funzioni *totali*; gli altri termini invece corrispondono a funzioni parziali, con domini più o meno stretti.

Prima di chiudere il capitolo, è interessante fornire un piccolo accenno sulla programmazione delle funzioni ricorsive nel λ-calcolo, cosa che non è a prima vista affatto ovvia.

Anzitutto, poiché le funzioni ricorsive sono funzioni su numeri interi, occorre trovare il corrispettivo dei numeri nel λ-calcolo. La rappresentazione universalmente più diffusa (ma non l'unica possibile) è quella degli *interi di Church*:

$$\begin{aligned} \overline{0} &= \lambda sz.z \\ \overline{n} &= \lambda sz.\underbrace{s(\ldots(s}_{n}\,z)\ldots) \end{aligned}$$

Su tale rappresentazione, si possono definire in modo semplicissimo le proiezioni:

$$\pi_i^n = \lambda x_1 \ldots x_n.x_i$$

e la funzione successore:

$$succ = \lambda nsz.s(nsz)$$

Per verificare il funzionamento di quest'ultima, basta effettuare il calcolo:

$$\begin{aligned} succ\,\overline{n} &\to \lambda sz.s((\lambda sz.\underbrace{s(\ldots(s}_{n}\,z)\ldots))sz) \to \\ &\to \lambda sz.s((\lambda z.\underbrace{s(\ldots(s}_{n}\,z)\ldots))z) \to \\ &\to \lambda sz.s(\underbrace{s(\ldots(s}_{n}\,z)\ldots)) = \overline{n+1} \end{aligned}$$

Naturalmente, si possono definire tutte le altre funzioni elementari sui numeri interi, come l'addizione, la sottrazione, la moltiplicazione, l'elevamento a potenza...

Non senza qualche difficoltà tecnica, è possibile dimostrare che il λ-calcolo è in grado di rappresentare tutti gli schemi di costruzione delle funzioni ricorsive: composizione, ricorsione primitiva, minimizzazione. In sostanza, vale il seguente teorema:

Teorema 2.3 *Se f è una funzione ricorsiva parziale da $\mathbb{N}^k$ a $\mathbb{N}$, allora esiste un λ-termine ϕ tale che:*

- $\phi\,\overline{x_1}\ldots\overline{x_k} \twoheadrightarrow \overline{f(x_1,\ldots,x_k)}$ *se* $f(x_1,\ldots,x_k)\downarrow$

– *$\phi\, \overline{x_1} \ldots \overline{x_k}$ non è normalizzabile se $f(x_1, \ldots, x_k) \uparrow$*

Dimostrazione. Si veda [16]. $\square$

Poiché il λ-calcolo non è altro che un sistema di riscrittura, non è difficile immaginare di programmare una macchina di Turing perché, data in input una qualche rappresentazione di un λ-termine, ne calcoli la forma normale (eventualmente girando all'infinito se questa non esiste). Dunque, ogni funzione λ-definibile (cioè per la quale esiste un λ-termine che la calcola) è anche Turing-calcolabile; per il teorema 2.1, abbiamo "chiuso il cerchio", ed ottenuto l'equivalenza tra i tre modelli computazionali.

La tesi di Church-Turing può dunque essere riformulata come segue:

Proposizione 2.7 (Tesi di Church-Turing, seconda versione) *Una funzione è "calcolabile" sse è ricorsiva sse è λ-definibile sse è Turing-calcolabile.*

2.3.2 Il λ-calcolo tipato semplice

Una delle peculiarità del λ-calcolo puro è che non c'è distinzione tra *dati* e *funzioni*; un termine è semplicemente un termine, e le variabili vincolate tramite la λ-astrazione possono essere sostituite con qualsiasi altro termine, senza alcun vincolo. E' un po' come se, in un linguaggio di programmazione, una stessa funzione accettasse argomenti di qualsiasi natura, siano essi, ad esempio, interi, stringhe, alberi... Il termine tipico che rappresenta questo "polimorfismo" esagerato è il famoso Δ, il cui input è contemporaneamente *funzione* e *argomento* della funzione stessa.

Quello che manca allora nel λ-calcolo puro è la nozione di *tipo*; in effetti, nel λ-calcolo tutti gli oggetti appartengono ad un unico tipo, che è contemporaneamente dato e funzione. E' possibile però definire una variante del λ-calcolo in cui il concetto di tipo sia esplicitamente presente: si tratta del cosiddetto *λ-calcolo tipato*, del quale noi introdurremo una versione ristretta chiamata *λ-calcolo tipato semplice* (STLC, *Simply Typed Lambda-Calculus*).

Occorre prima di tutto definire che cos'è un tipo. I tipi di STLC sono definiti in modo molto simile alle formule della logica:

- I simboli $X, Y, Z \ldots$, appartenenti ad un insieme numerabile di simboli, sono i *tipi atomici* di STLC.
- Se A e B sono due tipi, allora $A \to B$ è un tipo, e sarà il tipo delle funzioni che prendono in input un oggetto di tipo A e restituiscono un oggetto di tipo B.
- Se A e B sono due tipi, allora $A \times B$ è un tipo, e sarà il tipo delle coppie ordinate di oggetti di tipo A e B.

I termini di STLC possono essere definiti in modo induttivo come segue:

(Var) Sia $\mathcal{V} = \{x, y, z, \ldots\}$ un insieme numerabile di simboli di variabile, a ciascuno dei quali è associato un tipo come definito sopra. Allora, $x, y, z, \ldots$ sono termini di STLC, il cui tipo è quello associato al simbolo stesso. Per annotare esplicitamente che il tipo associato a x è A, scriveremo x^A. Per i termini in generale utilizzeremo la notazione $t : A$ per dire che t è un termine di tipo A; ad esempio dunque, $x^A : A$.

(λ) Sia $t : B$ un termine e x una variabile di tipo A. Allora, $\lambda x^A.t$ è un termine di tipo $A \to B$, e le eventuali occorrenze di x in t saranno dette *vincolate*.

(@) Siano $t : A \to B$ e $u : A$ due termini. Allora, tu è un termine di tipo B.

(Pair) Siano $s : A$ e $t : B$ due termini. Allora, $\langle s, t \rangle$ è un termine di tipo $A \times B$.

(Proj) Sia $t : A \times B$ un termine. Allora, $\mathsf{fst}(t)$ e $\mathsf{snd}(t)$ sono due termini rispettivamente di tipo A e B.

Le regole (λ) e (@) dicono rispettivamente come costruire e come utilizzare un termine di tipo $A \to B$, cioè una funzione che da un termine di tipo A ne restituisce uno di tipo B. Le regole (Pair) e (Proj) dicono invece come costruire e utilizzare le coppie di termini, cioè i termini di tipo $A \times B$.

Ci servono ora le regole per la riscrittura dei termini, analoghe alla β_0-conversione del λ-calcolo. Siano $s : A \times B$, $t : B$, $u : A$ e $v : A \to B$; le regole di riduzione sono le seguenti:

$$(\lambda x^A.t)u \rightsquigarrow_0 t[u/x]$$

$$\mathsf{fst}(\langle t, u \rangle) \rightsquigarrow_0 t$$

$$\mathsf{snd}(\langle t, u \rangle) \rightsquigarrow_0 u$$

La prima è praticamente identica alla β_0-conversione del λ-calcolo puro; le altre due invece si occupano di ridurre le coppie di termini. Naturalmente, la riduzione $\rightsquigarrow$ si definisce in modo analogo al λ-calcolo puro, come chiusura riflessiva, transitiva e contestuale di $\rightsquigarrow_0$.

Per il λ-calcolo tipato semplice continua naturalmente a valere la proprietà di Church-Rosser. Tuttavia, a differenza che nel λ-calcolo puro, in STLC tutti i termini sono fortemente normalizzabili; la forma normale si raggiunge sempre, non importa quale strategia di riduzione si scelga.

Intuitivamente, ciò è possibile perché il termine Δ del λ-calcolo puro non è tipabile: un termine non può essere applicato a sé stesso, qualunque sia il suo tipo. Questo ovviamente non dimostra la normalizzazione forte, ma è un elemento chiave perché questa possa verificarsi.

Capitolo 3

La corrispondenza di Curry-Howard

Il presente capitolo è dedicato all'illustrazione del legame logica/informatica, stabilito dalla cosiddetta Corrispondenza di Curry-Howard. *Presenteremo anzitutto la versione "costruttiva" della Logica Classica, ossia la Logica Intuizionista; di seguito, introdurremo le Deduzioni Naturali, gli oggetti sintattici che rappresentano effettivamente le dimostrazioni delle formule della logica. Con l'aiuto di queste, mostreremo come ad ogni dimostrazione possa essere associato un termine del λ-calcolo tipato.*

3.1 La logica intuizionista

3.1.1 Il problema del weakening

Alla luce di quanto visto per il λ-calcolo nella sezione 2.3, l'insieme delle derivazioni di **LK** può essere visto come un sistema di riscrittura in cui le forme normali sono le derivazioni cut-free, e le regole di riscrittura (analoghe alla β-riduzione) sono le regole di riduzione fornite dal Teorema di Cut-Elimination (sezione 1.3). Nello stesso modo in cui nel λ-calcolo si parte da un termine contenente dei redex per arrivare alla forma normale (se essa esiste), in logica classica è possibile prendere una derivazione di **LK** contenente dei tagli e applicare la procedura di cut-elimination fino ad ottenere una derivazione cut-free. Questa semplice intuizione è alla base di tutto ciò che diremo nel seguito; cercheremo infatti di ridurre il processo di cut-elimination ad un processo di calcolo, nell'ambito del quale la derivazione di partenza costituisce un programma da eseguire, e la derivazione cut-free di arrivo rappresenta invece il risultato del calcolo.

La prima differenza fondamentale tra λ-calcolo e logica classica è che quest'ultima non gode della proprietà di Church-Rosser. La ragione del problema è la presenza della regola strutturale del weakening; prendiamo infatti una derivazione di questo tipo:

$$\dfrac{\dfrac{\dfrac{\vdots\ \pi}{\Gamma \vdash \Delta}}{\Gamma \vdash A, \Delta}\,\mathsf{WR} \qquad \dfrac{\dfrac{\vdots\ \pi'}{\Gamma' \vdash \Delta'}}{\Gamma', A \vdash \Delta'}\,\mathsf{WL}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\,\mathsf{cut}$$

L'eliminazione del taglio per questa derivazione ci pone davanti ad una scelta: possiamo prediligere il ramo contenente π, e ridurre la derivazione a

$$\dfrac{\dfrac{\vdots\ \pi}{\Gamma \vdash \Delta}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\,\mathsf{W}$$

oppure prediligere π', e trovare

$$\dfrac{\dfrac{\vdots\ \pi'}{\Gamma' \vdash \Delta'}}{\Gamma, \Gamma' \vdash \Delta, \Delta'}\,\mathsf{W}$$

Entrambe le riduzioni sono perfettamente lecite; tuttavia, esse danno chiaramente luogo a due derivazioni che, a priori, sono totalmente diverse. La procedura di cut-elimination per **LK** dunque *non è confluente*; in prospettiva dell'analogia informatica che vogliamo creare, ciò è molto poco simpatico: significherebbe che i calcoli effettuati non hanno un risultato ben definito, ma ne possono avere più d'uno.

Inoltre, questa volta però in analogia con quanto succede nel λ-calcolo, è possibile trovare derivazioni di **LK** per le quali, applicando una certa "strategia" di riduzione, non si arriva mai ad una forma normale. Dal punto di vista informatico, il Teorema di Cut-Elimination garantisce dunque l'esistenza di un qualche risultato al calcolo effettuato, ma non assicura né che tale risultato sia unico, né che ci si arrivi sempre eliminando i cut in qualsiasi ordine.

3.1.2 La limitazione delle regole strutturali

Il problema del weakening, che in Logica Classica non ha speranza di risoluzione, è invece affrontabile nell'ambito della Logica Intuizionista. La Logica Intuizionista, nata per motivi completamente diversi da quelli per i quali ce ne serviremo noi, consiste in una restrizione della logica classica che, nel calcolo dei sequenti, può essere formulata in modo banale:

Definizione 3.1 (Sequente intuizionista) *Un* sequente intuizionista *è un sequente* $\Gamma \vdash \Delta$ *in cui* $|\Delta| \leq 1$.

Il calcolo dei sequenti intuizionista, che chiameremo **LJ**, si ricava da **LK** semplicemente modificando le regole in modo che esse agiscano esclusivamente su sequenti intuizionisti. Nel nostro caso, partiremo dalla versione proposizionale al secondo ordine di **LK**. Il primo ordine infatti non è di alcuna utilità a livello computazionale, mentre il secondo ordine fornisce un potere espressivo altissimo; essendo la formulazione delle regole per i quantificatori praticamente identica indipendentemente dall'ordine, tanto vale presentare direttamente la versione del calcolo alla quale faremo sempre riferimento in futuro.

Ecco dunque le regole di derivazione della nostra versione di **LJ**; la formula C che compare a destra dei sequenti indica una formula che può esserci o meno:

- **Regole dell'identità**

$$\frac{}{A \vdash A}\,\mathsf{ax} \qquad\qquad \frac{\Gamma \vdash A \qquad A, \Delta \vdash C}{\Gamma, \Delta \vdash C}\,\mathsf{cut}$$

- **Regole strutturali**

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\,\mathsf{X} \qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\,\mathsf{C} \qquad \frac{\Gamma \vdash C}{\Gamma, A \vdash C}\,\mathsf{WL} \qquad \frac{\Gamma \vdash}{\Gamma \vdash A}\,\mathsf{WR}$$

- **Regole logiche**

$$\frac{\Gamma \vdash A}{\Gamma, \neg A \vdash}\,\neg\mathsf{L} \qquad\qquad \frac{\Gamma, A \vdash}{\Gamma \vdash \neg A}\,\neg\mathsf{R}$$

$$\frac{\Gamma \vdash A \qquad B, \Delta \vdash C}{\Gamma, \Delta, A \Rightarrow B \vdash C}\,\Rightarrow\mathsf{L} \qquad\qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\,\Rightarrow\mathsf{R}$$

- **Regole logiche additive**

(nessuna regola sinistra per $\mathcal{T}$) $\qquad \dfrac{}{\Gamma \vdash \mathcal{T}}\,\mathcal{T}\mathsf{aR}$

$$\frac{\Gamma, A \vdash C}{\Gamma, A \wedge B \vdash C}\,\wedge\mathsf{aL1} \qquad \frac{\Gamma, B \vdash C}{\Gamma, A \wedge B \vdash C}\,\wedge\mathsf{aL2} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\,\wedge\mathsf{aR}$$

$\dfrac{}{\Gamma, \mathcal{F} \vdash C}\,\mathcal{F}\mathsf{aL}$ $\qquad$ (nessuna regola destra per $\mathcal{F}$)

$$\frac{\Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma, A \vee B \vdash C}\ \vee\mathsf{aL} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\mathsf{aR1} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\mathsf{aR2}$$

▶ **Regole logiche moltiplicative**

$$\frac{\Gamma \vdash C}{\Gamma, \mathcal{T} \vdash C}\ \mathcal{T}\mathsf{mL} \qquad \frac{}{\vdash \mathcal{T}}\ \mathcal{T}\mathsf{mR}$$

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \wedge B \vdash C}\ \wedge\mathsf{mL} \qquad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B}\ \wedge\mathsf{mR}$$

$$\frac{}{\mathcal{F} \vdash}\ \mathcal{F}\mathsf{mL} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathcal{F}}\ \mathcal{F}\mathsf{mR}$$

(nessuna regola moltiplicativa per $\vee$)

▶ **Regole logiche per i quantificatori**

$$\frac{\Gamma, A[B] \vdash C}{\Gamma, \forall X.A \vdash C}\ \forall\mathsf{L} \qquad \frac{\Gamma \vdash A[X]}{\Gamma \vdash \forall X.A}\ \forall\mathsf{R}\ (\star)$$

$$\frac{\Gamma, A[X] \vdash C}{\Gamma, \exists X.A \vdash C}\ \exists\mathsf{L}\ (\star) \qquad \frac{\Gamma \vdash A[B]}{\Gamma \vdash \exists X.A}\ \exists\mathsf{R}$$

La condizione $(\star)$ è sempre la stessa: la variabile proposizionale X non deve essere libera nel contesto della formula A (Γ nel caso di $\forall\mathsf{R}$ e Γ e C nel caso di $\exists\mathsf{L}$).

Si può evidentemente notare come le regole strutturali abbiano subìto una limitazione in **LJ**; in particolare, la contrazione può operare solo sul lato sinistro del sequente. Osservando l'esempio di riduzione non confluente presentato sopra, ci accorgiamo che se anche il weakening fosse sottoposto alla stessa restrizione, avremmo risolto il problema della non validità della proprietà di Church-Rosser.

In logica intuizionista, il weakening a destra si può applicare esclusivamente nel caso in cui il lato destro del sequente sia vuoto. Da un'ispezione accurata delle regole di **LJ**, emerge chiaramente che esistono solo due modi in cui un sequente può essere "svuotato" del suo lato destro: l'applicazione di una regola $\mathcal{F}\mathsf{mL}$, che crea dal vuoto un sequente con zero formule a destra, oppure l'applicazione di una regola $\neg\mathsf{L}$. Dall'abolizione di queste due regole nasce la cosiddetta Logica Minimalista, i cui sequenti $\Gamma \vdash \Delta$ soddisfano la condizione $|\Delta| = 1$.

La logica minimalista risolve completamente il problema della non confluenza; l'esempio di derivazione dato in precedenza non è più costruibile in **LJ** se non sono mai state utilizzate le due regole $\mathcal{F}\mathsf{mL}$ e $\neg\mathsf{L}$. Nel seguito, anziché introdurre un nuovo calcolo (che potremmo chiamare ad esempio **LM**), faremo sempre riferimento a derivazioni di **LJ** in cui non si utilizzano mai le due regole incriminate. Ciò ci permetterà di confondere logica intuizionista e logica minimalista, riferendoci sempre alla prima pur intendendo in verità la seconda. Tale confusione è ammissibile in ambito informatico, giacché per le regole per la negazione non è ancora stata trovata un'interpretazione computazionale soddisfacente, e la costante $\mathcal{F}$ è totalmente negletta in qualsiasi contesto di programmazione[1].

Diretta conseguenza dell'esclusione delle regole $\mathcal{F}\mathsf{mL}$ e $\neg\mathsf{L}$ è l'eliminazione di altre regole, che diventano a questo punto inutili giacché, a causa della presenza sicura di esattamente una formula a destra, non ci si troverà mai in condizioni di poterle applicare: le regole in questione sono il famigerato WR, la $\neg\mathsf{R}$, e la $\mathcal{F}\mathsf{mR}$, che esclude totalmente la presenza di un "falso moltiplicativo" in logica intuizionista.

Osserviamo infine che, con la logica intuizionista, abbiamo cambiato completamente la semantica delle formule. Esempi sconcertanti sono le due tautologie classiche $\neg A \vee A$ (il *tertium non datur*) e $\neg\neg A \Rightarrow A$, le quali non sono più derivabili in **LJ**. Questo fatto, apparentemente drammatico, non rappresenterà per noi un grosso problema; d'ora in avanti infatti sposteremo sempre di più la nostra attenzione dalle *formule* (e dunque la loro verità e dimostrabilità), alle *dimostrazioni* in quanto tali.

3.1.3 Cut-elimination in LJ

Procederemo ora con la presentazione dettagliata delle regole di riduzione per l'eliminazione del taglio in **LJ**. Tuttavia, opereremo su una versione ancor più ristretta del sistema, quella limitata al cosiddetto *frammento negativo* della logica intuizionista, in cui sono presenti solo i connettivi $\wedge$, $\Rightarrow$ e $\forall$. Le ragioni di tale scelta saranno più chiare nel seguito; per il momento, a supporto di una tale decisione possiamo chiamare in causa il fatto che, al secondo ordine, i connettivi logici $\wedge$, $\vee$ e $\exists$ possono essere espressi in funzione

[1] Ciò è vero per le costanti logiche in generale, ma se per il "vero moltiplicativo" si può vedere qualche utilizzo pratico (in genere nella definizione *naïve* del tipo dei booleani; in **LLL** invece l'unità moltiplicativa gioca un ruolo diverso, alquanto curioso), a quanto ne sappiamo noi il falso non ha mai trovato applicazione alcuna.

dei soli $\Rightarrow$ e $\forall$:

$$\begin{array}{rcl} A \wedge B & \iff & \forall X.((A \Rightarrow (B \Rightarrow X)) \Rightarrow X \\ A \vee B & \iff & \forall X.((A \Rightarrow X) \Rightarrow ((B \Rightarrow X) \Rightarrow X)) \\ \exists X.A & \iff & \forall Y.(\forall X.((A \Rightarrow Y) \Rightarrow Y)) \end{array}$$

Di conseguenza, a livello di espressività logica non si hanno problemi nel limitarsi al frammento negativo. Inoltre, giacché le regole strutturali ce lo consentono, ci sbarazzeremo completamente di tutta la formulazione moltiplicativa, ovverosia delle regole per la congiunzione moltiplicativa e delle regole moltiplicative per $\mathcal{T}$ (le regole moltiplicative per $\mathcal{F}$ erano già state eliminate dalla condizione minimalista). Per questa ragione, non essendoci più motivo di specificarla, nelle derivazioni che seguiranno non riporteremo accanto alle regole la formulazione cui appartengono.

Ecco dunque le 8 regole di riduzione per il frammento negativo di **LJ**:

$$\dfrac{\overset{\vdots\,\pi}{\Gamma \vdash A} \quad \dfrac{}{A \vdash A}\,\mathsf{ax}}{\Gamma \vdash A}\,\mathsf{cut} \quad\longrightarrow\quad \overset{\vdots\,\pi}{\Gamma \vdash A}$$

$$\dfrac{\dfrac{}{A \vdash A}\,\mathsf{ax} \quad \overset{\vdots\,\pi}{\Gamma, A \vdash C}}{\Gamma, A \vdash C}\,\mathsf{cut} \quad\longrightarrow\quad \overset{\vdots\,\pi}{\Gamma, A \vdash C}$$

$$\dfrac{\overset{\vdots\,\pi}{\Gamma \vdash A} \quad \dfrac{\overset{\vdots\,\pi'}{\Gamma', A, A \vdash C}}{\Gamma', A \vdash C}\,\mathsf{C}}{\Gamma, \Gamma' \vdash C}\,\mathsf{cut} \quad\longrightarrow\quad \dfrac{\dfrac{\overset{\vdots\,\pi}{\Gamma \vdash A} \quad \dfrac{\overset{\vdots\,\pi}{\Gamma \vdash A} \quad \overset{\vdots\,\pi'}{\Gamma', A, A \vdash C}}{\Gamma, \Gamma', A \vdash C}\,\mathsf{cut}}{\Gamma, \Gamma, \Gamma' \vdash C}\,\mathsf{cut}}{\Gamma, \Gamma' \vdash C}\,\mathsf{C}$$

$$\dfrac{\overset{\vdots\,\pi}{\Gamma \vdash A} \quad \dfrac{\overset{\vdots\,\pi'}{\Gamma' \vdash C}}{\Gamma', A \vdash C}\,\mathsf{W}}{\Gamma, \Gamma' \vdash C}\,\mathsf{cut} \quad\longrightarrow\quad \dfrac{\overset{\vdots\,\pi'}{\Gamma' \vdash C}}{\Gamma, \Gamma' \vdash C}\,\mathsf{W}$$

$$\dfrac{\dfrac{}{\vdash \mathcal{V}}\,\mathcal{T}\mathsf{R} \quad \dfrac{\overset{\vdots\,\pi}{\Gamma \vdash C}}{\Gamma, \mathcal{V} \vdash C}\,\mathcal{T}\mathsf{L}}{\Gamma \vdash C}\,\mathsf{cut} \quad\longrightarrow\quad \overset{\vdots\,\pi}{\Gamma \vdash C}$$

$$\dfrac{\dfrac{\overset{\vdots\,\pi}{\Gamma, A \vdash B}}{\Gamma \vdash A \Rightarrow B}\,{\Rightarrow}\mathsf{R} \quad \dfrac{\overset{\vdots\,\pi'}{\Gamma' \vdash A} \quad \overset{\vdots\,\pi''}{\Gamma'', B \vdash C}}{\Gamma', \Gamma'', A \Rightarrow B \vdash C}\,{\Rightarrow}\mathsf{L}}{\Gamma, \Gamma', \Gamma'' \vdash C}\,\mathsf{cut} \quad\longrightarrow\quad \dfrac{\dfrac{\overset{\vdots\,\pi'}{\Gamma' \vdash A} \quad \overset{\vdots\,\pi}{\Gamma, A \vdash B}}{\Gamma, \Gamma' \vdash B}\,\mathsf{cut} \quad \overset{\vdots\,\pi''}{\Gamma'', B \vdash C}}{\Gamma, \Gamma', \Gamma'' \vdash C}\,\mathsf{cut}$$

$$\dfrac{\dfrac{\dfrac{\vdots\,\pi_1}{\Gamma\vdash A_1}\quad\dfrac{\vdots\,\pi_2}{\Gamma\vdash A_2}}{\Gamma\vdash A_1\wedge A_2}\wedge\mathsf{R}\qquad\dfrac{\dfrac{\vdots\,\pi'}{\Gamma',A_i\vdash C}}{\Gamma',A_1\wedge A_2\vdash C}\wedge\mathsf{Li}}{\Gamma,\Gamma'\vdash C}\mathsf{cut}\quad\longrightarrow\quad\dfrac{\dfrac{\vdots\,\pi_i}{\Gamma\vdash A_i}\quad\dfrac{\vdots\,\pi'}{\Gamma',A_i\vdash C}}{\Gamma,\Gamma'\vdash C}\mathsf{cut}\qquad i\in 1,2$$

$$\dfrac{\dfrac{\dfrac{\vdots\,\pi}{\Gamma\vdash A[X]}}{\Gamma\vdash\forall X.A}\forall\mathsf{R}\qquad\dfrac{\dfrac{\vdots\,\pi'}{\Gamma',A[B]\vdash C}}{\Gamma',\forall X.A\vdash C}\forall\mathsf{L}}{\Gamma,\Gamma'\vdash C}\mathsf{cut}\quad\longrightarrow\quad\dfrac{\dfrac{\vdots\,\pi[B/X]}{\Gamma\vdash A[B/X]}\qquad\dfrac{\vdots\,\pi'}{\Gamma',A[B]\vdash C}}{\Gamma,\Gamma'\vdash C}\mathsf{cut}$$

Come al solito, nel caso in cui non si possa applicare una di queste regole perché la formula tagliata non è conclusione di entrambe le regole che si trovano prima del cut, si può dimostrare come in **LK** che la regola del taglio commuta con tutte le altre, e dunque i cut possono sempre essere spinti in sù verso le foglie dell'albero, fino ad incontrare le regole che hanno introdotto la formula tagliata.

In base alle regole di riduzione logiche e a quelle commutative, si può allora enunciare la versione dell'Hauptsatz per **LJ**:

Teorema 3.1 (Teorema di Cut-Elimination per LJ) *Se π è una derivazione del sequente $\Gamma \vdash A$ nel frammento negativo di* **LJ**, *allora esiste una procedura che consente di costruire una derivazione π' del medesimo sequente, tale che π' è cut-free.*

Il frammento negativo di **LJ**, con le regole di riduzione presentate, costituisce dunque un sistema di riscrittura nello stile del λ-calcolo. Per esso (e, a dir la verità, per tutto **LJ** in generale) vale la proprietà di confluenza:

Teorema 3.2 (Church-Rosser) *Se π è una derivazione di* **LJ**, *ed esistono due sequenze di regole di riduzione σ e τ tali che $\pi \xrightarrow{\sigma} \pi'$ e $\pi \xrightarrow{\tau} \pi''$, allora esiste una derivazione ρ e due sequenze di riduzione σ' e τ' tali che $\pi' \xrightarrow{\sigma'} \rho$ e $\pi'' \xrightarrow{\tau'} \rho$.*

La confluenza di **LJ**, assieme alla normalizzazione debole garantita dalla procedura di cut-elimination, forniscono la proprietà che cercavamo, e cioè che tutti i processi di calcolo abbiano un unico risultato. Tuttavia, nel corso dell'eliminazione del taglio potrebbero presentarsi contemporaneamente più cut, e dunque più regole di riduzione da poter applicare. Come nel λ-calcolo, potrebbe a priori presentarsi una situazione in cui la forma normale non viene mai raggiunta, e il processo di eliminazione prosegue all'infinito. Il seguente risultato, del quale omettiamo la dimostrazione, serve proprio a stabilire che le scelte che si effettuano nel corso della normalizzazione non hanno rilevanza:

Teorema 3.3 (Normalizzazione forte, Girard) *La procedura di eliminazione del taglio in* **LJ** *è fortemente normalizzante, vale a dire non esistono sequenze di riduzione di lunghezza infinita.*

La normalizzazione forte è un risultato importantissimo; essa ci assicura che qualsiasi strategia di riduzione avrà sempre termine, generando sempre la stessa forma normale.

3.2 Le deduzioni naturali

Il lettore con buona memoria si ricorderà che, quando abbiamo presentato il calcolo dei sequenti per la logica classica, abbiamo precisato che le derivazioni effettuabili al suo interno non sono propriamente *dimostrazioni*; una derivazione di **LK** (o di **LJ**) che si concluda con il sequente $\vdash A$ non è una dimostrazione di A, ma soltanto una prova del fatto che tale dimostrazione esiste, e che essa non utilizza alcuna ipotesi per arrivare ad A. Per quanto concerne la *dimostrabilità*, asserire l'esistenza di una dimostrazione o esibirne una non fa alcuna differenza, e il Teorema di Completezza è lì a confermarcelo. Tuttavia, la logica intuizionista ci offre la possibilità di definire degli oggetti che siano più identificabili a vere dimostrazioni di formule, nei quali sia percorribile in qualche modo il ragionamento che porta dalle ipotesi alla conclusione.

Questi oggetti, per il loro carattere più vicino alla struttura di un ragionamento umano, sono state appunto chiamate *Deduzioni Naturali*. Il calcolo delle deduzioni naturali in realtà esiste sia per la logica classica che per quella intuizionista, e viene chiamato rispettivamente **NK** e **NJ**. Tuttavia, nel caso della logica classica, le deduzioni naturali non riescono ad esprimere le simmetrie profonde che sono invece alla base del calcolo dei sequenti, e il calcolo **NK** non è dunque molto interessante. Al contrario, **NJ** riesce a far funzionare le cose piuttosto bene, fornendo dimostrazioni intuizioniste che emulano il calcolo dei sequenti in modo tutto sommato accettabile. In effetti, per il frammento negativo della logica intuizionista, le deduzioni naturali si comportano in maniera meravigliosa, ed è proprio grazie ad esse che potremo vedere in modo trasparente la corrispondenza di Curry-Howard. Viene così svelato il "mistero" della nostra scelta di limitare la cut-elimination per **LJ** alle sole regole logiche per i connettivi $\Rightarrow$, $\wedge$ e $\forall$: in questo modo potremo ottenere una corrispondenza perfetta con il frammento "buono" di **NJ**.

Cominciamo allora con il definire cos'è una deduzione naturale:

Definizione 3.2 (Pre-deduzione naturale) *Una* pre-deduzione naturale *è un albero i cui nodi, inclusa la radice, sono etichettati con (occorrenze di) formule intuizioniste, mentre le foglie sono etichettate o da formule, o* *da* formule scaricate*, indicate con* $[A]$*, dove* A *è una formula qualunque. La radice dell'albero sarà detta* conclusione *della deduzione; le foglie non scaricate saranno invece dette* ipotesi *della deduzione.*

Non tutti gli alberi costruiti con formule intuizioniste sono vere deduzioni naturali. Per essere tale, l'albero dev'essere stato costruito secondo le regole di **NJ**, che ci accingiamo ad esporre. Nel seguito, un albero T la cui radice è la formula A e le cui foglie sono formule qualsiasi sarà indicato come segue:

$$\begin{array}{c} \vdots\, T \\ A \end{array}$$

Se B è tra le foglie di T, per evidenziarlo scriveremo

$$\begin{array}{c} B \\ \vdots\, T \\ A \end{array}$$

Ecco dunque le regole di **NJ**:

- ▶ **Assioma**

  Se A è una formula qualsiasi, l'albero

  $$A$$

  che rappresenta l'albero la cui radice è contemporaneamente l'unica foglia, è una deduzione naturale.

- ▶ **Eliminazione dell'implicazione**

  Se S è un albero con conclusione A, e T un albero con conclusione $A \Rightarrow B$, allora l'albero la cui conclusione è

  $$\dfrac{A \qquad A \Rightarrow B}{B}\,\mathcal{E}\Rightarrow$$

  è una deduzione naturale.

- ▶ **Introduzione dell'implicazione**

  Se T è un albero con conclusione B, tra le cui ipotesi compare almeno un'occorrenza della formula A non scaricata, l'albero

  $$\dfrac{\begin{array}{c} [A] \\ \vdots\, T \\ B \end{array}}{A \Rightarrow B}\,\mathcal{I}\Rightarrow$$

è una deduzione naturale, nella quale sono state scaricate un numero arbitrario di foglie etichettate con A.

Allo stesso modo, se W è una formula qualsiasi, contenuta o no tra le ipotesi non scaricate di T, anche

$$\dfrac{\begin{array}{c}\vdots\, T\\ B\end{array}}{W \Rightarrow B}\,\mathcal{I}\Rightarrow$$

è una deduzione naturale.

▶ **Eliminazione della congiunzione**

Se T è un albero con conclusione $A \wedge B$, gli alberi con conclusioni

$$\dfrac{A \wedge B}{A}\,\mathcal{E}\wedge 1 \qquad\qquad \dfrac{A \wedge B}{B}\,\mathcal{E}\wedge 2$$

sono deduzioni naturali.

▶ **Introduzione della congiunzione**

Se S è un albero con conclusione A e T un albero con conclusione B, allora l'albero con conclusione

$$\dfrac{A \qquad B}{A \wedge B}\,\mathcal{I}\wedge$$

▶ **Eliminazione del quantificatore universale**

Se T è un albero con conclusione $\forall X.A$, allora l'albero con conclusione

$$\dfrac{\forall X.A}{A[B/X]}\,\mathcal{E}\forall$$

▶ **Introduzione del quantificatore universale**

Se T è un albero con conclusione $A[X]$, e la variabile proposizionale X non ha alcuna occorrenza libera tra le ipotesi non scaricate di T, allora l'albero con conclusione

$$\dfrac{A[X]}{\forall X.A}\,\mathcal{I}\forall$$

è una deduzione naturale.

Definizione 3.3 (Deduzione naturale) *Una* deduzione naturale *è una pre-deduzione naturale costruita secondo le regole fornite sopra.*

E' evidente la similitudine tra tutte le regole di introduzione di **NJ** e le regole destre di **LJ**; in particolare, è chiaro come l'operazione di scaricare un'ipotesi corrisponda all'aver portato la formula corrispondente da sinistra a destra del sequente, mediante la regola $\Rightarrow \mathsf{R}$. Intuitivamente quindi, la conclusione di una deduzione naturale è la formula a destra del sequente intuizionista, mentre le ipotesi non scaricate sono le formule a sinistra del medesimo sequente.

Sulla base di questa considerazione, è possibile costruire una "simulazione" delle regole di **LJ** mediante deduzioni naturali. Consideriamo il caso già accennato dell'implicazione. Supponiamo che in una derivazione di **LJ** ci sia una situazione di questo tipo:

$$\dfrac{\Gamma \vdash A \qquad \Delta, B \vdash C}{\Gamma, \Delta, A \Rightarrow B \vdash C} \Rightarrow \mathsf{L}$$

Quello che sta succedendo in termini di deduzioni naturali può essere spiegato come segue: abbiamo una deduzione naturale le cui ipotesi non scaricate sono le formule di Γ, e la conclusione è A, e un'altra deduzione naturale le cui ipotesi non scaricate sono le formule di Δ e B, e la cui conclusione è C; a partire da queste, costruiamo la deduzione naturale

$$\begin{array}{c}
\dfrac{\begin{array}{c}\Gamma \\ \vdots \\ A\end{array} \qquad A \Rightarrow B}{B, \Delta}\,\mathcal{E}\Rightarrow \\
\vdots \\
C
\end{array}$$

Per contro, se in **LJ** abbiamo

$$\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \Rightarrow \mathsf{R}$$

in **NJ** ciò equivale ad aver applicato esattamente la regola d'introduzione dell'implicazione su una deduzione naturale tra le cui ipotesi c'è A e la cui conclusione è B, scaricando una sola occorrenza di A.

Questa "simulazione" è estendibile a tutto il resto del calcolo. Osserviamo che in **NJ** non c'è però traccia delle regole strutturali; in realtà, esse sono implicite nella regola di introduzione dell'implicazione. La contrazione equivale infatti alla possibilità di poter scaricare un numero arbitrario di occorrenze della stessa formula; in **LJ** sarebbe come effettuare tante contrazioni seguite da una regola per l'implicazione a destra. Il weakening invece è implicito nella possibilità di introdurre l'implicazione su una premessa arbitraria, non contenuta tra le foglie non scaricate della deduzione naturale; di nuovo, in **LJ** ciò equivale appunto a fare un weakening seguito ancora da un'implicazione a destra.

In **NJ** manca anche un'altra cosa essenziale: non c'è una regola esplicita per il cut. Infatti, tagliare due dimostrazioni è in **NJ** un'operazione del tutto naturale, per la quale non serve introdurre una regola. Supponiamo di avere due deduzioni, una, che chiamiamo π, con ipotesi Γ e conclusione A, l'altra, che chiamiamo σ, con ipotesi Δ, A e conclusione C; il semplice "aggancio" di π sopra σ (facendo coincidere la conclusione A dell'una con l'ipotesi A dell'altra) crea un nuovo albero, che è ancora una deduzione naturale, le cui ipotesi sono le formule di Γ e Δ e la cui conclusione è C. A è diventato un nodo qualsiasi, né ipotesi, né conclusione; è proprio quello che succede in **LJ** con l'applicazione di un cut.

La differenza fondamentale tra i cut espliciti di **LJ** e quelli impliciti di **NJ** è che questi ultimi sono sempre "diretti", nel senso che non sono mai oscurati da regole strutturali o da altre regole che non agiscono sulla formula tagliata. Di conseguenza, l'eliminazione del taglio nelle deduzioni naturali è qualcosa di tremendamente più semplice che non nel caso già ampiamente esaminato del calcolo dei sequenti. In particolare, non servono regole di riduzione per weakening e contrazione, e non occorre perdere tempo a definire tutte quelle noiosissime regole di commutazione di cui abbiamo visto un esempio per **LK**.

La situazione è dunque semplicissima; in effetti, si hanno solo tre casi da analizzare, corrispondenti alle tre possibili strutture che può avere la formula su cui si fa l'"aggancio" dei due alberi: $A \Rightarrow B$, $A \wedge B$, $\forall X.A$. Il caso in cui la formula sia atomica non è riducibile; esso corrisponde ad un taglio su un assioma in **LJ**, ma nelle deduzioni naturali tale taglio non è visibile perché, per così dire, è già normalizzato.

Le regole di riduzione sono dunque le seguenti:

$$\dfrac{\begin{array}{c}\Delta\\ \vdots\\ A\end{array} \qquad \dfrac{\begin{array}{c}\Gamma, [A]\\ \vdots\\ B\end{array}}{A \Rightarrow B}\,\mathcal{I}\Rightarrow}{B}\,\mathcal{E}\Rightarrow \qquad \longrightarrow \qquad \begin{array}{c}\Delta\\ \vdots\\ \Gamma, A\\ \vdots\\ B\end{array}$$

$$\cfrac{\cfrac{\begin{array}{c}\Gamma\\ \vdots\\ A_1\end{array} \qquad \begin{array}{c}\Gamma\\ \vdots\\ A_2\end{array}}{A_1 \wedge A_2}\,\mathcal{I}\wedge}{A_i}\,\mathcal{E}\wedge i \qquad \longrightarrow \qquad \begin{array}{c}\Gamma\\ \vdots\\ A_i\end{array} \qquad\qquad i \in \{1,2\}$$

$$\cfrac{\cfrac{\begin{array}{c}\Gamma\\ \vdots\\ A[X]\end{array}}{\forall X.A}\,\mathcal{I}\forall}{A[B/X]}\,\mathcal{E}\forall \qquad \longrightarrow \qquad \begin{array}{c}\Gamma\\ \vdots\, B/X\\ A[B/X]\end{array}$$

Tutte e tre le regole sono caratterizzate dalla presenza di una regola di introduzione seguita immediatamente da una regola di eliminazione. I cut dunque nelle deduzioni naturali assumono forme quantomeno stupide, e la loro riduzione è praticamente ovvia. Solo nella prima regola, quella per l'implicazione, bisogna fare una precisazione: l'albero con ipotesi Δ e conclusione A viene "agganciato" a *tutte* le occorrenze di A che erano state scaricate dalla regola d'introduzione dell'implicazione; le duplicazioni generate da tale riduzione sono dovute alla presenza delle contrazioni implicite che, seppur nascoste, giocano lo stesso ruolo che giocavano nella cut-elimination di **LJ**. Analogamente, se la formula scaricata dall'introduzione dell'implicazione non è tra le foglie dell'albero, significa che essa è stata aggiunta per weakening implicito e quindi, proprio come succedeva nel calcolo dei sequenti, la deduzione con ipotesi Δ e conclusione A viene semplicemente buttata via.

3.3 Formule come tipi e dimostrazioni come programmi

Siamo giunti finalmente alla definizione della corrispondenza di Curry-Howard, che è effettivamente un isomorfismo tra due strutture completamente differenti: le deduzioni naturali da un lato (e dunque la logica intuizionista) e il λ-calcolo tipato semplice dall'altro. La corrispondenza di Curry-Howard dunque costruisce un ponte tra il mondo della logica e quello dell'informatica.

Per il momento, restringeremo ancora di più il frammento di logica intuizionista da prendere in considerazione; in particolare, ci limiteremo ai soli connettivi $\Rightarrow$ e $\wedge$. Il quantificatore universale al secondo ordine verrà reintrodotto nel prossimo capitolo, quando parleremo del Sistema $\mathbb{F}$.

Stabiliamo anzitutto la corrispondenza formule/tipi:

Corrispondenza formule/tipi	
Logica Intuizionista	**λ-calcolo Tipato Semplice**
Formule atomiche (variabili proposizionali): $X, Y, Z, \ldots$	Tipi atomici: $X, Y, Z, \ldots$
$A \Rightarrow B$	$A \rightarrow B$
$A \wedge B$	$A \times B$

La formula $A \Rightarrow B$ corrisponde dunque al tipo dei programmi che prendono in input un oggetto di tipo A e restituiscono un oggetto di tipo B; la formula $A \wedge B$ corrisponde invece al tipo delle coppie di oggetti di tipo A e B.

La seguente invece è la corrispondenza dimostrazioni/programmi:

Corrispondenza dimostrazioni/programmi	
Logica Intuizionista	**λ-calcolo Tipato Semplice**
A	$x^A : A$
$\dfrac{\begin{matrix}[A]\\ \vdots \\ B\end{matrix}}{A \Rightarrow B}\ \mathcal{I} \Rightarrow$	$\lambda x^A.t : A \to B$
$\dfrac{A \qquad A \Rightarrow B}{B}\ \mathcal{E} \Rightarrow$	$ts : B$
$\dfrac{A \qquad B}{A \wedge B}\ \mathcal{I}\wedge$	$\langle s, t\rangle : A \times B$
$\dfrac{A \wedge B}{A}\ \mathcal{E}\wedge 1$	$\mathsf{fst}(t) : A$
$\dfrac{A \wedge B}{B}\ \mathcal{E}\wedge 2$	$\mathsf{snd}(t) : B$

La corrispondenza dimostrazioni/programmi associa a ciascuna deduzione naturale un termine del λ-calcolo tipato semplice; una dimostrazione della formula $A \Rightarrow B$ può dunque essere vista a tutti gli effetti come un programma che fornisce un risultato ti tipo B utilizzando un input di tipo A.

La corrispondenza di Curry-Howard copre in pieno anche gli aspetti "dinamici" dei due sistemi. Nell'ambito delle deduzioni naturali, sappiamo che una sequenza di due regole che introducono e subito dopo eliminano lo stesso connettivo è riscrivibile per mezzo di regole precise in modo da generare un'altra deduzione naturale che non contenga più tale sequenza; nel λ-calcolo tipato semplice abbiamo invece le tre relazioni di riscrittura che trasformano un cosiddetto *redex* in un ben determinato *contractum*. Ebbene, le tre regole di riduzione definite per le deduzioni naturali corrispondono esattamente alle tre relazioni di riscrittura applicate sui λ-termini corrispondenti; in altre parole, se la deduzione π, cui è associato il termine t, si trasforma nella deduzione π', allora a tale deduzione è associato un λ-termine t' tale che t si riduce a t', e le regole di riduzione nei due processi di riscrittura sono in corrispondenza esatta.

Infatti,

$$\dfrac{\begin{array}{c}\vdots\\ A\end{array} \qquad \dfrac{\begin{array}{c}[A]\\ \vdots\\ B\end{array}}{A \Rightarrow B}\,\mathcal{I}\Rightarrow}{B}\,\mathcal{E}\Rightarrow \qquad \longrightarrow \qquad \begin{array}{c}\vdots\\ A\\ \vdots\\ B\end{array}$$

corrisponde alla riduzione

$$(\lambda x^A.t)u \rightsquigarrow t[u/x]$$

E' qui di estrema importanza l'osservazione fatta in precedenza, riguardante l'eventualità che la deduzione con conclusione A sia copiata un numero arbitrario (anche nullo) di volte, corrispondenti al numero di occorrenze di A scaricate dalla regola d'introduzione dell'implicazione. Infatti, nel λ-calcolo, la notazione $t[u/x]$ significa che u viene sostituito a x in *tutte le sue occorrenze precedentemente vincolate dalla λ-astrazione*. Ciò significa che, se x è presente più di una volta, u va copiato più di una volta, mentre se x non è presente affatto, di u non ci sarà più traccia nel contractum. In tutto ciò, emerge chiaramente la corrispondenza tra *scaricare una formula* e *astrarre su una variabile*.

Allo stesso modo,

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ A\end{array} \qquad \begin{array}{c}\vdots\\ B\end{array}}{A \wedge B}\,\mathcal{I}\wedge}{A}\,\mathcal{E}\wedge 1 \qquad \longrightarrow \qquad \begin{array}{c}\vdots\\ A\end{array}$$

corrisponde alla riduzione $\mathsf{fst}(\langle s,t\rangle) \rightsquigarrow s$, mentre

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ A\end{array} \qquad \begin{array}{c}\vdots\\ B\end{array}}{A \wedge B}\,\mathcal{I}\wedge}{B}\,\mathcal{E}\wedge 2 \qquad \longrightarrow \qquad \begin{array}{c}\vdots\\ B\end{array}$$

corrisponde alla riduzione $\mathsf{snd}(\langle s,t\rangle) \rightsquigarrow t$.

Per comprendere meglio il funzionamento dell'isomorfismo, mostriamo un esempio banale ma efficace. Consideriamo la deduzione naturale (che chiameremo κ)

$$\dfrac{\dfrac{[A]}{B \Rightarrow A}\,\mathcal{I}\Rightarrow}{A \Rightarrow (B \Rightarrow A)}\,\mathcal{I}\Rightarrow$$

corrispondente alla dimostrazione di uno dei cosiddetti *assiomi di Hilbert*. Ripercorrendo la deduzione, quello che è stato fatto è più o meno quanto segue: si è partiti dall'ipotizzare A, che è una deduzione naturale cui corrisponde il λ-termine x, dove x è di tipo A. Si è applicata una regola per l'introduzione dell'implicazione, utilizzando come premessa una formula che non è tra le ipotesi; a tale formula corrisponde, ad esempio, la variabile y (di tipo B ovviamente), e dunque il λ-termine corrispondente alla deduzione naturale che abbiamo ottenuto fin qui è l'astrazione su y di x, e cioè $\lambda y^B.x : B \to A$. Infine, abbiamo effettuato un'altra introduzione dell'implicazione, stavolta scaricando l'ipotesi A. A tale ipotesi corrispondeva la variabile x, e dunque il termine risultante, associato all'intera deduzione, è $\lambda x^A.\lambda y^B.x : A \to (B \to A)$. Il tipo del termine corrisponde esattamente alla conclusione della deduzione naturale; il fatto che non ci siano variabili libere nel λ-termine è in corrispondenza invece con il fatto che non ci sono ipotesi nella deduzione (tutte le foglie sono scaricate).

Supponiamo ora di disporre di due derivazioni π e π', con conclusioni rispettivamente A e B, cui corrispondono i λ-termini $t : A$ e $t' : B$. Utilizziamo ora le deduzioni π e π'assieme a κ per costruire la seguente dimostrazione:

$$\cfrac{\begin{matrix}\vdots\,\pi' \\ B\end{matrix} \qquad \cfrac{\begin{matrix}\vdots\,\pi \\ A\end{matrix} \quad \cfrac{\cfrac{[A]}{B \Rightarrow A}\,\mathcal{I}\Rightarrow}{A \Rightarrow (B \Rightarrow A)}\,\mathcal{I}\Rightarrow}{B \Rightarrow A}\,\mathcal{E}\Rightarrow}{A}\,\mathcal{E}\Rightarrow$$

cui è ovviamente associato il termine $((\lambda x^A.\lambda y^B.x)t)t' : A$. Osserviamo ora che quest'ultimo λ-termine contiene un redex, e infatti nella deduzione naturale c'è la corrispondente coppia introduzione/eliminazione. Applichiamo dunque tutte le trasformazioni possibili alla deduzione:

$$\cfrac{\begin{matrix}\vdots\,\pi' \\ B\end{matrix} \qquad \cfrac{\begin{matrix}\vdots\,\pi \\ A\end{matrix} \quad \cfrac{\cfrac{[A]}{B \Rightarrow A}\,\mathcal{I}\Rightarrow}{A \Rightarrow (B \Rightarrow A)}\,\mathcal{I}\Rightarrow}{B \Rightarrow A}\,\mathcal{E}\Rightarrow}{A}\,\mathcal{E}\Rightarrow \quad \longrightarrow \quad \cfrac{\begin{matrix}\vdots\,\pi' \\ B\end{matrix} \quad \cfrac{\begin{matrix}\vdots\,\pi \\ A\end{matrix}}{B \Rightarrow A}\,\mathcal{I}\Rightarrow}{A}\,\mathcal{E}\Rightarrow \quad \longrightarrow \quad \begin{matrix}\vdots\,\pi \\ A\end{matrix}$$

Se a questo punto proviamo ridurre il λ-termine corrispondente alla deduzione iniziale, scopriamo che la sequenza di passi per arrivare alla forma normale è in corrispondenza perfetta con le trasformazioni effettuate:

$$((\lambda x^A.\lambda y^B.x)t)t' \rightsquigarrow (\lambda y^B.t)t' \rightsquigarrow t$$

In effetti, il λ-termine corrispondente a κ è il cosiddetto *combinatore* K che, assieme al combinatore $S := \lambda x^{A\to(B\to C)}.\lambda y^{A\to B}.\lambda z^A.(xz)(yz) : C$, è sufficiente a programmare tutte le funzioni possibili del λ-calcolo tipato semplice, proprio come i due assiomi di Hilbert corrispondenti ai tipi di K ed S ($A \Rightarrow (B \Rightarrow C)$ e $(A \Rightarrow (B \Rightarrow C)) \Rightarrow ((A \Rightarrow B) \Rightarrow (A \Rightarrow C))$) sono sufficienti a derivare (grazie alle regole dell'implicazione) tutte le tautologie della logica intuizionista.

La corrispondenza di Curry-Howard apre una serie incredibile di nuove possibilità, stabilendo un legame fortissimo tra i sistemi logici e i linguaggi di programmazione funzionali. Le due strutture coinvolte nell'isomorfismo sono a priori talmente lontane che sembra quasi impossibile l'esistenza di una corrispondenza così stretta. Un isomorfismo è però tanto più interessante quanto più diverse sono le nature dei due sistemi coinvolti, e in questo caso logica e informatica potrebbero, tramite Curry-Howard, fornirci l'una sull'altra intuizioni molto profonde, come effettivamente è già avvenuto.

3.3.1 Curry-Howard nel calcolo dei sequenti

Torniamo un momento al calcolo dei sequenti. Quando abbiamo introdotto le deduzioni naturali, abbiamo detto che esse possono essere considerate come le dimostrazioni di cui parlano i sequenti di **LJ**; pertanto, **NJ** e **LJ** non sono privi di legami, anzi, quello che si fa in uno dei due calcoli ha una qualche corrispondenza in quello che si fa nell'altro.

Tenendo conto di ciò, ci si può domandare che fine faccia la corrispondenza di Curry-Howard nel calcolo dei sequenti. La risposta è che questa è ancora pienamente valida, anche se le raffinatezze di **LJ**, pur consentendo ancora di vedere le derivazioni come programmi, non permettono di mettere in corrispondenza biunivoca la dinamica dei λ-termini con quella del processo di cut-elimination. In particolare, in genere un passo di riduzione nel λ-calcolo corrisponde a più passi nel calcolo dei sequenti.

Per esplicitare la corrispondenza derivazioni/programmi si può formulare una versione "decorata" di **LJ**, in cui ogni formula di un sequente viene appunto decorata con un termine del λ-calcolo. Il λ-termine che decorerà la formula a destra della conclusione della derivazione sarà proprio il termine associato alla derivazione stessa.

Ecco dunque la versione annotata di **LJ**, naturalmente ristretto al solo frammento $\Rightarrow$ e $\wedge$:

$$\frac{}{x : A \vdash x : A}\,\mathsf{ax} \qquad \frac{\Gamma \vdash s : A \qquad x : A, \Delta \vdash t : C}{\Gamma, \Delta \vdash t[s/x] : C}\,\mathsf{cut}$$

$$\frac{\Gamma, x : A, y : A \vdash t : C}{\Gamma, z : A \vdash t[z/x, z/y] : C}\,\mathsf{C} \qquad \frac{\Gamma \vdash t : C}{\Gamma, x : A \vdash t : C}\,\mathsf{W}$$

$$\frac{\Gamma \vdash s : A \qquad x : B, \Delta \vdash t : C}{\Gamma, \Delta, y : A \Rightarrow B \vdash t[ys/x] : C}\,{\Rightarrow}\mathsf{L} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \Rightarrow B}\,{\Rightarrow}\mathsf{R}$$

$$\frac{\Gamma, x : A \vdash t : C}{\Gamma, y : A \wedge B \vdash t[\mathsf{fst}(y)/x] : C}\,\wedge\mathsf{aL1} \qquad \frac{\Gamma, x : B \vdash t : C}{\Gamma, y : A \wedge B \vdash t[\mathsf{snd}(y)/x] : C}\,\wedge\mathsf{aL2}$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash \langle s, t\rangle : A \wedge B}\,\wedge\mathsf{aR}$$

Naturalmente, i tipi dei termini sono ancora in corrispondenza esatta con le formule; ecco perché abbiamo utilizzato, ad esempio, $A \Rightarrow B$ anziché $A \to B$.

A titolo di esempio, mostriamo la derivazione dell'assioma di Hilbert $A \Rightarrow (B \Rightarrow A)$, corrispondente alla deduzione κ introdotta sopra:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{x : A \vdash x : A}\,\mathsf{ax}}{y : B, x : A \vdash x : A}\,\mathsf{W}}{x : A \vdash \lambda y^B.x : B \Rightarrow A}\,{\Rightarrow}\mathsf{R}}{\vdash \lambda x^A.\lambda y^B.x : A \Rightarrow (B \Rightarrow A)}\,{\Rightarrow}\mathsf{R}$$

In realtà, nel calcolo dei sequenti è sconveniente astrarre tutte le variabili che saranno poi gli "input" del programma rappresentato dalla derivazione. Infatti ad una funzione come questa, che nel λ-calcolo tipato semplice prende il tipo $A \to (B \to A)$, è più pratico associare una derivazione con conclusione $A, B \vdash A$, che corrisponderebbe dunque alla sotto-derivazione comprendente solo la regola dell'assioma e del weakening. In questo modo, è più semplice tagliare gli input della funzione, per poi ridurli mediante le regole della cut-elimination:

$$\dfrac{\begin{array}{c}\vdots\\ \Gamma\vdash A\end{array} \quad \dfrac{\begin{array}{c}\vdots\\ \Delta\vdash B\end{array}\quad \dfrac{\dfrac{}{A\vdash A}\,\mathsf{ax}}{A,B\vdash A}\,\mathsf{W}}{\Delta, A\vdash A}\,\mathsf{cut}}{\Gamma,\Delta\vdash A}\,\mathsf{cut} \quad\rightarrow\quad \dfrac{\begin{array}{c}\vdots\\ \Gamma\vdash A\end{array} \quad \dfrac{\dfrac{}{A\vdash A}\,\mathsf{ax}}{\Delta, A\vdash A}\,\mathsf{W}}{\Gamma,\Delta\vdash A}\,\mathsf{cut} \quad\twoheadrightarrow$$

$$\twoheadrightarrow\quad \dfrac{\dfrac{\begin{array}{c}\vdots\\ \Gamma\vdash A\end{array} \quad \dfrac{}{A\vdash A}\,\mathsf{ax}}{\Gamma\vdash A}\,\mathsf{cut}}{\Gamma,\Delta\vdash A}\,\mathsf{W} \quad\rightarrow\quad \dfrac{\begin{array}{c}\vdots\\ \Gamma\vdash A\end{array}}{\Gamma,\Delta\vdash A}\,\mathsf{W}$$

Le cose dunque continuano a funzionare, ma in modo un po' diverso. In particolare, in **LJ** servono più passi di cut-elimination per arrivare alla formale normale rispetto alle deduzioni naturali. In questo esempio specifico, l'applicazione multipla della regola che scambia weakening e cut (c'è un'applicazione per ciascuna formula di Δ) non ha alcuna corrispondenza in **NJ**. Nel calcolo dei sequenti dunque emergono dinamiche che nelle deduzioni naturali (e quindi nel λ-calcolo) non sono visibili. La corrispondenza formule/tipi dimostrazioni/programmi resta valida, ma non è un vero e proprio isomorfismo; il calcolo dei sequenti è più "fine", e in esso appaiono dettagli assenti nel λ-calcolo.

Per una trattazione dettagliata della corrispondenza di Curry-Howard nell'ambito del calcolo dei sequenti, rimandiamo il lettore interessato a [15].

Capitolo 4

Funzioni rappresentabili nei sistemi logici

In questo capitolo ci occuperemo di "mettere in pratica" l'idea fornita dalla corrispondenza di Curry-Howard, vale a dire il fatto che ogni dimostrazione logica possa essere vista come un programma. In particolare, introdurremo un sistema di assegnazione di tipi molto potente, conosciuto come Sistema $\mathbb{F}$*, e dimostreremo qual è la potenza espressiva di tale sistema, in termini di funzioni rappresentabili al suo interno.*

4.1 Il Sistema $\mathbb{F}$

Il Sistema $\mathbb{F}$ è stato introdotto nel 1971 da Jean-Yves Girard (si veda [13] per un'esposizione dettagliata del sistema), ed è stato più tardi (1974) "riscoperto" individualmente da John Reynolds, che l'ha chiamato λ-*calcolo polimorfico*. Il sistema $\mathbb{F}$ è in sostanza un'estensione di STLC, e si ottiene a partire da esso aggiungendo i quantificatori al secondo ordine.

Ovviamente, per poter parlare di corrispondenza dimostrazioni/programmi, è necessario lavorare in ambito intuizionista, e tale sarà dunque il nostro "ambiente di sviluppo" per la tipizzazione di $\mathbb{F}$. Inoltre, proseguendo con l'approccio del capitolo precedente, considereremo esclusivamente il cosiddetto "frammento negativo" della logica intuizionista, costituito dai connettivi $\Rightarrow$, $\wedge$ e $\forall$ al secondo ordine. In realtà, $\wedge$ non è necessario alla definizione di $\mathbb{F}$; tuttavia, noi lo includeremo perché può rendere la programmazione più semplice. La versione di $\mathbb{F}$ che trattiamo qui sarà dunque quella conosciuta come *Sistema* $\mathbb{F}$ *con prodotti.* Abbiamo inoltre scelto di presentare l'assegnazione dei tipi di $\mathbb{F}$ nel calcolo dei sequenti (dunque in **LJ**), sapendo che, sebbene meno esatta, la corrispondenza di Curry-Howard continua a funzionare come per le deduzioni naturali.

Ecco dunque l'assegnazione dei tipi di $\mathbb{F}$:

$$\frac{}{x : A \vdash x : A}\,\mathsf{ax} \qquad \frac{\Gamma \vdash s : A \qquad x : A, \Delta \vdash t : C}{\Gamma, \Delta \vdash t[s/x] : C}\,\mathsf{cut}$$

$$\frac{\Gamma, x : A, y : A \vdash t : C}{\Gamma, z : A \vdash t[z/x, z/y] : C}\,\mathsf{C} \qquad \frac{\Gamma \vdash t : C}{\Gamma, x : A \vdash t : C}\,\mathsf{W}$$

$$\frac{\Gamma \vdash s : A \qquad x : B, \Delta \vdash t : C}{\Gamma, \Delta, y : A \Rightarrow B \vdash t[ys/x] : C}\,\Rightarrow\!\mathsf{L} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x^A.t : A \Rightarrow B}\,\Rightarrow\!\mathsf{R}$$

$$\frac{\Gamma, x : A \vdash t : C}{\Gamma, y : A \wedge B \vdash t[\mathsf{fst}(y)/x] : C}\,\wedge\mathsf{aL1} \qquad \frac{\Gamma, x : B \vdash t : C}{\Gamma, y : A \wedge B \vdash t[\mathsf{snd}(y)/x] : C}\,\wedge\mathsf{aL2}$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash \langle s, t \rangle : A \wedge B}\,\wedge\mathsf{aR}$$

$$\frac{\Gamma, x : A[B] \vdash t : C}{\Gamma, y : \forall X.A \vdash t[yB/x] : C}\,\forall\mathsf{L} \qquad \frac{\Gamma \vdash t : A[X]}{\Gamma \vdash \Lambda X.t : \forall X.A}\,\forall\mathsf{R} \quad (\star)$$

$(\star)$ è la solita condizione di sempre sull'introduzione a destra del quantificatore universale.

Come è possibile vedere, la tipizzazione è identica a quella di STLC, salvo l'aggiunta della quantificazione al secondo ordine. Andiamo dunque a guardare nel dettaglio le due regole per il $\forall$.

La regola a destra è detta *astrazione universale* del tipo X nel termine t. E' molto simile all'astrazione standard del λ-calcolo, ma invece di astrarre variabili, astrae *tipi*. L'utilizzo dell'astrazione universale è visibile nella regola a sinistra, che corrisponde all'*applicazione universale*. Se nell'applicazione standard del λ-calcolo si applicava un termine ad un altro termine, l'applicazione universale prevede invece di applicare un termine *ad un tipo*.

Per capire meglio come funzioni la quantificazione al secondo ordine, e sufficiente osservare la regola di riduzione per questo nuovo costrutto sintattico, che si aggiunge alle altre tre regole già conosciute, che riportiamo per comodità:

$$(\lambda x^A.t)u \rightsquigarrow t[u/x]$$

$$\mathsf{fst}(\langle t, u\rangle) \rightsquigarrow t$$

$$\mathsf{snd}(\langle t, u\rangle) \rightsquigarrow u$$

$$(\Lambda X.t)A \rightsquigarrow t[A/X]$$

In altre parole, applicare un termine della forma $\Lambda X.t$ ad un tipo corrisponde a sostituire quel tipo ovunque occorra la variabile proposizionale X in t. Ecco perché l'altro nome del sistema $\mathbb{F}$ è λ-calcolo polimorfico: grazie alla quantificazione al secondo ordine, il tipo di un temine può essere *parametrizzato*, e deciso esattamente solo a *runtime*, proprio come nei linguaggi funzionali polimorfici come ML.

Forniremo nel seguito qualche esempio per ottenere un minimo di familiarizzazione con la programmazione in $\mathbb{F}$. Prima di cominciare, facciamo qualche precisazione sulla notazione:

· La convenzione per la sintassi dei λ-termini sarà esattamente la stessa utilizzata per il λ-calcolo puro (si veda 2.3.1).

· Per quanto riguarda le formule, il connettivo $\wedge$ avrà la precedenza su $\Rightarrow$, e dunque scriveremo, ad esempio, $A \wedge B \Rightarrow C \wedge D$ al posto di $(A \wedge B) \Rightarrow (C \wedge D)$.

· Il connettivo $\Rightarrow$ sarà associativo a destra: con la formula $A \Rightarrow B \Rightarrow C$ si intenderà la formula $A \Rightarrow (B \Rightarrow C)$.

· Un quantificatore al secondo ordine posto davanti ad una formula legherà tutte le istanze della variabile quantificata presenti nella formula, vale a dire che $\forall X.X \wedge Y \Rightarrow X$ sarà equivalente a $\forall X.((X \wedge Y) \Rightarrow X)$.

· Nelle derivazioni del calcolo dei sequenti, come al solito una linea di derivazione più spessa indicherà un certo numero (maggiore di uno) di istanze della stessa regola applicate consecutivamente.

Partiamo dalla rappresentazione dei numeri, che è quella consueta degli interi di Church. Posto, per il generico tipo A,

$$\mathsf{int}_A := (A \Rightarrow A) \Rightarrow A \Rightarrow A$$

il tipo degli interi in $\mathbb{F}$ sarà

$$\mathsf{int} := \forall X.\mathsf{int}_X$$

Della formula int esistono infatti in $\mathbb{F}$ *infinite derivazioni*, ciascuna associabile ad un numero naturale, e ciascuna "magicamente" (grazie all'assegnazione dei tipi fornita in precedenza) corrispondente ad un intero di Church. Per quanto riguarda $\overline{0}$, abbiamo infatti:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{z : X \vdash z : X}\,\mathsf{ax}}{\vdash (\lambda z.z) : X \Rightarrow X}\,{\Rightarrow}\mathsf{R}}{s : X \Rightarrow X \vdash (\lambda z^{X}.z) : X \Rightarrow X}\,\mathsf{W}}{\vdash (\lambda s^{X\Rightarrow X} z^{X}.z) : \mathsf{int}_X}\,{\Rightarrow}\mathsf{R}}{\vdash (\Lambda X.\lambda s^{X\Rightarrow X} z^{X}.z) : \mathsf{int}}\,\forall\mathsf{R}$$

Per quanto riguarda il generico $\overline{n}$, abbiamo invece

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{z : X \vdash z : X}\,\mathsf{ax} \qquad \begin{array}{c} \dfrac{\dfrac{}{x_1 : X \vdash x_1 : X}\,\mathsf{ax} \quad \dfrac{}{y : X \vdash y : X}\,\mathsf{ax}}{x_1 : X, s_1 : X \Rightarrow X \vdash (s_1 x_1) : X}\,{\Rightarrow}\mathsf{L} \\ \vdots \\ x_{n-1} : X, s_{n-1} : X \Rightarrow X, \ldots, s_1 : X \Rightarrow X \vdash (s_1(\ldots(s_{n-1}x_{n-1})\ldots)) : X \end{array}}{z : X, s_n : X \Rightarrow X, \ldots, s_1 : X \Rightarrow X \vdash (s_1(\ldots(s_n z)\ldots)) : X}\,{\Rightarrow}\mathsf{L}}{s_n : X \Rightarrow X, \ldots, s_1 : X \Rightarrow X \vdash (\lambda z^{X}.s_1(\ldots(s_n z)\ldots)) : X \Rightarrow X}\,{\Rightarrow}\mathsf{R}}{s : X \Rightarrow X \vdash (\lambda z^{X}.s(\ldots(sz)\ldots)) : X \Rightarrow X}\,\mathsf{C}}{\vdash (\lambda s^{X\Rightarrow X} z^{X}.s(\ldots(sz)\ldots)) : \mathsf{int}_X}\,{\Rightarrow}\mathsf{R}}{\vdash (\Lambda X.\lambda s^{X\Rightarrow X} z^{X}.\underbrace{s(\ldots(s}_{n}\,z)\ldots)) : \mathsf{int}}\,\forall\mathsf{R}$$

Come è evidente, i λ-termini associati alle derivazioni mediante le regole di tipizzazione corrispondono esattamente alla versione tipata degli interi di Church.

Per non appesantire troppo la notazione, presenteremo le derivazione che programma il successore sugli interi in versione "non decorata", vale a dire con i soli tipi (formule), senza riportare in modo esplicito i termini corrispondenti:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X \Rightarrow X \vdash X \Rightarrow X}\,\mathsf{ax} \qquad \dfrac{\dfrac{}{X \vdash X}\,\mathsf{ax} \quad \dfrac{\dfrac{}{X \vdash X}\,\mathsf{ax} \quad \dfrac{}{X \vdash X}\,\mathsf{ax}}{X, X \Rightarrow X \vdash X}\,{\Rightarrow}\mathsf{L}}{X \Rightarrow X, X, X \Rightarrow X \vdash X}\,{\Rightarrow}\mathsf{L}}{\mathsf{int}_X, X, X \Rightarrow X, X \Rightarrow X \vdash X}\,{\Rightarrow}\mathsf{L}}{\mathsf{int}_X, X, X \Rightarrow X \vdash X}\,\mathsf{C}}{\mathsf{int}, X, X \Rightarrow X \vdash X}\,\forall\mathsf{L}}{\mathsf{int} \vdash \mathsf{int}_X}\,{\Rightarrow}\mathsf{R}}{\mathsf{int} \vdash \mathsf{int}}\,\forall\mathsf{R}$$

Il λ-termine associato alla derivazione sopra è

$$(\Lambda X.\lambda s^{X\Rightarrow X}z^X.s(nXsz)) : \mathsf{int}$$

dove $n : \mathsf{int}$ è la variabile associata alla formula int a sinistra del sequente che conclude la derivazione. Aggiungendo una regola $\Rightarrow \mathsf{R}$, otterremmo un termine di tipo $\mathsf{int} \Rightarrow \mathsf{int}$, cioè una funzione da interi a interi, che è il giusto tipo per la funzione successore. Tuttavia, sappiamo che nel calcolo dei sequenti è più "comodo" lasciare le formule che rappresentano gli input a sinistra del sequente, in modo da poterle tagliare direttamente con gli argomenti della funzione.

Per controllare che il nostro successore funzioni davvero, effettuiamo il calcolo nel caso in cui l'input sia $\overline{0}$:

$$\begin{array}{rcl}(\lambda n^{\mathsf{int}}\Lambda X.\lambda s^{X\Rightarrow X}z^X.s(nXsz))\overline{0} & \rightsquigarrow & \Lambda X.\lambda s^{X\Rightarrow X}z^X.s((\Lambda X.\lambda s^{X\Rightarrow X}z^X.z)Xsz) \\ & \rightsquigarrow & \Lambda X.\lambda s^{X\Rightarrow X}z^X.s((\lambda s^{X\Rightarrow X}z^X.z)sz) \\ & \rightsquigarrow & \Lambda X.\lambda s^{X\Rightarrow X}z^X.sz = \overline{1}\end{array}$$

E' chiaro come l'esecuzione ricalchi esattamente quella già vista nel caso non tipato, e dunque ci risparmiamo di dimostrare che il λ-termine funzioni nel caso del generico input $\overline{n}$.

Il tipo degli interi non è il solo a poter essere rappresentato in $\mathbb{F}$. In pratica, tutti i tipi induttivi di un normale linguaggio di programmazione funzionale trovano posto nel sistema. Ad esempio, se poniamo, essendo A una formula qualsiasi,

$$\mathsf{bool}_A := A \wedge A \Rightarrow A$$

il tipo dei booleani è rappresentato dalla formula

$$\mathsf{bool} := \forall X.\mathsf{bool}_X$$

La formula bool ha due possibili derivazioni in $\mathbb{F}$, che possiamo far corrispondere (arbitrariamente) una al valore booleano *true*, l'altra a *false*:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{y : X \vdash y : X}\,\mathsf{ax}}{x : X \wedge X \vdash (\mathsf{fst}(x)) : X}\,\wedge\mathsf{aL1}}{\vdash (\lambda x^{X\wedge X}.\mathsf{fst}(x)) : \mathsf{bool}_X}\,\Rightarrow\mathsf{R}}{\vdash (\Lambda X.\lambda x^{X\wedge X}.\mathsf{fst}(x)) : \mathsf{bool}}\,\forall\mathsf{R} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{y : X \vdash y : X}\,\mathsf{ax}}{x : X \wedge X \vdash (\mathsf{snd}(x)) : X}\,\wedge\mathsf{aL2}}{\vdash (\lambda x^{X\wedge X}.\mathsf{snd}(x)) : \mathsf{bool}_X}\,\Rightarrow\mathsf{R}}{\vdash (\Lambda X.\lambda x^{X\wedge X}.\mathsf{snd}(x)) : \mathsf{bool}}\,\forall\mathsf{R}$$

Possiamo scegliere, ad esempio, di prendere la prima come *true* e la seconda come $false$. Grazie ai booleani, possiamo ora rappresentare costrutti come l'"if... then... else":

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{t:Y\vdash t:Y}\,\mathsf{ax}}{t:Y,e:Y\vdash t:Y}\,\mathsf{W}\qquad\dfrac{\dfrac{}{e:Y\vdash e:Y}\,\mathsf{ax}}{t:Y,e:Y\vdash t:Y}\,\mathsf{W}}{t:Y,e:Y\vdash\langle t,e\rangle:Y\wedge Y}\,\wedge\mathsf{aR}\qquad\dfrac{}{x:Y\vdash x:Y}\,\mathsf{ax}}{t:Y,e:T,y:\mathsf{bool}_Y\vdash(y\langle t,e\rangle):Y}\,\Rightarrow\mathsf{L}}{t:Y,e:T,b:\mathsf{bool}\vdash(bY\langle t,e\rangle):Y}\,\forall\mathsf{L}}{\vdash(\lambda b^{\mathsf{bool}}t^Y e^Y.bY\langle t,e\rangle):\mathsf{bool}\Rightarrow Y\Rightarrow Y\Rightarrow Y}\,\Rightarrow\mathsf{R}}{\vdash(\Lambda Y.\lambda b^{\mathsf{bool}}t^Y e^Y.bY\langle t,e\rangle):\forall Y.\mathsf{bool}\Rightarrow Y\Rightarrow Y\Rightarrow Y}\,\forall\mathsf{R}
$$

Quella che abbiamo programmato è una funzione polimorfica che agisce in questo modo: preso in input un tipo qualsiasi A, la funzione attende tre input ulteriori, il primo di tipo bool e gli altri due di tipo A; a seconda del valore booleano dato in ingresso, viene restituito uno dei due termini di tipo A.

Vediamo il tutto con un esempio. Supponiamo di disporre di due termini di tipo A, siano essi u e v. Sia inoltre $p : \mathsf{bool}$ un termine la cui forma normale (essendo booleano) è o $true$ o $false$. Possiamo allora calcolare

$$
\begin{aligned}
(\Lambda Y.\lambda b^{\mathsf{bool}}t^Y e^Y.bY\langle t,e\rangle)Apuv &\rightsquigarrow (\lambda b^{\mathsf{bool}}t^A e^A.bA\langle t,e\rangle)puv\\
&\rightsquigarrow pA\langle u,v\rangle
\end{aligned}
$$

Ora, se $p \rightsquigarrow true$, abbiamo

$$(\Lambda X.\lambda x^{X\wedge X}.\mathsf{fst}(x))A\langle u,v\rangle \rightsquigarrow (\lambda x^{A\wedge A}.\mathsf{fst}(x))\langle u,v\rangle \rightsquigarrow \mathsf{fst}(\langle u,v\rangle) \rightsquigarrow u$$

se, al contrario, $p \rightsquigarrow false$, si ha

$$(\Lambda X.\lambda x^{X\wedge X}.\mathsf{snd}(x))A\langle u,v\rangle \rightsquigarrow (\lambda x^{A\wedge A}.\mathsf{snd}(x))\langle u,v\rangle \rightsquigarrow \mathsf{snd}(\langle u,v\rangle) \rightsquigarrow v$$

che è il risultato atteso da una funzione if... then... else come quella da noi programmata.

Un altro tipo interessante è quello delle liste di tipo generico Y, la cui formula corrispondente è

$$\mathsf{list} := \forall Y.\forall X.(Y \Rightarrow X \Rightarrow X) \Rightarrow X \Rightarrow X$$

Istanziando list su una particolare formula A, otteniamo il tipo delle liste di termini di tipo A:

$$\mathsf{list}^A := \forall X.(A \Rightarrow X \Rightarrow X) \Rightarrow X \Rightarrow X$$

Senza fornire le derivazioni corrispondenti, mostriamo di seguito alcune costanti e funzioni di base sulle liste; ad esempio, la lista vuota è rappresentata dal termine

$$nil := (\Lambda Y.\Lambda X.\lambda c^{Y\Rightarrow X\Rightarrow X} n^X.n) : \mathsf{list}$$

mentre, se $t_1, \ldots, t_k$ sono k termini di tipo A, la lista che li contiene sarà

$$(\Lambda X.\lambda c^{A\Rightarrow X\Rightarrow X} n^X.ct_1(\ldots(ct_k n)\ldots)) : \mathsf{list}^A$$

La funzione *cons*, che aggiunge un elemento in testa ad una lista, è rappresentata dal termine

$$cons := (\Lambda Y.\lambda t^Y l^{\mathsf{list}}.\Lambda X.\lambda c^{Y\Rightarrow X\Rightarrow X} n^X.ct(lYXcn)) : \forall Y.Y \Rightarrow \mathsf{list} \Rightarrow \mathsf{list}^Y$$

Come al solito, per testare gli esempi possiamo provare a fornire in input alla funzione *cons* l'intero di Church $\overline{2}$ e la lista vuota, e aspettarci dunque in uscita la lista di interi contenente $\overline{2}$ come unico elemento:

$$\begin{array}{lll}
cons\ \mathsf{int}\ \overline{2}\ nil & \rightsquigarrow & (\lambda t^{\mathsf{int}} l^{\mathsf{list}}.\Lambda X.\lambda c^{\mathsf{int}\Rightarrow X\Rightarrow X} n^X.ct(l\mathsf{int}Xcn))\ \overline{2}\ nil \\
& \rightsquigarrow & \lambda l^{\mathsf{list}}.\Lambda X.\lambda c^{\mathsf{int}\Rightarrow X\Rightarrow X} n^X.c\overline{2}(l\mathsf{int}Xcn))\ nil \\
& \rightsquigarrow & \Lambda X.\lambda c^{\mathsf{int}\Rightarrow X\Rightarrow X} n^X.c\overline{2}((\Lambda Y.\Lambda X.\lambda c^{Y\Rightarrow X\Rightarrow X} n^X.n)\mathsf{int}Xcn) \\
& \rightsquigarrow & \Lambda X.\lambda c^{\mathsf{int}\Rightarrow X\Rightarrow X} n^X.c\overline{2}((\Lambda X.\lambda c^{\mathsf{int}\Rightarrow X\Rightarrow X} n^X.n)Xcn) \\
& \rightsquigarrow & \Lambda X.\lambda c^{\mathsf{int}\Rightarrow X\Rightarrow X} n^X.c\overline{2}((\lambda c^{\mathsf{int}\Rightarrow X\Rightarrow X} n^X.n)cn) \\
& \rightsquigarrow & \Lambda X.\lambda c^{\mathsf{int}\Rightarrow X\Rightarrow X} n^X.c\overline{2}((\lambda n^X.n)n) \\
& \rightsquigarrow & (\Lambda X.\lambda c^{\mathsf{int}\Rightarrow X\Rightarrow X} n^X.c\overline{2}n) : \mathsf{list}^{\mathsf{int}}
\end{array}$$

Per quanto riguarda gli esempi di programmazione, ci fermeremo qui, anche se è chiaro che le possibilità offerte dal sistema $\mathbb{F}$ sono quasi illimitate. Piuttosto, menzioneremo il seguente risultato fondamentale, che estende il risultato a cui si è fatto cenno nell'introdurre il λ-calcolo tipato semplice:

Teorema 4.1 (Normalizzazione forte di $\mathbb{F}$) *Se t è un termine del λ-calcolo tipabile in $\mathbb{F}$ (cioè al quale corrisponde una derivazione del sistema $\mathbb{F}$), allora tutte le strategie di normalizzazione terminano.*

Dimostrazione. Si veda [13]. $\square$

Il teorema di normalizzazione forte sostanzialmente dice che tutti le derivazioni di $\mathbb{F}$ rappresentano programmi *ben costruiti*; non è possibile che un programma di $\mathbb{F}$ contenga un ciclo infinito. Naturalmente, valendo sempre la confluenza, il risultato di tutte le strategie di normalizzazione è anche unico.

Nella prossima sezione, ci prenderemo cura di descrivere esattamente il potere espressivo di $\mathbb{F}$, stabilendo quali siano effettivamente le funzioni che vi possono trovare posto.

4.2 Teoremi di rappresentabilità

Vogliamo ora descrivere, nel modo più preciso possibile, quali funzioni sono rappresentabili all'interno di $\mathbb{F}$, possibilmente in termini di un sottoinsieme delle funzioni ricorsive.

Cominciamo con un'osservazione: grazie alla normalizzazione forte, sappiamo che tutte le derivazioni di $\mathbb{F}$ possono essere ridotte a derivazioni cut-free in un numero finito di passi, indipendentemente dalla strategia di cut-elimination scelta. Ciò significa che un programma fornisce un risultato *qualunque sia il suo input*. Di conseguenza, è chiaro che in $\mathbb{F}$ non potremo rappresentare che funzioni *totali*; una funzione parziale non ha infatti alcuna speranza di trovare posto in $\mathbb{F}$, giacché una derivazione che la rappresenti dovrebbe ammettere almeno un input che, tagliato con tale derivazione, dia luogo ad una sequenza di normalizzazione infinita.

Per contro, poiché il calcolo della normalizzazione di una derivazione (o il termine ad essa associato) di $\mathbb{F}$ è chiaramente una procedura effettiva per calcolare il valore di una certa funzione per un certo input, non c'è ombra di dubbio sul fatto che le funzioni rappresentate in $\mathbb{F}$ siano *ricorsive*.

Possiamo allora essere tentati di dire che $\mathbb{F}$ sia in grado di rappresentare tutte e sole le funzioni ricorsive totali. Tuttavia, il seguente argomento[1] mostra chiaramente che non è questo il caso:

Teorema 4.2 *Esiste una funzione ricorsiva totale non rappresentabile in* $\mathbb{F}$.

Dimostrazione. Supponiamo che sia rappresentabile, in $\mathbb{F}$, una funzione totale in grado di enumerare le funzioni ad un argomento rappresentabili nello stesso sistema $\mathbb{F}$. Tale funzione, che chiamiamo Φ, sarà di arità 2; avremo cioè, se $f_0, f_1, f_2, \dots$ sono le funzioni unarie rappresentabili in $\mathbb{F}$ (chiaramente numerabili):

$$\Phi(i,n) = f_i(n) \qquad \forall i,n \in \mathbb{N}$$

Consideriamo ora la funzione definita nel seguente modo:

$$\varphi(n) = \Phi(n,n) + 1 \qquad \forall n \in \mathbb{N}$$

φ è chiaramente ricorsiva totale, ed è per di più unaria. Ora, se essa non è rappresentabile in $\mathbb{F}$, abbiamo concluso; al contrario, se essa è rappresenta-

[1] Come si potrà notare, quello utilizzato è chiaramente un argomento diagonale; abbiamo così colto l'occasione per utilizzare questa tecnica almeno una volta nella tesi, sembrandoci davvero un peccato lasciar fuori dall'esposizione uno dei metodi dimostrativi più simpatici della logica e dell'informatica.

bile, allora esisterà un certo intero m tale che

$$\Phi(m, n) = \varphi(n) \qquad \forall n \in \mathbb{N}$$

Se ora proviamo a calcolare φ in m, otteniamo, per definizione

$$\varphi(m) = \Phi(m, m) + 1$$

ma, poiché m è proprio l'indice di φ nell'enumerazione, abbiamo anche

$$\varphi(m) = \Phi(m, m)$$

che è un'evidente contraddizione. L'ipotesi che Φ fosse rappresentabile ha condotto ad un assurdo, e dunque Φ è un esempio di funzione totale non rappresentabile in $\mathbb{F}$. $\square$

Il sistema $\mathbb{F}$ è dunque un po' meno potente di quanto avessimo inizialmente sperato. Esso però soddisfa un'importante *proprietà di correttezza*, che enunciamo per il tipo degli interi, che è quello d'interesse in questo momento:

Proposizione 4.1 *Un termine chiuso e in forma normale t è tipabile con la formula $\mathsf{int} = \forall X.(X \Rightarrow X) \Rightarrow X \Rightarrow X$ se e solo se questo è un intero di Church $\overline{n}$ per qualche $n \in \mathbb{N}$.*

Dimostrazione. Si veda [13][2]. $\square$

La proposizione sopra è importantissima: in pratica, ci garantisce che, se t è un temine di tipo $\mathsf{int} \Rightarrow \mathsf{int}$ (ovvero una derivazione τ del sequente $\mathsf{int} \vdash \mathsf{int}$), allora la normalizzazione del termine $t\,\overline{n}$ (cioè l'eliminazione del taglio dalla derivazione costruita tagliando τ con la derivazione del sequente $\vdash \mathsf{int}$ associata all'intero di Church $\overline{n}$) produce un termine che è senz'altro anch'esso un intero di Church; le funzioni programmabili in $\mathbb{F}$ sono dunque davvero funzioni "da interi a interi".

Per arrivare a parlare del potere espressivo del nostro sistema, occorre introdurre la nozione di funzione *dimostrabilmente totale*[3]. Nel seguito, se $\mathbf{A}$ è un

[2] In realtà, c'è un piccolo dettaglio tecnico da sottolineare: nella versione enunciata, il teorema vale se per il tipo degli interi si è scelta la formula $\forall X.X \Rightarrow (X \Rightarrow X) \Rightarrow X$, anziché la nostra int; ciò comporta che, in questa notazione, gli interi di Church hanno le due variabili nel λ "scambiate": nel caso di $\overline{2}$, si ha ad esempio $\Lambda X.\lambda z^X s^{X \Rightarrow X}.s(sz)$ anziché il nostro familiare $\Lambda X.\lambda s^{X \Rightarrow X} z^X.s(sz)$. Con la nostra formula, il teorema continua a valere, ma esiste in più anche il termine $\Lambda X.\lambda x^{X \Rightarrow X}.x$ che è chiuso, normale e di tipo int, pur non essendo nella forma di un intero di Church. Questa è una delle piccole imperfezioni della sintassi; nell'ambito di una semantica estensionale, l'intruso in questione avrebbe la stessa interpretazione di $\overline{1}$, poiché è η-equivalente ad esso; non avendo nemmeno definito cosa sia l'η-equivalenza nel λ-calcolo, preferiamo dimenticarci di questa storia e far finta di nulla.

[3] Non abbiamo trovato una traduzione migliore del termine *provably total*...

sistema di assiomi (anche detto *teoria assiomatica*), scrivendo $\mathbf{A} \vdash F$ intenderemo che la formula F è dimostrabile (derivabile nel calcolo dei sequenti) a partire dagli assiomi di $\mathbf{A}$.

Definizione 4.1 (Funzione dimostrabilmente totale) *Sia* $\mathbf{A}$ *una teoria assiomatica, in cui sia rappresentabile, mediante un termine del linguaggio di* $\mathbf{A}$*, la funzione*

$$\mathsf{fun}(i)$$

che per ogni intero non negativo i fornisce il "codice" dell'i-esima funzione ricorsiva (le funzioni ricorsive sono chiaramente numerabili, dunque è possibile "indicizzarle" in qualche modo). Sia inoltre rappresentabile nel linguaggio di $\mathbf{A}$ *la formula*

$$\mathsf{Def}_1[f, m, n]$$

il cui significato è il seguente: la funzione il cui codice è f *è* definita *in* m*, e il suo valore è* n.
Diremo allora che la funzione f_i *è* dimostrabilmente totale *nel sistema* $\mathbf{A}$ *se e solo se*

$$\mathbf{A} \vdash \forall m.\exists n.\mathsf{Def}_1[\mathsf{fun}(i)/f]$$

Introduciamo ora una teoria assiomatica famosissima in ambito logico, la cosiddetta *aritmetica di Peano*:

Definizione 4.2 (Aritmetica di Peano al primo ordine) *Sia* $\mathcal{L}_{\mathbf{PA}}$ *il linguaggio al primo ordine contenente:*

- *il simbolo di costante 0*
- *il simbolo di funzione unario* s *e i simboli di funzione binari (che utilizzeremo in notazione infissa)* $+$ *e* $\cdot$
- *il simbolo di predicato* $=$*, la sua negazione* $\neq$ *(che utilizzeremo entrambi in notazione infissa)*

Il sistema $\mathbf{PA}$ *è costituito dai seguenti assiomi, dove* A *è una qualsiasi formula del linguaggio* $\mathcal{L}_{\mathbf{PA}}$ *contenente una variabile libera:*

1. $\forall x.s(x) \neq 0$
2. $\forall x.\forall y.(x = y \Rightarrow A[x] = A[y])$
3. $\forall x.\,(A[0] \Rightarrow \forall y.\,(A[y] \Rightarrow A[s(y)]) \Rightarrow A[x])$
4. $\forall x.x + 0 = x$
5. $\forall x.\forall y.x + s(y) = s(x + y)$
6. $\forall x.x \cdot 0 = 0$

7. $\forall x.\forall y.x \cdot s(y) = x + s(x \cdot y)$

In realtà, noi non avremo bisogno proprio di **PA**, ma di qualcosa di più. Utilizzeremo ora per la prima volta un linguaggio più complesso di quelli utilizzati fin qui (cui si è brevemente accennato in 1.1), in cui compariranno contemporaneamente sia i quantificatori al primo che al secondo ordine; in un qualsiasi linguaggio del genere, possiamo definire la seguente formula "parametrica", che chiamiamo *schema di comprensione*:

$$\exists X.\forall x_1.\dots\forall x_n.\,(X(x_1,\dots,x_n) \Leftrightarrow F)$$

dove X è una variabile di relazione n-aria, F è una formula qualunque, e ovviamente $A \Leftrightarrow B$ è un'abbreviazione di $(A \Rightarrow B) \wedge (B \Rightarrow A)$.

Possiamo allora finalmente definire il sistema di cui abbiamo bisogno:

Definizione 4.3 (Aritmetica di Peano al secondo ordine) *Sia $\mathcal{L}_{\mathbf{PA}_2}$ l'estensione al secondo ordine di $\mathcal{L}_{\mathbf{PA}}$, e sia X una variabile di relazione unaria. Il sistema $\mathbf{PA}_2$, le cui formule sono costruite sul linguaggio $\mathcal{L}_{\mathbf{PA}_2}$, è costituito dagli assiomi derivanti dallo schema di comprensione e dai medesimi assiomi di* **PA**, *tranne che per i numero 2 e 3, che vengono sostituiti dai seguenti:*

2b. $\forall X.\forall x.\forall y.\,(x = y \Rightarrow X(x) = X(y))$

3b. $\forall X.\forall x.\,(X(0) \Rightarrow \forall y.\,(X(y) \Rightarrow X\,(s(y))) \Rightarrow X(x))$

E' possibile dimostrare che sia **PA** che $\mathbf{PA}_2$ contengono la loro rappresentazione del predicato Def_1 definito in precedenza; abbiamo allora il seguente risultato:

Teorema 4.3 (Espressività di $\mathbb{F}$) *Una funzione è rappresentabile nel Sistema $\mathbb{F}$ se e solo se essa è dimostrabilmente totale in $\mathbf{PA}_2$.*

Dimostrazione. Si veda [13]. $\square$

Detto in parole più semplici, il potere espressivo di $\mathbb{F}$ è mostruoso; in esso è possibile programmare più funzioni di quante non ne siano effettivamente calcolabili in pratica. Con questo, intendiamo che, dal punto di vista della complessità computazionale (che sarà l'oggetto del prossimo capitolo), il sistema $\mathbb{F}$ offre la possibilità di programmare funzioni il cui calcolo richiede tempi fuori dalla portata di qualsiasi essere umano o, addirittura, superiori alla vita operativa di qualsiasi computer che potrà mai essere costruito.

Il sistema $\mathbb{F}$ non è l'unico sistema logico a poter essere visto come un "linguaggio di programmazione". In particolare, insistendo sui linguaggi che ammettono la presenza contemporanea del primo e secondo ordine di quantificazione, esiste un sistema conosciuto come *Aritmetica Funzionale al secondo ordine*. L'aritmetica funzionale al secondo ordine, abbreviata di solito con $\mathbf{AF}_2$, è un sistema introdotto negli anni '80 dal logico francese Jean-Louis Krivine, che in sostanza costituisce un'estensione del sistema $\mathbb{F}$, nel quale viene reintrodotta la quantificazione al primo ordine e, conseguentemente, dai simboli proposizionali si passa ai *simboli di relazione* n-ari.

In $\mathbf{AF}_2$ ci sono inoltre, come nell'aritmetica di Peano, il simbolo di costante 0 e il simbolo di funzione unario s (che rappresenta come al solito il successore), nonché i predicati di uguaglianza e disuguaglianza. In $\mathbf{AF}_2$, il tipo degli interi diventa una formula aperta, contenente una variabile libera (al primo ordine):

$$\mathsf{nat}[x] := \forall X.(X(0) \Rightarrow \forall y.(X(y) \Rightarrow X(s(y))) \Rightarrow X(x))$$

Come è evidente, $\mathsf{nat}[x]$ non è altro che l'assioma che rappresenta il principio di induzione in $\mathbf{PA}_2$, cui è stata tolta la quantificazione sulla variabile x. Se con $s^n 0$ abbreviamo il termine

$$\underbrace{s(\ldots s}_{n}(0)\ldots)$$

abbiamo che la rappresentazione in $\mathbf{AF}_2$ del generico intero n prende il tipo $\mathsf{nat}[s^n 0]$. In particolare, gli interi di Church in $\mathbf{AF}_2$ hanno la seguente forma, non lontana da quella del sistema $\mathbb{F}$:

$$\begin{array}{l} \overline{0} := (\lambda zs.z) : \mathsf{nat}[0] \\ \overline{n} := (\lambda zs.\underbrace{s(\ldots(s}_{n}\,z)\ldots)) : \mathsf{nat}[s^n 0] \end{array}$$

Osserviamo che nella sintassi del λ-calcolo di $\mathbf{AF}_2$ viene omessa la tipizzazione; naturalmente è possibile fare ciò anche in $\mathbb{F}$.

L'aritmetica funzionale al secondo ordine ha il medesimo potere espressivo di $\mathbb{F}$; tuttavia, oltre alla confluenza, alla normalizzazione forte e alla proprietà di correttezza sugli interi già menzionata per $\mathbb{F}$, essa gode anche di una proprietà più forte, che è la seguente:

Teorema 4.4 (Correttezza dei programmi di $\mathbf{AF}_2$) *Sia $\mathcal{E}$ è un sistema di assiomi consistente su $\mathbb{N}$ (vale a dire $\mathcal{E} \vdash s^m 0 = s^n 0 \Rightarrow m = n$).*
Se

$$\mathcal{E} + \mathbf{AF}_2 \vdash t : \mathsf{nat}[x] \Rightarrow \mathsf{nat}[f(x)]$$

allora t è un programma *per f, vale a dire*

$$t\,\overline{m} \twoheadrightarrow \overline{n} \quad \textit{sse} \quad \mathcal{E} \vdash f(s^m 0) = s^n 0$$

Dimostrazione. Si veda [16]. □

La proprietà di correttezza appena introdotta è qualcosa di fenomenale: essa asserisce che, in $\mathbf{AF}_2$, una derivazione che rappresenta una funzione da interi ad interi è automaticamente una *dimostrazione di correttezza* del programma rappresentato da quella derivazione. In altre parole, $\mathbf{AF}_2$ è un linguaggio di programmazione in cui non si possono fare "errori": la correttezza di un programma è sempre garantita.

Capitolo 5

La complessità computazionale

Questo capitolo è dedicato alla presentazione dell'argomento centrale della trattazione, vale a dire la complessità computazionale. *Introdurremo la complessità secondo la definizione canonica, che utilizza le macchine di Turing, e definiremo le classi di complessità più importanti. In seguito mostreremo come sia possibile fornire delle caratterizzazioni assiomatiche di tali classi di complessità, indipendenti dal particolare modello di calcolo. Infine analizzeremo la procedura di cut-elimination dal punto di vista della complessità computazionale, cercando di stabilire l'esistenza di limiti superiori alla lunghezza di una riduzione.*

5.1 Definizione di complessità e classi di complessità

La complessità computazionale è uno dei campi più importanti dell'informatica teorica. Sviluppatasi in modo formale soltanto a partire dagli anni '70, essa si occupa essenzialmente di studiare il legame tra le risorse che un algoritmo utilizza per risolvere un problema e la dimensione della rappresentazione delle istanze del problema stesso.

Intuitivamente, le risorse fondamentali di un algoritmo sono due: tempo e spazio. In termini di calcolatori elettronici, queste si possono tradurre concretamente, ad esempio, in numero di cicli di clock e quantità di memoria necessari all'esecuzione del programma che implementa l'algoritmo. In genere, gli algoritmi hanno più di un input possibile; possiamo dunque immaginare di poter caratterizzare in qualche modo la dimensione di ciascuna istanza di input, in termini ad esempio di memoria occupata per rappresentarla. Inoltre, praticamente tutti gli algoritmi che servono a risolvere problemi di un qualche interesse pratico sono caratterizzati dal fatto, peraltro più che ragionevole, che più aumenta la dimensione dell'input, più aumentano le risorse necessarie per processarlo. L'obiettivo dell'analisi della complessità computazionale di un algoritmo è dunque questo: stabilire formalmente quanto aumentano le risorse utilizzate all'aumentare della dimensione dell'input, esprimendo tale legame per mezzo di una ben precisa funzione matematica.

In realtà, è possibile estendere l'ambito del discorso non solo agli algoritmi, ma ai *problemi* stessi e alle *funzioni* in generale. L'oggetto di studio diventa quindi la ricerca di limiti inferiori e superiori alla quantità di risorse necessarie alla risoluzione di un problema, trovando quindi quali sono gli algoritmi migliori e peggiori per risolverlo. Allo stesso modo, dai problemi si può generalizzare alle funzioni, giacché ogni calcolo della soluzione di un problema può essere visto come il calcolo del valore di un'opportuna funzione.

E' chiaro però che la complessità computazionale di un problema o di una funzione, così come l'abbiamo informalmente definita, dipende da come si misurano sia le risorse che la dimensione dell'input; in sostanza, essa dipende in un certo senso dall'*hardware* su cui facciamo girare il nostro algoritmo. Del resto, noi calcoliamo da secoli le moltiplicazioni a mano, e dunque nulla ci vieta di analizzare la complessità computazionale della funzione "prodotto" prendendo come dimensione dell'input la lunghezza in base 10 dei due numeri, come risorsa tempo il numero di moltiplicazioni "elementari" e somme da dover effettuare e come risorsa spazio la porzione di foglio occupata dal calcolo e i milligrammi di grafite o di inchiostro impiegati per portarlo avanti...

Qualunque definizione dunque è ammissibile, purché si precisi l'*hardware* utilizzato e purché il tutto sia matematicamente ben fondato. Tradizionalmente, la scelta per una formalizzazione rigorosa della complessità computazionale è ricaduta sulle macchine di Turing; queste infatti soddisfano il requisito di essere manipolabili a livello matematico e, soprattutto, rispetto a tutti gli altri principali modelli di calcolo (quelli presentati nel capitolo 2), forniscono una caratterizzazione estremamente minimalista delle risorse computazionali: il tempo è rappresentato dai *passi di calcolo*, la cui idiozia fa quasi spavento, e lo spazio dalle *celle di nastro* utilizzate; è difficile concepire qualcosa di ancor più elementare. In particolare, utilizzeremo in questa sede le macchine di Turing multinastro, le quali consentono in generale di implementare gli algoritmi in modo più semplice.

La misura della dimensione dell'input sarà dunque il numero di celle di nastro occupate dalla sua rappresentazione al momento in cui la macchina viene avviata. Dobbiamo poi definire le funzioni che legano tale misura al numero di passi impiegati e al numero di celle occupate nel corso del calcolo, i quali caratterizzeranno rispettivamente la complessità temporale e spaziale del calcolo stesso. A priori, qualsiasi funzione ricorsiva (da interi non negativi a interi non negativi) può andar bene. Tuttavia, la mancanza di restrizioni alle funzioni utilizzabili come bound di complessità porta a conseguenze molto spiacevoli a livello teorico. Di conseguenza, è opportuno dare una definizione precisa delle funzioni che verranno considerate valide per rappresentare un legame dimensione-input/tempo o dimensione-input/spazio. Nel seguito, utilizzeremo la notazione "O grande" con il solito significato:

Definizione 5.1 *Siano f e g due funzioni da interi non negativi a interi non negativi; se esiste un intero non negativo m e una costante positiva c tale che*

$$f(n) \leq c \cdot g(n) \quad \textit{per ogni } n \geq m$$

allora diremo che f è dell'ordine di *g, e scriveremo*

$$f(n) = O(g(n))$$

La definizione che segue è tratta da [19]:

Definizione 5.2 (Funzione di complessità propria) *Sia f una funzione da interi non negativi a interi non negativi. Diremo che f è una* funzione di complessità propria *se f è monotòna non decrescente (cioè $f(n+1) \geq f(n)$ per ogni n) e inoltre è verificata la seguente condizione: esiste una macchina di Turing a k nastri M_f, il cui alfabeto contiene almeno il simbolo 1, tale che, per qualunque input x di lunghezza n, M_f termina dopo $O(n + f(n))$ passi i k nastri sono occupati da: l'input x, $k - 2$ stringhe costituite da una sequenza di $O(f(n))$ simboli 1 e una stringa costituita da una sequenza di 1 di lunghezza esattamente $f(n)$.*

In sostanza, dato l'input x, se $|x|$ denota la lunghezza di tale input, M_f calcola la stringa $1^{f(|x|)}$, e lo fa in tempo $O(|x| + f(|x|))$, utilizzando uno spazio $O(f(|x|))$ a parte l'input stesso.

La definizione di funzione di complessità propria serve a togliere di mezzo alcune funzioni "patologiche" che genererebbero fenomeni molto poco realistici (ad esempio, si potrebbero definire funzioni in grado di far collassare classi di complessità completamente diverse — la definizione esatta di "classe di complessità" sarà data fra breve). In effetti dunque, tutte le funzioni di interesse generale sono funzioni di complessità proprie: le funzioni costanti, i polinomi, la funzione $\lceil \log n \rceil$, le funzioni "miste" come $n \log n$, gli esponenziali, e anche le funzioni come $\sqrt{n}$ e $n!$ rientrano tutte nella definizione data sopra.

Possiamo a questo punto definire le *classi di complessità*:

Definizione 5.3 *Sia f una funzione di complessità propria. Denoteremo con*

$$\mathbf{TIME}(f)$$

$$\mathbf{SPACE}(f)$$

l'insieme dei problemi risolvibili da una macchina di Turing deterministica rispettivamente in tempo e spazio $O(f(n))$, dove n è la misura della rappresentazione dell'input. Con

$$\mathbf{NTIME}(f)$$

$$\mathbf{NSPACE}(f)$$

denoteremo invece l'insieme dei problemi risolvibili da una macchina di Turing non deterministica rispettivamente in tempo e spazio $O(f(n))$, dove n è sempre la misura dell'input. Allo stesso modo, parleremo delle classi $\mathbf{FTIME}(f)$, $\mathbf{FSPACE}(f)$ *(tempo e spazio deterministici),* $\mathbf{FNTIME}(f)$ *e* $\mathbf{FNSPACE}(f)$ *(tempo e spazio non deterministici) quando considereremo il caso più generale delle funzioni anziché dei problemi.*

E' chiaro che qualunque classe di complessità in ambito di funzioni include strettamente la corrispondente classe in ambito di problemi; poiché però quest'ultimo tipo di classe è largamente più noto, nel seguito utilizzeremo sempre la dicitura, ad esempio, **TIME** pur riferendoci a **FTIME**.

Le classi di complessità più note e studiate sono, ad esempio:

$$\mathbf{P} = \mathbf{PTIME} = \bigcup_{k>0} \mathbf{TIME}(n^k)$$

e

$$\mathbf{NP} = \mathbf{NPTIME} = \bigcup_{k>0} \mathbf{NTIME}(n^k)$$

che rappresentano la classe dei problemi (o funzioni) calcolabili in tempo polinomiale rispettivamente da un algoritmo deterministico o non deterministico. Ci sono poi:

$$\mathbf{PSPACE} = \bigcup_{k>0} \mathbf{SPACE}(n^k)$$

$$\mathbf{NPSPACE} = \bigcup_{k>0} \mathbf{NSPACE}(n^k)$$

$$\mathbf{EXP} = \mathbf{EXPTIME} = \bigcup_{k>0} \mathbf{TIME}(2^{n^k})$$

$$\mathbf{NEXP} = \mathbf{NEXPTIME} = \bigcup_{k>0} \mathbf{NTIME}(2^{n^k})$$

e, in ambito di spazio, le classi sub-lineari

$$\mathbf{L} = \mathbf{LOGSPACE} = \mathbf{SPACE}(\log n)$$

e la sua controparte non deterministica

$$\mathbf{NL} = \mathbf{NLOGSPACE} = \mathbf{NSPACE}(\log n)$$

C'è poi un'altra classe poco considerata per le applicazione pratiche, ma di grande interesse teorico, che è la classe delle cosiddette *funzioni elementari.* Sia $tow(b, k, n)$ la funzione così definita:

$$tow(b, k, n) = \begin{cases} n & \text{se } k = 0 \\ b^{tow(k-1,n)} & \text{se } k > 0 \end{cases}$$

In sostanza, $tow(b, k, n)$ è una *torre di esponenziali* di altezza k; in particolare,

$$tow(2, k, n) = 2^{2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}}$$

con k iterazioni dell'esponenziale. La classe delle funzioni elementari è dunque

$$\mathbf{ELEMENTARY} = \bigcup_{k>0} tow(2, k, n)$$

Le funzioni elementari richiedono tempi mostruosi per essere calcolate anche su dimensioni risibili dell'input; il contesto in cui tale classe fu definita era quello della ricorsività, e ciò giustifica un termine apparentemente molto poco appropriato come "elementare". Naturalmente, esistono funzioni ricorsive definibili in modo molto semplice che non sono neppure elementari.

Quando si parla di funzioni elementari, non ha più importanza la distinzione tra determinismo e non-determinismo, e non ha più senso neanche definire limiti di complessità spaziale. Per quanto riguarda le altre classi, il rapporto fra non determinismo e non determinismo è al contrario uno dei problemi aperti più importanti della teoria della complessità computazionale; la famosa equivalenza $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ è l'esponente più illustre di tale problema, ma anche le equivalenze come $\mathbf{L} \stackrel{?}{=} \mathbf{NL}$ e $\mathbf{EXP} \stackrel{?}{=} \mathbf{NEXP}$ sono altrettanto lontane dall'essere verificate o smentite.

5.2 Caratterizzazioni assiomatiche di classi di complessità

Come abbiamo visto, lo studio della complessità computazionale di un problema o di una funzione si effettua considerando gli algoritmi implementati per mezzo di macchine di Turing. Questa è una scelta che, sebbene ben giustificata, porta a favorire in modo arbitrario un modello di calcolo sopra tutti gli altri; così facendo, se si vuole stabilire un risultato di complessità computazionale nell'ambito di un altro modello, si deve necessariamente definire una qualche *traduzione* che rapporti il tutto alle macchine di Turing.

A partire dagli anni '60 si è cominciata a considerare la possibilità di astrarre la definizione di complessità dalle macchine di Turing, fornendo delle caratterizzazioni puramente assiomatiche delle classi di complessità. L'approccio è identico a quello della teoria della ricorsione; l'obiettivo è catturare una certa classe di complessità definendola in termini di un certo numero di funzioni di base che, combinate per mezzo di una serie di schemi, generino tutte le funzioni appartenenti alla classe stessa. Il vantaggio è grandissimo: la definizione di classe di complessità rimane quella canonica, basata sulle macchine di Turing, ma è allo stesso tempo possibile rapportare tutti gli altri modelli di computazione ad uno schema completamente astratto, come quello delle funzioni ricorsive. Si ha in pratica a disposizione una definizione *machine-independent* delle classi di complessità.

La prima caratterizzazione assiomatica ad essere definita è quella di Kalmar (vedi [21]), che si occupa di definire la classe delle funzioni elementari:

Definizione 5.4 *La classe* **ELEMENTARY** *è il più piccolo insieme contenente le seguenti funzioni di base:*

- *funzioni costanti*
- *proiezioni*
- *addizione*
- *moltiplicazione*
- *sottrazione (o predicato di uguaglianza)*

e chiusa rispetto a

- *composizione*
- *somme e prodotti limitati*

In seguito, si è riusciti a fornire caratterizzazioni assiomatiche per altre classi di complessità, a partire dalla caratterizzazione di **P** fornita da Cobham in [6]:

Definizione 5.5 *Sia* $|\cdot|$ *la funzione così definita:*

$$|n| = \begin{cases} 1 & se\ n = 0 \\ \lfloor \log_2 n \rfloor + 1 & se\ n > 0 \end{cases}$$

La classe **P** *è il più piccolo insieme contenente le seguenti funzioni di base:*

- *funzione costante* 0
- *proiezioni*
- *successori binari s_0 e s_1, definiti come segue:*

$$s_i(n) = 2n + i \quad i \in \{0, 1\}$$

- *"smash function"*

$$\sigma(m, n) = 2^{|m| \cdot |n|}$$

e chiuso rispetto a

- *composizione*
- *ricorsione limitata: se h_i ($i \in \{0,1\}$), g e u sono in* **P**, *allora la funzione f definita con*

$$\begin{array}{rcl} f(0, \overline{n}) & = & g(\overline{n}) \\ f(m, \overline{n}) & = & \begin{cases} h_0(\lfloor \frac{m}{2} \rfloor, \overline{n}, f(\lfloor \frac{m}{2} \rfloor, \overline{n})) & \textit{se } m \bmod 2 = 0 \\ h_1(\lfloor \frac{m}{2} \rfloor, \overline{n}, f(\lfloor \frac{m}{2} \rfloor, \overline{n})) & \textit{se } m \bmod 2 = 1 \end{cases} \qquad \textit{con } m \neq 0 \end{array}$$

  è anch'essa in **P** *se e soltanto se*

$$f(m, \overline{n}) \leq u(m, \overline{n}) \quad \textit{per ogni } m, \overline{n}$$

Come si vede, la caratterizzazione di Cobham è un po' particolare perché lavora sugli interi nella loro rappresentazione binaria; del resto, quando si parla di complessità in genere si scarta a priori la rappresentazione unaria, perché troppo sconveniente. Tuttavia, questa non è la sola peculiarità della definizione. Lo schema di ricorsione prevede infatti una postilla alquanto scomoda, che è quella di dover verificare che la funzione che si sta definendo sia *in ogni punto* limitata da una funzione che è già nella classe. Questo tra l'altro è l'unico motivo per cui quella strana *smash function* viene introdotta tra le funzioni di base; senza di questa non si potrebbe definire nulla per ricorsione.

Recentemente (all'inizio degli anni '90), Bellantoni e Cook [4] hanno presentato una caratterizzazione della classe **P** che evita i problemi della definizione di Cobham. L'intuizione chiave di Bellantoni e Cook è quella di separare gli argomenti delle funzioni in *argomenti normali* e *argomenti safe*; intuitivamente, questi ultimi possono influire sul risultato solo in modo lineare, mentre per gli altri non è previsto alcun vincolo. La limitazione al tempo polinomiale si ottiene quindi mediante la *safe recursion*[1], che permette di iterare le funzioni solamente passandole in argomento *safe*.

[1] L'articolo originale parla di *predicative recursion on notation*, ma in seguito è stata adottata una varietà di altri nomi, tra i quali anche *ricorsione ramificata*, nonché quello utilizzato da noi.

Le funzioni definite sono dunque nella forma

$$f(x_1, \ldots, x_m; a_1, \ldots, a_n)$$

dove $x_1, \ldots, x_m$ sono gli argomenti normali e $a_1, \ldots, a_n$ quelli *safe*. La classe delle funzioni calcolabili in tempo polinomiale corrisponderà al sottoinsieme costituito dalle funzioni a soli argomenti normali, del tipo $f(x_1, \ldots, x_m;)$, costruite nel modo seguente:

Definizione 5.6 *La classe* **P** *è il più piccolo insieme contenente le seguenti funzioni base:*

- *funzione costante 0*
- *proiezioni:*

$$\pi_j^{m,n}(x_1, \ldots, x_m; x_{m+1}, \ldots, x_{m+n}) = x_j \quad 1 \leq j \leq m+n$$

- *successori binari:*

$$s_i(;a) = 2a+1 \quad i \in \{0,1\}$$

- *predecessore:*

$$p(;a) = \left\lfloor \frac{a}{2} \right\rfloor$$

- *condizionale:*

$$C(;a,b,c) = \begin{cases} b & \textit{se } a \bmod 2 = 0 \\ c & \textit{altrimenti} \end{cases}$$

e chiuso rispetto a

- *safe recursion: se h_i ($i \in \{0,1\}$) e g sono in* **P**, *allora*

$$\begin{array}{rcl} f(0,\overline{x};\overline{a}) & = & g(\overline{x};\overline{a}) \\ f(y,\overline{x};\overline{a}) & = & \begin{cases} h_0(\lfloor \frac{y}{2} \rfloor, \overline{x}; \overline{a}, f(\lfloor \frac{y}{2} \rfloor, \overline{x}; \overline{a})) & \textit{se } y \bmod 2 = 0 \\ h_1(\lfloor \frac{y}{2} \rfloor, \overline{x}; \overline{a}, f(\lfloor \frac{y}{2} \rfloor, \overline{x}; \overline{a})) & \textit{altrimenti} \end{cases} \qquad y \neq 0 \end{array}$$

  è anch'essa in **P**

- *safe composition: se h, $r_1, \ldots, r_p$ e $t_1, \ldots, t_q$ sono in* ***P***, *allora*

$$f(\overline{x};\overline{a}) = h(r_1(\overline{x};), \ldots, r_p(\overline{x};); t_1(\overline{x};\overline{a}), \ldots, t_q(\overline{x};\overline{a}))$$

  è in **P**.

Osserviamo che, poiché le funzioni polinomiali saranno alla fine quelle a soli argomenti normali, deve essere possibile costruire delle versioni non-safe delle funzioni di base, alcune delle quali operano invece solo su argomenti "sicuri". Ciò è possibile grazie alla composizione; nel caso del predecessore, possiamo ad esempio definire la funzione

$$p'(x;) = p(;\pi_1^{1,0}(x;))$$

e in modo analogo si possono costruire le versioni normali dei successori e del condizionale. Il fatto fondamentale però è che il contrario non è possibile; una funzione ad un argomento normale non può essere trasformata in una funzione ad un argomento safe, naturalmente tale che questa calcoli ancora la stessa cosa. Ecco perché le funzioni di base lavorano su argomenti safe; definire una funzione safe è qualcosa di più forte che definirla in modo normale.

La safe recursion ha permesso di fornire definizioni assiomatiche per altre importanti classi di complessità, come ad esempio **PSPACE**. Al momento attuale, la caratterizzazione di Bellantoni-Cook è un argomento centrale di ricerca nel campo della logica. Si sta tentando infatti di trovare l'interpretazione della safe recursion nell'ambito dei sistemi logici, che al momento sembrano non riuscire ad esprimere uno schema di questo tipo.

5.3 La complessità della cut-elimination

Nei capitoli 3 e 4 abbiamo visto come, tramite l'isomorfismo di Curry-Howard, esista una corrispondenza tra dimostrazioni e λ-calcolo, e come sia possibile dunque rappresentare e calcolare funzioni all'interno dei sistemi logici. Ora che abbiamo introdotto il concetto di complessità computazionale, è indubbiamente d'interesse studiare i sistemi logici anche da questo punto di vista. Quello che ci si può domandare è se esiste un limite superiore alla complessità delle funzioni (e quindi dei programmi) definibili mediante la logica; poiché l'esecuzione di un programma in logica corrisponde all'eliminazione dei tagli dalle derivazioni, la questione si sposta immediatamente in un ambito più generale, che è quello di analizzare la complessità della procedura di cut-elimination.

In questo senso, lavorando quindi strettamente nel campo della teoria delle dimostrazioni, senza alcun riferimento esplicito all'informatica, si può cercare un legame tra la dimensione che una derivazione ha prima dell'inizio della cut-elimination e la dimensione di una delle sue forme normali[2], tentando

[2] A livello di teoria delle dimostrazioni non ha importanza che le derivazioni rappresentino programmi. L'analisi dunque può essere fatta nel caso più generale possibile, che è quello della logica classica, nel quale sappiamo che la procedura di cut-elimination non è confluente.

di stabilire degli upper bound.

Per il calcolo dei sequenti classico non è complicatissimo arrivare a dimostrare l'esistenza di un upper bound iperesponenziale. Anzitutto, occorre definire una qualche misura della complessità di una derivazione di **LK**. A tal scopo sceglieremo qualcosa di molto intuitivo, vale a dire l'altezza dell'albero di derivazione:

Definizione 5.7 *L'*altezza *di una derivazione π di* **LK**, *che denoteremo con $h(\pi)$ è il numero di regole che compaiono nel ramo più alto dell'albero di π.*

Possiamo ora dimostrare la seguente proposizione:

Proposizione 5.1 (Upper-bound iperesponenziale) *Applicando la procedura di cut-elimination ad una derivazione π di* **LK**, *tale che $h(\pi) = \eta$ e tale che δ è il grado massimo dei cut presenti in π, si ottiene una derivazione cut-free σ la cui altezza è limitata dalla seguente maggiorazione:*

$$h(\sigma) \leq tow(k, \delta, \eta)$$

dove $k \geq 2$ e tow è la funzione definita al termine della sezione 5.1.

Dimostrazione. In 1.3 abbiamo visto che l'eliminazione del taglio consiste nell'iterare una procedura in grado di normalizzare derivazioni quasi-cut-free (definizione 1.16). Tale procedura è descritta nel cruciale lemma 1.1, per il quale si può fare la seguente analisi di complessità. Pur non avendo mostrato esplicitamente tutte le regole di riduzione, non è difficile convincersi che le trasformazioni indotte dalle riduzioni strutturali sono le più costose in termini di aumento dell'altezza della derivazione. In particolare, la contrazione aggiunge alla fine un numero di nuove contrazioni pari al numero di formule presenti nel contesto della formula tagliata (vedi pagina 34). Il numero di formule è senz'altro limitato dall'altezza dell'albero di derivazione; questa particolare regola dunque, alla peggio, duplica l'altezza dell'albero. Abbiamo dunque che l'applicazione di una qualsiasi regola di riduzione strutturale causa un aumento lineare dell'altezza. Se π' è la derivazione dopo l'applicazione di una tale regola, si ha dunque:

$$h(\pi') \leq k\eta$$

per qualche k intero fissato. Le regole di riduzione logiche e commutative non comportano invece aumenti significativi dell'altezza; al contrario, in qualche caso questa si riduce.

La dimostrazione del lemma 1.1 in sostanza itera l'applicazione delle regole di riduzione strutturali e commutative fino ad arrivare a creare un cut in cui la formula tagliata è stata appena introdotta in entrambe le premesse. A quel punto, si applica una regola di riduzione logica e si fa diminuire il grado di 1. La "migrazione" dei cut verso le regole che introducono le occorrenze della formula tagliata costa al massimo un aumento lineare per ogni scambio; alla peggio, si faranno proprio η scambi (ricordiamo che η è l'altezza originale dell'albero), dunque, se π'' è la sotto-derivazione quasi-cut-free in cui il grado del cut è stato abbassato di un'unità, avremo

$$h(\pi'') \leq k^{\eta}$$

Abbassare il grado dei cut ha dunque un costo esponenziale; poiché occorre far scendere il grado fino a zero, tale operazione esponenziale sarà iterata al massimo un numero di volte pari al grado del cut di grado maggiore, che è δ. Otteniamo allora il bound desiderato:

$$h(\sigma) \leq tow(k, \delta, \eta)$$

□

L'esatto valore di k nel bound fornito dipende da come vengono definite le regole di riduzione: in [13], viene posto $k = 4$; in [5], $k = 2$ per **LK** e $k = 4$ per **LJ**. Si osservi che, ad ogni modo, pur essendo ciascuna delle funzioni $tow(k, \delta, \eta)$ elementare in η, poiché il grado massimale di una derivazione non può essere maggiorato in generale, non esiste una funzione elementare che limiti in assoluto l'esplosione dell'altezza di una derivazione cut-free, e dunque il bound è iperesponenziale.

Nel caso della logica proposizionale, è possibile trovare delle strategie di cut-elimination che operano in tempo elementare, addirittura esponenziale. I lavori di Orevkov [18] e Statman [23] pongono invece un limite inferiore alla complessità dell'eliminazione del taglio nella logica del primo ordine, negando la possibilità che strategie simili possano essere trovate in questo caso. A tal proposito, definiamo la funzione iperesponenziale come

$$hyp(n) = tow(2, n, 1)$$

ed enunciamo il seguente risultato:

Teorema 5.1 (Lower bound iperesponenziale, Orevkov) *Esiste una sequenza di formule al primo ordine $(C_k)_{k\in\mathbb{N}}$ tali che per ogni k esiste in* **LK** *una derivazione di $\vdash C_k$ di altezza lineare in k, la cui forma normale ha invece altezza almeno pari a $hyp(k)$.*

Dimostrazione. La dimostrazione di questo teorema è estremamente tecnica, e va oltre gli scopi della trattazione. La prova originale di Orevkov può essere trovata in [18]; un'altra versione, non esageratamente più comprensibile, si può trovare in [5]. □

Il teorema appena enunciato restituisce in qualche modo importanza "deduttiva" alla regola del taglio. Come abbiamo già detto più volte, inizialmente sembra che questa sia indispensabile a formalizzare qualsiasi ragionamento; l'Hauptsatz ha invece contraddetto spudoratamente quest'intuizione, rivelando l'inutilità dei cut. La presenza di lower-bound iperesponenziali alla dimensione delle prove cut-free ci porta invece di nuovo verso la posizione di partenza: è vero che si può fare tutto senza *modus ponens*, ma il prezzo da pagare è talmente alto che una deduzione che non utilizzi tale strumento può diventare facilmente di una lunghezza non gestibile. Basti riflettere sul fatto che $hyp(5)$ è un numero che in base 10 occupa circa 20.000 cifre... in termini temporali, in questo caso (che è tutto sommato uno dei meno drammatici che ci si possa immaginare, visto che 5 non è certo il più grande numero di regole logiche concepibili per derivare qualcosa d'interessante...), si passerebbe da tempi dell'ordine, ad esempio, dei microsecondi a tempi dell'ordine di un migliaio di volte l'età dell'Universo.

Il risultato di Orevkov ha invece un'interpretazione perentoria se letto nell'ottica di Curry-Howard: le funzioni rappresentabili in logica intuizionista non sono, in generale, contenute in nessuna classe di complessità. Il tempo di esecuzione della procedura di cut-elimination è infatti strettamente legato alla dimensione della forma normale: dimensioni iperesponenziali significano tempi iperesponenziali.

Il motivo principale dell'effetto disastroso che la cut-elimination ha sulla grandezza delle prove è, come abbiamo evidenziato nella dimostrazione di 5.1, la presenza delle regole strutturali. La logica classica non offre strumenti sufficientemente potenti per controllare il loro potere espressivo; ecco perché al suo interno non è possibile immaginare alcuna possibilità di caratterizzare le classi di complessità in modo puramente logico.

Una speranza è invece data dalla Logica Lineare, introdotta nella metà degli anni '80. Nata per tutti altri motivi, questa offre finalmente la possibilità di manipolare in modo più accurato le regole strutturali, e lascia dunque aperta la possibilità di definire sistemi logici corrispondenti in modo esatto a determinate classi di complessità, proprio come le caratterizzazioni di Kalmar e Bellantoni-Cook corrispondono esattamente a **ELEMENTARY** e **FP**.

Parte II

Caratterizzazioni logiche di classi di complessità

Capitolo 6

La logica lineare

Il presente capitolo è dedicato all'introduzione (piuttosto rapida e superficiale) della Logica Lineare, il sistema del quale ci serviremo nel resto della trattazione, grazie al quale è possibile affrontare a livello puramente logico la definizione delle classi di complessità introdotte nel capitolo precedente.

6.1 Le regole strutturali

Quando abbiamo presentato il calcolo dei sequenti per la logica classica, del quale nel seguito utilizzeremo la versione "one-sided" definita in 1.2.1, abbiamo fornito due formulazioni per le regole logiche che introducono i connettivi $\wedge$, $\vee$ e i rispettivi elementi neutri $\mathcal{T}$ e $\mathcal{F}$. Abbiamo chiamato queste due formulazioni *additiva* e *moltiplicativa*, e abbiamo fatto spesso affidamento sulla loro equivalenza dal punto di vista dimostrativo: in **LK**, se un sequente è derivabile, allora è derivabile facendo uso di sole regole additive o di sole regole moltiplicative.

Ci occuperemo ora di dimostrare tale equivalenza, vale a dire che deriveremo le regole di una formulazione a partire da quelle dell'altra. In tutto questo, vedremo come le regole strutturali giochino un ruolo fondamentale.

Partiamo anzitutto dall'esprimere le regole moltiplicative utilizzando quelle additive. Per la costante $\mathcal{T}$, la cosa è ovvia: la regola $\mathcal{T}\mathsf{m}$ non è che un caso particolare della controparte additiva (vale a dire il caso in cui la lista Γ è vuota). Per la congiunzione, è possibile costruire la seguente derivazione:

$$\dfrac{\dfrac{\vdash \Gamma, A}{\vdash \Gamma, \Delta, A}\,\mathsf{W} \qquad \dfrac{\vdash \Delta, B}{\vdash \Gamma, \Delta, B}\,\mathsf{W}}{\vdash \Gamma, \Delta, A \wedge B}\,\wedge\mathsf{a}$$

Allo stesso modo, è chiaro come la regola $\mathcal{F}\mathsf{m}$ non sia altro che un caso particolare di weakening; per la disgiunzione, abbiamo la seguente derivazione:

$$\dfrac{\dfrac{\dfrac{\vdash \Gamma, A, B}{\vdash \Gamma, A \vee B, B}\,\mathsf{\vee a1}}{\vdash \Gamma, A \vee B, A \vee B}\,\mathsf{\vee a2}}{\vdash \Gamma, A \vee B}\,\mathsf{C}$$

Le regole moltiplicative sono dunque perfettamente simulabili per mezzo delle sole regole additive *e delle regole strutturali.*

Passiamo dunque al caso contrario; vogliamo mostrare come le regole moltiplicative possano, sempre grazie alle regole strutturali, sostituire quelle additive. Partiamo sempre dalla costante $\mathcal{T}$:

$$\dfrac{\dfrac{}{\vdash \mathcal{T}}\,\mathcal{T}\mathsf{m}}{\vdash \Gamma, \mathcal{T}}\,\mathsf{W}$$

Per quanto riguarda la congiunzione, abbiamo

$$\dfrac{\dfrac{\vdash \Gamma, A \qquad \vdash \Gamma, B}{\vdash \Gamma, \Gamma, A \wedge B}\,\mathsf{\wedge m}}{\vdash \Gamma, A \wedge B}\,\mathsf{C}$$

Non c'è alcuna regola additiva per $\mathcal{F}$ da simulare, ma in compenso ci sono due regole per la disgiunzione, ottenibili in modo del tutto analogo:

$$\dfrac{\dfrac{\vdash \Gamma, A}{\vdash \Gamma, A, B}\,\mathsf{W}}{\vdash \Gamma, A \vee B}\,\mathsf{\vee m} \qquad\qquad \dfrac{\dfrac{\vdash \Gamma, B}{\vdash \Gamma, A, B}\,\mathsf{W}}{\vdash \Gamma, A \vee B}\,\mathsf{\vee m}$$

La formulazione additiva e quella moltiplicativa sono dunque perfettamente equivalenti in **LK**, *modulo le regole strutturali.* Senza queste ultime, non sarebbe più possibile derivare una formulazione a partire dall'altra. L'equivalenza delle due versioni ha perfettamente senso a livello di dimostrabilità; se consideriamo, ad esempio, la congiunzione, è naturale che aver ottenuto A e B a partire da due contesti diversi o averle ottenute a partire dallo stesso contesto non cambia nulla ai fini del significato di $A \wedge B$.

Tuttavia, se analizziamo le due formulazioni sotto l'ottica della cut-elimination, ci rendiamo conto che queste hanno comportamenti completamente diversi. Nella formulazione moltiplicativa, un taglio congiunzione/disgiunzione si riduce creando due nuovi cut, che continueranno poi ad essere ridotti "in parallelo":

$$\dfrac{\dfrac{\overset{\vdots\,\pi_1}{\vdash \Gamma, A} \quad \overset{\vdots\,\pi_2}{\vdash \Delta, B}}{\vdash \Gamma, \Delta, A \wedge B}\,{\scriptstyle \wedge\mathsf{m}} \qquad \dfrac{\overset{\vdots\,\pi_3}{\vdash \Sigma, \neg A, \neg B}}{\vdash \Sigma, \neg A \vee \neg B}\,{\scriptstyle \vee\mathsf{m}}}{\vdash \Gamma, \Delta, \Sigma}\,{\scriptstyle \mathsf{cut}} \quad \longrightarrow \quad \dfrac{\overset{\vdots\,\pi_2}{\vdash \Delta, B} \qquad \dfrac{\overset{\vdots\,\pi_1}{\vdash \Gamma, A} \quad \overset{\vdots\,\pi_3}{\vdash \Sigma, \neg A, \neg B}}{\vdash \Gamma, \Sigma, \neg B}\,{\scriptstyle \mathsf{cut}}}{\vdash \Gamma, \Delta, \Sigma}\,{\scriptstyle \mathsf{cut}}$$

La formulazione additiva invece rende chiaramente l'idea di una "scelta" tra due possibili rami di esecuzione; in questo caso viene scelto il ramo π_1:

$$\dfrac{\dfrac{\overset{\vdots\,\pi_1}{\vdash \Gamma, A} \quad \overset{\vdots\,\pi_2}{\vdash \Gamma, B}}{\vdash \Gamma, A \wedge B}\,{\scriptstyle \wedge\mathsf{a}} \qquad \dfrac{\overset{\vdots\,\pi_3}{\vdash \Delta, \neg A}}{\vdash \Delta, \neg A \vee \neg B}\,{\scriptstyle \vee\mathsf{a1}}}{\vdash \Gamma, \Delta}\,{\scriptstyle \mathsf{cut}} \quad \longrightarrow \quad \dfrac{\overset{\vdots\,\pi_1}{\vdash \Gamma, A} \quad \overset{\vdots\,\pi_3}{\vdash \Delta, \neg A}}{\vdash \Gamma, \Delta}\,{\scriptstyle \mathsf{cut}}$$

Questa profonda differenza computazionale viene completamente offuscata dalle regole strutturali. Queste ultime hanno anch'esse un chiaro significato dal punto di vista della cut-elimination: la contrazione serve a *duplicare* una parte di dimostrazione, che, nell'ottica di Curry-Howard, corrisponde a duplicare un input; allo stesso modo, il weakening serve invece a *cancellare* un input.

E' proprio su questi punti che la Logica Lineare apporta alcune novità fondamentali; le regole strutturali vengono modificate in modo da mantenere separati i comportamenti additivo e moltiplicativo dei connettivi logici, consentendo di duplicare o cancellare solo determinate formule.

6.2 Il calcolo dei sequenti lineare

La Logica Lineare, introdotta nella metà degli anni '80 da Jean-Yves Girard [10], si presenta come la "logica dietro alla logica", nel senso che costituisce un *raffinamento* della Logica Classica, le cui caratteristiche consentono di far emergere strutture che in quest'ultimo sistema rimangono, per così dire, nascoste.

La logica lineare nasce da uno studio approfondito di una particolare classe di oggetti, chiamati *spazi coerenti*, la cui origine è legata alla ricerca di una *semantica denotazionale* per la logica intuizionista. La semantica denotazionale, come la semantica dei modelli presentata nel capitolo 1, serve a dare un "significato" agli oggetti che vengono manipolati in un sistema logico. Grossolanamente, la differenza tra le semantiche dei modelli e le semantiche denotazionali sta nel fatto che queste ultime non attribuiscono un significato solo alle formule, ma anche e soprattutto alle *dimostrazioni* di tali formule. In altre parole, in ambito denotazionale non ha più molta importanza stabilire se una formula è "vera" o "falsa", ma piuttosto ha importanza attribuire un significato alla struttura delle derivazioni, in particolare stabilendo le proprietà che una dimostrazione conserva sotto il processo di cut-elimination. Possiamo vedere le cose in questo modo: la semantica "tradizionale" si occupa di ricercare cos'è che è invariante sotto le regole di derivazione; la semantica denotazionale si occupa invece di studiare cos'è che è invariante sotto le regole che trasformano una derivazione in una equivalente, ad esempio per mezzo della cut-elimination.

Un'introduzione completa alla logica lineare dovrebbe dunque senz'altro passare attraverso gli spazi coerenti; tuttavia, in questa sede non è possibile analizzare la questione in modo così approfondito, e quindi presenteremo la logica lineare direttamente come un calcolo che nasce sostanzialmente da **LK**, al quale sono state però apportate alcune modifiche.

Introduciamo anzitutto la sintassi delle formule della logica lineare (che da qui in poi abbrevieremo spesso con **LL**)[1]:

Definizione 6.1 *Sia $\mathcal{V} = \{X, Y, Z, \ldots\}$ un insieme numerabile di variabili proposizionali. Le formule di* **LL** *sono definite nel seguente modo:*

- $X, Y, Z, \ldots$ *e* $X^\perp, Y^\perp, Z^\perp, \ldots$ *sono formule di* **LL**
- *Le costanti logiche* $\top$, 0, 1 *e* $\perp$ *sono formule di* **LL**
- *Se A e B sono formule, allora $A \mathbin{\&} B$, $A \oplus B$, $A \otimes B$ e $A \parr B$ sono formule.*
- *Se A è una formula, $!A$ e $?A$ sono formule.*

La negazione lineare *(o* duale, *o* ortogonale*) è il connettivo definito dalle seguenti leggi di De Morgan:*

$(A \mathbin{\&} B)^\perp := A^\perp \oplus B^\perp$

$(A \oplus B)^\perp := A^\perp \mathbin{\&} B^\perp$

$(A \otimes B)^\perp := A^\perp \parr B^\perp$

$(A \parr B)^\perp := A^\perp \otimes B^\perp$

$(!A)^\perp := ?A^\perp$

[1] La definizione che diamo prende in considerazione il solo frammento proposizionale, perché è qui che emergono le novità cruciali della logica lineare rispetto alla logica classica; l'introduzione dei quantificatori a qualsiasi ordine è, dal punto di vista sintattico, identica al caso classico.

$(?A)^\perp := !A^\perp$

*L'*implicazione lineare *(o* entails*) è il connettivo definito come segue:*

$$A \multimap B := A^\perp \parr B$$

Iniziamo dunque con il presentare il gruppo delle regole dell'identità per il calcolo dei sequenti di **LL**:

$$\frac{}{\vdash A^\perp, A}\,\mathsf{ax} \qquad\qquad \frac{\vdash \Gamma, A \qquad \vdash \Delta, A^\perp}{\vdash \Gamma, \Delta}\,\mathsf{cut}$$

Le regole dell'identità sono dunque sostanzialmente identiche a quelle di **LK**, con l'unica differenza che qui si considera la negazione lineare delle formule.

Le regole strutturali sono invece ridotte alla sola regola di scambio, identica anche in **LK**:

$$\frac{\vdash \Gamma, A, B, \Delta}{\vdash \Gamma, B, A, \Delta}\,\mathsf{X}$$

Le regole logiche sono anch'esse praticamente identiche a quelle di **LK**, con la differenza fondamentale che esse introducono però connettivi diversi a seconda della formulazione. Cominciamo dalla congiunzione additiva. L'elemento neutro di tale connettivo è $\top$ (*top*), la cui regola è

$$\frac{}{\vdash \Gamma, \top}\,\top$$

La congiunzione additiva, che è rappresentata dal connettivo $\&$ (*with*), ha invece la regola

$$\frac{\vdash \Gamma, A \qquad \vdash \Gamma, B}{\vdash \Gamma, A \,\&\, B}\,\&$$

L'elemento neutro della disgiunzione additiva (0, *zero*) non ha una regola d'introduzione nel calcolo dei sequenti lineare, esattamente come succedeva in **LK**. Il connettivo corrispondente è $\oplus$ (*plus*), e esso ha invece due regole:

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, A \oplus B}\,\oplus 1 \qquad\qquad \frac{\vdash \Gamma, B}{\vdash \Gamma, A \oplus B}\,\oplus 2$$

Passiamo ora alla formulazione moltiplicativa. L'elemento neutro della congiunzione è 1 (*uno*), la cui regola è

$$\frac{}{\vdash 1}\,1$$

Il connettivo corrispondente alla congiunzione moltiplicativa è $\otimes$ (*times*, o *tensore*), con la seguente regola:

$$\frac{\vdash \Gamma, A \qquad \vdash \Delta, B}{\vdash \Gamma, \Delta, A \otimes B}\otimes$$

Allo stesso modo, l'elemento neutro della disgiunzione moltiplicativa è $\perp$ (*bottom*), con la corrispondente regola

$$\frac{\vdash \Gamma}{\vdash \Gamma, \perp}\perp$$

Il connettivo che rappresenta la disgiunzione moltiplicativa è una delle peculiarità della logica lineare, e si indica con il simbolo $⅋$ (*par*). La regola corrispondente è

$$\frac{\vdash \Gamma, A, B}{\vdash \Gamma, A ⅋ B}⅋$$

Abbiamo così terminato l'esposizione di tutte le regole logiche che discendono direttamente da **LK**. Le altre regole costituiscono la novità assoluta di **LL**, e servono a recuperare in modo più "controllato" le regole strutturali della contrazione e del weakening che fin'ora sembrano essere scomparse; ricordiamo che è proprio grazie a questa scomparsa che i connettivi additivi e moltiplicativi hanno potuto prendere vita propria indipendente.

La regole che stiamo per introdurre riguardano due nuovi connettivi, che chiameremo *esponenziali*. Il primo è ? (*why not*), la cui regola è

$$\frac{\vdash \Gamma, A}{\vdash \Gamma, ?A}\,?\mathsf{D}$$

La regola appena introdotta viene chiamata *dereliction*. Il secondo esponenziale è ! (*of course*), e la regola corrispondente è detta *promotion*:

$$\frac{\vdash ?\Gamma, A}{\vdash ?\Gamma, !A}\,!\mathsf{P}$$

dove $?\Gamma$ rappresenta una sequenza di formule che sono tutte della forma $?C$ per qualche formula C.

Possiamo a questo punto introdurre la versione lineare delle regole strutturali:

$$\frac{\vdash \Gamma, ?A, ?A}{\vdash \Gamma, ?A}\,?\mathsf{C} \qquad\qquad \frac{\vdash \Gamma}{\vdash \Gamma, ?A}\,?\mathsf{W}$$

La morale della contrazione e del weakening in logica lineare è chiara: le sole formule a poter essere contratte (cioè le sole risorse a poter essere duplicate, in ottica Curry-Howard) sono le formule esponenziali; allo stesso modo, una formula può essere introdotta per weakening (cioè una risorsa può essere cancellata) solo se è anch'essa esponenziale.

Per quanto riguarda i quantificatori, non cambia praticamente nulla rispetto a **LK**. Considerando ad esempio la quantificazione al secondo ordine (che è di gran lunga più interessante a livello computazionale), basta anzitutto estendere le leggi di De Morgan con le due seguenti:

$$(\forall X.A)^{\perp} := \exists X.A^{\perp}$$
$$(\exists X.A)^{\perp} := \forall X.A^{\perp}$$

e introdurre poi le due solite regole per la quantificazione universale ed esistenziale, che non riportiamo perché davvero identiche a quelle di **LK**.

Come per la logica classica, anche in logica lineare è possibile dimostrare alcune equivalenze fondamentali, come ad esempio la distributività fra i connettivi. Indicando con $A \leftrightarrow B$ la formula $(A \multimap B) \,\&\, (B \multimap A)$, abbiamo

$$A \otimes (B \oplus C) \leftrightarrow (A \otimes B) \oplus (A \otimes C)$$
$$A \parr (B \,\&\, C) \leftrightarrow (A \parr B) \,\&\, (A \parr C)$$

Non è invece vero che $\otimes$ si distribuisce su $\parr$ e che $\&$ si distribuisce su $\oplus$. Le proprietà di distributività forniscono ai due gruppi di connettivi il loro nome: proprio come la moltiplicazione si distribuisce sull'addizione, così i connettivi moltiplicativi si distribuiscono su quelli additivi.

Le seguenti equivalenze logiche[2] sono invece quelle che danno ai connettivi esponenziali il loro nome:

$$!(A \,\&\, B) \leftrightarrow\, !A \otimes !B$$
$$?(A \oplus B) \leftrightarrow\, ?A \parr ?B$$

Se vediamo i connettivi moltiplicativi come dei prodotti, quelli additivi come delle somme e quelli esponenziali come, appunto, degli esponenziali, le formule di sopra non esprimono nient'altro che la ben nota equivalenza

$$2^{a+b} = 2^a \cdot 2^b$$

6.3 Alcune considerazioni su LL

E' opportuno fare alcune osservazioni fondamentali riguardo alla logica appena introdotta. In primo luogo, è chiaro come la logica lineare continui

[2] In realtà queste, come anche le precedenti, sono molto di più che semplici equivalenze logiche; si tratta infatti di *isomorfismi denotazionali*, radicati dunque nella struttura profonda della semantica di **LL** (a tal proposito, si veda [10].

sempre di più nella direzione, già intrapresa dalla logica intuizionista, di far perdere quasi completamente rilevanza al significato "classico" delle formule. In logica lineare, non ha alcun senso parlare della verità o della falsità di una formula, perché non esiste più una semantica dei modelli come quella per la logica classica[3]. Questa situazione è tipica della visione moderna della logica; l'interesse non è più quello di dimostrare la verità o la falsità di una formula, ma di considerare piuttosto la struttura delle dimostrazioni stesse, il che, nell'ottica di Curry-Howard, significa ad esempio studiare le proprietà computazionali dei sistemi logici.

Dal punto di vista della derivazione di formule, è comunque importante il fatto che la logica lineare costituisce un *raffinamento* della logica classica, in cui quindi non si perde nulla. Infatti, è possibile trovare una traduzione dalle formule classiche a quelle di **LL** tale che, se una certa formula A è derivabile in **LK**, allora la sua traduzione è derivabile nel calcolo dei sequenti lineare; la stessa cosa è vera anche per **LJ**.

Se però non ha molto senso cercare di fornire un'intuizione sul significato di una formula come $A \,\&\, B$ a partire da un qualche significato di A e di B, è al contrario di grandissimo interesse cercare di scoprire il significato che ha una *dimostrazione* di tale formula a partire dal significato delle dimostrazioni delle sue sottoformule. In tal senso, possiamo dare una spiegazione intuitiva di tre dei connettivi "più semplici" della logica lineare:

- & Una dimostrazione di $A \,\&\, B$ è una coppia di dimostrazioni, una di A e una di B, le quali derivano tali formule *a partire dalle stesse premesse*. Ciò significa, in termini computazionali, che le due formule presenti in $A \,\&\, B$ utilizzano *le stesse risorse*; la congiunzione additiva quindi indica la presenza contemporanea di due possibilità, delle quali però solo una sarà resa disponibile, giacché entrambe consumano lo stesso insieme di risorse. Un programma di tipo $A \,\&\, B$ dunque mette l'utente di fronte ad una scelta: è lui a dover selezionare quale dei due output ottenere, se quello di tipo A o quello di tipo B. Osserviamo dunque come la congiunzione additiva abbia un evidente "contaminazione" disgiuntiva.

- $\otimes$ Una dimostrazione di $A \otimes B$ è una coppia di dimostrazioni, una di A e una B, le quali però derivano tali formule *a partire da due premesse diverse*. Da un punto di vista informatico, i due possibili output di $A \otimes B$ dipendono dunque da due risorse separate, o comunque da

[3] In realtà anche **LL** ha una sua semantica per le formule, la cosiddetta *semantica delle fasi*. In base ad essa è possibile dimostrare gli analoghi, per il calcolo dei sequenti lineare, dei teoremi di correttezza e completezza già visti per **LK** in 1.2. Naturalmente, la semantica delle fasi si basa su strutture parecchio più complesse che i semplici valori di verità utilizzati in logica classica.

due istanze della stessa risorsa, il che è la stessa cosa; la congiunzione moltiplicativa dunque mette a disposizione entrambe i risultati, contemporaneamente. E' importante notare come quest'idea metta in luce un legame tra il tensore e il parallelismo computazionale.

$\oplus$ Una dimostrazione di $A \oplus B$ è una dimostrazione di A *oppure* una dimostrazione di B. Essendo il duale del connettivo $\&$, la disgiunzione additiva induce in ambito computazionale una situazione duale rispetto a quella della congiunzione additiva. Anche in questo caso infatti, solo uno dei due risultati sarà disponibile, ma *la scelta non dipende dall'utente*: chi ha davanti una dimostrazione di $A \oplus B$ non può decidere se la dimostrazione da cui essa proviene è una dimostrazione di A o di B.

Il connettivo $⅋$, che è la disgiunzione moltiplicativa, ha un significato molto meno facile da intuire. Il modo più semplice di vedere le cose è quello di considerare il connettivo $\multimap$, che come abbiamo visto è definito proprio in termini del par:

$$A \multimap B := A^{\perp} ⅋ B$$

Una dimostrazione di $A \multimap B$ è una funzione che induce dimostrazioni di B a partire da dimostrazioni di A, ma la cosa fondamentale è che il processo per mezzo del quale si ottiene la dimostrazione "in output" utilizza la dimostrazione "in input" *esattamente una volta*. In termini computazionali, un programma di tipo $A \multimap B$ è un programma che calcola un risultato di tipo B utilizzando un input di tipo A esattamente una volta. Un programma che abbia tale proprietà si dice *lineare* (in un particolare input). Questa funzione C è ad esempio senz'altro lineare nel parametro `x`, ma non lo è in `y`:

```
int f(int x, int y)
{
    // siano g ed h due funzioni lineari
    return g(y, h(x, y));
}
```

Nell'introdurre questo concetto di linearità, abbiamo in effetti trovato una differenza radicale tra l'implicazione lineare $\multimap$ e l'implicazione classica (o intuizionista) $\Rightarrow$; nell'interpretazione computazionale di **LJ**, i programmi di tipo $A \Rightarrow B$ sono infatti ancora funzioni da oggetti di tipo A a oggetti di tipo B, ma essi possono utilizzare il loro input *un numero arbitrario di volte, anche nullo*. E' questo uno dei punti cruciali di tutta la logica lineare, che è alla base della sua stessa nascita. Il sistema sintattico **LL** è infatti scaturito in primo luogo da un'analisi della semantica denotazionale del connettivo $\Rightarrow$ della logica intuizionista, per il quale si è scoperta una *decomposizione* in termini di due connettivi più primitivi: l'implicazione lineare, appunto, e una modalità del tutto assente in logica classica, la quale consente di esprimere che una risorsa verrà utilizzata un numero arbitrario, anche nullo, di volte. Tale modalità è il connettivo esponenziale !, grazie al quale si può scrivere l'isomorfismo denotazionale che abbiamo appena descritto:

$$A \Rightarrow B \simeq (!A) \multimap B$$

L'implicazione lineare e gli esponenziali mettono in luce il vero significato dell'implicazione classica: $A \Rightarrow B$ significa che si può ottenere B a partire da A, *utilizzandola però un numero qualsiasi di volte*. In ambito matematico, ciò ha perfettamente senso. Supponiamo infatti che A esprima il concetto "t è un triangolo rettangolo" e che B sia invece "il quadrato del lato più lungo del triangolo t è pari alla somma dei quadrati degli altri due lati"; in tal caso $A \Rightarrow B$ non è null'altro che il Teorema di Pitagora. Supponiamo ora di aver dimostrato che un qualche triangolo t sia rettangolo; chi mai avrebbe a lamentarsi se, nel corso della dimostrazione del Teorema di Pitagora per t, utilizzassimo una, due, o dieci volte il fatto che esso è rettangolo? Una volta che t è rettangolo, lo è per sempre, indipendentemente dal numero di volte che questa sua proprietà è invocata per dimostrarne qualche altra. Allo stesso modo, chiunque troverebbe alquanto singolare una geometria nella quale, una volta applicato il Teorema di Pitagora a t, questo cessasse improvvisamente di essere un triangolo rettangolo.

Una delle novità fondamentali della logica lineare è proprio questa: in **LL** è possibile esprimere il *deterioramento delle risorse*: in $A \multimap B$, l'aver fornito A per ottenere B comporta la perdita di A nel processo, e affinché questo possa mettersi in moto serve esattamente un'istanza di A, né una di più, né una di meno. Al contrario, in $(!A) \multimap B$, si può ottenere B soltanto da una risorsa $!A$ che sia disponibile in quantità arbitrarie; $!A$ non verrà dunque esaurita nel corso del processo, proprio come un triangolo rettangolo resta tale dopo avervi applicato il Teorema di Pitagora.

Questo controllo più raffinato che la logica lineare mette a disposizione nei confronti dell'utilizzazione delle risorse, apre possibilità inconcepibili a livello classico. In particolare, vedremo come grazie ad un'analisi approfondita dei connettivi esponenziali sia possibile trovare in essi una sorta di "manopola di controllo" della complessità computazionale di **LL**.

Capitolo 7

Le proof-net

In questo capitolo introdurremo una sintassi alternativa per le dimostrazioni della logica lineare, le cosiddette proof-net. *Le proof-net (o reti dimostrative) sono l'analogo delle deduzioni naturali per la logica intuizionista; esse riescono infatti a presentare le dimostrazioni del calcolo dei sequenti in modo il più possibile* parallelo. *Vedremo dapprima la definizione di proof-net nel solo frammento moltiplicativo della logica lineare, sistema nel quale tale formalismo è particolarmente elegante; in seguito, estenderemo la definizione all'intera logica lineare.*

7.1 Il frammento moltiplicativo

7.1.1 La definizione induttiva

Abbiamo visto in 3.2 come per la logica intuizionista sia possibile rappresentare le dimostrazioni per mezzo di alberi particolari, detti deduzioni naturali. Le deduzioni naturali hanno la proprietà di essere, per così dire, più "essenziali" delle derivazioni di **LJ**, nel senso che ad una singola deduzione naturale possono corrispondere più derivazioni; è come se **LJ** costringesse ad esibire informazioni sulle dimostrazioni che non sono in effetti necessarie. A ben guardare, l'informazione ridondante del calcolo dei sequenti è la *sequenzialità*: si pensi ad esempio a quante derivazioni diverse esistono del sequente $A_1 \wedge A_2, A_3 \wedge A_4, \ldots, A_{2n-1} \wedge A_{2n} \vdash C$ a partire dal sequente $A_1, A_3, \ldots, A_{2n-1} \vdash C$, ciascuna delle quali introduce le formule A_{2i} in ordine diverso mediante regole $\wedge aL1$; tutte queste derivazioni corrispondono ad un'unica deduzione naturale in **NJ**, nella quale dalle foglie $A_{2i-1} \wedge A_{2i}$ vengono eliminate le formule A_{2i} mediante quella che possiamo vedere come un'applicazione *parallela*, contemporanea, di regole $\mathcal{E} \wedge 1$. Abbiamo anche visto come questa proprietà delle deduzioni naturali consenta di mettere in luce in modo più limpido il processo di cut-elimination, così da poter costruire un vero e proprio isomorfismo con la riduzione dei termini del λ-calcolo, mettendo in piedi la corrispondenza fondamentale tra dimostrazioni e programmi.

La domanda che nasce a questo punto, nell'ottica della logica lineare, è se sia possibile costruire un formalismo simile per questo nuovo sistema logico, il quale nasce già di per sé con un fortissimo legame verso i processi di calcolo. La risposta è affermativa, anche se, nel caso generale dell'intera logica lineare, non è ancora del tutto soddisfacente.

Iniziamo allora con il considerare un sottosistema estremamente semplice di **LL**, che è il cosiddetto *frammento moltiplicativo*, a cui ci si riferisce con l'abbreviazione **MLL** (*Multiplicative Linear Logic*). Le formule di **MLL** sono definite nel seguente modo:

Definizione 7.1 *Sia $\{X, Y, Z, \ldots\}$ un insieme numerabile di variabili proposizionali. Le formule di* **MLL** *sono generate induttivamente come segue:*

- $X, Y, Z, \ldots$ *e* $X^\perp, Y^\perp, Z^\perp, \ldots$ *sono formule di* **MLL**
- *se A e B sono due formule di* **MLL***, $A \otimes B$ e $A \parr B$ sono formule di* **MLL**.

La negazione lineare (ortogonale) è definita mediante le seguenti leggi di De Morgan:

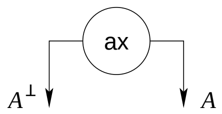
$$(A \otimes B)^\perp := A^\perp \parr B^\perp$$
$$(A \parr B)^\perp := A^\perp \otimes B^\perp$$

Il calcolo dei sequenti di **MLL** è quello di **LL** ristretto alle sole quattro regole ax, cut, $\otimes$ e $\parr$.

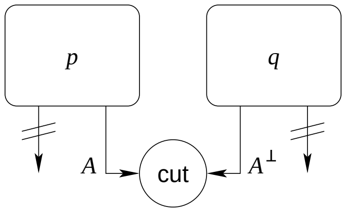
Possiamo a questo punto fornire una definizione induttiva, che ricalca in modo esplicito le regole del calcolo dei sequenti, dell'equivalente delle deduzioni naturali per **MLL**, che chiameremo *proof-net*:

Definizione 7.2 (Proof-net moltiplicativa, versione induttiva) *Una* proof-net *moltiplicativa è un grafo orientato costituito da quattro tipi di nodi (*ax*,* cut*, $\otimes$ e $\parr$), i cui archi sono etichettati da (occorrenze di) formule di* **MLL**. *Gli archi entranti in un nodo sono detti* premesse *del nodo, mentre quelli uscenti sono detti* conclusioni*. Gli archi che non sono premesse di alcun nodo saranno detti* conclusioni *della proof-net, così come le formule che li etichettano. Le proof-net sono definite induttivamente nel modo seguente:*

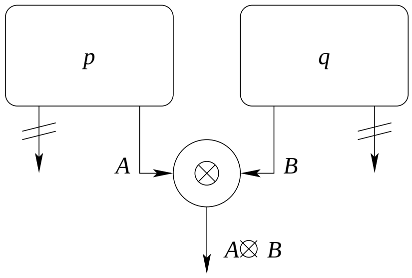
ax *Il seguente grafo, con un nodo e due conclusioni A e $A^\perp$, è una proof-net:*



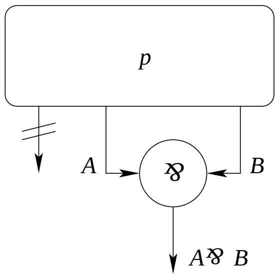
cut *Se p è una proof-net tra le cui conclusioni c'è la formula A, e q è un'altra proof-net tra le cui conclusioni c'è la formula $A^{\perp}$, il seguente grafo è una proof-net:*



$\otimes$ *Se p è una proof-net tra le cui conclusioni c'è la formula A, e q è un'altra proof-net tra le cui conclusioni c'è la formula B, il seguente grafo è una proof-net:*



$\parr$ *Se p è una proof-net tra le cui conclusioni ci sono le formule A e B, il seguente grafo è una proof-net:*



Le proof-net non sono dunque alberi come le deduzioni naturali, ma grafi; ciò appare piuttosto naturale se si considera che i sequenti di **MLL** (e, più in generale, di **LL**) hanno più di una conclusione. Osserviamo inoltre che, in effetti, le conclusioni di una proof-net sono archi uscenti che non entrano in alcun nodo, la qual cosa è un po' scomoda dal punto di vista formale visto che le strutture definite in questo modo non sono propriamente grafi; tuttavia, la situazione può essere recuperata agevolmente dando per scontata l'esistenza di un altro tipo di nodi aventi un solo arco entrante, costituito appunto da una conclusione della proof-net. In questo modo, le strutture dimostrative diventano dei veri e propri grafi; tuttavia, nel seguito della trattazione ci dimenticheremo di questi inutili "nodi conclusione".

E' chiaro che ad ogni derivazione del calcolo dei sequenti di **MLL** può essere associata una proof-net: basta ripercorrere l'albero di derivazione e applicare la costruzione iterativa delle proof-net descritta nella definizione. Anche il viceversa è vero:

Teorema 7.1 (Sequenzializzazione delle proof-net di MLL) *Se π è una proof-net di* **MLL** *con conclusioni $A_1, \ldots, A_n$, allora esiste nel calcolo dei sequenti di* **MLL** *una derivazione del sequente $\vdash A_1, \ldots, A_n$.*

Dimostrazione. La dimostrazione si può trovare in [10]. □

7.1.2 I criteri di correttezza e le definizioni strutturali

Le proof-net definite nella sezione precedente sembrano effettivamente essere ciò che cercavamo: in esse non c'è più alcuna traccia della sequenzialità delle regole applicate. Tuttavia, la definizione induttiva è ancora di per sé sequenziale, nel senso che una proof-net viene costruita passo passo proprio come una derivazione del calcolo dei sequenti. Al contrario, trovandoci di fronte ad un grafo orientato già costruito, i cui nodi siano dei quattro tipi dati nella definizione, non sarebbe per noi facile dire se questo sia o no una proof-net, cioè se sia possibile sequenzializzarlo in una derivazione del calcolo dei sequenti. Iniziamo allora con l'introdurre il concetto di *proof-structure*:

Definizione 7.3 (Proof-structure moltiplicativa) *Una* proof-structure *(o* struttura dimostrativa*) moltiplicativa è un grafo orientato costituito da nodi a cui è associata un'arità e una coarità, indicanti rispettivamente il numero di archi entranti e il numero di archi uscenti. Gli archi entranti saranno detti* premesse *del nodo, gli archi uscenti* conclusioni*. Tali nodi si distinguono in quattro tipi:*

- ax *Il nodo assioma, con zero premesse e due conclusioni, etichettate con le formule $A^\perp$ e A*
- cut *Il nodo cut, con due premesse, etichettate dalle formule A e $A^\perp$, e zero conclusioni*
- $\otimes$ *Il nodo tensore, con due premesse etichettate dalle formule A e B e una conclusione etichettata dalla formula $A \otimes B$*
- $⅋$ *Il nodo par, con due premesse etichettate dalle formule A e B e una conclusione etichettata dalla formula $A ⅋ B$*

Gli archi che sono conclusione di un nodo ma non sono premessa di alcun altro nodo sono dette le conclusioni *della proof-structure.*

E' chiaro che una proof-net è senz'altro una proof-structure, mentre il viceversa non è assolutamente vero; il cuore della teoria delle proof-net è proprio questo: cercare delle proprietà puramente strutturali per caratterizzare le proof-net a partire dalle proof-structure, senza ricorrere alla definizione induttiva.

Ciò che stiamo cercando dunque è un *criterio di correttezza* che ci consenta di dire che una proof-structure è effettivamente una proof-net, cioè che questa rappresenta una derivazione di **MLL**. Ad oggi sono conosciuti diversi criteri di questo tipo; quello più semplice da descrivere è il cosiddetto criterio ACC introdotto da Danos e Regnier in [9]. Per descrivere tale criterio, è necessario introdurre i concetti di *switching* e di *grafo di correttezza*:

Definizione 7.4 (Switching) *Sia σ una proof-structure di* **MLL**. *Uno* switching *di σ è una funzione che associa ad ogni nodo $⅋$ di σ uno dei due nodi di σ la cui conclusione è premessa del $⅋$ stesso.*

Definizione 7.5 (Grafo di correttezza) *Sia σ una proof-structure, e s un suo switching. Il* grafo di correttezza *di σ sotto lo switching s, che denoteremo con $R(\sigma, s)$, è il grafo non orientato costruito a partire da σ nel seguente modo:*

- *Ad ogni nodo di σ corrisponde un nodo di $R(\sigma, s)$.*
- *Se (a, b) è un arco di σ, dove b non è un nodo $⅋$, allora $\{a, b\}$ è un arco non orientato di $R(\sigma, s)$; se invece (a_0, b) e (a_1, b) sono due archi incidenti su un nodo $⅋$, allora $\{s(b), b\}$ è un arco non orientato di $R(\sigma, s)$.*

In sostanza, uno switching vede ogni nodo $⅋$ come un interruttore, e nel grafo di correttezza vengono eliminati quegli archi che collegano i nodi $⅋$ alle premesse "disabilitate" dall'interruttore. Osserviamo che ad ogni proof-structure σ contenente n nodi $⅋$ sono associabili 2^n switching, e dunque 2^n grafi di correttezza. In figura 7.1.2 è mostrata una semplicissima proof-structure con un solo nodo $⅋$ e con i suoi due possibili grafi di correttezza.

Ecco dunque la definizione della condizione ACC:

Definizione 7.6 (Condizione ACC) *Si dice che una proof-structure σ soddisfa la condizione ACC (o, più semplicemente, che è ACC) se per ogni possibile switching s di σ, $R(\sigma, s)$ è un grafo aciclico e connesso (ovverosia un albero). ACC in effetti sta per* Acyclic-Connected.

Il criterio di correttezza ACC, che consente di fornire un definizione non induttiva delle proof-net, è il risultato del seguente teorema:

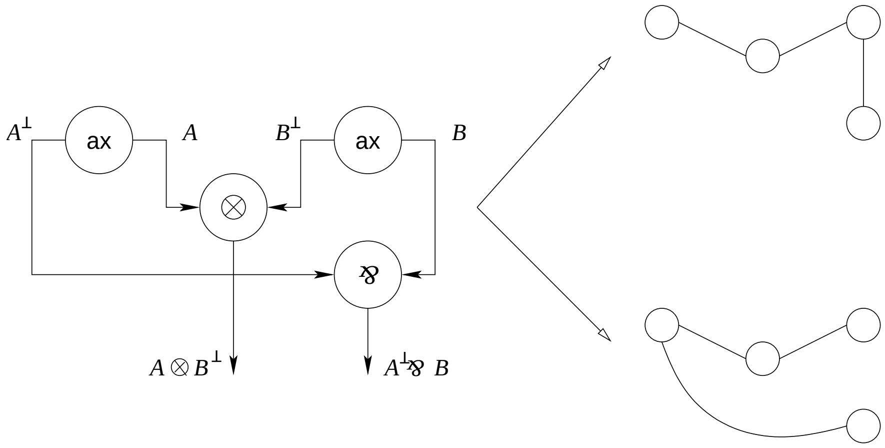


Figura 7.1: Una semplice proof-net moltiplicativa con i suoi due possibili grafi di correttezza.

Teorema 7.2 (Danos-Regnier) *Una proof-structure σ è una proof-net se e soltanto se essa soddisfa la condizione ACC.*

Dimostrazione. La dimostrazione di questo teorema si compone di due parti, la *validità* e la *completezza* del criterio. La validità è l'implicazione *se una proof-structure è una proof-net, allora essa è ACC*, la completezza è l'implicazione inversa, *se una proof-structure è ACC, allora essa è una proof-net.* La dimostrazione della validità è piuttosto semplice, e consta in un'induzione sulla struttura delle proof-net. La correttezza invece è una questione molto più complicata, nella quale non possiamo entrare in dettaglio. Una dimostrazione della completezza di ACC può essere trovata in [9]. □

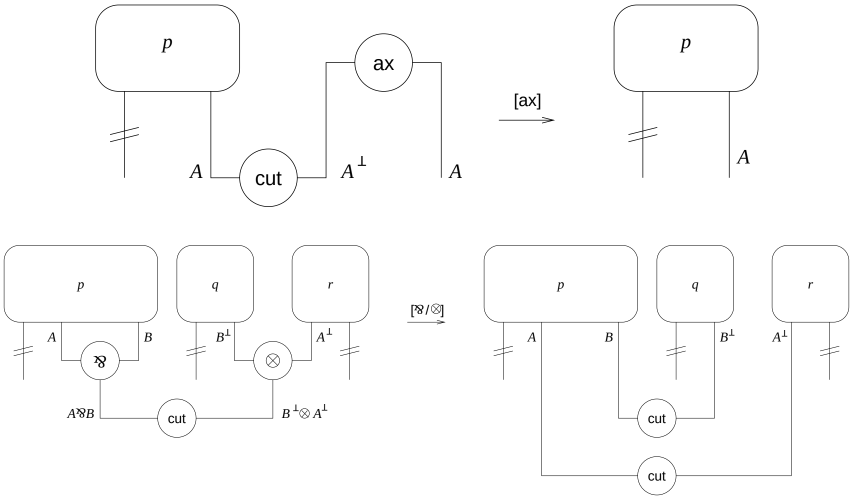
La definizione migliore di proof-net può dunque essere data nel modo seguente:

Definizione 7.7 (Proof-net moltiplicativa, versione ACC) *Una proof-net è una proof-structure che soddisfa ACC.*

Osserviamo che, pur essendo molto elegante, questa definizione è ben lontana dal fornire un criterio concreto per verificare che una proof-structure sia una proof-net; infatti, come abbiamo già fatto notare in precedenza, una proof-structure con n nodi $⅋$ possiede 2^n grafi di correttezza, e per verificare ACC è necessario controllare l'aciclicità e la connessione di tutti. Anche se queste due operazioni possono essere portate avanti in tempo polinomiale, la verifica di ACC resta un'operazione che richiede, nel caso di un algoritmo deterministico, un tempo $O(p(s) \cdot 2^s)$, dove s è la dimensione della proof-structure (cioè il suo numero di nodi) e p un polinomio. In realtà, è possibile fornire criteri di correttezza per **MLL** molto più efficienti, addirittura lineari nella dimensione (si veda [14]).

7.1.3 La cut-elimination

Il grandissimo vantaggio di disporre di una sintassi "parallela" per la logica è quello di poter descrivere il processo di cut-elimination in modo molto più limpido di quanto non sia possibile nel calcolo dei sequenti. In particolare, per **MLL**, l'eliminazione del taglio consiste in un processo di *riscrittura di grafi*, governato da due sole regole:



Non è difficilissimo verificare che la riscrittura mediante le due regole di cut-elimination preserva la condizione ACC; di conseguenza, la proof-structure ottenuta dopo l'applicazione di un numero qualsiasi di regole di riduzione è una proof-net se quella originale lo era (si veda [9]).

Per **MLL**, la cut-elimination gode di tutte le buone proprietà immaginabili per un processo di riduzione: è confluente, fortemente normalizzante, e per di più tutte le sequenze di normalizzazione convergono nello stesso numero di passi. Osservando le due regole di riscrittura, è facile constatare che il numero di nodi di una proof-net si riduce sempre ad ogni passo di eliminazione del taglio; ogni passo dunque "consuma" un pezzo di proof-net, e quindi qualsiasi riduzione per una proof-net π non può essere costituita da un numero di passi superiore al numero di nodi di π. Composto con le proprietà elencate sopra, questo ragionamento induce un bound sub-lineare alla durata della cut-elimination in **MLL**: se π è una proof-net di dimensione (numero di nodi) s, tutte le strategie di normalizzazione conducono alla stessa forma normale, nel medesimo numero di passi, e tale numero di passi è $O(s)$. La dimostrazione dettagliata di tutti questi risultati è in [10].

7.2 Estensione all'intera logica lineare

Il funzionamento delle proof-net nel frammento moltiplicativo è talmente limpido ed elegante che sarebbe meraviglioso riuscire ad estendere il formalismo all'intera logica lineare. Sfortunatamente, nel caso generale non si è ancora arrivati a soluzioni altrettanto buone, e sotto molti aspetti è improbabile che si riesca a fare di meglio. Tuttavia, per quanto riguarda gli aspetti fondamentali strettamente legati al calcolo e, conseguentemente, alla complessità computazionale, l'estensione delle proof-net a tutta **LL** si comporta comunque in modo più che soddisfacente.

Cominciamo dunque con il fornire l'estensione delle proof-structure alla logica lineare completa, con i quantificatori al secondo ordine, e senza costanti logiche. La definizione che segue è adattata da [25]:

Definizione 7.8 (Pre-proof-structure) *Una pre-proof-structure è un grafo orientato i cui archi sono etichettati da (occorrenze di) formule di* ***LL****, e i cui nodi sono provvisti di un'arità e di una coarità, corrispondenti rispettivamente al numero di archi entranti, dette* premesse *del nodo, e al numero di archi uscenti, dette* conclusioni. *I nodi sono di uno dei tipi seguenti:*

- ax *Un nodo* assioma *ha due conclusioni etichettate da due formule duali, e nessuna premessa*
- cut *Un nodo* cut *ha due premesse etichettate da due formule duali, e nessuna conclusione*
- $\otimes$ *Un nodo* tensore *ha due premesse e una conclusione. Se la premessa sinistra del nodo è etichettata dalla formula A e la premessa destra da B, allora la conclusione sarà etichettata con la formula $A \otimes B$*
- $⅋$ *Un nodo* tensore *ha due premesse e una conclusione. Se la premessa sinistra del nodo è etichettata dalla formula A e la premessa destra da B, allora la conclusione sarà etichettata con la formula $A ⅋ B$*
- ! *Un nodo* of course *(o* bang*) ha una premessa e una conclusione, la quale è etichettata con l'*of course *dell'etichetta della premessa*
- ?D *Un nodo* dereliction *ha una premessa e una conclusione, la quale è etichettata con il* why not *dell'etichetta della premessa*
- ?W *Un nodo* weakening *non ha alcuna premessa e ha una conclusione, etichettata con $?A$, dove A è una formula.*
- ?C *Un nodo* contrazione *ha due premesse e una conclusione, tutte etichettate dalla stessa formula $?A$*

pax *Un nodo* pax *(porta ausiliaria) ha una premessa e una conclusione, entrambe etichettate dalla stessa formula* $?A$

$\oplus_1$ *e* $\oplus_2$ *Un nodo* plus uno *[*plus due*] ha una premessa e una conclusione, tali che se* A *è l'etichetta della premessa, allora* $A \oplus B$ $[B \oplus A]$ *è l'etichetta della conclusione, dove* B *è una formula. In generale, parleremo semplicemente di un nodo* $\oplus$ *(*plus*) quando non sarà necessario specificare se si tratti di un* $\oplus_1$ *o* $\oplus_2$.

& *Un nodo* with *ha due premesse e una conclusione. Se la premessa sinistra del nodo è etichettata dalla formula* A *e la premessa destra da* B*, allora la conclusione sarà etichettata con la formula* $A \,\&\, B$

coad *Un nodo* contrazione additiva *ha due premesse e una conclusione, tutte etichettate dalla medesima formula*

$\forall$ *Un nodo* per ogni *ha una premessa e una conclusione, tali che se la premessa è etichettata dalla formula* A*, allora la conclusione è etichettata da* $\forall X.A[X]$. *La variabile* X *sarà detta* variabile propria *del nodo per ogni*

$\exists$ *Un nodo* esiste *ha una premessa e una conclusione, tali che se la premessa è etichettata da* $A[B/X]$, *dove* B *è una qualche formula, allora la conclusione è etichettata da* $\exists X.A$

Sia ora G *un grafo orientato ottenuto utilizzando i nodi appena descritti, e tale che:*

(α) *ogni arco di* G *è conclusione di un unico nodo;*

(β) *ogni arco di* G *è premessa di al più un nodo.*

Gli archi di G *che non sono premesse di alcun nodo sono dette* conclusioni *di* G. *Diremo che* G *è una pre-proof-structure se soddisfa le tre condizioni seguenti:*

1. *Scatola ! (*!-box*):*

   (a) *a ciascun nodo* of course n *è associato un (unico) sotto-grafo* $B^!$ *di* G *(che soddisfa* (α) *e* (β)*), tale che una delle conclusioni di* $B^!$ *è la conclusione di* n *e tutte le altre conclusioni (potrebbero anche non essercene altre) è conclusione di un nodo* pax. $B^!$ *è detta* scatola esponenziale *ed è rappresentata come in figura 7.2; diremo che* n *è la* porta principale *(*pal *in breve) di* $B^!$

   (b) *a ciascun nodo* pax n *è associata una scatola esponenziale* $B^!$ *di* G *tale che una delle conclusioni di* $B^!$ *è la conclusione di* n *(vedi fig. 7.2); diremo che* n *è una* porta ausiliaria *di* $B^!$

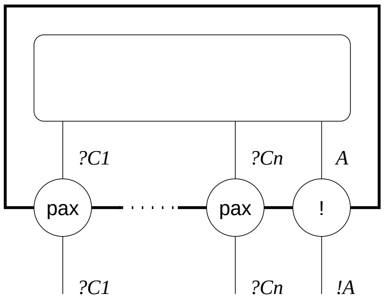


Figura 7.2: Una scatola esponenziale.

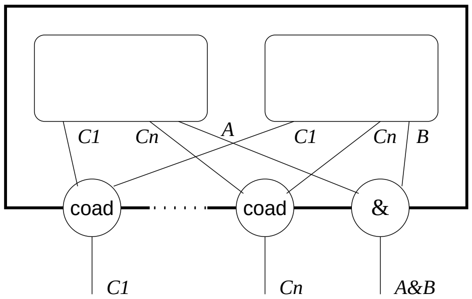


Figura 7.3: Una scatola additiva.

2. *Scatola additiva:*

   (a) *a ciascun nodo* with n *è associato un (unico) sotto-grafo* B *di* G *(che soddisfa* (α) *e* (β)*), tale che una delle conclusioni di* B *è la conclusione di* n *e tutte le altre conclusioni (potrebbero anche non essercene altre) è conclusione di un nodo* contrazione additiva. B *è detta* scatola additiva *ed è rappresentata come in figura 7.2; diremo che* n *è la* porta & *di* B

   (b) *a ciascun nodo* contrazione additiva n *è associata una scatola additiva* B *di* G *tale che una delle conclusioni di* B *è la conclusione di* n *(vedi fig. 7.2); diremo che* n *è una* porta coad *di* B

3. *Condizione di inscatolamento: due scatole qualsiasi (additive o esponenziali) sono disgiunte, oppure sono l'una contenuta nell'altra.*

Gli archi di una pre-proof-structure che non sono premessa di alcun nodo sono dette conclusioni *della pre-proof-structure.*

I nodi $⅋$, $\otimes$, !, ?D, $\oplus_1$, $\oplus_2$, & , $\forall$ e $\exists$ sono detti *nodi logici.*

Poiché gli archi di una pre-proof-structure sono orientati "dagli assiomi verso le conclusioni", potremo parlare senza ambiguità di una cammino che "sale" o che "scende" nella pre-proof-structure. Per lo stesso motivo, nelle rappresentazioni grafiche di cui ci serviremo nel seguito si utilizzeranno, per semplicità di notazione, archi non orientati.

Parleremo spesso di scatole, nodi o archi di una pre-proof-structure ρ contenuti in una scatola B (additiva o esponenziale) di ρ. Nel caso dei nodi, non considereremo le porte di B come nodi contenuti in B. Scrivendo "un nodo n [un arco a] di una scatola B (additiva o esponenziale)" di una data pre-proof-structure, sottintenderemo sempre che n [a] è contenuto in B oppure che n [a] è una porta [conclusione] di B. Se B è una scatola (additiva o esponenziale) di ρ, allora la più grande e la più piccola scatola di ρ contenente B è chiaramente definita, grazie alla condizione di inscatolamento della definizione 7.8.

Diremo che un nodo o un arco di una pre-proof-structure ρ è a *profondità* esponenziale [additiva] n in ρ se esso è contenuto in esattamente n scatole esponenziali [additive] di ρ. Nel caso di una scatola (additiva o esponenziale) B, diremo che B è a profondità esponenziale [additiva] n se essa è contenuta in esattamente n scatole esponenziali [additive], tutte diverse da B.

Osserviamo inoltre che, se ρ è una pre-proof-structure e B una sua scatola additiva o esponenziale, una conseguenza immediata della definizione 7.8 è che il contenuto di B è anch'esso una pre-proof-structure.

Definizione 7.9 (Proof-structure) *Sia ρ una pre-proof-structure. Diremo che ρ è una proof-structure se:*

(i) se una variabile appare libera nella formula che etichetta una delle conclusioni di ρ, allora questa non è la variabile propria di alcun nodo $\forall$ di ρ

(ii) ciascun(a occorrenza di un) nodo $\forall$ usa una variabile propria diversa

(iii) se X è la variabile propria di un nodo $\forall$ di una scatola B di ρ, allora tutte le occorrenze di X sono contenute in B

(iv) per ogni scatola B di ρ, la pre-proof-structure contenuta in B è una proof-structure.

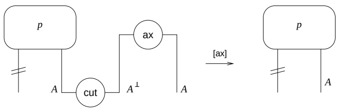
Un'osservazione importante: dietro alla definizione 7.9 si nasconde il fenomeno della *ridenominazione* delle variabili; se ad esempio abbiamo due proof-structure σ_1 e σ_2 e vogliamo connettere una conclusione di σ_1 con una conclusione di σ_2 per mezzo di un nodo $\otimes$, affinché il grafo così ottenuto soddisfi le condizioni della definizione si può essere costretti a dover rinominare alcune variabili che appaiono libere nelle formule di σ_1 e/o di σ_2.

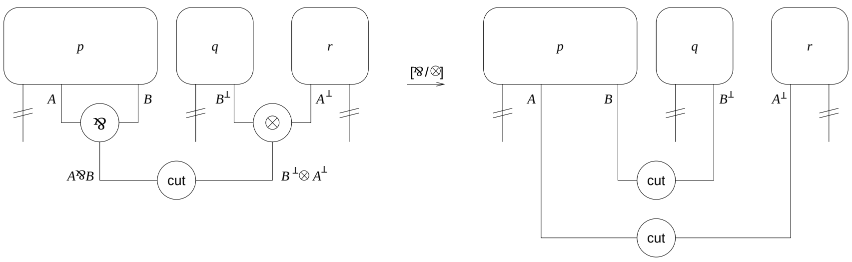
A questo punto è chiaro che, modulo ridenominazione delle variabili, a ciascuna derivazione di **LL** è associabile una proof-structure; possiamo dunque definire le proof-net della logica lineare completa nel modo seguente:

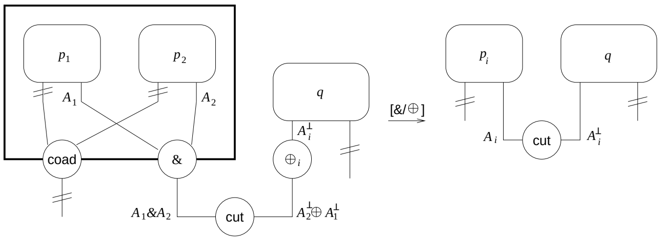
Definizione 7.10 (Proof-net) *Una proof-net è una proof-structure che proviene da una derivazione di **LL**.*

La definizione sopra sembra prendere una "scorciatoia" rispetto alla definizione mediante ACC data per le proof-net di **MLL** a partire dalle proof-structure moltiplicative. In effetti si tratta davvero di una scorciatoia: è possibile infatti introdurre criteri di correttezza anche per le proof-structure nel caso generale, in modo da separare mediante soli criteri strutturali quelle che vengono da derivazioni del calcolo dei sequenti (cioè quelle che chiamiamo proof-net) da quelle che invece non rappresentano dimostrazioni. Per definire tali criteri si deve però pagare un costo altissimo in termini di complessità; ciò ci spinge, in questa sede, a ignorare completamente la questione rimandando il lettore interessato al lavoro di Tortora [25], il quale espone approfonditamente i risultati più recenti in merito.

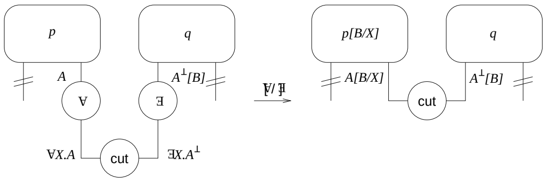
Anche per le proof-net di **LL** è possibile definire una procedura di cut-elimination a partire da un certo numero di passi elementari di riduzione, i quali sono, come per **MLL**, regole di riscrittura di grafi che producono proof-net a partire da proof-net. Riportiamo nel seguito la rappresentazione grafica di tutti i passi elementari di riduzione:



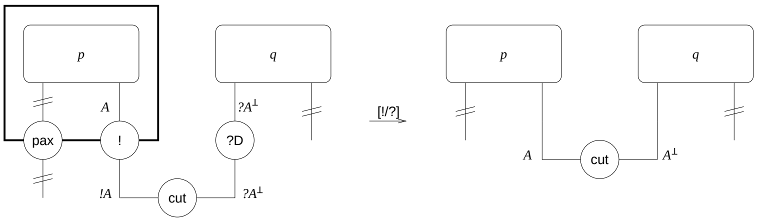




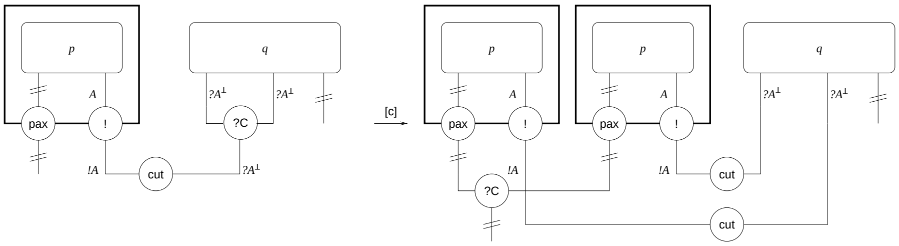
p1
p2
A1
A2
q
coad
&
⊕i
cut
A1&A2
[&/⊕]
pi
q
Ai
cut



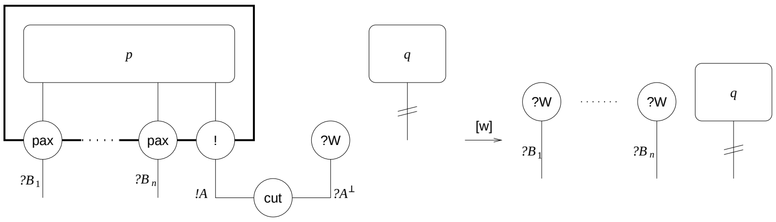
p
q
A
∀
∃
cut
∀X.A
p[B/X]
A[B/X]
q
cut



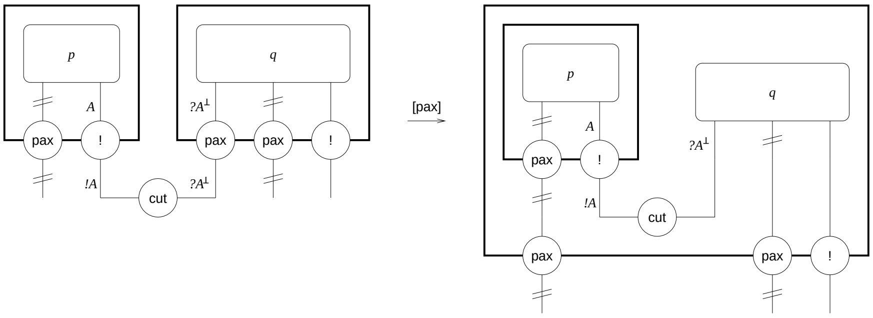
p
q
A
pax
!
?D
cut
!A
[!/?]
p
q
A
cut



p
A
pax
!
q
?C
cut
!A
[c]
p
A
pax
!
p
A
pax
!
q
!A
?C
!A
cut
cut



p
pax
pax
!
q
?W
cut
!A
[w]
?W
?W
q



p
A
pax
!
q
pax
pax
!
cut
!A
[pax]
p
A
pax
!
q
!A
cut
pax
pax
!

E' opportuno fare alcune considerazioni sul significato computazionale dei passi elementari di riduzione, in base al quale si può comprendere meglio anche la semantica dei connettivi della logica lineare, e la grande novità che questi introducono rispetto alle loro controparti classiche.

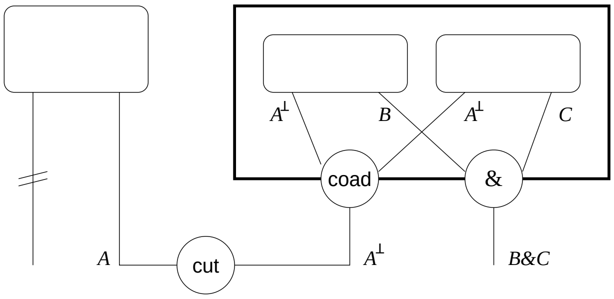
Il passo [ax] è il cardine della cut-elimination; è questo a far "sparire" definitivamente i cut, terminando il processo di comunicazione innescato dall'aver tagliato la conclusione di una proof-net con quella di un'altra proof-net. Come abbiamo già osservato per il calcolo dei sequenti, l'intero processo di cut-elimination può essere visto come una "migrazione" dei cut verso gli assiomi.

L'eliminazione del taglio [$\parr$ /$\otimes$] innesca un processo "parallelo" in cui i due nuovi cut si possono ridurre in modo il più possibile indipendente. L'eliminazione di un taglio [& /$\oplus$] rappresenta chiaramente la scelta di una delle due sotto-proof-net contenute in una scatola additiva, operata da parte del nodo $\oplus$, che può essere naturalmente un $\oplus_1$ o un $\oplus_2$; i tagli additivi possono dunque essere visti come la base di un'esecuzione *condizionale*, come in una sorta di "if…then…else". Il passo [$\forall/\exists$] è l'"applicazione universale" del sistema $\mathbb{F}$; esso consente di applicare una funzione polimorfa ad un tipo, in modo da ottenere una funzione che opera su quel tipo specifico.

I tagli esponenziali sono invece la caratteristica fondamentale di **LL**, che permette di esprimere in dettaglio l'utilizzo delle risorse nel corso della computazione. Essenzialmente, una scatola esponenziale può essere vista come una risorsa non-lineare (nel senso dato nella sezione 6.3), inesauribile; questa può essere *cancellata*, *duplicata*, e infine *acceduta* per essere utilizzata. I tre passi [w], [c] e [!/?] rappresentano ciascuno una di queste tre situazioni.

Il passo [pax] non ha un vero e proprio contenuto computazionale; è semplicemente una di quelle che nei sistemi di riscrittura vengono dette "conversioni commutative" (*commuting conversions*). E' in effetti un residuo delle famose "riduzioni commutative" del calcolo dei sequenti, che abbiamo incontrato quando abbiamo discusso la cut-elimination per **LK** e **LJ**, le quali servivano a far "salire" un cut in modo che alla fine questo incontrasse le regole che introducevano la formula tagliata. La presenza di un tale "relitto" è dovuta al fatto che, in realtà, le scatole esponenziali sono dei veri e propri sequenti: in generale, le scatole (sia esponenziali che additive) costituiscono un momento di "sincronizzazione" delle proof-net con il calcolo dei sequenti, e creano dunque una situazione in cui il parallelismo tipico delle proof-net viene perso, e si ritorna alla sequenzialità propria invece del calcolo dei sequenti.

A dir la verità, i cut di tipo [pax] non sono gli unici tagli commutativi possibili; non sono infatti state prese in considerazione le scatole additive, per le quali si può verificare la seguente situazione:



dove A non è conclusione di un nodo assioma. Le riduzioni dei tagli di questo tipo, chiamate [ccad] (*conversione commutativa additiva*), presentano non poche difficoltà ad essere definite; noi non entreremo in dettaglio, lasciando al lettore curioso la facoltà di documentarsi in [25].

Tra l'altro, le riduzioni commutative additive sono responsabili di spiacevoli fenomeni di non confluenza in **LL**. Tuttavia, è stato dimostrato che la procedura di normalizzazione cosiddetta *lazy*, che noi però chiameremo *additive-lazy*[1], la quale consiste nell'applicare tutti i passi elementari di riduzione ad eccezione di [ccad] e in cui non si normalizzano tagli interni a scatole additive, è confluente (si vedano [3] e [26]). Inoltre, dalle considerazioni fatte da Girard in [10] (dove tra l'altro si introduce il termine *lazy* per questo tipo di normalizzazione), se nelle conclusioni della proof-net che si vuole ridurre non ci sono sottoformule positive della forma $A \mathbin{\&} B$, allora la procedura *additive-lazy* termina con una forma normale, ciò dà luogo a proof-net cut-free. Questo fatto è di estrema importanza perché, come vedremo, tale situazione si applica a tutte le proof-net che rappresentano funzioni che operano su tipi di dato di un qualche interesse, come ad esempio interi, stringhe, booleani, alberi, ecc. Di conseguenza, da un punto di vista strettamente computazionale, il passo di riduzione [ccad] può essere lasciato indefinito, e questa sarà proprio la scelta che faremo noi.

In realtà, in merito ai tagli commutativi additivi vale un risultato ancora più forte, dimostrato da Tortora in [25]: nel caso in cui una proof-net non contenga conclusioni con occorrenze positive del connettivo $\&$, la riduzione *additive-lazy* e la riduzione che utilizza anche i passi [ccad] *conducono alla stessa forma normale*. Ciò significa che l'aggiunta del passo [ccad], è davvero una questione che, ai nostri scopi, può essere messa in secondo piano.

[1] Questo per distinguerla dalle strategie di normalizzazione per il λ-calcolo che hanno lo stesso nome

Capitolo 8

I sistemi logici a bassa complessità

In questo capitolo effettueremo una "ricognizione" sullo stato attuale della ricerca nel campo della caratterizzazione di classi di complessità mediante sotto-sistemi della logica lineare. Cercheremo dunque di ricapitolare brevemente i risultati trovati fino a questo momento, introducendo in un qualche dettaglio i sistemi più utilizzati.

8.1 La complessità della riduzione delle proof-net

Come accennato nel capitolo 6, in logica lineare si possono "tradurre" tutte le derivazioni sia della logica classica che della logica intuizionista. Di conseguenza, non possiamo aspettarci che **LL** *di per sé* possa risultare in alcun modo "meglio" (dal punto di vista della complessità della cut-elimination) dei sistemi **LK** e **LJ**. A priori, dunque, anche in logica lineare l'eliminazione dei tagli da una dimostrazione produce derivazioni la cui dimensione non è maggiorabile da nessuna funzione elementare, e quindi anche in termini temporali (per l'osservazione fatta in 5.3), la procedura di cut-elimination in **LL** risulta essere iperesponenziale.

Tuttavia, in logica lineare abbiamo a disposizione uno strumento inesistente negli altri sistemi, che abbiamo visto essere molto vantaggioso nella rappresentazione dell'eliminazione dei tagli dalle derivazioni: stiamo parlando naturalmente delle *proof-net*, introdotte nel capitolo precedente. Quando si parla di proof-net, esistono due grandezze fondamentali che ne caratterizzano la struttura, già introdotte in 7.1 e 7.2: la prima è la *dimensione* (o *size*), che corrisponde al numero di nodi presenti nella proof-net stessa, la seconda è invece la *profondità*, che in questo momento sarà per noi esclusivamente *esponenziale*. Ricordiamo che la profondità esponenziale, riferita ai nodi, è semplicemente il numero di scatole esponenziali dentro le quali un nodo si trova, mentre riferita all'intera proof-net è nient'altro che la profondità del suo nodo di profondità massima. Se π è una proof-net, indichiamo con $\sharp(\pi)$ la sua dimensione, e con $\partial(\pi)$ la sua profondità.

Ora, la cosa più naturale che viene in mente è caratterizzare la complessità della procedura di cut-elimination mettendo in relazione la dimensione di una proof-net con il numero di step elementari di riduzione richiesti per eliminare tutti i cut. Infatti, in base alla corrispondenza di Curry-Howard sappiamo che una proof-net (essendo una dimostrazione) è un programma, e il processo di cut-elimination è l'esecuzione di tale programma. Se prendiamo una proof-net cut-free π, diciamo di conclusioni (almeno) $A^{\perp}$ e B, e la tagliamo contro un'altra proof-net cut-free ι con conclusione (almeno) A, possiamo considerare π come il programma e ι come il suo input; la proof-net θ risultante dal *cut* delle due è dunque il programma applicato al suo input, mentre la proof-net cut-free ν ottenuta dopo l'eliminazione del taglio è il risultato restituito dal programma. Supponiamo ora di aver trovato una funzione f tale che il numero di step elementari di riduzione necessario per arrivare da θ a ν sia $O(f(\sharp(\theta)))$; poiché $\sharp(\theta) = \sharp(\pi) + \sharp(\iota) + 1$ (c'è solo un cut in più rispetto alle due) e poiché, qualunque sia la proof-net ι scelta come input, la dimensione di π resta costante, possiamo ben dire che il numero di step richiesti è anche $O(f(\sharp(\iota)))$. Caratterizzare la complessità della riduzione di una proof-net in funzione della sua dimensione equivale dunque a trovare il legame tra "tempo di esecuzione" e dimensione dell'input, che è esattamente ciò che si vuol fare in un'analisi di complessità temporale.

Seguendo questo approccio, è possibile iniziare a fare qualche osservazione. Affinché in una proof-net π sia possibile eseguire uno step elementare di riduzione, dev'esserci in π almeno un nodo cut. Il numero di nodi cut presenti in π è senz'altro limitato da $\sharp(\pi)$; allo stesso modo, se $\pi \to \pi'$, il numero di cut presenti in π' è anch'esso limitato da $\sharp(\pi')$. Di conseguenza, se si riesce a trovare in che modo la riduzione di un cut influisce sulla dimensione della proof-net ridotta, si può anche stimare la lunghezza del processo di cut-elimination.

Osservando gli step elementari di riduzione di **LL** definiti a pagina 138, possiamo dividere le riduzioni in due classi, una costituita dalle regole di riscrittura che diminuiscono la size, l'altra da regole che, in generale, la fanno aumentare:

– $[\mathsf{ax}]$, $[\parr/\otimes]$, $[\&/\oplus]$, $[\forall/\exists]$, $[!/?]$, $[\mathsf{w}]$ e $[\mathsf{pax}]$ fanno tutti, in generale, diminuire la size della proof-net ridotta rispetto alla proof-net di partenza

+ $[\mathsf{c}]$ è l'unico step in grado di far aumentare in ogni caso la dimensione della proof-net ridotta dagli step elementari di riduzione abbiamo sempre escluso, come al solito, lo step [coad].

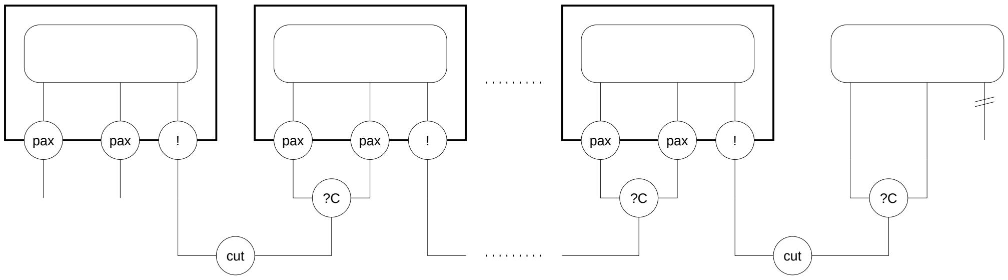
In **LL**, il responsabile dell'esplosione della complessità del processo di cut-elimination viene messo indiscutibilmente allo scoperto: l'analisi della riduzione dei cut sulle contrazioni è la chiave per catturare la complessità dell'eliminazione del taglio. Se, infatti, eliminiamo dalla logica lineare i connettivi esponenziali, costruendo quella a cui in genere ci si riferisce con l'abbreviazione **MALLq** (*Multiplicative-Additive Linear Logic with quantifiers* — i quantificatori possono essere a qualsiasi ordine, noi diamo sempre per scontato che si tratti del secondo ordine), otteniamo il seguente risultato:

Teorema 8.1 *Sia π una proof-net di* **MALLq**, *tale che $\sharp(\pi) = s$. Allora, la procedura di cut-elimination additive-lazy applicata a π conduce ad una proof-net cut-free in $O(s)$ step elementari di riduzione.*

Dimostrazione. Gli unici step elementari di riduzione applicabili in **MALLq** appartengono alla classe di step che conducono sempre ad una proof-net ridotta di dimensione più piccola di quella di partenza. Di conseguenza, se $\pi \to \pi'$, avremo $\sharp(\pi') < \sharp(\pi)$, e così via dopo ogni step applicato. Poiché ad ogni istante il numero di cut non può eccedere la size della proof-net, e poiché s è un bound superiore alla dimensione di qualsiasi proof-net generata nel corso della cut-elimination, la procedura termina in un numero di step addirittura *sub-lineare* in s, vale a dire che il numero di step applicati è strettamente minore della dimensione di π. □

I soli programmi a poter essere rappresentati in **MALLq** sono dunque programmi la cui esecuzione impiega un tempo lineare nell'input. Il problema è che **MALLq** è un sistema terribilmente inespressivo, e al suo interno praticamente non trova posto alcun programma interessante; ad esempio, in **MALLq** non è neanche possibile rappresentare adeguatamente gli interi di Church.

L'aggiunta degli esponenziali consente di recuperare il potere espressivo perduto, tornando però ai livelli del sistema $\mathbb{F}$, e dunque facendoci uscire da qualsiasi classe di complessità definibile. Tuttavia, dei quattro step di riduzione elementari introdotti dalla presenza dei connettivi esponenziali, sappiamo che quello davvero problematico è esclusivamente [c]. Tale step è in grado di *duplicare* un pezzo di proof-net; per l'esattezza ciò che viene duplicato è il contenuto di una scatola esponenziale. Iterare n volte una funzione che duplica l'input significa banalmente costruire una funzione esponenziale, 2^n nel nostro caso. Nell'ambito delle proof-net, la seguente situazione corrisponde proprio all'iterazione che abbiamo in mente:



Se supponiamo che le scatole che formano la "catena" siano n e, per semplicità, che esse siano tutte identiche, è chiaro che una volta ridotte tutte le contrazioni otterremo una proof-net contenente in totale $\sum_{i=1}^{n} 2^i = 2(2^n - 1)$ copie della nostra scatola: il !-box "più a destra" viene semplicemente duplicato, quello alla sua sinistra viene invece quadruplicato, e così via fino al !-box "più a sinistra", del quale avremo esattamente 2^n repliche.

Abbiamo così scoperto una *dinamica esponenziale* nel processo di cut-elimination delle proof-net. In realtà, una volte escluse (come nel nostro caso) le riduzioni commutative additive, la dinamica in questione è l'unica in grado di generare esplosioni esponenziali nella size delle proof-net; infatti, solo l'attivazione di un taglio su una contrazione può duplicare una scatola, e solo la ripetuta attivazione di tanti tagli del genere "in cascata" può iterare la duplicazione.

Sappiamo che in **LL** è possibile fare molto di più; ad esempio, si dovrebbe poter iterare la dinamica esponenziale per ottenerne una *iperesponenziale*. Tuttavia, il "prototipo" di dinamica iperesponenziale è molto più complesso di quello della dinamica esponenziale appena descritta, e non è dunque il caso di entrare nei dettagli in questa sede. Possiamo però, in modo intuitivo, mettere in luce quella che è la caratteristica fondamentale delle dinamiche iperesponenziali. Abbiamo visto che la "catena" disegnata sopra, composta da n scatole, una volta normalizzata genera $O(2^n)$ copie di tali scatole; per ottenere un processo iperesponenziale occorrerebbe a questo punto essere in grado di prendere queste $O(2^n)$ scatole e costruire una simile catena con esse, in modo da ottenere, dopo la normalizzazione, $O(2^{2^n})$ scatole, e così via. Il problema è che a partire dalle $O(2^n)$ scatole generate dalla riduzione della catena non è possibile creare un'altra catena, per il semplice fatto che la "struttura" che esse formano ora non contiene più tagli su contrazioni, mentre nelle catene in questione è indispensabile che le scatole siano tagliate l'una contro l'altra tramite cut su contrazioni.

Si può allora immaginare la seguente situazione. Supponiamo di aver costruito una proof-net $\chi(n)$ che, se fatta interagire con un !-box, crea una "catena esponenziale" (cioè una struttura capace di generare una dinamica esponenziale come quella che abbiamo visto) composta da n copie della scatola. Supponiamo che dentro questa scatola ci sia proprio $\chi(1)$ e che, facendo interagire due di queste scatole tagliando la porta principale di una con la porta ausiliaria dell'altra, si crei un'unica scatola (e questo è sicuro perché lo step [pax] non cancella scatole) contenente $\chi(2)$, la quale interagendo con un'altra scatola ancora generi una scatola con dentro $\chi(3)$, e così via. Per semplicità, chiameremo $!\chi(n)$ la proof-net costituita da un !-box contenente $\chi(n)$. Abbiamo dunque che, facendo interagire $!\chi(1)$ con $\chi(n)$, si ottiene una catena esponenziale costituita da n copie di $!\chi(1)$, la quale, eliminando i tagli sulle contrazioni, conduce ad una "catena senza contrazioni" contenente $O(2^n)$ copie di $!\chi(1)$. Queste ultime, interagendo fra loro, si normalizzeranno alla fine in $!\chi(O(2^n))$.

Ricapitolando il tutto, dall'interazione di $!\chi(1)$ con $\chi(n)$ abbiamo ottenuto $!\chi(O(2^n))$; osserviamo che il fatto che l'"input" di $\chi(n)$ debba essere un box è assolutamente necessario, poiché solo i box possono essere copiati, e per creare una catena esponenziale di lunghezza n non si può fare a meno di copiare n volte un box. Con $!\chi(O(2^n))$ siamo arrivati quasi al punto di partenza, ma soltanto "quasi". Ciò che vorremmo fare ora infatti è ripetere il gioco tagliando $!\chi(O(2^n))$ con $!\chi(1)$, ottenendo $!\chi(O(2^{2^n}))$; chiaramente però ciò è impossibile, visto che contro $!\chi(1)$ può essere tagliata $\chi(\cdot)$, e non $!\chi(\cdot)$. È questo il punto chiave delle dinamiche iperesponenziali che volevamo mettere in luce: le catene esponenziali non possono produrre che scatole, e dunque per iterare il processo bisogna necessariamente, all'inizio di ogni iterazione, "aprire" tali scatole. Aprire una scatola significa eseguire uno step [!/?], cioè far interagire un nodo dereliction con la porta principale della scatola della quale si vuole accedere l'interno. In questo caso, possiamo supporre che, prima di essere fatta interagire di nuovo con $!\chi(1)$, la proof-net $!\chi(O(2^n))$ si ritrovi ad essere tagliata contro una dereliction, in modo da tirar fuori $\chi(O(2^n))$ dalla scatola per poter così iterare il processo. Come abbiamo già sottolineato, tutto ciò è stato descritto in modo molto informale, ma ciò che ci interessa è di aver fornito l'intuizione fondamentale sulle dinamiche che consentono, nel corso della riduzione di una proof-net, di generare normalizzazioni non elementari.

In sintesi, i risultati trovati finora sulla complessità della riduzione delle proof-net sono i seguenti:

(a) per ottenere dinamiche iperesponenziali serve lo step [!/?]

(b) per ottenere dinamiche esponenziali bisogna costruire una catena di scatole legate tra loro per mezzo di tagli su contrazioni

vedremo nel seguito come l'aver isolato queste due semplici "regole" si riveli estremamente utile per controllare la complessità in logica lineare.

8.2 Controllare la complessità in logica lineare

8.2.1 I sistemi affini

Il primo sotto-sistema a bassa complessità della logica lineare ad essere introdotto è stato quello noto con il nome di *Light Linear Logic* (abbreviato con **LLL**), introdotto da Girard in [12][1]. In sostanza, **LLL** sfrutta le intuizioni di cui abbiamo parlato nella sezione precedente in modo da inibire le dinamiche esponenziali, conducendo così ad un sistema che corrisponde esattamente alla classe **P**, vale a dire la classe delle funzioni[2] calcolabili in tempo polinomiale da una macchina di Turing deterministica. Inoltre, liberalizzando una certa regola di **LLL**, si ottiene una logica che corrisponde esattamente alla classe **ELEMENTARY**, cui è stato dato il nome di **ELL** (*Elementary Linear Logic*).

Tuttavia, il sistema originale di Girard presenta parecchie complicazioni, prima fra tutte un calcolo dei sequenti un po' particolare, in cui non esistono solo formule ma anche "blocchi di formule" e "formule scaricate", il tutto per far funzionare le cose in modo che l'isomorfismo fondamentale degli esponenziali ($!A \otimes !B \simeq !(A \ \& \ B)$) sia derivabile anche in **LLL**. In questa sede preferiamo dunque introdurre direttamente la variante "affine" di **LLL**, che ha il pregio di essere enormemente più semplice.

La Logica Affine (introdotta da Asperti in [1]) è semplicemente la logica lineare in cui la regola del weakening è completamente liberalizzata; in altre parole, si può introdurre per weakening una formula qualsiasi, non necessariamente esponenziale. Naturalmente, la contrazione resta possibile solo su formule di tipo $?A$, altrimenti il sistema collasserebbe sulla vecchia logica classica.

La logica affine ha due grandissime differenze rispetto alla logica lineare. In primo luogo, aver liberalizzato il weakening fa riapparire lo spettro della non confluenza nella riduzione delle derivazioni; per questo motivo, poiché siamo interessati esclusivamente agli aspetti computazionali della logica affine, ci sposteremo definitivamente nel suo frammento intuizionista, denominato **IAL** (*Intuitionistic Affine Logic*). In secondo luogo, e questa invece è una differenza "in positivo", la liberalizzazione del weakening rende la congiunzione moltiplicativa "più forte" di quella additiva (diviene cioè derivabile $A \otimes B \vdash A \ \& \ B$); di conseguenza, il frammento additivo diventa

[1] A dir la verità c'erano già stati altri tentativi in precedenza, il più noto dei quali è **BLL** (*Bounded Linear Logic*), tutti però lontani dall'essere davvero soddisfacenti.

[2] Ad essere precisi ciò corrisponderebbe a **FP**, di cui **P** è un sottoinsieme (ogni problema è una funzione, mentre non vale il viceversa, come abbiamo già discusso in 2.1); tuttavia, essendo la nomenclatura **FP** molto meno conosciuta, d'ora in poi confonderemo le due cose.

rappresentabile per mezzo del solo frammento moltiplicativo (naturalmente non è vero il viceversa), e i connettivi $\&$ e $\oplus$ non servono più. Questa è naturalmente una grossa semplificazione dal punto di vista della sintassi di **IAL**. Per chiarezza, riportiamo qui di seguito il calcolo dei sequenti del frammento intuizionista della logica affine:

▶ **Regole dell'identità**

$$\frac{}{A \vdash A}\,\mathsf{ax} \qquad \frac{\Gamma \vdash A \quad \Delta, A \vdash C}{\Gamma, \Delta \vdash C}\,\mathsf{cut}$$

▶ **Regole strutturali**

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\,\mathsf{X} \qquad \frac{\Gamma \vdash C}{\Gamma, A \vdash C}\,\mathsf{W}$$

▶ **Regole logiche moltiplicative**

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\,\otimes\mathsf{L} \qquad \frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\,\otimes\mathsf{R}$$

$$\frac{\Gamma \vdash A \quad \Delta, B \vdash C}{\Gamma, \Delta, A \multimap B \vdash C}\,\multimap\mathsf{L} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\,\multimap\mathsf{R}$$

▶ **Regole logiche esponenziali**

$$\frac{\Gamma, A \vdash C}{\Gamma, !A \vdash C}\,?\mathsf{D} \qquad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A}\,!\mathsf{P}$$

$$\frac{\Gamma, !A, !A \vdash C}{\Gamma, !A \vdash C}\,?\mathsf{C}$$

▶ **Regole logiche per i quantificatori**

$$\frac{\Gamma, A[B] \vdash C}{\Gamma, \forall X.A \vdash C}\,\forall\mathsf{L} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \forall X.A}\,\forall\mathsf{R} \quad (\star)$$

La regola $(\star)$ è sempre la solita: X non dev'essere libera in Γ.

Naturalmente, anche per **IAL** sono definibili le proof-net, che differiranno da quelle di **ILL** per l'assenza[3] dei nodi e delle scatole additive, e per la presenza di un nodo weakening liberalizzato, che può avere come conclusione una formula A qualsiasi, invece che obbligatoriamente una formula $?A$. Anche la cut-elimination resta praticamente identica, limitandola al frammento moltiplicativo e modificando lo step [w] in modo che questo cancelli non solo scatole esponenziali ma, più genericamente, sotto-proof-net qualsiasi. In

[3] Molto gradita!

particolare, in **IAL** troviamo senz'altro lo step [!/?], eseguibile qualora venga tagliata la porta principale di un !-box con un nodo dereliction.

Il lettore con buona memoria ricorderà senz'altro la "regola" (a) esposta alla fine della sezione precedente: per generare dinamiche iperesponenziali è strettamente necessario lo step [!/?]. Eliminare la possibilità di eseguire tale step significa far passare la logica nel dominio delle funzioni elementari, poiché è sempre possibile costruire "individualmente" torri esponenziali di altezza qualsiasi, ma non è più possibile rappresentare un processo che le generi tutte. Come fare dunque per mettere in pratica tale intuizione? La soluzione è più semplice di quanto si possa pensare: basta eliminare le regole della dereliction e della promotion, e rimpiazzarle con un unica regola che le "ingloba" entrambe:

$$\frac{\Gamma \vdash A}{!\Gamma \vdash !A}\,!$$

La regola in questione, denominata *soft-promotion*, costringe a dover "inscatolare" una proof-net ogni volta che si voglia effettuare una dereliction su una sua conclusione; non solo, con la soft-promotion la dereliction è "totale", cioè non può più essere effettuata su una sola formula.

È chiaro che, in presenza della soft-promotion, nelle proof-net non esistono più nodi dereliction; le pax di un !-box diventano, per così dire, dereliction e porte ausiliarie al tempo stesso. I due step elementari di riduzione [!/?] e [pax] verranno dunque rimpiazzati da un unico step [!mrg], in base al quale due scatole esponenziali semplicemente si "fondono" in un'unica scatola, con il taglio tra le due che entra in questa nuova scatola e mette in comunicazione quelle che erano le premesse della porta principale di una scatola e della porta ausiliaria dell'altra. Dal punto di vista del calcolo dei sequenti, succede quanto segue:

$$\frac{\dfrac{\Gamma \vdash A}{!\Gamma \vdash !A}\,! \qquad \dfrac{\Delta, A, \vdash C}{!\Delta, !A \vdash !C}\,!}{!\Gamma, !\Delta \vdash !C}\,\mathsf{cut} \quad \longrightarrow \quad \dfrac{\dfrac{\Gamma \vdash A \qquad \Delta, A, \vdash C}{\Gamma, \Delta \vdash C}\,\mathsf{cut}}{!\Gamma, !\Delta \vdash !C}\,!$$

Osserviamo che, senza lo step [!/?], *non è più possibile aprire alcuna scatola*; in sostanza, la profondità di una proof-net resta praticamente inalterata nel corso della cut-elimination. Tutto quello che può succedere è che uno step [w] elimini qualche scatola, possibilmente facendo diminuire la profondità complessiva della proof-net, ma ciò non ha alcun effetto nell'ottica della dinamica iperesponenziale che vogliamo proibire.

È possibile dimostrare che, con quest'intuizione dell'inibizione della dereliction, abbiamo effettivamente colto nel segno: la logica che abbiamo appena definito, nota come **IEAL** (*Intuitionistic Elementary Affine Logic*), corrisponde esattamente alla classe delle funzioni calcolabili in tempo elementare. Naturalmente, verificando che non è più possibile generare dinamiche iperesponenziali dimostreremmo solo la *correttezza* di **IEAL**; al fine di realizzare la corrispondenza, occorre una prova della *completezza* del sistema rispetto a **ELEMENTARY**, vale a dire che tutte le funzioni calcolabili in tempo elementare sono rappresentabili in **IEAL**. Ciò può essere fatto, ad esempio, sfruttando la caratterizzazione assiomatica di Kalmar, di cui abbiamo parlato in 5.2.

Con **IEAL** abbiamo varcato il primo confine di complessità possibile, visto che **ELEMENTARY** è la più grande classe di complessità di un qualche interesse. Tornando alle osservazioni fatte nella sezione precedente (in particolare alla "regola" (b)), notiamo che in **IEAL** è ancora possibile, come del resto potevamo aspettarci, costruire "catene" di scatole in modo che la loro riduzione generi una dinamica esponenziale. Se riuscissimo a proibire anche queste configurazioni, avremmo fatto un salto enorme: avendo impedito ogni dinamica esponenziale, saremmo scesi addirittura nel mondo del tempo polinomiale. Anche realizzare quest'idea è quasi banale (dal punto di vista sintattico). Poiché le nostre "catene esponenziali" sono basate sul fatto che ogni scatola, prima di essere tagliata contro un'altra, manda le conclusioni delle sue porte ausiliarie in una contrazione, una buona idea è quella di restringere il contesto della regola di soft-promotion (cioè la parte a sinistra del sequente) in modo che esso contenga *al massimo una formula.* In termini di proof-net, ciò corrisponde all'aver forzato tutti i !-box ad avere al massimo una porta ausiliaria; poiché per realizzare una catena esponenziale servono box con almeno due porte ausiliarie, così facendo abbiamo scongiurato il pericolo che si possano mai formare simili configurazioni.

La regola della soft-promotion va dunque rimpiazzata con

$$\frac{B \vdash A}{!B \vdash !A}\,!$$

dove B può anche non esserci. Questa restrizione fortissima alla soft-promotion ha però un effetto collaterale molto spiacevole: il sistema perde drammaticamente potere espressivo, tanto da non poter più rappresentare al suo interno oggetti fondamentali come gli interi di Church. In altre parole, il sistema è senz'altro corretto dal punto di vista della complessità polinomiale, come lo era **MALLq** per le funzioni lineari, ma è ben lontano dall'essere completo.

Per recuperare il potere espressivo bisogna ricorrere ad un nuovo connettivo esponenziale (già presente in **LLL**, la quale soffre dello stesso problema) denominato *neutral*, e rappresentato dal simbolo §(per questo motivo ci si riferisce ad esso anche con il termine *paragrafo*), per il quale si introduce la seguente regola:

$$\frac{\Gamma, \Delta \vdash A}{!\Gamma, \S\Delta \vdash \S A}\ \S$$

Il connettivo paragrafo è *autoduale*, vale a dire che, per qualsiasi formula A,

$$(\S A)^{\perp} = \S A^{\perp}$$

Questa nuova modalità esponenziale induce la comparsa nelle proof-net di un'altra scatola, il cosiddetto §-box. Naturalmente, si può definire uno step di riduzione elementare per i tagli §/§, che possiamo chiamare [§mrg] e che è sostanzialmente identico a [!mrg], salvo il fatto che le scatole coinvolte sono entrambe §-box. Può chiaramente verificarsi che la porta principale di un !-box venga tagliata con la porta ausiliaria di un §-box; ciò non causa assolutamente problemi, visto che la riduzione di questo taglio ([!§mrg]), come al solito, "fonde" le due scatole in un unico §-box e fa "entrare" il taglio dentro tale scatola.

La definizione autoduale di § non è ancora del tutto stabilita. In effetti, le interpretazioni semantiche di **LLL** sembrerebbero spingere in una direzione non autoduale del connettivo; tuttavia, da un punto di vista strettamente informatico, le due soluzioni sono del tutto equivalenti. La cosa fondamentale infatti è che i §-box *non possono essere duplicati*; non avendo porte !, non possono infatti essere tagliati contro alcuna contrazione. Il paragrafo dunque ha un utilizzo esclusivamente espressivo: esso non influisce minimamente sulle dinamiche di complessità.

La logica così definita, che prende il nome di **ILAL** (*Intuitionistic Light Affine Logic*, [1], [2]), è ciò che cercavamo. Anche per **ILAL** si possono dimostrare i due risultati richiesti per la corrispondenza con la classe delle funzioni polinomiali: le funzioni programmabili nel sistema sono incluse in **P** (correttezza) e l'insieme di tali funzioni include **P** (completezza). Sfortunatamente, per quest'ultima dimostrazione non esistono caratterizzazioni assiomatiche interpretabili in logica come quella di Kalmar; di conseguenza, la prova si basa su una tediosissima codifica delle macchine di Turing polinomiali in **ILAL**, per la quale invitiamo il lettore curioso a documentarsi in [22] e [2]. Dal canto nostro, la sezione 9.4 del prossimo capitolo sarà dedicata ad una dimostrazione pressoché identica, e dunque non approfondiremo ulteriormente la questione in questo momento.

8.2.2 Altri sistemi

Tornando alla logica lineare, abbiamo già detto come esistano due sistemi basati su di essa, detti **LLL** e **ELL**, i quali corrispondono rispettivamente a **P** e **ELEMENTARY**. Recentemente, Danos e Joinet hanno condotto un'opera di semplificazione del lavoro di Girard, dimostrando che esiste una caratterizzazione logica di **ELEMENTARY** basata su un sistema costituito da un *sottoinsieme* delle proof-net di **LL**. In sostanza, anziché definire un nuovo sistema logico, Danos e Joinet hanno imposto una semplice condizione sulle proof-net della logica lineare (detta *condizione di stratificazione*), per mezzo della quale di possono "separare" tutte quelle la cui normalizzazione è elementare e il cui potere espressivo è sufficientemente elevato da includere tutta la classe **ELEMENTARY**. La definizione del sistema, che noi chiameremo $\overline{\mathbf{ELL}}$ per distinguerlo da quello originale di Girard, con la dimostrazione della sua correttezza e completezza è esposta dagli autori in [8]. Noi non scenderemo ulteriormente in dettagli, giacché il prossimo capitolo sarà interamente dedicato alla definizione di un sistema, che abbiamo chiamato $\overline{\mathbf{LLL}}$, il quale sostanzialmente estende l'approccio di Danos e Joinet a **LLL**, costituendo dunque una caratterizzazione logica della classe **P**.

Tutti sistemi a bassa complessità visti finora, assieme a quello che presenteremo nel prossimo capitolo, seguono quello che potremo chiamare *approccio light*. In particolare, nel definire un sistema a complessità polinomiale, abbiamo osservato come sia fondamentale l'interdizione di "catene esponenziali", le quali presentano un'applicazione iterata dello step [c]. Ma un cut ridotto da tale step, come tutti i cut, ha due premesse: una è la porta principale di un !-box, l'altra è una contrazione.

Quello che abbiamo chiamato "approccio light" si occupa in sostanza di inibire cut di questo tipo *agendo sulle scatole*: restringendo l'uso della dereliction e limitando il numero di pax dei !-box si costringono le scatole ad avere una struttura tale da rendere impossibile la formazione di strutture in grado di generare dinamiche esponenziali. Come abbiamo più volte sottolineato, la chiave di queste dinamiche è che un !-box non solo viene duplicato ma, tramite le sue porte ausiliarie contratte, diventa lui stesso il "motore" della successiva duplicazione di un'altra scatola, e così via. La morale dell'approccio light è dunque la seguente: *i !-box sono gli unici a poter essere duplicati, ma non possono essi stessi duplicare.*

Se ci rivolgiamo all'altra premessa del cut, possiamo immaginare di poter modificare in qualche modo, piuttosto che le scatole, le contrazioni, così da ottenere un risultato analogo. Questa strada alternativa, che chiameremo *approccio soft*, è stata intrapresa con successo da Yves Lafont in [17], nel quale viene definito un sistema a complessità polinomiale denominato **ISLL** (*Intuitionistic Soft Linear Logic*).

La definizione di **SLL** è molto elegante. In primo luogo, si rimpiazzano le quattro regole esponenziali della logica lineare (?P, ?D, ?C e ?W) con le seguenti tre regole (una delle quali è una nostra recente conoscenza):

$$\frac{\Gamma \vdash A}{!\Gamma \vdash !A}\,! \qquad \frac{\Gamma, A^{(n)} \vdash C}{\Gamma,\ !A \vdash C}\,\mathsf{mux} \qquad \frac{\Gamma,\ !!A \vdash C}{\Gamma,\ !A \vdash C}\,\mathsf{dig}$$

Nella seconda regola, detta *multiplexing*, $A^{(n)} = \underbrace{A, \ldots, A}_{n}$; la terza regola è invece detta *digging*.

Non è difficile dimostrare che mediante queste tre regole (e il cut) si possono simulare le quattro regole esponenziali originali, e viceversa. Di conseguenza, fin qui non abbiamo fatto null'altro che "ridefinire" **ILL**. Il nostro sistema si definisce allora come segue: **ISLL** è semplicemente questa "riformulazione" di **ILL** dalla quale viene tolta la regola del digging. È qui che ritroviamo l'intuizione di quello che abbiamo chiamato "approccio soft". Consideriamo infatti la seguente derivazione:

$$\dfrac{\dfrac{\Gamma,\ !A,\ !A \vdash C}{\Gamma,\ !!A \vdash C}\,\mathsf{mux}}{\Gamma,\ !A \vdash C}\,\mathsf{dig}$$

Come si vede, questa è la simulazione della regola della contrazione nella "riformulazione soft" di **LL**. La morale è che, una volta eliminato il digging, *la contrazione non esiste più nella sua forma consueta.* Al posto suo, esiste solo il multiplexing, che è una forma "indebolita" di contrazione. Infatti, contrarre una formula mediante multiplexing costringe necessariamente ad aumentare la profondità dell'eventuale proof-net con cui la conclusione del multiplexing stesso sarà tagliata. Ciò significa che se proviamo a costruire una delle nostre famigerate "catene esponenziali" in **ISLL**, scopriamo che, ogni volta che vogliamo tagliare le porte ausiliarie contratte di una scatola contro la porta principale di un'altra scatola, *questa dev'essere necessariamente più profonda dell'altra*, di esattamente un livello. Non è dunque possibile generare una catena "uniforme" di qualunque lunghezza, poiché più aumenta la lunghezza, più aumenta la profondità richiesta. Tutto ciò è vero anche in tutti i sistemi polinomiali *light*, anche se lì il fenomeno prende una piega leggermente diversa (il lettore è invitato a controllare cosa succede in questo caso).

Ovviamente, oltre ad essere corretto, il sistema **ISLL** è anche completo rispetto alle funzioni polinomiali; si veda [17] per i dettagli. Osserviamo che, in **ISLL**, la completezza non richiede l'aggiunta di alcun connettivo, la qual cosa presenta indubbiamente alcuni vantaggi a livello sintattico.

Possiamo a questo punto fornire un "quadro riassuntivo" dei sistemi a bassa complessità trovati fino a questo momento:

	Approccio soft	Approccio light		
–	**ILL**	**LL**		**IAL**
ELEMENTARY		**ELL**	$\overline{\mathbf{ELL}}$	**IEAL**
P	**ISLL**	**LLL**	$\overline{\mathbf{LLL}}$	**ILAL**

Il sistema $\overline{\mathbf{LLL}}$ menzionato nella tabella, corrispondente alla classe **P** e facente parte di sistemi *light*, sarà l'oggetto del prossimo capitolo.

Capitolo 9

Una nuova caratterizzazione logica di P

*Verrà nel seguito presentata una variante semplificata della Light Linear Logic (***LLL***), il sistema a complessità polinomiale introdotto da Girard. L'approccio utilizzato è quello di caratterizzare* **LLL** *in termini di un sottoinsieme delle derivazioni della logica lineare completa, in modo analogo a quanto fatto da Danos e Joinet per* **ELL**.

9.1 Il sistema $\overline{\mathbf{LLL}}$

9.1.1 Motivazioni principali

Il sistema che stiamo introducendo vuole essere in qualche modo una semplificazione della Light Linear Logic (**LLL**, [12]) che, come abbiamo visto, costituisce una caratterizzazione della classe **PTIME** delle funzioni calcolabili in tempo polinomiale da una macchina di Turing deterministica. La logica di Girard presenta infatti alcuni elementi, dal punto di vista sintattico, che ne limitano la chiarezza; in particolare, il calcolo dei sequenti a "strati" (moltiplicativo, additivo ed esponenziale, caratterizzati rispettivamente dai separatori ';', ',' e '[ ]') e le proof-net con conclusioni additive possono risultare di difficile comprensione e manipolazione.

Oltre ad aumentare la chiarezza di **LLL**, questo lavoro si pone l'obiettivo, certamente più importante, di trovare una caratterizzazione delle funzioni polinomiali senza modificare (o modificando in modo pressoché irrilevante) le regole della logica lineare completa **LL**, semplicemente imponendo alcune condizioni che ne restringono l'utilizzo e, dunque, il potere espressivo. Il sistema quindi, al contrario di **LLL**, non si presenterà come una "nuova" logica, ma piuttosto come un *sottoinsieme* di (una versione estesa di) **LL**: le derivazioni a complessità polinomiale saranno esattamente le derivazioni di **LL** che soddisfano alcuni criteri strutturali. In tal senso, questo lavoro estende alle funzioni polinomiali l'approccio di Danos e Joinet esposto in [8], dove si fornisce una caratterizzazione delle funzioni ricorsive elementari in termini di un sottoinsieme delle derivazioni della logica lineare completa, definendo così un sistema con lo stesso potere espressivo di quello abbozzato da Girard in [12], al quale noi ci riferiremo come $\overline{\mathbf{ELL}}$ (Elementary Linear Logic)[1].

9.1.2 Il sistema $\overline{\mathbf{LLL}}$: le condizioni di stratificazione

Per definire correttamente $\overline{\mathbf{LLL}}$ occorre prima aggiungere alla logica lineare una modalità aggiuntiva, il connettivo esponenziale '§' (*neutral*, o paragrafo):

Definizione 9.1 *Il sistema* $\mathbf{LL}_\S$ *è la logica lineare completa alla cui sintassi è aggiunto il connettivo* § *tale che, se* A *è una formula,*

$$(\S A)^\perp := \S(A^\perp)$$

e con la seguente regola per il calcolo dei sequenti:

$$\frac{\vdash\ ?\Gamma, \Delta}{\vdash\ ?\Gamma, \S\Delta}\ \S$$

Il connettivo *neutral* non trova alcun uso significativo in $\mathbf{LL}_\S$; esso serve esclusivamente a recuperare il potere espressivo delle modalità esponenziali "tradizionali" quando l'utilizzo di queste, come avviene in $\overline{\mathbf{LLL}}$, sia soggetto a limitazioni.

Naturalmente, anche le derivazioni di $\mathbf{LL}_\S$ possono essere tradotte nella sintassi alternativa delle proof-net, che è esattamente la stessa di **LL**, ma con l'aggiunta di una nuova scatola esponenziale, il §-*box*:

Definizione 9.2 *Se* π *è una proof-net con conclusioni* $?\Gamma$, $|\Gamma| = m$ *e* Δ, $|\Delta| = n$, *la rete* π' *ottenuta costruendo attorno a* π *una scatola con* m *porte ausiliarie (*pax*) ed* n *porte* neutral *(§), è una proof-net con conclusioni* $?\Gamma$ *e* $\S\Delta$ *(figura 9.1.2). In* $\mathbf{LL}_\S$, *con il termine* scatola esponenziale *ci riferiremo dunque (a meno che non sia specificato altrimenti) sia ad un* !-*box che ad un* §-*box.*

Sempre nell'ambito delle proof-net, diamo le due seguenti definizioni preliminari:

[1] In questa sede abbiamo scelto di chiamare tale sistema $\overline{\mathbf{ELL}}$ (in analogia con il nostro $\overline{\mathbf{LLL}}$) per distinguerlo dal sistema originale di Girard, che continueremo a chiamare semplicemente **ELL**.

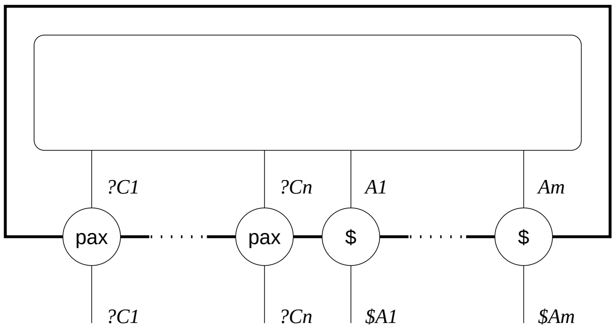


Figura 9.1: La struttura tipica di un §-box. Nella figura, il simbolo $ rimpiazza §.

Definizione 9.3 (Ramo esponenziale) *Consideriamo un cammino orientato di una proof-net i cui archi sono tutti etichettati con la stessa formula di tipo* $?A$. *Essendo orientato, un siffatto cammino* p *possiede sempre una* sorgente *e un* target*, che sono rispettivamente il nodo la cui conclusione è il primo arco di* p *e il nodo la cui premessa è l'ultimo arco di* p. *Chiameremo* ramo esponenziale *un cammino di questo tipo tale che la premessa della sorgente, o non c'è, oppure non è etichettata con* $?A$, *e la conclusione del target, o non c'è, oppure non è neanche questa etichettata con* $?A$.

La sorgente di un ramo esponenziale può dunque essere uno tra i seguenti tre tipi di nodi: *dereliction*, *assioma* o *weakening*; il target può essere un nodo qualunque, ad eccezione ovviamente dei nodi che non hanno premesse (*assioma* e *weakening*) e dei nodi *pax*, *contrazione* e *contrazione additiva.* Osserviamo inoltre che due o più rami esponenziali possono avere un tratto in comune, ad esempio nel caso in cui due istanze di una formula esponenziale vengano contratte in una sola.

Definizione 9.4 (Pax-tree) *A ciascuna porta ausiliaria (*pax gate*) di una scatola esponenziale di qualsiasi tipo, la cui premessa (nonché conclusione) è* $?A$, *viene associato un albero, detto* pax-tree*, la cui radice è la porta ausiliaria stessa e le cui foglie sono i nodi che hanno introdotto la formula* $?A$.

I nodi di un pax-tree possono essere *pax*, *contrazioni* o *contrazioni additive*, mentre le foglie possono essere *dereliction*, *assiomi* o *weakening*.

Siamo ora pronti a definire formalmente il sottoinsieme di $\mathbf{LL}_{\S}$ che chiameremo $\overline{\mathbf{LLL}}$, e che corrisponderà esattamente alle funzioni della classe **PTIME**:

Definizione 9.5 ($\overline{\mathbf{LLL}}$) *Sia* Π *l'insieme delle proof-net di* $\mathbf{LL}_{\S}$*; l'insieme delle proof-net di* $\overline{\mathbf{LLL}}$ *è il sottoinsieme di* Π *costituito da tutte le proof-net che soddisfano le seguenti condizioni:*

i. *Ogni !-box ha al massimo una porta ausiliaria (pax).*

ii. (condizione di stratificazione a scendere, Danos-Joinet) *Ogni ramo esponenziale la cui sorgente è un nodo dereliction [assioma] attraversa esattamente un [zero] box esponenziale[i] (di qualsiasi tipo, sia !-box che §-box).*

iii. (condizione di stratificazione a salire) *Tra le foglie del pax tree di ogni !-box c'è al massimo una foglia dereliction.*

Nella condizione **iii** si parla *del* pax tree di un !-box perché, per la condizione **i**, questo è sempre unico (può al limite essere vuoto). Inoltre, per la **ii**, in $\overline{\mathbf{LLL}}$ i pax-tree non possono più avere tra le foglie alcun nodo assioma.

9.1.3 Caratteristiche logiche di $\overline{\mathbf{LLL}}$

E' opportuno a questo punto analizzare le caratteristiche logiche del sistema appena definito. Come tutti i sistemi che seguono l'approccio *light*, $\overline{\mathbf{LLL}}$ si basa sull'eliminazione di due principi base della logica lineare, la *dereliction* ($!A \multimap A$) e il *digging* ($!A \multimap !!A$), la cui rimozione è garantita dalla condizione di stratificazione a scendere. Inoltre, sempre seguendo l'approccio *light*, $\overline{\mathbf{LLL}}$ rifiuta la *multifuntorialità* dell'*of course* ($!A\otimes\, !B \multimap !(A\otimes B)$), rimpiazzandola con la multifuntorialità del nuovo connettivo *neutral*, $\S A \otimes \S B \multimap \S(A \otimes B)$; tale caratteristica è imposta dalla condizione **i**.

Purtroppo, la medesima condizione è responsabile di un "effetto collaterale" che, pur essendo innocuo dal punto di vista computazionale, può risultare spiacevole in ambito semantico: $\overline{\mathbf{LLL}}$ non verifica l'isomorfismo fondamentale degli esponenziali, $!A\otimes\, !B \simeq\, !(A \,\&\, B)$. In particolare, pur rimanendo vero che $!(A \,\&\, B) \multimap\, !A\otimes\, !B$, a causa della condizione **i** non è più possibile derivare il viceversa, $!A\otimes\, !B \multimap\, !(A \,\&\, B)$. Quest'ultimo principio è invece derivabile in **LLL**, dove le acrobazie sintattiche del calcolo dei sequenti "multi strato" sembrano essere state costruite proprio attorno ad esso. Come accennato poc'anzi, la mancata verifica dell'isomorfismo base degli esponenziali non costituisce tuttavia un danno computazionale, cosa confermata dal fatto che nessuno degli altri sistemi polinomiali (ci riferiamo a **LAL** e **SLL**) riesce a catturare tale proprietà.

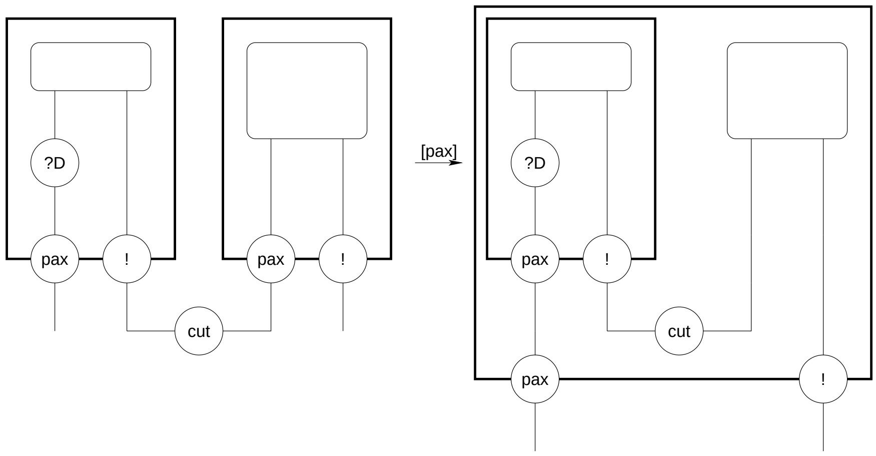
La condizione di stratificazione a salire non ha invece una diretta conseguenza logica. Essa serve piuttosto a inibire la possibilità di un !-box di duplicare altri !-box, innescando così dinamiche esponenziali. In altre parole, il principio di base di $\overline{\mathbf{LLL}}$, che poi è anche quello di tutti sistemi *light*, è che *i !-box sono i soli a poter essere duplicati, ma non possono essi stessi duplicare.*

La presenza della promozione "tradizionale" al posto della soft-promotion preferita dai sistemi sia *light* che *soft*, e dunque la possibilità di utilizzare in modo "temporaneo" la dereliction, permette di mantenere l'utilizzo del weakening limitatamente alle formule esponenziali. Il principio base del weakening resta quindi $!A \multimap 1$, al contrario di quanto accade in logica affine, in cui il weakening è liberalizzato ($A \multimap 1$), e in **LLL**, in cui, accanto al weakening tradizionale, esiste uno strano *weakening additivo*, che altro non è che una forma temporanea di weakening liberalizzato. Anche il principio base della contrazione ($!A \multimap !A \otimes !A$) resta invariato, a differenza di **SLL**, in cui la contrazione costringe necessariamente ad aumentare la profondità esponenziale ($!!A \multimap !A \otimes !A$). $\overline{\mathbf{LLL}}$ si presenta dunque come una "via di mezzo" tra i vari sistemi polinomiali oggi esistenti.

9.2 Cut-elimination in $\overline{\mathbf{LLL}}$

9.2.1 Instabilità di $\overline{\mathbf{LLL}}$

Essendo un semplice sottoinsieme di $\mathbf{LL}_\S$, $\overline{\mathbf{LLL}}$ gode senz'altro della cut-elimination; è sufficiente definire, in modo piuttosto ovvio, un passo di riduzione per i tagli $\S/\S$. Purtroppo però le cose non funzionano così facilmente: $\overline{\mathbf{LLL}}$ infatti non è stabile rispetto ai passi elementari di riduzione di $\mathbf{LL}_\S$, che sono quelli definiti in 7.2 più il passo per il paragrafo appena discusso. Ciò significa che, se π è una proof-net di $\overline{\mathbf{LLL}}$ e π' una proof-net ottenuta mediante l'applicazione di uno dei passi elementari di riduzione di $\mathbf{LL}_\S$, π' potrebbe non essere più una proof-net di $\overline{\mathbf{LLL}}$. Per verificarlo, basta considerare il seguente esempio:



Nella proof-net ridotta c'è un ramo esponenziale che attraversa due scatole esponenziali, in contraddizione con la condizione **ii** della definizione 9.5.

In realtà, è possibile dimostrare che la forma normale di una proof-net di

$\overline{\mathbf{LLL}}$ è ancora una proof-net di $\overline{\mathbf{LLL}}$, ma è piuttosto scomodo avere una procedura di cut elimination che fa continuamente "uscire" e "rientrare" nel sistema. Occorre dunque ridefinire i passi elementari di riduzione in modo da ottenere una procedura di cut-elimination rispetto alla quale $\overline{\mathbf{LLL}}$ sia completamente stabile.

L'idea fondamentale è quella di eliminare il passo [pax] per inglobarlo in un passo di riduzione "più grande" che includa anche i passi [w], [c] e [!/?], uno dei quali segue certamente l'applicazione di (un certo numero di) [pax], a causa della condizione di stratificazione a scendere. La definizione di un simile "super-passo" nell'ambito delle proof-net convenzionali è però troppo problematica, data la presenza di una quantità non indifferente di possibili situazioni da dover considerare, vista anche la presenza (da questo punto di vista scomodissima) degli additivi. La soluzione allora è quella di ricorrere ad una certa classe di proof-net, in cui la definizione dei passi di riduzione sia più semplice. Tale classe è quella delle cosiddette proof-net *nouvelle syntaxe*, secondo la quale l'applicazione delle regole strutturali è ristretta in modo da forzare le proof-net ad avere una determinata struttura. I princìpi di base della nouvelle syntaxe sono sostanzialmente due: un weakening seguito da una contrazione viene semplicemente eliminato, e le regole strutturali vengono spinte il più possibile "in basso" verso le conclusioni della proof-net[2].

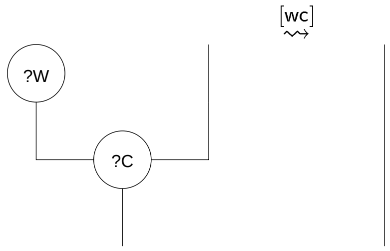
Nel caso di quella che potremmo chiamare $\mathbf{ME}\overline{\mathbf{LLL}}$, cioè il solo frammento moltiplicativo ed esponenziale di $\overline{\mathbf{LLL}}$, si potrebbe arrivare facilmente alla definizione di una procedura di cut-elimination rispetto alla quale le proof-net nouvelle syntaxe (e dunque l'intero sistema) siano stabili; sfortunatamente, la presenza degli additivi rende le cose parecchio più complicate, e il risultato di un "super-passo" di normalizzazione applicato ad una proof-net nouvelle syntaxe potrebbe non essere più una proof-net nouvelle syntaxe. Per risolvere il problema, è possibile definire una procedura che converta una proof-net tradizionale in una proof-net nouvelle syntaxe, e sottoporre le proof-net a questa procedura prima di applicare qualsiasi passo di riduzione. La cut-elimination di $\overline{\mathbf{LLL}}$ sarà dunque costituita da una sequenza di passi di riduzione inframmezzati da passi di conversione per passare dalla forma tradizionale alla nouvelle syntaxe, e viceversa. Presa una qualsiasi proof-net π di $\overline{\mathbf{LLL}}$, una tipica sequenza di riduzione comincerà dunque con diversi passi di conversione volti a trasformarla in una proof-net π' nouvelle syntaxe; a questo punto, verrà applicato un passo di eliminazione del taglio, il quale indurrà una proof-net π_1 non necessariamente nouvelle syntaxe. Un'ulteriore serie di passi di conversione porterà ad una proof-net π_1' la quale, essendo

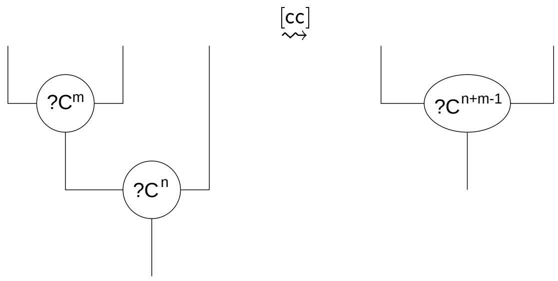
[2]E' importante osservare che entrambe queste due operazioni sono *semanticamente invisibili*. In altre parole, l'interpretazione semantica di due dimostrazioni che differiscano solo nei termini descritti è identica; si veda, ad esempio, [25].

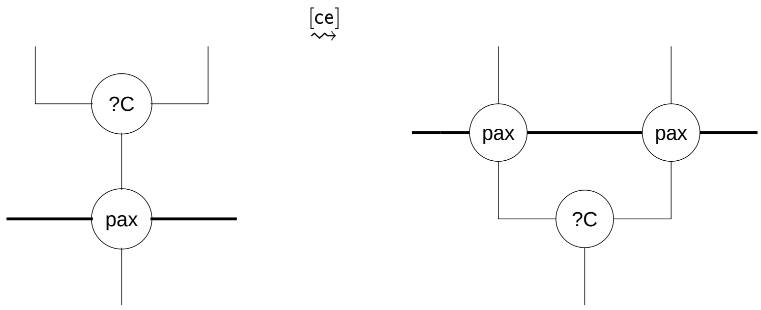
nouvelle syntaxe, potrà essere soggetta all'applicazione di un passo di riduzione che fornirà una nuova proof-net π_2, e così via. Una volta ottenuta una proof-net cut-free, si applicherà una semplice trasformazione per eliminare gli eventuali "residui" di nouvelle syntaxe in modo da riportare il risultato finale in forma tradizionale.

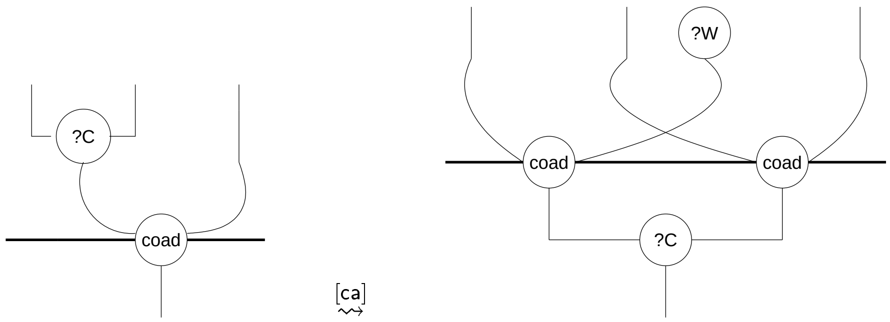
9.2.2 Dalla forma tradizionale alla *nouvelle syntaxe*

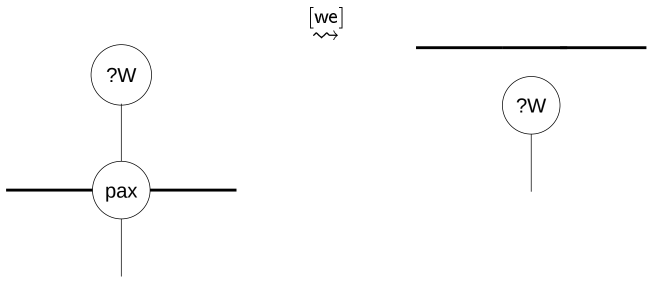
Quelli riportati nel seguito sono i *passi elementari di conversione* che serviranno a trasformare una proof-net dalla forma tradizionale alla nouvelle syntaxe:

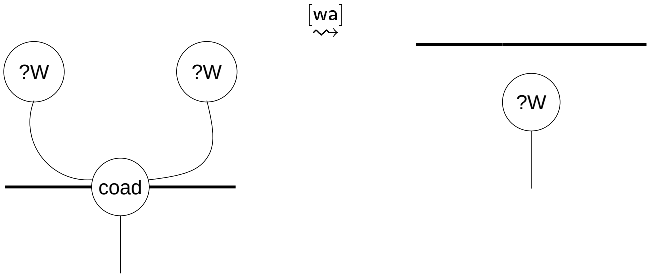












Una precisazione importante riguardo a [ca]: in realtà, di questo step esiste una versione duale, perfettamente ammissibile, nella quale la conclusione del nodo weakening che viene introdotto, invece di essere premessa della contrazione additiva "a sinistra", è premessa di quella "a destra" (e naturalmente la conclusione "più a destra di tutte", la quale già esisteva prima della trasformazione e che è ora premessa della contrazione additiva a destra, diventa premessa di quella a sinistra). In effetti non c'è modo di distinguere, a livello locale, le due premesse della contrazione che viene ridotta da [ca]; questo comporta che quando si applica tale step si sceglie, *senza poterlo sapere*, una delle sue due possibili versioni. Fortunatamente, ciò non causa problemi ai fini della cut-elimination.

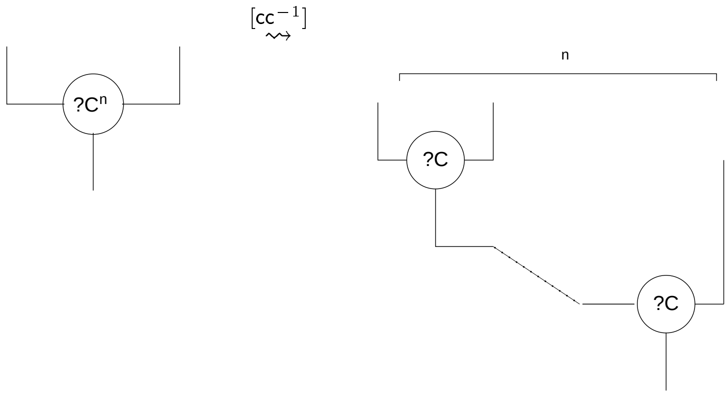
Nel seguito, scriveremo $\pi \stackrel{\varepsilon}{\rightsquigarrow} \pi'$ se la proof-net π si converte in π' dopo l'applicazione del passo elementare di conversione ε, oppure scriveremo semplicemente $\pi \rightsquigarrow \pi'$ se π si converte in π' dopo l'applicazione di un numero arbitrario (ma finito) di passi di conversione.

Definizione 9.6 (Proof-net nouvelle syntaxe) *Una proof-net ν di $\mathbf{LL}_{\S}$ appartiene alla* nouvelle syntaxe *se e soltanto se è normale rispetto ai passi di conversione definiti sopra, cioè se nessuno di tali passi è applicabile a ν.*

Osserviamo che una proof-net nouvelle syntaxe non è propriamente una proof-net nella definizione tradizionale; la differenza, l'unica che c'è in effetti, sta nella presenza di contrazioni n-arie, assenti nella forma tradizionale[3]. In

[3]Naturalmente, le "proof-net" con contrazioni n-arie, ottenute mediante l'applicazione degli step descritti a partire da proof-net tradizionali, restano sempre sequenzializzabili nel calcolo dei sequenti in cui la regola della contrazione venga sostituita con una versione anch'essa n-aria.

tal senso, basta definire un semplice passo di "riconversione", che associ ad una proof-net nouvelle syntaxe un'unica proof-net tradizionale:



L'applicazione ripetuta di $[\mathsf{cc}^{-1}]$ consente chiaramente di ripristinare la "tradizionalità" di una proof-net nouvelle syntaxe, e il risultato di tale applicazione è ovviamente unico.

E' senz'altro possibile che due proof-net diverse π' e π'' si riducano alla stessa proof-net nouvelle syntaxe; potremmo dunque definire una relazione $\approx_\nu$ sulle proof-net di $\mathbf{LL}_\S$ tale che $\pi' \approx_\nu \pi''$ se e solo se π' e π'' hanno la stessa immagine in nouvelle syntaxe, modulo la riconversione mediante $[\mathsf{cc}^{-1}]$. Non è difficile verificare che $\approx_\nu$ è una relazione d'equivalenza, e dunque le proof-net nouvelle syntaxe non sono altro che i rappresentanti delle classi d'equivalenza dell'insieme delle proof-net di $\mathbf{LL}_\S$ quozientato rispetto a $\approx_\nu$. Quest'operazione di quozientatura non è certo una novità; le stesse proof-net tradizionali sono in verità rappresentanti di classi d'equivalenza di derivazioni del calcolo dei sequenti.

Forniremo ora una procedura specifica per l'applicazione dei passi appena descritti, e dimostreremo poi alcune proprietà importanti su di essa:

Definizione 9.7 (Procedura di conversione standard) *La* procedura di conversione standard *consiste nell'applicare ad una proof-net* π *di* $\mathbf{LL}_\S$ *i passi di conversione descritti sopra, nel modo seguente (per "profondità" s'intenderà qui la profondità di qualsiasi tipo, sia additiva che esponenziale):*

- *Si inizia a profondità massima, che supponiamo essere* m*, alla quale si applicano tutti i possibili step di tipo* $[\mathsf{wc}]$*,* $[\mathsf{we}]$*,* $[\mathsf{wa}]$ *e* $[\mathsf{ca}]$*, con la seguente strategia: lo step* $[\mathsf{wc}]$ *ha la priorità su tutti gli altri; se ce n'è uno, sarà sempre lui ad essere eseguito. Infine, si eseguono gli step* $[\mathsf{ce}]$ *rimasti a profondità massima. Questo sarà il "primo passo" della procedura.*

- *Si ricomincia, nel medesimo modo, a profondità $m-1$; questo sarà ovviamente il "secondo passo" della procedura. Proseguendo in modo analogo, si arriverà all'$m+1$-esimo passo, che si occuperà della profondità 0, alla quale non si possono che applicare step* [wc].
- *Alla fine si eseguono, in qualsiasi ordine, tutti gli step* [cc].

Proposizione 9.1 (Stabilità di $\overline{\mathbf{LLL}}$ sotto la conversione standard) *Se π è una proof-net di $\overline{\mathbf{LLL}}$, l'applicazione della procedura di conversione standard a π produce una proof-net π' che, modulo* $[\mathsf{cc}^{-1}]$, *è ancora in $\overline{\mathbf{LLL}}$.*

Dimostrazione. Iniziamo dalle due condizioni di stratificazione. Per quanto riguarda quella a scendere, nessun passo di conversione è in grado di creare rami esponenziali il cui nodo sorgente è un nodo assioma o dereliction; inoltre, il numero di scatole esponenziali attraversate da un qualsiasi ramo esponenziale con sorgente assioma o dereliction resta invariato sotto la conversione. Di conseguenza, nella proof-net risultante dalla conversione i soli rami esponenziali che vìolano la condizione **ii** sono quelli che già esistevano inizialmente. Riguardo invece alla condizione di stratificazione a salire, sebbene tutti gli step siano in grado di modificare la struttura di un pax tree, nessuno di questi è grado di cambiare il tipo delle foglie; l'unico ad aggiungere un'eventuale foglia è [ca], ma si tratta di una foglia weakening, dunque "innocua" dal punto di vista della condizione **iii**.

Resta allora da dimostrare che venga preservata la prima condizione, cioè che il numero di porte ausiliarie di ciascun !-box rimanga minore o uguale a uno. In effetti, un aumento di tale numero sarebbe possibile nel caso dello step [ce], il quale produce due pax a partire da una; dobbiamo dunque concentrarci su questo tipo di step.

Supponiamo che la profondità massima (nel senso che abbiamo adottato per la procedura di conversione, cioè profondità "cumulativa", sia additiva che moltiplicativa) di π sia m; la conversione standard viene effettuata in m passi (più un round preliminare all'inizio), i quali operano sempre a profondità i e $i-1$, senza toccare le altre profondità; inoltre, gli step eseguiti a profondità $i-1$ sono solo di tipo [wc], e non ci riguardano in questo momento. Basterà dunque dimostrare che ogni passo operante a profondità i preserva la proprietà per i !-box che racchiudono tale profondità, e il risultato globale sarà evidentemente ottenibile in modo induttivo.

Il passo generico della procedura di conversione standard adotta una politica molto particolare per gli step a cui siamo interessati, vale a dire i [ce]: questi vengono eseguiti solo dopo l'esecuzione di tutti gli altri. Mostreremo allora che, a causa delle condizioni di stratificazione (le quali sono preservate dall'applicazione di un qualsiasi passo di conversione), i passi [ce] saranno applicati esclusivamente a contrazioni che si trovano sopra la porta ausiliaria di un §-box, e dunque il numero di nodi pax di ciascun !-box resterà costante.

Supponiamo allora per assurdo che ci si trovi di fronte ad una situazione in cui è possibile applicare un [ce] ad una contrazione in un !-box; entrambe le premesse della contrazione in questione sono diverse da assiomi (condizione di stratificazione a scendere) e da weakening, altrimenti si potrebbe ancora applicare un passo [wc]. Non possono neanche essere entrambe dereliction, altrimenti non sarebbe soddisfatta la condizione di stratificazione a salire. Supponiamo allora che solo una delle due sia una dereliction; l'altra può essere o una pax, o una coad. Nel primo caso, se sopra la pax ci fosse un weakening, questo sarebbe stato "tirato giù" dalla procedura al passo precedente mediante uno step [we], e sarebbe poi stato eliminato da uno step [wc] alla profondità corrente. Se, al contrario, sopra la pax ci fosse una dereliction, non sarebbe soddisfatta nessuna delle due condizioni di stratificazione. Non possiamo dunque avere una pax. Il caso della coad è analogo: sopra non possono esserci né weakening, né contrazioni, perché sarebbero stati ridotti, e non può esserci un assioma o una dereliction perché sarebbe violata una delle due condizioni di stratificazione. Per quanto riguarda la pax, vale il discorso già fatto sopra. Un simile argomento si può applicare al caso in cui entrambe le premesse della contrazione non siano dereliction, e dunque l'ipotesi che la proof-net sia originariamente in $\overline{\mathbf{LLL}}$ garantisce che nessuno step [ce] possa essere applicato ad un !-box a qualsiasi profondità. □

Proposizione 9.2 *Il pax-tree di ogni porta ausiliaria di ogni scatola esponenziale (di qualsiasi tipo) di una proof-net nouvelle syntaxe di* $\overline{\mathbf{LLL}}$ *ha sempre la seguente forma: la radice è un nodo* pax*, le foglie sono dei nodi* ?D *e* ?W*, e i nodi intermedi sono solo* coad*.*

Dimostrazione. Supponiamo per assurdo che ci sia un nodo pax nell'albero (esclusa ovviamente la radice). Il pax-tree di tale porta ausiliaria, che è un sotto albero del nostro pax-tree, può avere solo foglie weakening, altrimenti esisterebbe un cammino esponenziale che attraverserebbe due porte ausiliarie, contravvenendo alla condizione di stratificazione a scendere (la quale vale per i box esponenziali di qualsiasi tipo). Ma se esistesse un sottoalbero che ha solo foglie weakening, certamente sarebbe possibile applicare uno step [we], [wa], o [wc] e dunque la proof-net non sarebbe nouvelle syntaxe contro l'ipotesi.

Supponiamo ora che ci sia un nodo contrazione nel pax-tree. Poiché questo fa parte di un pax-tree, la sua conclusione non può essere che premessa di uno dei seguenti nodi: una pax, una coad o un'altra contrazione. In tutti e tre i casi, si potrebbe applicare uno step di conversione, rispettivamente [ce], [ca] e [cc], e dunque la proof-net non sarebbe nouvelle syntaxe, conducendo ad un assurdo. $\square$

La proposizione 9.2 sarà fondamentale ai fini della definizione dei passi di cut-elimination, mentre la proposizione 9.1 garantisce che la conversione non fa "uscire" da $\overline{\mathbf{LLL}}$.

Poichè la procedura di conversione verrà applicata prima di ogni passo di eliminazione del taglio, è necessario che questa termini sempre, altrimenti non sarebbe possibile applicare il passo di riduzione in questione. Il seguente teorema garantisce che tutte le applicazioni della procedura di conversione standard terminano in un numero finito di passi:

Teorema 9.1 (Normalizzazione (debole) della conversione) *Sia π una proof-net di $\mathbf{LL}_\S$. L'applicazione della procedura di conversione standard a π termina in un numero finito di passi.*

Dimostrazione. La procedura di conversione standard opera a profondità via via decrescente, e non viene eseguito alcuno step elementare di conversione a profondità $i-1$ se prima non sono stati esauriti tutti quelli a profondità i. Poiché la conversione non modifica la profondità della proof-net a cui viene applicata, e poiché questa non può avere profondità infinita, basta dimostrare che il generico passo della conversione standard termina sempre.

Supponiamo dunque di cominciare ad operare a profondità i. Definiamo la coppia $\chi_i(\pi) = \langle c, w\rangle$, dove c è il numero di nodi contrazione a profondità i e w il numero di nodi weakening sempre a profondità i. Sulle coppie $\langle c, w\rangle$ è definita la relazione d'ordine (lessicografico)

$$\langle c, w\rangle \preceq \langle c', w'\rangle \qquad \text{sse } c < c' \text{ oppure } c = c' \text{ e } w \leq w'$$

Il minimo dell'ordine è, chiaramente, la coppia $\langle 0, 0\rangle$. Ragioniamo allora per induzione su $\chi_i(\pi)$:

- Se $\chi_i(\pi) = \langle 0, 0\rangle$, non c'è bisogno di applicare il passo a profondità i, e la procedura termina in 0 step.

- Supponiamo che la procedura termini per tutti i valori strettamente inferiori a $\chi_i(\pi)$, cioè per ogni coppia p tale che $p \prec \chi_i(\pi) = \langle c, w\rangle$. Chiamiamo π' la proof-net risultante dall'applicazione di un qualsiasi step elementare di conversione. Abbiamo le seguenti possibilità:

  [wc] $\chi_i(\pi') = \langle c-1, w-1\rangle$

  [we] $\chi_i(\pi') = \langle c, w-1\rangle$

  [wa] $\chi_i(\pi') = \langle c, w-2\rangle$

[ce] $\chi_i(\pi') = \langle c-1, w \rangle$

[ca] $\chi_i(\pi') = \langle c-1, w+1 \rangle$

In tutti i casi, $\chi_i(\pi') \prec \chi_i(\pi)$, e dunque la procedura termina per ipotesi d'induzione.

□

Se osserviamo la procedura di conversione standard, troviamo che l'applicazione degli step [we], [wa] e [ca] ad una data profondità è lasciata libera; ciò significa che, in presenza di due possibili conversioni elementari da poter applicare, la procedura standard accetta qualsiasi ordine di applicazione. Ciò potrebbe portare, a priori, alla generazione di più forme normali al termine del processo, cosa disastrosa nell'ottica della cut elimination, perché produrrebbe sequenze di riduzione non confluenti. In realtà, la forma nouvelle syntaxe di una proof-net sotto la conversione standard è "quasi" unica. C'è infatti un problema fondamentale, che riguarda lo step [ca]: come abbiamo già osservato, di questo step in realtà esistono due versioni e, cosa più grave, *non si ha alcun modo di distinguere quale delle due si stia applicando*, nel senso che le proof-net, per come sono definite, non forniscono informazioni sufficienti a caratterizzare in qualche modo le due diverse applicazioni di [ca]. E' però evidente che, a priori, ciascuna delle due genera un risultato completamente diverso dall'altra. Di conseguenza, non c'è speranza di ottenere una piena confluenza della conversione.

Questa è in realtà un'istanza microscopica di un problema più generale, che è quello della rappresentazione degli additivi nell'ambito delle proof-net. Si dà il caso infatti che gli additivi siano, a ben guardare, i responsabili di praticamente tutte le anomalie più fastidiose della teoria delle proof-net, a cominciare dalla non confluenza nel caso generale (si veda ad esempio [25]). La soluzione al problema dello step [ca] è dunque lontana dal poter essere trovata nell'ambito delle proof-net così come le abbiamo definite; più verosimilmente, occorrerà trovare una sintassi più raffinata, nell'ambito della quale o le due versioni di [ca] in effetti collassino in una sola, oppure tra le due versioni sia possibile identificarne una "canonica".

Fortunatamente, è possibile verificare che l'interpretazione semantica delle due proof-net derivanti dalle due possibili applicazioni di [ca] è identica; di conseguenza, dal punto di vista della cut-elimination (che è il nostro unico interesse in questa sede), la non confluenza è assolutamente innocua. In sostanza, tutte le possibili versioni nouvelle syntaxe di una proof-net interagiranno con un'altra proof-net esattamente nello stesso modo o, detto in altri termini, la cut-elimination additive-lazy resta confluente nonostante la presenza di più possibilità intermedie.

Possiamo allora definire la *procedura di conversione standard additive-lazy*, la quale semplicemente non esegue i passi [ca]. In tal caso, è possibile dimostrare la validità della proprietà di Church-Rosser, servendoci del seguente lemma:

Lemma 9.1 *L'applicazione della procedura additive-lazy a profondità i alla generica proof-net π genera un'unica proof-net π'.*

Dimostrazione. Iniziamo con l'osservare che gli step [cc] vengono applicati solo al termine di tutta la procedura, quindi in questo momento non ci riguardano. Inoltre, l'applicazione della conversione standard a profondità i esegue gli step [ce] solo dopo aver applicato tutti gli altri; poiché gli step [ce] non possono interferire l'uno con l'altro, basta mostrare che l'applicazione degli altri step è confluente. Per quanto riguarda [we] e [wa], questi agiscono entrambe su "isole" sconnesse dal resto della scatola all'interno della quale vengono applicati (i nodi weakening infatti non hanno premesse), e dunque sono permutabili con qualsiasi altro step. Restano gli step [wc]. La conversione di due step di tale tipo che interferiscono l'uno con l'altro (una coppia weakening/contrazione è "premessa" di un'altra coppia) è chiaramente confluente. Per il resto, [wc] ha sempre la precedenza sugli altri step, e dunque qualsiasi conflitto si risolve automaticamente mediante l'ordine imposto dalla strategia. □

Possiamo ora facilmente dimostrare la confluenza a livello globale:

Teorema 9.2 (Confluenza (ristretta) per la conversione standard) *Se π è una proof-net di $\mathbf{LL}_\S$, allora l'applicazione della procedura di conversione standard additive-lazy a π produce un'unica proof-net ν "quasi" nouvelle syntaxe.*

Dimostrazione. Basta iterare il lemma 9.1 per ottenere la confluenza fino al momento in cui la procedura termina l'ultimo passo a profondità zero; a questo punto, l'applicazione degli step [cc] è chiaramente confluente, dunque l'intera conversione standard soddisfa la proprietà di Church-Rosser. In generale, la proof-net ottenuta è chiaramente "quasi" nouvelle syntaxe poiché non è stato applicato alcuno step [ca], dei quali invece avrebbe potuto esserci bisogno per arrivare alla forma propriamente nouvelle syntaxe. □

Dimostrata la confluenza ristretta al caso della versione *additive-lazy* della conversione standard, possiamo parlare nel caso generale di proprietà di Church-Rosser *modulo le differenti applicazioni dello step* [ca].

Riguardo alla normalizzazione della procedura di conversione standard vale anche un risultato più forte, che sarà di fondamentale importanza ai fini della **PTIME**-correttezza di $\overline{\mathbf{LLL}}$:

Teorema 9.3 (Normalizzazione polinomiale per la conversione standard) *L'applicazione della procedura di conversione standard ad una proof-net π di $\mathbf{LL}_\S$ termina in un numero di passi lineare nella dimensione di π; vale a dire, se n è il numero di nodi di π, la procedura fornisce una proof-net nouvelle syntaxe in $O(n)$ step.*

Dimostrazione. Riprendiamo la funzione χ_i della dimostrazione di 9.1, per la quale è, per definizione,

$$\chi_i(\pi) = \langle c, w \rangle$$

dove c e w sono rispettivamente il numero di nodi contrazione e weakening a profondità i in π. In base a tale funzione, definiamo un indice ι tale che, se $\chi_i(\pi) = \langle c_i, w_i \rangle$ e m è la profondità massima di π,

$$\iota(\pi) = \sum_{i=0}^{m} 2c_i + w_i$$

Non è difficile verificare che ι decresce ad ogni applicazione di qualsiasi step elementare di conversione (senza considerare per il momento lo step [cc]). Come nella dimostrazione di 9.1, chiamiamo π' la proof-net risultante da π dopo l'applicazione di un generico step elementare di conversione, il quale viene eseguito a profondità j arbitraria, e con c_i e w_i indichiamo il numero di contrazioni e weakening presenti a profondità i prima dell'applicazione dello step:

$$[\mathsf{wc}] \quad \iota(\pi') = 2(c_j - 1) + w_j - 1 + \sum_{\substack{i=0 \\ i \neq j}}^{m} 2c_i + w_i = \iota(\pi) - 3$$

$$[\mathsf{we}] \quad \iota(\pi') = 2c_j + w_j - 1 + \sum_{\substack{i=0 \\ i \neq j}}^{m} 2c_i + w_i = \iota(\pi) - 1$$

$$[\mathsf{wa}] \quad \iota(\pi') = 2c_j + w_j - 2 + \sum_{\substack{i=0 \\ i \neq j}}^{m} 2c_i + w_i = \iota(\pi) - 2$$

$$[\mathsf{ce}] \quad \iota(\pi') = 2(c_j - 1) + w_j + \sum_{\substack{i=0 \\ i \neq j}}^{m} 2c_i + w_i = \iota(\pi) - 2$$

$$[\mathsf{ca}] \quad \iota(\pi') = 2(c_j - 1) + w_j + 1 + \sum_{\substack{i=0 \\ i \neq j}}^{m} 2c_i + w_i = \iota(\pi) - 1$$

Poichè dunque ad ogni step ι diminuisce, e ι non può certo essere minore di zero, il valore iniziale $\iota(\pi)$ limita superiormente il numero totale di step che la procedura di conversione standard dovrà eseguire; poiché il numero di contrazioni e weakening presenti a qualsiasi profondità è (brutalmente) maggiorato dal numero di nodi di π, se quest'ultimo è n abbiamo

$$\iota(\pi) \leq \sum_{i=0}^{m} 3n = 3(m+1)n$$

Bisogna poi aggiungere gli step [cc], i quali non possono anch'essi essere più di n. Il numero di step è allora limitato da $(3m+4)n$; poichè la profondità è costante nell'intero processo, abbiamo che la procedura di conversione standard termina in $O(n)$ step, ed è dunque lineare nella dimensione della proof-net. $\Box$

9.2.3 I passi elementari di riduzione

Siamo finalmente in grado di definire i passi elementari di riduzione per la cut-elimination di $\overline{\mathbf{LLL}}$. Anzitutto, imporremo che questi agiscano solo ed esclusivamente su proof-net *nouvelle syntaxe*; queste, come abbiamo visto, sono in pratica proof-net "semplificate", in cui l'unica differenza a livello di nodi è che le contrazioni possono essere n-arie. La cut-elimination per $\overline{\mathbf{LLL}}$ funzionerà dunque così: prima di eseguire un qualsiasi step elementare di riduzione, si applicherà la procedura di conversione standard descritta nella sezione precedente; una volta arrivati ad una proof-net cut-free, si eseguiranno tutti i passi [cc^{-1}] possibili per "riconvertirla" alla forma tradizionale. In questo modo, il processo di cut-elimination è ancora una funzione che manda proof-net in proof-net, nella definizione solita del termine.

Poiché il punto centrale di $\overline{\mathbf{LLL}}$ sarà la complessità computazionale della sua procedura di cut-elimination, la quale dovrà risultare polinomiale, è di fondamentale importanza il teorema 9.3; in base ad esso, sappiamo che l'applicazione della conversione standard prima di ogni step di cut-elimination non altera la complessità della riduzione, giacché questa aggiunge solamente una componente lineare. Ciò significa che, ad esempio, se per eliminare tutti i tagli da una proof-net di dimensione s ci vogliono $O(p(s))$ passi elementari di riduzione, e nel corso della riduzione la proof-net non supera la dimensione $O(q(s))$ (dove p e q sono polinomi), in considerazione del fatto che ogni passo porta con sé un'applicazione della conversione standard, la complessità totale non potrà essere più di $O(p(s) \cdot q(s))$, che è ancora polinomiale. Di conseguenza, ai fini della complessità computazionale potremo ignorare completamente la conversione, e ragionare esclusivamente sui passi elementari di riduzione che definiremo fra breve.

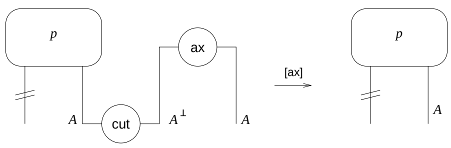
Nel seguito dunque considereremo sottintesa l'applicazione della procedura di conversione standard *prima* dell'applicazione di uno step elementare di riduzione; questi ultimi quindi saranno definiti solo ed esclusivamente per le proof-net nouvelle syntaxe. Ad esempio, nel dire che π si riduce a π' mediante l'applicazione dello step $[s]$ (cosa che denoteremo, commettendo un abuso, semplicemente con $\pi \xrightarrow{[s]} \pi'$), intenderemo in realtà che sussiste la relazione $\pi \rightsquigarrow \nu \xrightarrow{[s]} \nu' \approx_\nu \pi'$, dove ν è la forma nouvelle syntaxe di π ottenuta mediante la conversione standard, e π' è uguale a ν' modulo la riconversione mediante $[\mathsf{cc}^{-1}]$. Allo stesso modo, quando scriveremo $\pi \twoheadrightarrow \pi'$, intendendo che la proof-net π si riduce a π' dopo l'applicazione di un numero arbitrario (finito) di step elementari di riduzione, in realtà sottintenderemo che esiste $n \in \mathbb{N}$, n proof-net nouvelle syntaxe $\nu_1, \ldots, \nu_n$, n proof-net $\nu'_1, \ldots, \nu'_n$ (non necessariamente nouvelle syntaxe) ed n step elementari di riduzione $[s_1], \ldots, [s_n]$ tali che

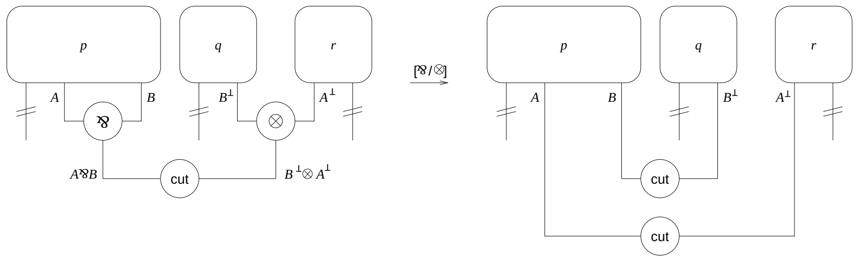
$$\pi \rightsquigarrow \nu_1 \xrightarrow{[s_1]} \nu'_1 \rightsquigarrow \nu_2 \xrightarrow{[s_2]} \nu'_2 \rightsquigarrow \ldots \xrightarrow{[s_{n-1}]} \nu'_{n-1} \rightsquigarrow \nu_n \xrightarrow{[s_n]} \nu'_n \approx_\nu \pi'$$

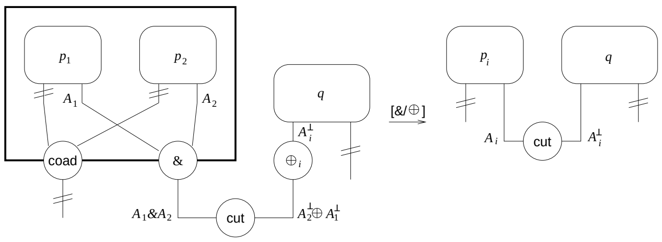
Divideremo gli step elementari di riduzione in tre categorie, secondo la nomenclatura introdotta da Asperti e Roversi per **ILAL**:

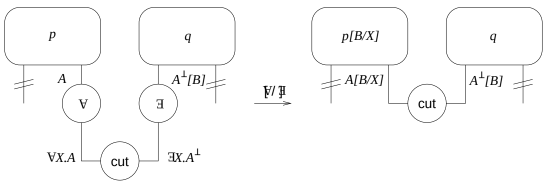
▶ **Step lineari**

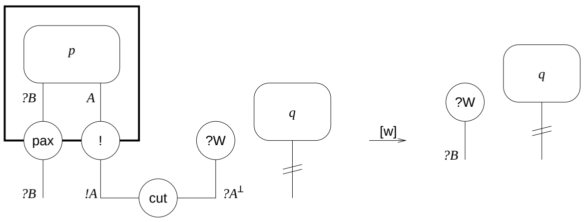
Gli step lineari riguardano i seguenti tagli (che saranno anch'essi detti "lineari"): ax, $⅋/\otimes$, $\&/\oplus$, $\forall/\exists$ e $!/?\mathsf{W}$. La riscrittura indotta dalla riduzione di tali tagli è identica a quella descritta in 7.2; per comodità, riportiamo qui di seguito gli step, che indicheremo rispettivamente con $[\mathsf{ax}]$, $[⅋/\otimes]$, $[\&/\oplus_1]$, $[\&/\oplus_2]$, $[\forall/\exists]$ e $[\mathsf{w}]$:







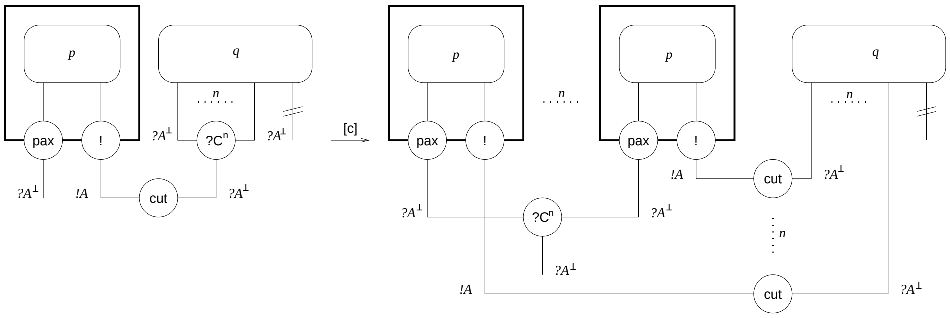




Occorre precisare che, per definizione, sono considerati tagli di tipo ax (ai quali dunque di applica lo step $[\mathsf{ax}]$) *tutti i tagli che coinvolgano un assioma*, indipendentemente da quale sia l'altro nodo coinvolto nel cut.

▶ Step polinomiale

Lo step polinomiale riguarda il taglio (che sarà anch'esso detto "polinomiale") $!/?\mathsf{C}^n$, $n \geq 2$. Anch'esso è praticamente identico al suo corrispondente in **LL**, con la piccola differenza che la contrazione non duplica ma, in generale, n-uplica una scatola. Lo step sarà identificato con $[\mathsf{c}]$:

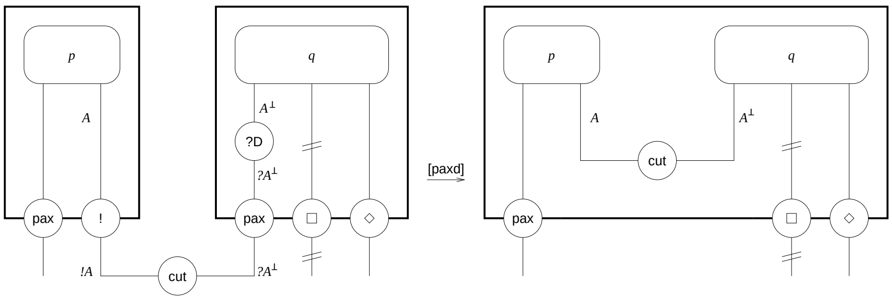


La porta ausiliaria del !-box duplicato dalla contrazione potrebbe anche non esserci, e a quel punto non si deve aggiungere alcuna contrazione.

▶ **Step shifting**

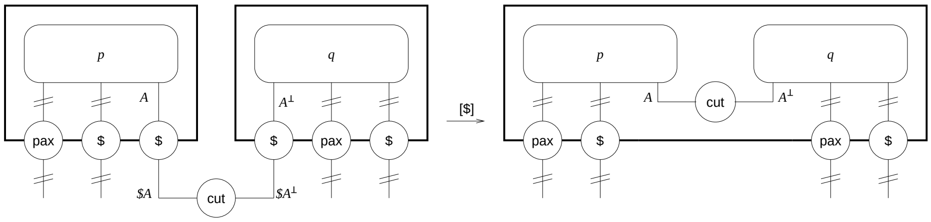
Gli step shifting riguardano i seguenti tagli (che saranno anch'essi detti "shifting"): !/pax e §/§. Il primo è il tradizionale taglio commutativo esponenziale, in cui però il nodo pax coinvolto può appartenere sia ad un !-box che ad un §-box; il secondo invece è uno step del tutto nuovo, che non ha controparte in **LL** semplicemente perché lì non esiste il connettivo §.

Cominciamo con il cut !/pax; quando ci si trova di fronte a questo tipo di taglio, in $\overline{\mathbf{LLL}}$ occorre esaminare il *pax-tree* della porta ausiliaria coinvolta. Infatti, la riduzione del cut !/pax *è definita esclusivamente se il pax-tree in questione non contiene nodi* coad. In base alla proposizione 9.2, sappiamo che i pax-tree delle proof-net nouvelle syntaxe non possono contenere né pax, né contrazioni; se eliminiamo anche il caso in cui questi contengano coad, l'unica possibilità rimasta è che il pax-tree sia filiforme, cioè che esso sia costituito semplicemente da un nodo dereliction (la foglia) e un nodo pax (la radice). Il passo elementare di riduzione per questo taglio è allora il seguente, che chiameremo [paxd]:



Nella figura, $\square \in \{\mathsf{pax}, \S\}$ e $\diamond \in \{!, \S\}$.

L'altro taglio shifting si riduce invece in modo evidente, mediante lo step che indicheremo con [§]:



Nella figura, il simbolo \$ rimpiazza il simbolo §.

Con questo abbiamo concluso l'esposizione dei passi elementari di riduzione per $\overline{\mathbf{LLL}}$; si può ora dimostrare facilmente il seguente teorema:

Teorema 9.4 (Stabilità di $\overline{\mathbf{LLL}}$ sotto cut-elimination) $\overline{\mathbf{LLL}}$ *è stabile rispetto ai passi elementari di riduzione descritti sopra.*

Dimostrazione. Le condizioni della definizione 9.5 sono evidentemente preservate dagli step lineari e dallo step polinomiale. Restano da verificare i due step shifting.

Per [paxd], se entrambe le scatole coinvolte erano !-box, il !-box risultante ha al massimo una porta ausiliaria, a seconda se l'aveva o meno il box di sinistra; nel caso in cui la scatola di destra sia un §-box, possono esserci tante porte ausiliarie quante si desidera, e dunque in entrambi i casi la condizione **i** è preservata. Anche la condizione di stratificazione a scendere (**ii**) non viene alterata, giacché si cancellano sia la dereliction che la pax ad essa associata, e tutto il resto rimane immutato. Per quanto riguarda la condizione di stratificazione a salire (**iii**), anche qui l'unico caso interessante è quello in cui entrambe le scatole siano !-box. Come già osservato prima, la porta ausiliaria del !-box risultante è la stessa del box a sinistra; dunque il suo pax-tree sarà valido se lo era quello di partenza.

Passiamo allora allo step [§]; poiché qui sono coinvolti solo §-box, resta da verificare solo la condizione di stratificazione a scendere, la quale è senz'altro soddisfatta perché le porte ausiliare non vengono alterate dallo step. □

Per descrivere la dinamica del processo di cut-elimination, saranno utili nel seguito le nozioni di *antenato* e *residuo* di un dato nodo:

Definizione 9.8 (Nodo antenato e nodo residuo) *Sia* $\pi \xrightarrow{[s]} \pi'$. *Ciascun nodo* ax *o nodo logico* n *di* π' *proviene da un'unica occorrenza dello "stesso" nodo* ax *o nodo logico* $\overleftarrow{n}$ *di* π. *Chiameremo* $\overleftarrow{n}$ l'antenato *del nodo* n *di* π'. *Reciprocamente, definiremo l'insieme dei* residui *del nodo logico [nodo* ax*]* n *di* π *come l'insieme di tutte le occorrenze* n' *di nodi logici [nodi* ax*] di* π' *tali che* $\overleftarrow{n'} = n$.

Allo stesso modo, ogni occorrenza c di un nodo cut *di* π' *proviene da un'unica occorrenza* $\overleftarrow{c}$ *di un nodo* cut *di* π *le cui premesse sono etichettate dalle stesse formule (a meno di sostituzioni nel caso in cui* $[s] = [\forall/\exists]$*), salvo il caso in cui c sia stato creato da uno step lineare. In quest'ultimo caso, diremo che c non ha antenati, e che il nodo* cut *ridotto da* $[s]$ *non ha residui. In questa sede non ci interesseranno i concetti (pur definibili) di antenato e residuo di un nodo* ?W *o di una scatola esponenziale o additiva.*

E' importante sottolineare che, durante il processo di cut-elimination, *la profondità di tutti i nodi che non siano cut resta costante.* Possiamo enunciare questa proprietà in termini formali:

Proposizione 9.3 *Se* π *è una proof-net di* $\overline{\mathbf{LLL}}$*, e* $\pi \xrightarrow{[s]} \pi'$*, allora la profondità di ogni nodo n di* π' *è uguale a quella del suo antenato, e può essere superiore solo se n è il residuo del cut ridotto da* $[s]$*, con* $[s] \in \{[\mathsf{paxd}], [\S]\}$.

Dimostrazione. Ovvia dall'analisi della presentazione grafica dei vari step di cut-elimination. $\Box$

La proposizione 9.3 sarà fondamentale nella dimostrazione della **PTIME**-correttezza di $\overline{\mathbf{LLL}}$.

Un'ultima osservazione riguardo alla cut-elimination. Come già fatto per **LL**, non è stato definito alcuno step di riduzione per i tagli commutativi additivi; inoltre, come già precisato nella definizione dello step [paxd], non viene nemmeno eseguita la riduzione di un taglio commutativo esponenziale se nel pax-tree della porta ausiliaria coinvolta ci sono scatole additive. Conviene allora dare la seguente definizione:

Definizione 9.9 (Proof-net lazy-cut-free) *Una proof-net* π *di* $\mathbf{LL}_\S$ *è detta* lazy-cut-free *se gli unici nodi* cut *in essa contenuti hanno almeno una premessa che è conclusione o di un nodo* coad*, oppure di un nodo* pax *il cui pax-tree contiene nodi* coad*. I tagli di quest'ultimo tipo saranno detti* tagli commutativi additivi estesi.

La cut-elimination di $\overline{\mathbf{LLL}}$ porta dunque, in generale, a proof-net lazy-cut-free. Tuttavia, in base ai risultati già discussi in 7.2, ciò non crea problemi dal punto di vista della normalizzazione dei tipi "comuni", in cui invece l'eliminazione del taglio di $\overline{\mathbf{LLL}}$ arriva a proof-net propriamente cut-free. Inoltre, l'assenza di riduzioni commutative additive garantisce la confluenza del sistema, sempre in base a quanto detto nel medesimo frangente.

9.3 PTIME-correttezza

9.3.1 La strategia standard

In questa sezione ci occupiamo di dimostrare che l'insieme delle funzioni programmabili in $\overline{\mathbf{LLL}}$ è contenuto in **PTIME**, vale a dire che tutti i programmi di $\overline{\mathbf{LLL}}$ sono eseguibili in tempo polinomiale in una qualche misura (che preciseremo) del loro input. Poiché l'esecuzione dei programmi corrisponde all'eliminazione dei tagli dalle derivazioni che li rappresentano, per dimostrare il risultato sarà sufficiente esibire una strategia di *cut-elimination* che termini sempre in un numero di passi limitato da un qualche polinomio.

Forniamo anzitutto alcune definizioni preliminari:

Definizione 9.10 *La* profondità esponenziale *o, più semplicemente, la* profondità *di un nodo di una proof-net è il numero di scatole esponenziali di qualsiasi tipo (!-box o §-box) che racchiudono il nodo stesso. I nodi* !, § *e* pax *sono considerati come se fossero al di fuori della scatola che introducono.*

Definizione 9.11 *La* profondità *di una proof-net π, che denoteremo con $\partial(\pi)$ è la massima tra le profondità dei nodi che la compongono.*

Definizione 9.12 *Per* livello *di una proof-net π intenderemo d'ora in avanti l'insieme dei nodi di π che si trovano ad una certa profondità. La* dimensione, *o* size, *del livello a profondità i di π, che denoteremo con $\sharp_i(\pi)$, è il numero di nodi la cui profondità è i. La dimensione $\sharp(\pi)$ dell'intera proof-net sarà semplicemente la somma delle dimensioni di tutti i livelli:*

$$\sharp(\pi) = \sum_{i=0}^{\partial(\pi)} \sharp_i(\pi)$$

Definizione 9.13 (Passo standard a profondità i) *Un* passo standard a profondità i *è una sequenza di step elementari di riduzione eseguiti tutti a profondità i, nel seguente ordine:*

Prima fase: *Si eseguono, in qualsiasi ordine, tutti gli step lineari possibili a profondità i.*

Seconda fase: *Si eseguono, in qualsiasi ordine, tutti gli step polinomiali possibili a profondità i e gli eventuali step lineari che possono emergere nel corso di tale riduzione sempre a profondità i (dimostreremo nel seguito che si potrà trattare solo di un certo tipo di step* [ax]*).*

Terza fase: *Si eseguono, in qualsiasi ordine, tutti gli step shifting a profondità i.*

D'ora in avanti, il termine “passo” sarà utilizzato esclusivamente in riferimento ai passi standard; per riferirci ai passi elementari di riduzione utilizzeremo invece il termine “step”.

La proposizione 9.3 ci assicura che, durante l'applicazione del passo standard a profondità i, la “struttura” a livelli della proof-net resta praticamente inalterata. L'unico caso in cui ci può essere una variazione è quello in cui venga eseguito uno step $[\mathsf{w}]$, il quale è in grado, a priori, di cancellare interi livelli di una proof-net.

La strategia di normalizzazione che utilizzeremo nel seguito, che chiameremo *strategia standard*, sarà definita come segue:

Definizione 9.14 (Strategia standard) *Data una proof-net π di* $\overline{\mathbf{LLL}}$*, l'applicazione della* strategia standard *a π prevede l'esecuzione dei passi standard a profondità crescente, a partire dalla profondità 0.*

In sostanza, la nostra strategia applica il passo standard a profondità 0, poi a profondità 1, e così via fino ad arrivare a profondità massima. Osserviamo anzitutto che questa strategia è ben definita: in base alla proposizione 9.3, tutti gli step elementari di riduzione non modificano la “struttura” a livelli della proof-net, e dunque ha sempre senso applicare un passo ad una data profondità. L'unico caso particolare è quello di $[\mathsf{w}]$, grazie al quale interi livelli della proof-net possono essere cancellati; tuttavia, il livello al quale si sta operando in un dato istante (supponiamo sia i) non può mai sparire, e una volta terminato il passo standard a profondità i ci si troverà sempre in una delle seguenti condizioni: o esiste il livello $i+1$ al quale applicare il passo successivo, oppure non esistono ulteriori livelli. Se ogni passo termina, per l'osservazione fatta riguardo alla preservazione della profondità della proof-net, allora l'intera procedura termina.

Per chiarezza, è opportuno fornire la struttura approssimativa del seguito della sezione, nella quale dimostreremo in dettaglio la **PTIME**-correttezza di $\overline{\mathbf{LLL}}$:

- Anzitutto, dedicheremo il resto del paragrafo alla verifica del fatto che la strategia standard sia ben definita, cioè che effettivamente sia possibile eliminare i tagli nel modo in cui li esegue il passo standard.
- In seguito, nel paragrafo 9.3.2, introdurremo la nozione di *foresta contrattiva*, la quale servirà a descrivere in modo estremamente sintetico la dinamica di cut-elimination indotta dagli step polinomiali, che sono la chiave della complessità di $\overline{\mathbf{LLL}}$.
- Nel paragrafo 9.3.3 studieremo la dinamica delle foreste contrattive dal punto di vista della complessità, e sfrutteremo i risultati trovati

per fornire un bound temporale all'esecuzione del generico passo della strategia standard a profondità i. Il bound legherà la dimensione iniziale del livello i al numero di step elementari di riduzione da eseguire per far diventare tale livello lazy-cut-free. Una volta trovato il bound temporale per l'esecuzione di ciascun passo, potremmo calcolare la complessità dell'intera strategia standard come somma delle complessità dei vari passi applicati ai vari livelli; tuttavia, un calcolo del genere è troppo *naïf*, poiché non prende in considerazione il fatto che il passo a profondità i fa in generale aumentare la dimensione del livello $i + 1$. Poiché il bound del passo successivo dipenderà proprio da tale dimensione, è evidentemente sbagliato non tenerne conto. Il resto del paragrafo sarà dunque dedicato all'analisi di un upper bound all'incremento della size sotto i passi della strategia standard, ricavato sempre utilizzando le foreste contrattive.

■ Infine, nel paragrafo 9.3.4 metteremo assieme tutti i risultati trovati e calcoleremo finalmente la complessità della strategia standard, mostrando che essa è polinomiale nella dimensione della proof-net di partenza. Più precisamente, se π è una proof-net di dimensione s e profondità d, l'applicazione della strategia standard a π condurrà alla sua forma (almeno) lazy-cut-free in un numero di passi limitato da $p_d(s)$, dove p_d è un polinomio il cui grado dipende da d.

Iniziamo dunque con il primo punto del "programma". Partiamo dalle seguenti definizioni:

Definizione 9.15 (Assioma speciale) *Un nodo* ax *di una proof-net è detto* speciale *se:*

- *le sue due conclusioni sono etichettate con formule del tipo* $?A^\perp$, $!A$
- *la conclusione etichettata con* $?A^\perp$ *è premessa di un nodo* cut*, la cui altra premessa non è conclusione di un assioma*
- *la conclusione etichettata con* $!A$ *è premessa di un nodo qualsiasi, fuorché un nodo* cut

Il nodo cut *la cui premessa è conclusione di un assioma speciale sarà anch'esso detto "cut speciale".*

Definizione 9.16 (Proof-net contrattiva) *Una proof-net* π *di* $\overline{\mathbf{LLL}}$ *è detta* i-contrattiva *se a profondità* i *essa gli unici nodi* cut *che essa contiene sono polinomiali, shifting, commutativi additivi (anche estesi) o speciali.*

In sostanza, una proof-net i-contrattiva è una proof-net che non contiene cut lineari al livello i, fatta eccezione per i cut speciali. Una conseguenza diretta della definizione del passo standard a profondità i è che, dopo l'esecuzione della prima fase (quella in cui si eliminano tutti i tagli lineari), ci si ritrova senz'altro con una proof-net i-contrattiva. La fase successiva è quella di eliminare i tagli polinomiali, cioè quelli che coinvolgono contrazioni. Dimostreremo ora che l'eliminazione di questi tagli può creare solo altri tagli polinomiali o tagli speciali:

Proposizione 9.4 *Sia π una proof-net di $\overline{\mathbf{LLL}}$, e sia $\pi \xrightarrow{[\mathsf{c}]} \pi'$, con* [c] *eseguito a profondità i. Allora, se π è i-contrattiva, anche π' lo è.*

Dimostrazione. Supponiamo che a profondità i in π ci siano $n+1$ nodi cut, dei quali c è quello ridotto dallo step polinomiale e $c_1, \dots, c_n$ sono tutti gli altri. Per ipotesi, sappiamo che c è un taglio che coinvolge una contrazione, mentre i vari c_i sono tagli o su altre contrazioni, o su porte ausiliare, o su contrazioni additive, oppure ancora su assiomi speciali. Ricordando che gli unici nodi cut senza antenati sono quelli creati da uno step $[\parr/\otimes]$, e che in una proof-net contrattiva come π tali step sono già stati tutti eseguiti, è sufficiente dimostrare quanto segue:

(a) se c' è un residuo di c in π', questo non è un cut lineare oppure, se lo è, si tratta di un cut speciale;

(b) se c_i' è un residuo di c_i in π', questo non è un cut lineare oppure, se lo è, si tratta di un cut speciale.

Per quanto riguarda (a), sappiamo che se la contrazione ridotta era k-aria, c avrà k residui, ciascuno dei quali taglierà un nodo ! e un nodo la cui conclusione è etichettata con una formula del tipo $?A$. Tale nodo è necessariamente uno tra i seguenti 6 tipi di nodi: ax, ?D, ?C, ?W, pax e coad. Il caso ?D è escluso per la condizione di stratificazione a scendere; i casi ?C e ?W sono esclusi perché lo step [c] si considera applicato sulla versione nouvelle syntaxe di π, nella quale dunque ci sarebbe stata una configurazione eliminabile dagli step di conversione [cc] o [wc]; dei restanti 3 casi, nel caso pax abbiamo un taglio shifting (o commutativo additivo esteso, se nel pax-tree ci sono scatole additive), mentre nel caso coad abbiamo un taglio commutativo additivo, che non sono dunque tagli lineari; nell'ultimo caso, quello del nodo ax, abbiamo sì in taglio lineare, ma esso è speciale: l'assioma coinvolto ha infatti una conclusione *why not* premessa di un cut, e l'altra conclusione (necessariamente di tipo *of course*) non può essere tagliata con nulla, altrimenti ci sarebbe stato in π un taglio lineare, contro l'ipotesi che essa sia i-contrattiva (ricordiamo che l'assioma ha "precedenza" su tutti gli altri nodi, nel senso che un cut che coinvolge un assioma è per definizione di tipo ax, dunque lineare, indipendentemente da quale sia l'altro nodo coinvolto).

Consideriamo ora (b). Anzitutto, ogni c_i di π ha un unico residuo c_i' in π', poiché lo step [c] non può duplicare nulla alla profondità a cui viene applicato (eccetto le porte ausiliare e principali dei box coinvolti nello step). Il generico residuo c_i', essendo un cut, ha solo premesse, e dunque può aver cambiato il suo "status" solo se è cambiata una delle premesse rispetto all'antenato c_i. Nel corso dello step [c], l'unico nodo che subisce modifiche in tal senso è l'eventuale nodo la cui premessa è conclusione della porta ausiliaria della scatola coinvolta nella duplicazione. In tal caso infatti, con la k-uplicazione delle scatole, le k porte ausiliarie vengono contratte, e la conclusione di questa nuova contrazione k-aria diventa la premessa del nodo che prima era legato all'unica pax esistente. Supponiamo che tale nodo sia proprio uno dei vari c_i. c_i non può essere un taglio su un assioma, altrimenti tale assioma non sarebbe speciale, contraddicendo l'ipotesi che π fosse contrattiva. c_i è dunque shifting, mentre c_i' è un taglio su una contrazione; siamo così passati da un taglio shifting ad uno polinomiale, senza però creare nuovi tagli lineari. $\square$

La proposizione appena dimostrata garantisce che le tre "fasi" del passo standard sono ben definite. Passiamo dunque all'analisi della complessità di tale strategia di cut-elimination.

9.3.2 Le foreste contrattive

In questo paragrafo introdurremo una particolare struttura, che chiameremo *foresta contrattiva*, in grado di catturare le dinamiche indotte in una proof-net i-contrattiva dall'esecuzione di step polinomiali. Questi ultimi, coinvolgendo le contrazioni, sono gli unici a contribuire in modo significativo all'aumento della dimensione di una proof-net nel corso della cut-elimination; descrivere la dinamica di tali aumenti significa dunque descrivere l'intera dinamica del processo di cut-elimination.

Nel resto della discussione lavoreremo con le seguenti convenzioni:

- Si farà praticamente sempre riferimento alla versione *nouvelle syntaxe* di una proof-net i-contrattiva, ottenuta dopo l'applicazione della "prima fase" del passo standard a profondità i; si eviterà dunque di specificare nuovamente tutto ciò ovunque non sia strettamente necessario.

- Se non esplicitamente chiarito, la proof-net in questione non sarà necessariamente una proof-net di $\overline{\mathbf{LLL}}$ ma sarà, più in generale, di $\mathbf{LL}_\S$.

Forniamo anzitutto una definizione preliminare:

Definizione 9.17 (Cammino diretto) *Sia π una proof-net. Un* cammino diretto *di π è definito induttivamente nel seguente modo:*

- *un cammino orientato (seguendo l'orientamento degli archi stabilito per le proof-net) tra due nodi n_1 ed n_2 di π (scritto in breve $n_1, \ldots, n_2$) è un cammino diretto di π, il cui orientamento può essere indifferentemente da n_1 a n_2 o viceversa (in altre parole, i cammini diretti non sono orientati); i nodi n_1 ed n_2 sono detti* estremi *del cammino diretto*

- *siano $m, \ldots, a$ e $a, \ldots, n$ due cammini diretti di π aventi un estremo in comune, e sia tale estremo un nodo* ax *o* cut*; allora $m, \ldots, n$ (o, indifferentemente, $n, \ldots, m$), ottenuto "congiungendo" tali cammini, è un cammino diretto di π; gli estremi di tale cammino saranno naturalmente m ed n*

- *nient'altro è un cammino diretto*

In sostanza, i cammini diretti sono cammini non orientati che consentono di passare da un nodo di una proof-net ad un altro "passando attraverso" assiomi e cut.

Introduciamo ora la nozione fondamentale di *ordine fra !-box*:

Definizione 9.18 *Siano $\mathcal{B}_1$ e $\mathcal{B}_2$ due !-box di una proof-net π, entrambi a profondità i (cioè le cui porte si trovano a profondità i). Diremo che $\mathcal{B}_1$* precede immediatamente *$\mathcal{B}_2$, e scriveremo*

$$\mathcal{B}_1 \lhd_1 \mathcal{B}_2$$

se e solo se esiste un cammino diretto che ha per estremi la porta principale di $\mathcal{B}_1$ e una delle porte ausiliarie di $\mathcal{B}_2$, e tale che tutti gli altri nodi del cammino sono cut *o* ?C*. In figura 9.3.2 sono riportate le due situazioni possibili in cui $\mathcal{B}_1 \lhd_1 \mathcal{B}_2$.*
Sia $\unlhd$ la chiusura riflessiva e transitiva di $\lhd_1$. Se $\mathcal{B}_1 \unlhd \mathcal{B}_2$, diremo che $\mathcal{B}_1$ precede *$\mathcal{B}_2$; se è anche $\mathcal{B}_1 \neq \mathcal{B}_2$, allora scriveremo*

$$\mathcal{B}_1 \lhd \mathcal{B}_2$$

e diremo che $\mathcal{B}_1$ precede strettamente *$\mathcal{B}_2$.*

E' opportuno sottolineare che, se $\mathcal{B} \lhd_1 \mathcal{B}'$, allora $\mathcal{B}$ e $\mathcal{B}'$ sono due !-box alla stessa profondità e, se tale profondità è maggiore di zero, esse sono contenute nella stessa scatola (che non è necessariamente un !-box).

Proposizione 9.5 *Se π è una proof-net di $\mathbf{LL}_\S$, la relazione $\unlhd$ è una relazione d'ordine parziale sull'insieme dei !-box a profondità i di π.*

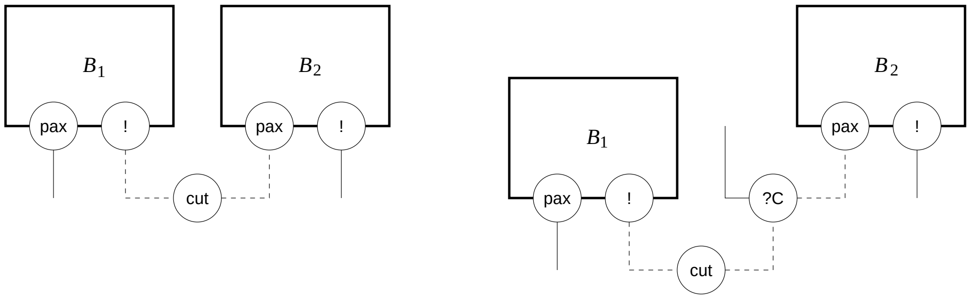


Figura 9.2: Le due situazioni in cui $B_1 \lhd_1 B_2$. I cammini diretti tra porta principale e porta ausiliaria sono rappresentati dagli archi tratteggiati. Naturalmente, la contrazione in generale può essere n-aria e, inoltre, se non siamo in $\overline{\mathbf{LLL}}$, possono esserci più pax in B_2, e la contrazione può contrarre le pax di una stessa scatola.

Dimostrazione. Che la relazione sia parziale è chiaro dalla definizione: possono senz'altro esistere due scatole tra le quali è impossibile costruire un cammino diretto.

Riflessività e transitività di $\unlhd$ sono verificate per definizione; resta da dimostrare l'antisimmetria della relazione. Supponiamo dunque per assurdo l'esistenza di due scatole $\mathcal{B}'$, $\mathcal{B}''$ che contraddicano l'asimmetria, cioè tali che

$$\mathcal{B}'' \lhd \mathcal{B}' \lhd \mathcal{B}''$$

Mostreremo ora che tale relazione induce un ciclo in uno dei grafi di correttezza[4] di π. Infatti, per definizione di $\unlhd$, esistono $j+k$!-box $(j,k \geq 1)$ $\mathcal{B}'_1, \ldots, \mathcal{B}'_j$ e $\mathcal{B}''_1, \ldots, \mathcal{B}''_k$ tali che

$$\mathcal{B}'' \lhd_1 \mathcal{B}'_1 \lhd_1 \ldots \lhd_1 \mathcal{B}'_j \lhd_1 \mathcal{B}' \lhd_1 \mathcal{B}''_1 \lhd_1 \ldots \lhd_1 \mathcal{B}''_k \lhd_1 \mathcal{B}''$$

Esiste dunque uno switching delle contrazioni a profondità i in grado di generare un grafo di correttezza in cui, per definizione di $\lhd_1$, possiamo trovare un cammino "tra scatole" (cioè un cammino in cui se si entra da una porta ausiliaria di un !-box si può uscire immediatamente dopo dalla sua porta principale — o viceversa, dato che i grafi di correttezza non sono orientati) tale che dal ! di $\mathcal{B}'$ si può arrivare ad una pax di $\mathcal{B}''$, e dal nodo ! di tale scatola si può tornare ad una pax di $\mathcal{B}'$. Poiché uno dei grafi di correttezza conterrebbe così un ciclo, grazie al criterio AC si può concludere che π non sarebbe una proof-net[5], e dunque questa situazione non può mai presentarsi.

[4] I grafi di correttezza per le proof-net nel caso generale non sono stati introdotti, ma si possono definire (in modo non proprio banale) come estensione di quelli delle proof-net di **MLL**; si veda [25].

[5] L'acicilicità semplice vale infatti anche per le proof-net nel caso generale. In realtà, grazie all'introduzione dei cosiddetti *jump* ([11], [25]), si può estendere ACC alle proof-net di **LL**.

$\Box$

Essendo una relazione d'ordine parziale, $\trianglelefteq$ induce un grafo orientato sull'insieme dei !-box del livello i di π, i cui nodi sono appunto le scatole stesse, e i cui archi rappresentano l'esistenza di una relazione $\lhd_1$ tra due scatole. Tale grafo può, a priori, presentare più di una componente connessa; la disconnessione si ha nonappena il livello i di π sia composto da più di una scatola, ma può anche verificarsi nel caso in cui, pur essendo il livello tutto "concentrato" in una sola scatola, all'interno di essa ci sia un box non confrontabile con nessun altro.

Andremo ora a caratterizzare in modo un po' più preciso le proprietà strutturali del grafo indotto da $\trianglelefteq$. Anzitutto, la seguente proposizione garantisce che in ciascuna componente connessa esiste sempre almeno un "nodo pozzo", che è un elemento minimale dell'ordine:

Proposizione 9.6 (Elementi minimali) *Sia π una proof-net. Allora*

- *Esiste (almeno) un !-box $\mathcal{M}_0$ a profondità 0 tale che, per ogni altro !-box $\mathcal{B}_0$ a profondità 0, $\mathcal{B}_0 \trianglelefteq \mathcal{M}$ se e solo se $\mathcal{B}_0 = \mathcal{M}_0$.*
- *Per ogni scatola esponenziale $\mathcal{B}$ (di qualsiasi tipo) a profondità i ($0 \leq i < \partial(\pi) - 1$), se $\mathcal{B}$ contiene !-box, allora esiste (almeno) un !-box $\mathcal{M}$ contenuto in $\mathcal{B}$ e di profondità $i+1$ tale che, per ogni altro !-box $\mathcal{B}'$ contenuto in $\mathcal{B}$ e di profondità $i+1$, $\mathcal{B}' \trianglelefteq \mathcal{M}$ se e solo se $\mathcal{B}' = \mathcal{M}_0$.*

Dimostrazione. Sia l'insieme dei !-box a profondità 0 che l'insieme dei !-box contenuti in una scatola esponenziale e posizionati alla stessa profondità sono sottoinsiemi parzialmente ordinati rispetto a $\trianglelefteq$. Poiché ciascuno di questi sottoinsiemi è finito (essendo finito l'insieme di tutti i !-box di π), l'esistenza di elementi minimali è garantita. $\Box$

Osserviamo che un !-box senza porte ausiliarie è sempre un elemento minimale; si verifica poi che, dato un !-box $\mathcal{B}$, possono esserci diversi $\mathcal{B}'$ "immediatamente maggiori di $\mathcal{B}$", cioè tali che $\mathcal{B} \lhd_1 \mathcal{B}'$. Inoltre, l'argomento utilizzato per la proposizione appena dimostrata garantisce anche l'esistenza di elementi massimali, e dunque di "nodi sorgente" per ciascuna componente connessa del grafo.

Mentre la proposizione 9.6 ha validità generale, quella che segue è specifica di $\overline{\mathbf{LLL}}$, ed è in effetti l'elemento cruciale della **PTIME**-correttezza:

Proposizione 9.7 *Sia π una proof-net di $\overline{\mathbf{LLL}}$, e $\mathcal{B}$ un suo !-box. Allora, esiste al più un !-box $\mathcal{B}'$ tale che $\mathcal{B}' \lhd_1 \mathcal{B}$.*

Dimostrazione. Supponiamo che esistano due scatole $\mathcal{B}'$ ed $\mathcal{B}''$, diverse tra loro, tali che $\mathcal{B}' \lhd_1 \mathcal{B}$ e $\mathcal{B}'' \lhd_1 \mathcal{B}$. Per definizione di $\lhd_1$, deve allora esistere un cammino diretto contenente solo cut e contrazioni sia tra il ! di $\mathcal{B}'$ e una pax di $\mathcal{B}$ che tra il ! di $\mathcal{B}''$ e una pax di $\mathcal{B}$. Ma, per ipotesi, π è una proof-net di $\overline{\mathbf{LLL}}$, e dunque $\mathcal{B}$ non può che avere una sola pax; poiché i cammini diretti in questione diramano solo nella direzione "dal ! alla pax", non è possibile che una sola pax sia collegata tramite un cammino diretto a due porte principali diverse. Di conseguenza, abbiamo necessariamente $\mathcal{B}' = \mathcal{B}''$. $\square$

Grazie a quanto appena dimostrato, a partire da un qualsiasi !-box $\mathcal{B}$ esiste, nel grafo indotto da $\trianglelefteq$, un unico cammino che da lui porta ad un nodo pozzo. Vale dunque banalmente la seguente proprietà:

Proposizione 9.8 (Minimi) *Sia π una proof-net di $\overline{\mathbf{LLL}}$, e $\mathcal{B}$ un !-box di π a profondità i. Allora, a tale profondità esiste un unico !-box $\mathcal{M}_{\mathcal{B}}$ tale che $\mathcal{M}_{\mathcal{B}}$ è il minimo dell'insieme dei box $\mathcal{B}' \trianglelefteq \mathcal{B}$.*

Limitandoci a **overlineLLL**, in base alle proposizioni viste fin qui possiamo concludere quanto segue. Sappiamo che il grafo indotto da $\trianglelefteq$ è costituito da un certo numero di componenti connesse; all'interno di tali componenti connesse, esistono pozzi e sorgenti (prop. 9.6), e i nodi hanno un numero qualsiasi di archi entranti ma *al più un arco uscente* (prop. 9.7). Di conseguenza, le sorgenti possono essere più d'una, ma, per ogni componente connessa, il pozzo è unico (prop. 9.8).

Quello che abbiamo appena descritto è un albero, in cui le foglie sono i nodi sorgente e la radice è il nodo pozzo; dato un !-box minimale $\mathcal{M}$, è dunque ben definito l'albero $\mathsf{Tree}(\mathcal{M})$ di $\mathcal{M}$ rispetto a $\trianglelefteq$. Naturalmente in $\overline{\mathbf{ELL}}$ tutto ciò non è affatto vero; essendo ammissibili più porte ausiliarie per un singolo !-box, il grafo indotto da $\trianglelefteq$ resta in generale tale, e le sue componenti connesse non possono essere ridotte ad alberi.

Siamo allora pronti a dare la prima versione della definizione di *foresta contrattiva*:

Definizione 9.19 (Foresta contrattiva semplice) *Sia π una proof-net i-contrattiva di $\overline{\mathbf{LLL}}$. La* foresta contrattiva semplice *a profondità i di π, che denoteremo con $\mathsf{SCF}_i(\pi)$ (o, ove sia chiara la profondità di cui si parla, semplicemente $\mathsf{SCF}(\pi)$), è definita nel modo seguente. Siano $\mathcal{M}_1, \ldots, \mathcal{M}_n$ i !-box minimali (rispetto a $\trianglelefteq$) del livello i di π; allora*

$$\mathsf{SCF}_i(\pi) = \bigcup_{j=1}^{n} \mathsf{Tree}(\mathcal{M}_j)$$

Definizione 9.20 *Se Φ è una foresta contrattiva semplice, denoteremo con $\sharp(\Phi)$ quella chiameremo la sua* dimensione *(o* size*), corrispondente al numero di nodi di Φ.*

Proposizione 9.9 *Gli alberi che compongono la foresta contrattiva semplice del livello i di una proof-net π di $\overline{\mathbf{LLL}}$ sono tutti disgiunti.*

Dimostrazione. L'asserto è un corollario della proposizione 9.8: infatti, se due alberi $\mathsf{Tree}(\mathcal{M}')$ e $\mathsf{Tree}(\mathcal{M}'')$ di due diversi box minimali $\mathcal{M}'$ e $\mathcal{M}''$ avessero un nodo in comune (eccetto ovviamente la radice), esisterebbe un box $\mathcal{B}$ tale che $\mathcal{M}' \trianglelefteq \mathcal{B}$ e $\mathcal{M}'' \trianglelefteq \mathcal{B}$, contraddicendo la proposizione 9.8. □

La foresta contrattiva semplice di una proof-net di $\overline{\mathbf{LLL}}$ è dunque effettivamente una foresta, nel senso che è composta da alberi disgiunti. Questa è la differenza cruciale rispetto a $\overline{\mathbf{ELL}}$; in quest'ultima anzitutto, non si può parlare dell'"albero" di un box minimale, ma piuttosto di un grafo; inoltre, tali grafi possono a priori intersecarsi, perché non esiste un elemento minimale univoco per ogni scatola. L'estensione del concetto di foresta contrattiva a $\overline{\mathbf{ELL}}$ comporta dunque, necessariamente, l'abbandono delle foreste in favore di strutture più complesse quali i grafi[6].

In realtà, noi non saremo interessati proprio alle foreste contrattive semplici, ma piuttosto ad alcune loro varianti. Cominciamo anzitutto con il definire una particolare categoria di !-box:

Definizione 9.21 (Scatola contrattiva) *Un !-box di una proof-net π di $\overline{\mathbf{LLL}}$ è detto* contrattivo *se la conclusione della sua porta principale è premessa di un cut la cui altra premessa è la conclusione di una contrazione.*

Osserviamo che, poiché stiamo assumendo di lavorare in nouvelle syntaxe, la contrazione contro cui è tagliata una scatola contrattiva non può avere come premesse altre contrazioni; ad ogni scatola contrattiva può dunque univocamente essere associata l'arità della "sua contrazione".

Considerando ora la relazione $\trianglelefteq$, è ben definito (grazie all'aciclicità delle proof-net) il concetto di *scatola contrattiva massimale*: questa è semplicemente una scatola contrattiva $\mathcal{C}$ per la quale non esiste alcuna scatola contrattiva $\mathcal{C}'$ tale che $\mathcal{C} \triangleleft \mathcal{C}'$.

Possiamo allora introdurre le foreste di nostro interesse:

Definizione 9.22 (Sotto-foresta contrattiva massimale) *Sia π una proof-net di $\overline{\mathbf{LLL}}$, e sia $\mathsf{SCF}_i(\pi)$ la sua foresta contrattiva semplice a profondità i. La* sotto-foresta contrattiva massimale *(o, più semplicemente,* sotto-foresta massimale*) a profondità i di π, che denoteremo con $\mathsf{MCF}_i(\pi)$, è $\mathsf{SCF}_i(\pi)$ "potata" ai box contrattivi massimali; vale a dire, il nodo n di $\mathsf{SCF}_i(\pi)$ corrispondente al generico !-box $\mathcal{B}$ di π sarà in $\mathsf{MCF}_i(\pi)$ se e solo se esiste un*

[6] Questo è forse l'unico punto in cui $\overline{\mathbf{ELL}}$ è "meno semplice" di $\overline{\mathbf{LLL}}$...

!-box contrattivo $\mathcal{C}$ tale che $\mathcal{B} \trianglelefteq \mathcal{C}$ (se $\mathcal{B}$ è contrattivo, $\mathcal{C}$ e $\mathcal{B}$ possono dunque coincidere). E' chiaro che, se un albero di $\mathsf{SCF}_i(\pi)$ non contiene alcun nodo corrispondente ad una scatola contrattiva, questo sarà completamente cancellato.

La dimensione di una sotto-foresta massimale resta naturalmente definita in modo identico a quello delle foreste contrattive semplici, ereditando anche la stessa notazione.

Oltre alle sotto-foreste massimali, ci serviremo anche di un'altra variante delle foreste contrattive semplici, la quale è definita come segue:

Definizione 9.23 (Foresta contrattiva pesata) *Sia π una proof-net di* $\overline{\textbf{LLL}}$. *La* foresta contrattiva pesata *(o, più semplicemente,* foresta contrattiva*) a profondità i di π, denotata con $\mathsf{CF}_i(\pi)$ (o $\mathsf{CF}(\pi)$ qualora non ci sia ambiguità), corrisponde a $\mathsf{SCF}_i(\pi)$ ai cui nodi n è però assegnato un peso $\mho(n)$, nel modo seguente:*

- *se n è un nodo facente parte anche di $\mathsf{MCF}_i(\pi)$, e la scatola $\mathcal{C}$ che esso rappresenta è contrattiva, allora $\mho(n) = k$, dove k è l'arità della contrazione con cui $\mathcal{C}$ è tagliata*
- *se n è un nodo facente parte anche di $\mathsf{MCF}_i(\pi)$, ma la scatola che esso rappresenta non è contrattiva, allora $\mho(n) = 1$*
- *se n è un nodo che non fa parte anche $\mathsf{MCF}_i(\pi)$, allora $\mho(n) = 0$*

Un nodo n tale che $\mho(n) > 0$ sarà detto vivo*; se esso verifica anche $\mho(n) > 1$, allora n sarà detto* contrattivo*. Al contrario, un nodo di peso nullo sarà detto* morto.

Anche per le foreste contrattive rimane ovviamente definita la dimensione, esattamente come per le foreste contrattive semplici e le sotto-foreste massimali.

E' importante segnalare il fatto che, in un qualsiasi albero di una foresta contrattiva, i predecessori di un nodo vivo sono tutti necessariamente vivi; detto altrimenti, se da un nodo vivo si percorre il cammino che da esso porta alla radice, non si possono mai incontrare nodi morti. La spiegazione di ciò segue banalmente dalla definizione di foresta contrattiva: i nodi vivi sono esattamente i nodi della sotto-foresta massimale; di conseguenza, partendo dalle foglie della foresta contrattiva e andando verso la radice, prima o poi si entrerà in tale sotto-foresta e si incontreranno solo nodi vivi.

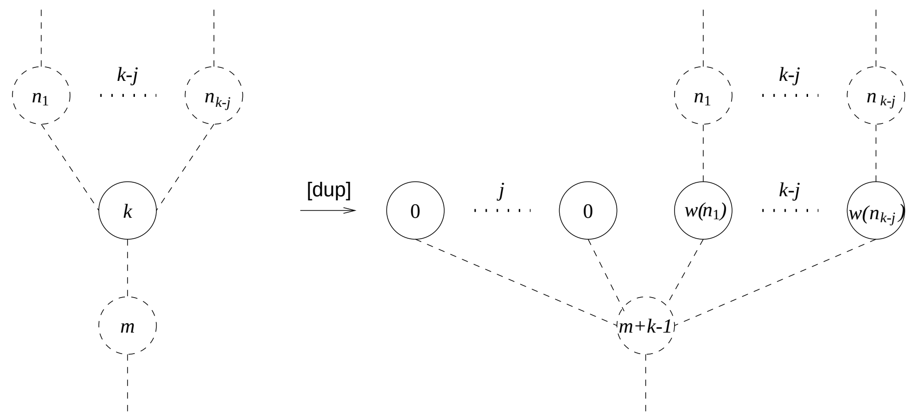
Le foreste contrattive e le sotto-foreste massimali saranno la base della dimostrazione della **PTIME**-correttezza di $\overline{\textbf{LLL}}$; le prime serviranno a descrivere la complessità *spaziale* del processo di cut-elimination, le seconde quella *temporale*.

Affinché sia possibile ricavare informazioni sulla dinamica dell'eliminazione del taglio nelle proof-net a partire dalle foreste contrattive, è necessario definire una qualche forma di dinamica anche per esse. In altre parole, introdurremo un sistema di riscrittura che renderà le foreste contrattive "isomorfe" alle proof-net sotto la cut-elimination.

Dobbiamo a questo punto ricordare che le foreste contrattive (e dunque le foreste minimali) sono definite per proof-net contrattive, vale a dire proof-net in cui, ad un certo livello, non ci sono più tagli lineari da ridurre (eccetto quelli "innocui" su assiomi speciali). Quello a cui siamo interessati è allora l'eliminazione dei tagli sulle contrazioni; le foreste contrattive saranno quindi dotate di una dinamica per mezzo della quale si possa rappresentare, in modo estremamente sintetico, ciò che avviene con l'esecuzione di uno step [c].

Dimentichiamoci per un momento di $\overline{\mathbf{LLL}}$ e delle sue proof-net e poniamoci in un ambito più generale. Consideriamo una foresta composta da alberi "radicati", ovvero alberi in cui ci sia un nodo speciale che identifichiamo come sua radice. Nelle rappresentazioni grafiche, la radice in questione sarà sempre il nodo "più in basso" di tutti. I nodi degli alberi che stiamo considerando saranno etichettati con un numero intero non negativo, che chiamiamo *peso* del nodo. Per i nodi utilizzeremo, in dipendenza dal loro peso, la stessa terminologia introdotta per i nodi delle foreste contrattive (ci riferiamo alle definizioni di nodo morto, vivo e contrattivo).

Se k è un intero strettamente maggiore di 1, definiamo la seguente regola di riscrittura:



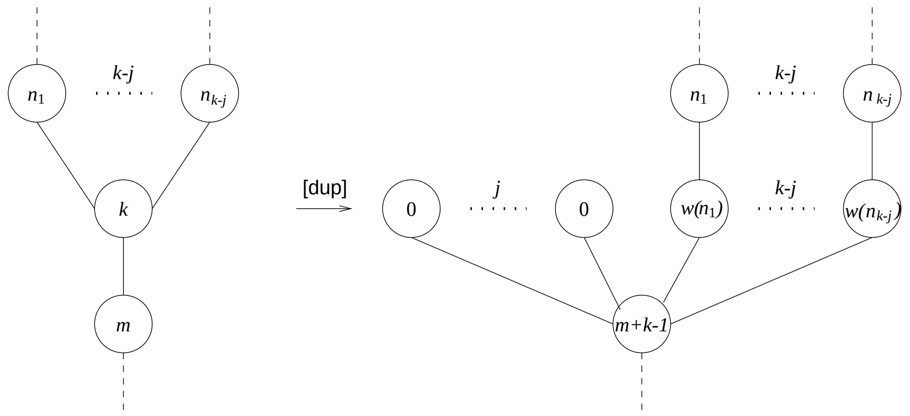
Nella figura, ogni nodo è etichettato con il suo peso; la regola [dup] è dunque definita solo per i nodi contrattivi[7]. Le linee tratteggiate indicano nodi e archi che potrebbero esserci o meno, e w è una funzione così definita:

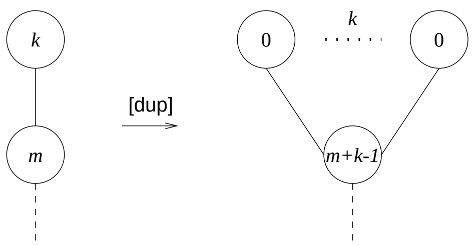
$$w(n) = 1 - \delta(n)$$

dove δ è l'"impulso di Kronecker"[8].

Con la regola [dup] si vuole in sostanza simulare ciò che avviene nelle proof-net durante l'esecuzione di uno step [c]. Infatti, se vediamo i nodi come scatole, e se vediamo il peso di un nodo contrattivo come l'arità della contrazione contro cui è tagliato, vediamo subito come [dup] non faccia altro che creare un numero di copie del nodo su cui opera in numero proprio pari al suo peso, spostando quest'ultimo sull'eventuale nodo genitore, proprio come una contrazione "si sposta" dopo l'esecuzione di [c] andando eventualmente a collassare su una contrazione preesistente.

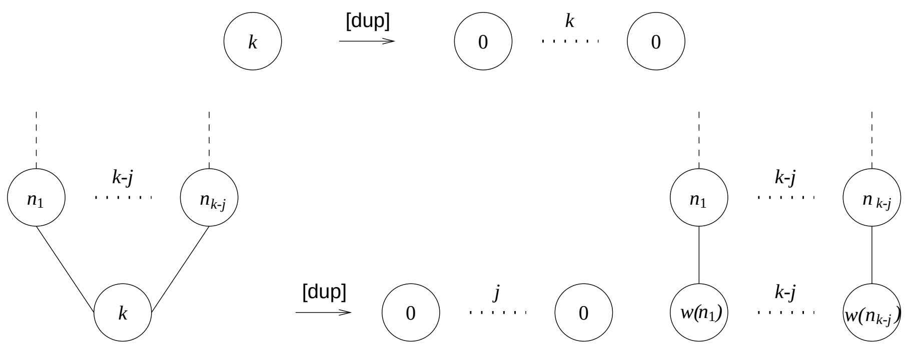
In base alla presenza o assenza degli elementi indicati con linee tratteggiate, è possibile istanziare la regola [dup] in quattro casi base:





[7]In realtà la regola in questione potrebbe essere definita anche per $k = 1$, e dunque estesa in generale a tutti i nodi vivi; tuttavia, l'applicazione della regola ad un nodo di peso unitario trasformerebbe un albero in sé stesso, e non è quindi significativa dal punto di vista dinamico. Vedremo più avanti che, inoltre, [dup] ha un'interpretazione in ambito di proof-net solamente quando essa è applicata a nodi contrattivi.

[8]$\delta(0) = 1$, mentre per n non nullo $\delta(n) = 0$



Diamo ora la seguente definizione:

Definizione 9.24 *Una foresta contrattiva è detta* irriducibile *se ad essa non può essere applicata alcuna istanza della regola* [dup]*, cioè se essa non contiene nodi contrattivi.*

Mostreremo ora che [dup] è in realtà la versione "sintetica" dello step [c] delle proof-net all'interno delle foreste contrattive. Per far ciò, sarà sufficiente dimostrare il seguente enunciato:

Proposizione 9.10 *Se π è una proof-net i-contrattiva di $\overline{\overline{\mathbf{LLL}}}$, allora $\pi \xrightarrow{[\mathsf{c}]} \pi'$ se e solo se $\mathsf{CF}_i(\pi) \xrightarrow{[\mathsf{dup}]} \mathsf{CF}_i(\pi')$, dove lo step* [c] *è applicato ad un cut tra un !-box e una contrazione a profondità i, e* [dup] *è applicato al nodo corrispondente alla scatola contrattiva coinvolta in* [c]*.*

Dimostrazione. Dimostriamo il primo verso dell'implicazione: sia $\pi \xrightarrow{[\mathsf{c}]} \pi'$ e $\mathsf{CF}(\pi) \xrightarrow{[\mathsf{dup}]} \Phi$; dobbiamo concludere che $\Phi = \mathsf{CF}(\pi')$.

Il taglio ridotto da [c] coinvolge un !-box $\mathcal{B}$ e una contrazione di arità, supponiamo, k. Di conseguenza, in π' saranno presenti, al livello i, k copie di $\mathcal{B}$ tagliate ciascuna con una premessa della contrazione, e tali scatole vedranno la loro eventuale porta ausiliaria contratta con una contrazione ancora di arità k. Abbiamo due possibilità: $\mathcal{B}$ era minimale in π oppure non lo era. Nel primo caso, le copie di $\mathcal{B}$ sono ancora minimali; per la proposizione 9.8 esse costituiscono dunque le radici di k alberi, e ciascuna radice ha al più un discendente diretto che (in $\mathsf{CF}(\pi)$ era uno dei discendenti di $\mathcal{B}$. La foresta contrattiva di π' in questo caso può dunque essere descritta come segue: tutti gli alberi tranne uno sono identici alla foresta di π; al posto dell'albero che non c'è più ce ne sono altri k, e le radici di questi nuovi alberi saranno vive solo se era vivo il corrispondente figlio della radice del vecchio albero. L'esecuzione di [dup] alla radice di tale albero in $\mathsf{CF}(\pi)$ produce una foresta identica a quella descritta, dunque $\Phi = \mathsf{CF}(\pi')$.

Resta da vedere il caso in cui $\mathcal{B}$ non sia minimale. Anche qui abbiamo due possibilità: la pax di $\mathcal{B}$ è direttamente tagliata con la porta principale del predecessore di $\mathcal{B}$, oppure essa è premessa di una contrazione j-aria. Nel primo caso, la contrazione che ha come premesse le pax delle k copie di $\mathcal{B}$ in π' è premessa di un cut contro il predecessore di $\mathcal{B}$; tale predecessore, che in π non era un box contrattivo, lo è ora diventato in π', e la sua arità è esattamente k. Nel secondo caso, il predecessore di $\mathcal{B}$ era già contrattivo, di arità j; in π', tale box è ancora contrattivo, ma (con la conversione in nouvelle syntaxe), l'arità della sua contrazione è aumentata, perché ora ci sono k box a rimpiazzarne uno solo. Di conseguenza, l'arità del predecessore di $\mathcal{B}$ sarà $j + k - 1$. In entrambi i casi, le copie di $\mathcal{B}$ in π' daranno luogo a nodi vivi se erano vivi i nodi dei rispettivi successori di $\mathcal{B}$ in π, con i quali ciascuna copia è ora tagliata. L'esecuzione di $[\mathsf{dup}]$ sul nodo corrispondente a $\mathcal{B}$ produce esattamente l'effetto desiderato: l'arità del predecessore del nodo corrispondente a $\mathcal{B}$ aumenta di $k-1$ (dunque, se questa era 1, diventa k), e i nodi generati dalla riscrittura sono vivi solo se lo sono i loro diretti discendenti. Anche qui, abbiamo allora $\Phi = \mathsf{CF}(\pi')$.

Dimostriamo ora l'altro verso dell'implicazione: sia $\mathsf{CF}(\pi) \xrightarrow{[\mathsf{dup}]} \Phi$; dobbiamo concludere che esiste una proof-net π' tale che $\pi \xrightarrow{[\mathsf{c}]} \pi'$ e $\mathsf{CF}(\pi') = \Phi$. La costruzione di una tale proof-net è molto semplice. Sia k il peso del nodo di $\mathsf{CF}(\pi)$ a cui è applicato $[\mathsf{dup}]$; per definizione, il box $\mathcal{B}$ di π corrispondente a tale nodo è senz'altro contrattivo ($k \geq 2$). Rimpiazziamo dunque $\mathcal{B}$ con k sue copie, le cui pax (se presenti) sono contratte per mezzo di una contrazione $m + k - 1$ aria (dove m è l'arità dell'eventuale contrazione di cui la pax di $\mathcal{B}$ era premessa, $m = 1$ se la pax non era premessa di una contrazione; se $\mathcal{B}$ non aveva pax, non servirà neanche aggiungere la contrazione in questione), e le cui porte principali sono tagliate ciascuna con una premessa della contrazione contro la quale era tagliato $\mathcal{B}$ in π. E evidente che abbiamo costruito una proof-net π' tale che $\mathsf{CF}(\pi') = \Phi$; questa proof-net, per definizione di $[\mathsf{c}]$, è anche quella che soddisfa $\pi \xrightarrow{[\mathsf{c}]} \pi'$. Abbiamo così concluso la dimostrazione. $\square$

La proposizione 9.10 stabilisce dunque una sorta di "isomorfismo" tra le proof-net sotto l'applicazione dello step $[\mathsf{c}]$ e le foreste contrattive sotto l'applicazione di $[\mathsf{dup}]$. In base a tale corrispondenza, è possibile fare delle constatazioni sulla dinamica della cut-elimination in una proof-net π a partire da considerazioni sulla dinamica della riscrittura di $\mathsf{CF}(\pi)$. Ad esempio, è chiaro che, per costruzione, il numero di nodi di $\mathsf{CF}_i(\pi)$ è esattamente uguale al numero di !-box contenuti nel livello i di π; inoltre, vale senz'altro la seguente proprietà:

Proposizione 9.11 *Sia π una proof-net i-contrattiva di $\overline{\mathbf{LLL}}$. π non am-*

mette applicazioni dello step [c] *a profondità* i *se e solo se* $\mathsf{CF}_i(\pi)$ *è irriducibile.*

Dimostrazione. Conseguenza immediata della corrispondenza indotta dalla proposizione 9.10. □

Dimostriamo ora il seguente lemma:

Lemma 9.2 *Sia* π *una proof-net* i*-contrattiva di* $\overline{\mathbf{LLL}}$. *Se* $\mathsf{CF}_i(\pi)$ *contiene nodi vivi, allora essa contiene anche almeno un nodo contrattivo.*

Dimostrazione. Se $\mathsf{CF}(\pi)$ contiene nodi vivi, significa che $\mathsf{MCF}(\pi)$ è non vuota; ma, per definizione di sotto-foresta massimale, esiste almeno una scatola contrattiva massimale. □

In base al lemma 9.2 si può dimostrare il seguente asserto riguardante le forme irriducibili delle foreste contrattive:

Proposizione 9.12 *Una foresta contrattiva è irriducibile se e solo se essa contiene solo nodi morti.*

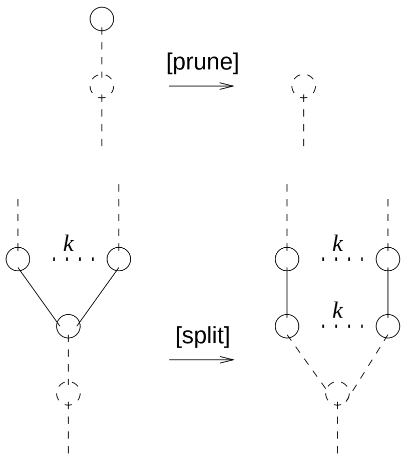
Dimostrazione. Per definizione, una foresta è irriducibile se e solo se essa non contiene nodi contrattivi; ma, per contrapposizione, il lemma 9.2 garantisce che se non ci sono nodi contrattivi, allora non ci sono neanche nodi vivi in assoluto, e dunque la foresta è composta da soli nodi morti. □

Se una foresta contrattiva contiene solo nodi morti, diremo che tale foresta è morta. Possiamo allora riformulare la proposizione 9.11 nel seguente modo, che enunceremo come lemma:

Lemma 9.3 *Sia* π *una proof-net* i*-contrattiva di* $\overline{\mathbf{LLL}}$. π *non ammette applicazioni dello step* [c] *a profondità* i *se e solo se* $\mathsf{CF}_i(\pi)$ *è morta.*

Il lemma 9.3 è fondamentale ai fini dell'analisi della cut-elimination di $\overline{\mathbf{LLL}}$: esso asserisce che la fase del passo a profondità i della strategia standard in cui si opera sulle contrazioni corrisponde esattamente all'uccisione di tutti i nodi della foresta contrattiva iniziale. La dimensione della foresta contrattiva morta indica dunque il numero di scatole presenti nella proof-net al termine della "fase contrattiva" della strategia standard; fornire un bound a tale dimensione significa dunque limitare la size della proof-net risultante. Inoltre, il numero di applicazioni di regole [dup] necessarie ad "uccidere" tutti i nodi della foresta contrattiva è esattamente uguale al numero di step elementari di cut elimination (a meno di quelli sugli assiomi speciali) necessari a terminare la fase contrattiva del passo standard, il quale porta ad una proof-net che contiene solo tagli shifting. A partire da un bound su tale numero si può dunque fornire un bound temporale al processo di cut-elimination.

Tuttavia, a tal scopo è più utile sfruttare le sotto-foreste contrattive massimali; per queste ultime è possibile definire un sistema di riscrittura composto dalle seguenti due regole, che sono la versione "non pesata" di [dup]:



In sostanza, [prune] consente di "potare" un albero, eliminando da esso una foglia (ovviamente assieme all'arco che la collega al nodo genitore); se la foglia in questione è anche radice dell'albero, l'applicazione di [prune] fornisce l'albero vuoto. [split] invece agisce sulle ramificazioni degli alberi, facendole "scendere" verso la radice. Quando la ramificazione è proprio la radice, [split] divide l'albero a cui viene applicata in k alberi, dove k è il numero di discendenti della radice stessa.

Osserviamo che la sotto-foresta massimale di una proof-net è perfettamente ricavabile dalla sua foresta contrattiva: essa è semplicemente la sotto-foresta comprendente i soli nodi vivi, dei quali viene completamente ignorato il peso. In quest'ottica, è chiaro come [prune] e [split] non siano altro che due specializzazioni di [dup] adattate alle sotto-foreste massimali; [prune] corrisponde all'applicazione di [dup] ad un nodo rappresentante un box contrattivo massimale, mentre [split] corrisponde all'applicazione di [dup] ad un nodo contrattivo non massimale.

In base a tale osservazione, possiamo trasferire tutti i risultati trovati per le foreste contrattive alle sotto-foreste massimali. In particolare, vale senza bisogno di dimostrarla esplicitamente la seguente proposizione:

Proposizione 9.13 *Se π è una proof-net i-contrattiva di $\overline{\mathbf{LLL}}$, allora $\pi \xrightarrow{[\mathsf{c}]} \pi'$ se e solo se $\mathsf{MCF}_i(\pi) \xrightarrow{s} \mathsf{MCF}_i(\pi')$, dove lo step [c] è applicato ad un cut tra un !-box e una contrazione a profondità i, e s, applicato al nodo corrispondente alla scatola contrattiva coinvolta in [c], è uguale a [prune] se tale scatola è massimale, altrimenti è uguale a [split].*

In corrispondenza diretta con il lemma 9.3 abbiamo il seguente lemma fondamentale:

Lemma 9.4 *Sia π una proof-net i-contrattiva di $\overline{\mathbf{LLL}}$. π non ammette applicazioni dello step* [c] *a profondità i se e solo se $\mathsf{MCF}_i(\pi)$ è vuota.*

La dinamica temporale della cut-elimination (limitata alle contrazioni) al livello i di una proof-net di $\overline{\mathbf{LLL}}$ si può dunque vedere come un processo che riduce la sotto-foresta contrattiva massimale della proof-net ad una foresta vuota; il numero di applicazioni di [prune] e [split] necessarie a "disboscare" la foresta è esattamente uguale al numero di step [c] necessari a terminare (sempre a meno dell'eliminazione dei tagli speciali) la "fase contrattiva" del passo standard a profondità i.

9.3.3 La dinamica delle foreste contrattive

In questa sezione condurremo un analisi dettagliata della complessità del sistema di riscrittura definito per le foreste contrattive. Anzitutto, diamo le seguenti definizioni preliminari:

Definizione 9.25 *Sia n un nodo di una sotto-foresta massimale Φ. Indicheremo con $d(n)$ la* profondità *in Φ di n, definita nel modo usuale: $d(n)$ è pari al numero di nodi che compongono il cammino da n alla radice Φ, inclusi n e la radice stessa.*
La profondità di Φ, che denoteremo con $\partial(\Phi)$, è il massimo tra le profondità dei suoi nodi.

Definizione 9.26 *Sia Φ una sotto-foresta massimale. Denoteremo con $\sharp_j(\Phi)$ il numero di nodi di Φ la cui profondità è j. La* dimensione *(o* size*) di Φ è invece il numero di nodi che compongono Φ, e sarà dunque la somma delle $\sharp_j$ (che sono dunque dette anche* dimensioni parziali*):*

$$\sharp(\Phi) = \sum_{j=1}^{\partial(\Phi)} \sharp_j(\Phi)$$

Definizione 9.27 *Sia Φ una sotto-foresta massimale. Un nodo b di Φ che ha più di un discendente sarà detto* ramificazione.
Denoteremo con $\mathsf{Br}_j(\Phi)$ l'insieme dei nodi ramificazione di Φ la cui profondità è j. L'insieme delle ramificazioni di Φ sarà l'unione dei Br_j:

$$\mathsf{Br}(\Phi) = \bigcup_{j=1}^{\partial(\Phi)} \mathsf{Br}_j(\Phi)$$

Definizione 9.28 (Ampiezza di una sotto-foresta massimale) *Sia Φ una sotto-foresta massimale. L'ampiezza di Φ, che denotiamo con $\alpha(\Phi)$, è definita come la somma delle profondità dei nodi ramificazione di Φ, ovvero*

$$\alpha(\Phi) = \sum_{b \in \mathsf{Br}(\Phi)} d(b) = \sum_{j=1}^{\partial(\Phi)} j \cdot |\mathsf{Br}_j(\Phi)|$$

dove $|\cdot|$ denota la cardinalità di un insieme.

Una conseguenza immediata della definizione 9.28 è che l'ampiezza di una sotto-foresta massimale è nulla se e solo se questa non contiene ramificazioni.

E' opportuno iniziare con alcune osservazioni, che seguono immediatamente dalla definizione delle regole di riscrittura. Nel seguito, sia Φ una sotto-foresta massimale, e Φ' la sotto-foresta da essa ottenuta mediante uno dei due step di riduzione:

(a) Lo step [prune] fa sempre diminuire la size di Φ di esattamente un'unità.

(b) Lo step [prune] può far diminuire l'ampiezza di Φ o, altrimenti, la può far restare uguale. Il primo caso si ha quando la foglia "potata" è discendente di una ramificazione binaria, che chiamiamo b; tale nodo in Φ' non è più una ramificazione, e dunque l'ampiezza diminuisce esattamente di $d(b)$. Viceversa, quando la foglia potata non discende da una ramificazione, l'ampiezza resta ovviamente invariata.

(c) Lo step [split] fa sempre aumentare la size di Φ. Infatti, dopo l'applicazione di [split] ad una ramificazione avente k discendenti, si ha

$$\sharp(\Phi') = \sharp(\Phi) + k - 1$$

Poiché $2 \leq k \leq \sharp(\Phi) - 1$ (una ramificazione ha almeno due discendenti, mentre la maggiorazione si ha nel caso in cui la foresta sia costituita da un solo albero, di cui la ramificazione in questione è la radice, e tutti gli altri nodi sono suoi discendenti), si ha $\sharp(\Phi) + 1 \leq \sharp(\Phi') \leq 2(\sharp(\Phi) - 1)$.

Per quanto riguarda il comportamento dell'ampiezza sotto l'esecuzione di [split], abbiamo la seguente proprietà:

Proposizione 9.14 *Se Φ e Φ' sono due sotto-foreste massimali tali che $\Phi \xrightarrow{[\mathsf{split}]} \Phi'$, allora $\alpha(\Phi') < \alpha(\Phi)$.*

Dimostrazione. Sia b la ramificazione a cui viene applicato lo step [split]. Abbiamo tre casi possibili: il genitore di b è a sua volta una ramificazione, il genitore di b non è una ramificazione, oppure b non ha genitori ed è dunque la radice di un albero.

Nel primo caso, le copie di b vengono fatte discendere direttamente dalla ramificazione genitore, la quale semplicemente aumenta il suo numero di discendenti. Le copie di b non sono tuttavia ramificazioni; abbiamo allora che $\mathsf{Br}(\Phi') = \mathsf{Br}(\Phi) \setminus \{b\}$, e dunque l'ampiezza è diminuita esattamente di $d(b)$.

Nel secondo caso, b si "trasferisce" sul nodo genitore, che diventa una ramificazione; il numero totale delle ramificazioni resta dunque uguale, ma una di esse è ora scesa di profondità, e l'ampiezza diminuirà di conseguenza di 1.

Nel terzo ed ultimo caso, poiché b era radice di un albero, $d(b) = 1$ in Φ; tuttavia, in Φ', b non c'è più, e quindi anche in questa situazione l'ampiezza è diminuita di 1. □

L'ampiezza dunque decresce sempre sotto l'applicazione degli step [split]; poiché una volta arrivati all'ampiezza nulla, non esistono più ramificazioni, essa è senz'altro un upper bound al numero di [split] che possono essere eseguiti, giacché [prune] non la può far aumentare. Allo stesso modo, la size determina il numero massimo di step [prune] consecutivi da poter effettuare. In effetti, per ottenere la foresta vuota, lo step [split] è ridondante, e la strategia che non ne fa uso si rivela essere ottimale:

Proposizione 9.15 *Sia Φ una sotto-foresta massimale. La strategia ottimale per la riduzione di Φ alla foresta vuota esegue esattamente $\sharp(\Phi)$ step* [prune]*, senza eseguire alcuno step* [split]*, e termina in un numero di passi lineare nella dimensione della foresta.*

Dimostrazione. E' chiaro che, tra i due step possibili, l'unico che serve davvero ad ottenere la foresta vuota è [prune], giacché esso è il solo a poter eliminare i nodi. Poiché ogni applicazione di [prune] ne elimina esattamente uno, e poiché non esistono situazioni in cui uno step di questo tipo non possa essere applicato (ogni albero di Φ ha necessariamente almeno una foglia, sia pure coincidente con la radice), è chiaro che dopo esattamente $\sharp(\Phi)$ applicazioni di [prune] si arriva alla foresta vuota.

L'ottimalità della strategia deriva dall'osservazione (c): l'esecuzione di uno step [split] fa aumentare almeno di 1 la dimensione della foresta a cui viene applicato; poiché tale nodo aggiuntivo dovrà essere eliminato da uno step [prune], si dovrà eseguire almeno uno step in più rispetto alla strategia che non usa mai [split]. □

La strategia ottimale appena trovata ha una chiara corrispondenza in ambito di proof-net: *per avere un'esecuzione ottima della "fase contrattiva" del passo standard si devono sempre eliminare i tagli sulle scatole contrattive massimali.*

Abbiamo così individuato il modo ottimale di ridurre una foresta; tuttavia, il nostro intento è quello di arrivare a definire un bound polinomiale per la strategia standard nel caso più generale possibile. Mostreremo allora una strategia che non utilizza mai step [prune], se non alla fine, e dimostreremo che questa è la peggiore di tutte. Anzitutto, verifichiamo la seguente proprietà:

Lemma 9.5 (Confluenza di [split]) *Sia Φ una sotto-foresta massimale. Dopo l'applicazione di al massimo $\alpha(\Phi)$ step* [split]*, senza che sia applicato alcuno step* [prune]*, la foresta si riduce ad un insieme di alberi filiformi, uno per ogni foglia di Φ, ciascuno di profondità pari alla profondità della foglia da cui proviene.*

Dimostrazione. Che $\alpha(\Phi)$ sia un upper bound al numero di step [split] da poter applicare è garantito dalla proposizione 9.14.

Dimostriamo anzitutto la confluenza. Sia dunque Φ una foresta in cui esistono almeno due ramificazioni r_1 e r_2, ed è dunque possibile applicare due step [split]. L'unico caso interessante è quando le due ramificazione in questione sono "una sopra l'altra", cioè la ramificazione r_1 discende direttamente dalla ramificazione r_2 (in tal caso, r_1 è "sopra" r_2); infatti, in tutti gli altri casi i due step sono del tutto indipendenti, e si ha chiaramente confluenza forte (cioè si può arrivare ad una stessa foresta applicando esattamente un passo ad entrambe le configurazioni generate dai sue step).

Supponiamo allora di essere in un caso simile; se [split] è applicato prima ad r_1, tale ramificazione "scenderà" su r_2, la quale avrà un numero di discendenti pari ai suoi più quelli di r_1; al contrario, se scegliamo di applicare [split] a r_2, sarà lei a scendere (o, eventualmente, a "sparire", nel caso in cui essa sia una radice), lasciando dietro di sé un certo numero di sue copie, nessuna di esse ramificazione, una delle quali sarà avrà come discendente r_1. I due alberi sono chiaramente differenti; tuttavia, se nel primo caso riduciamo r_2, e nel secondo caso riduciamo r_1 più la ramificazione che viene creata dalla sua riduzione, otteniamo lo stesso identico albero. Si ha dunque confluenza in uno step in un caso, in due step nell'altro.

Abbiamo così dimostrato la *confluenza locale* dell'applicazione degli step [split]. Poiché però, sempre per la già menzionata proposizione 9.14, non possono esistere sequenze infinite di applicazioni di tale step (abbiamo cioè la normalizzazione forte per [split][9], dalla confluenza locale possiamo dedurre automaticamente la confluenza.

Passiamo al resto dell'enunciato. Per quanto riguarda il fatto che le foreste siano filiformi, questa è un'ovvia proprietà che tutte le forme normali rispetto a [split] devono avere; se una foresta non è filiforme, contiene certamente una ramificazione, e dunque ad essa può ancora essere applicato [split]. La profondità dei "fili" si deduce dal fatto che la profondità delle foglie di una sotto-foresta massimale è invariante sotto l'applicazione di [split]. La verifica di ciò è banale, e segue direttamente dall'osservazione della rappresentazione grafica dello step.

□

Il lemma 9.5 dimostra la confluenza delle trasformazioni indotte dall'applicazione di soli step [split]: indipendentemente dall'ordine con il quale si riducono le ramificazioni, alla fine si ottiene sempre una foresta composta da alberi filiformi. Riguardo alla dimensione di tale foresta si può dimostrare il seguente lemma:

Lemma 9.6 *Sia Φ una sotto-foresta massimale, tale che $\sharp(\Phi) = s$. La sotto-foresta massimale Ψ ottenuta una volta eseguiti tutti gli step* [split] *possibili (senza aver eseguito alcuno step* [prune]*) è tale che $\sharp(\Psi) = O(s^2)$.*

Dimostrazione. In base al lemma 9.5, sappiamo che Ψ è composta da alberi filiformi in numero pari alle foglie di Φ, profondi quanto la profondità delle stesse foglie in Φ. Se denotiamo con $\mathsf{Le}(\cdot)$ l'insieme delle foglie di una sotto-foresta massimale, abbiamo che la dimensione di Ψ sarà data da

$$\sharp(\Psi) = \sum_{l \in \mathsf{Le}(\Phi)} d(l) \leq \sum_{l \in \mathsf{Le}(\Phi)} \partial(\Phi) = \partial(\Phi) \cdot |\mathsf{Le}(\Phi)| \leq s^2$$

che fornisce il bound dell'ipotesi. L'ultima maggiorazione è giustificata dal fatto che sia la profondità che il numero di foglie di una foresta sono minori o uguali alla dimensione della foresta stessa. □

Componendo i lemmi 9.5 e 9.6, sappiamo quindi che l'esecuzione, in qualsiasi ordine, di tutti i possibili step [split] (senza eseguire alcuno step [prune]) porta ad un'unica sotto-foresta massimale, la cui dimensione è quadratica nella dimensione della foresta di partenza. Tra tutte le possibili sequenze di riduzione delle ramificazioni, ne esiste almeno una che, in termini di numero di step eseguiti, è la peggiore di tutte:

[9]É comunque banale mostrare che la normalizzazione forte vale per l'intero sistema di riscrittura, considerando cioè anche [prune]: data una sotto-foresta massimale Ψ, basta considerare il fatto che, rispetto al solito ordine lessicografico, la coppia $\langle \alpha(\Psi), \sharp(\Psi) \rangle$ "decresce" sempre sotto l'applicazione di qualsiasi passo.

Proposizione 9.16 *Sia Φ una sotto-foresta massimale. La strategia che, ad ogni passo, riduce sempre la ramificazione a profondità minima (cioè quella "più vicina" alla radice), elimina tutte le ramificazioni in esattamente $\alpha(\Phi)$ step, ed è dunque la peggiore di tutte.*

Dimostrazione. Poiché l'ampiezza decresce di almeno un'unità ad ogni applicazione di [split], trovare una strategia che utilizza esattamente $\alpha(\Phi)$ step significa senza dubbio aver trovato la peggiore. Dobbiamo dunque verificare che la strategia descritta impieghi effettivamente un tempo pari all'ampiezza della sotto-foresta iniziale.

Siano Φ' e Φ'' due sotto-foreste contrattive tali che $\Phi' \stackrel{[\mathsf{split}]}{\longrightarrow} \Phi''$. L'unico caso in cui $\alpha(\Phi'') < \alpha(\Phi') - 1$ è quello in cui la ramificazione b su cui opera [split] discenda da un'altra ramificazione g; in questo caso infatti, g "assorbe" le ramificazioni di b, la quale si duplica in un certo numero di nodi che non sono a loro volta ramificazioni. Nel complesso, l'insieme delle ramificazioni di Φ'' ha dunque perso un elemento rispetto a $\mathsf{Br}(\Phi')$, e l'ampiezza sarà dunque diminuita esattamente di $d(b)$, la quale può essere in generale ben maggiore di 1. Se, al contrario, b non discende da un'altra ramificazione, il genitore di b, che non era una ramificazione in Φ', lo diventerà in Φ'', e $\mathsf{Br}(\Phi'')$ avrà lo stesso numero di elementi di $\mathsf{Br}(\Phi')$, ma uno di essi vedrà la sua profondità scendere di esattamente un'unità; altrettanto farà allora l'ampiezza. Anche nel caso in cui b sia la radice di un albero, $\mathsf{Br}(\Phi'')$ avrà un elemento in meno rispetto a $\mathsf{Br}(\Phi')$, ma tale elemento, essendo la radice di un albero, contribuiva all'ampiezza con una profondità unitaria, e dunque sarà ancora $\alpha(\Phi'') = \alpha(\Phi') - 1$.

Ora, se operiamo nel modo descritto nell'ipotesi, la ramificazione su cui operano i nostri [split], essendo supposta di profondità minima, non può avere altre ramificazioni da cui discendere; di conseguenza, ogni volta l'ampiezza decresce esattamente di un'unità, e la foresta "filiforme" non potrà che essere raggiunta in esattamente $\alpha(\Phi)$ step. $\square$

Dalla dimostrazione appena vista deduciamo che, in realtà, la strategia descritta nella proposizione 9.16 non è l'unica "pessima"; in realtà, basta scegliere ogni volta di eseguire [split] su una ramificazione che non sia discendente di un'altra ramificazione, indipendentemente dalla sua profondità. In tal modo, si è sicuri di non far mai scendere l'ampiezza di più di un'unità, e l'eliminazione delle ramificazioni continua ad essere ottenuta in esattamente $\alpha(\Phi)$ step, dove chiaramente Φ è la sotto-foresta massimale di partenza.

Possiamo allora dimostrare il seguente lemma:

Lemma 9.7 *Se Φ è una sotto-foresta contrattiva massimale, esiste una sequenza di applicazione degli step* [prune] *e* [split] *per mezzo della quale di* *ottiene la foresta vuota in $O(\alpha(\Phi) + (\sharp(\Phi))^2)$ passi, e tale sequenza è, in termini di numero di step applicati, la peggiore possibile.*

Dimostrazione. La dimostrazione è una semplice composizione delle proposizioni e dei lemmi dimostrati in precedenza. Basta infatti scegliere una qualsiasi sequenza di riduzioni "[split]-only" suggerita dalla proposizione 9.16 per ottenere la sotto-foresta massimale "filiforme" (che è unica in base al lemma 9.5) in esattamente $\alpha(\Phi)$ passi. La size di tale sotto-foresta è, come ci garantisce il lemma 9.6, dell'ordine di $(\sharp(\Phi))^2$; ma, per la proposizione 9.15, l'applicazione degli step [prune] (che sono i soli a poter essere applicati a questo punto) porta alla foresta vuota in un numero di passi lineare con la dimensione della foresta da cui si parte. In totale, si hanno dunque $O(\alpha(\Phi) + (sharp(\Phi))^2)$ passi, come voluto nella tesi. $\Box$

La riduzione di una sotto-foresta contrattiva massimale alla foresta vuota, che corrisponde all'eliminazione di tutti i tagli polinomiali da una proof-net i-contrattiva, si può dunque eseguire alla peggio in un numero di passi lineare nell'ampiezza della sotto-foresta e quadratico nella sua dimensione. Mentre quest'ultima grandezza è direttamente rapportabile ad un parametro strutturale ben preciso delle proof-net (che abbiamo chiamato sempre dimensione), l'ampiezza non è immediatamente riconducibile ad alcuna proprietà delle proof-net. Il seguente lemma offre una soluzione al problema:

Lemma 9.8 *Sia Φ una sotto-foresta contrattiva massimale, tale che $\sharp(\Phi) = s$ e $\alpha(\Phi) = a$. Allora, $a < s^2$.*

Dimostrazione. La dimostrazione si ottiene mediante una serie di semplici maggiorazioni:

$$a = \sum_{j=1}^{\partial(\Phi)} j \cdot \mathsf{Br}_j(\Phi) \leq \sum_{j=1}^{\partial(\Phi)} j \cdot \sharp_j(\Phi) <$$

$$< \partial(\Phi) \sum_{j=1}^{\partial(\Phi)} \sharp_j(\Phi) = \partial(\Phi) \cdot s \leq s^2$$

L'ultima maggiorazione tiene conto del fatto che, al massimo, $\partial(\Phi) = \sharp(\Phi)$ (è il caso di una foresta composta da un solo albero filiforme). $\Box$

Bound temporale

Possiamo ora enunciare il risultato finale sulla complessità della riduzione delle sotto-foreste contrattive massimali:

Teorema 9.5 *Se Φ è una sotto-foresta massimale, tale che $\sharp(\Phi) = s$, la riduzione alla foresta vuota impiega $O(s^2)$ passi.*

Dimostrazione. Sia $\alpha(\Phi) = a$. In base al lemma 9.7, abbiamo che la riduzione alla foresta vuota si ottiene, mediante una strategia "pessima", in $O(a+s^2)$ passi. Per il lemma 9.8, possiamo concludere che l'esecuzione non può impiegare più di $O(2s^2)$ passi, da cui la tesi. □

La riduzione di una sotto-foresta è dunque polinomiale (quadratica, per l'esattezza) nella sua dimensione. Questa è un'ottima notizia, visto che, in sostanza, la dimensione di una foresta contrattiva è proporzionale alla dimensione del livello della proof-net a cui si applicano i passi polinomiali. Prima di arrivare al bound temporale, dimostriamo il seguente lemma:

Lemma 9.9 *Sia π una proof-net i-contrattiva di $\overline{\mathbf{LLL}}$, risultante dall'applicazione della prima fase del passo standard a profondità i, e tale che $\sharp_i(\pi) = s_i$. Allora, durante tutta la seconda fase del passo standard, la somma delle arità delle contrazioni a profondità i resta sempre minore o uguale a s_i.*

Dimostrazione. Durante la seconda fase non vengono che eseguiti step [c] e step [ax] *speciali*. Questi ultimi non sono chiaramente in grado di modificare l'arità di alcuna contrazione. Per quanto riguarda i primi, osserviamo che dopo l'esecuzione di uno qualunque di essi una contrazione ha semplicemente "cambiato posto", ma non ha modificato la propria arità. Dopo lo "spostamento", potrebbero verificarsi due situazioni:

(a) La conclusione della nuova contrazione è premessa di un'altra contrazione; in tal caso, la conversione a nouvelle syntaxe effettuata dallo step successivo "collasserà" le due contrazioni in una, di arità pari alla somma delle due *meno uno*: abbiamo dunque diminuito di un'unità la somma delle arità di tutte le contrazioni a profondità i.

(b) La conclusione della nuova contrazione non è premessa di un'altra contrazione; in tal caso, non si applica alcuna conversione, e sia il numero che la somma delle arità delle contrazioni a profondità i restano invariati.

Poichè ogni step eseguito nella seconda fase, alla peggio, preserva la somma delle arità delle contrazioni, e poiché questa è senz'altro limitata dalla dimensione iniziale del livello i, otteniamo la tesi. □

Enunciamo ora (e dimostriamo) il risultato finale di questo paragrafo:

Teorema 9.6 (Bound temporale) *Sia π una proof-net di $\overline{\mathbf{LLL}}$, e sia $\sharp_i(\pi) = s_i$. Allora, l'applicazione del passo standard a profondità i elimina tutti i tagli a tale profondità (eccetto quelli contrattivi additivi e contrattivi additivi estesi) in $O(s_i s_{i+1}^2)$ step elementari di riduzione. Nel caso in cui $i = \partial(\pi)$, si effettuano semplicemente $O(s_i)$ step.*

Dimostrazione. Il passo standard comincia con l'eliminare tutti i tagli lineari; sotto l'esecuzione di questo tipo di tagli, la dimensione del livello a cui vengono eseguiti decresce sempre, e dunque s_i costituisce un upper bound al numero di step impiegati per concludere la prima fase del passo (sostanzialmente la situazione è identica a quella di **MALLq**, vedi [10]).

Nella seconda fase, ci troviamo con una proof-net π' i-contrattiva, per la quale, grazie al lemma 9.4, sappiamo di poter contare il numero di passi necessari all'eliminazione di tutti i tagli polinomiali contando il numero di passi necessari a "disboscare" $\mathsf{MCF}_i(\pi')$. Il teorema 9.5 ci dice che, se $\sharp(\mathsf{MCF}_i(\pi')) = \sigma$, tale operazione richiede al massimo $O(\sigma^2)$ passi. Ora, la dimensione della sotto-foresta contrattiva massimale del livello i di π' è al più pari al numero di !-box contenuti in tale livello di π'. Poiché un !-box deve contenere almeno un nodo, possiamo senza dubbio scrivere $\sigma \leq s_{i+1}$, dunque l'eliminazione dei tagli polinomiali richiede al più s_{i+1}^2 step. Inoltre, nel corso della seconda fase del passo standard vengono eliminati anche i cosiddetti "tagli speciali" che si possono creare dalla riduzione dei tagli sulle contrazioni. Ogni esecuzione di $[\mathsf{c}]$ può al massimo creare un numero di tagli speciali pari all'arità della contrazione ridotta; ora, poichè l'arità di una contrazione di profondità i è senz'altro limitata da s_i (lemma 9.9), il numero di tagli speciali che si possono creare (e ridurre) durante la seconda fase è senz'altro inferiore a $s_i \cdot s_{i+1}^2$. C'è poi da tener conto dell'aumento di dimensione al livello i indotto dall'eliminazione dei tagli polinomiali. Durante l'esecuzione di un taglio di questo tipo, in cui la contrazione attivata è, ad esempio, di arità k, si ha che il livello i vede aumentare la sua dimensione di $3(k-1)$ nodi. Essendo sempre, per il lemma 9.9, $k \leq s_i$, ed essendo al massimo s_{i+1}^2 il numero di step polinomiali eseguiti, la dimensione aumenterà al massimo di $3(s_i - 1)s_{i+1}^2$.

Con la fine della seconda fase, in base alla proposizione 9.4, si ottiene dunque una proof-net π'' in cui rimangono solo tagli contrattivi additivi, contrattivi additivi estesi e tagli shifting. Questi ultimi non possono essere più di quanti siano i nodi al livello i di π''; abbiamo visto che la size a profondità i di π'' è inferiore a $3(s_i-1)s_{i+1}^2 + s_i$, e dunque basterà eseguire al massimo un tale numero di step shifting per completare il passo standard.

Mettendo insieme il tutto, abbiamo:

- s_i step lineari
- s_{i+1}^2 step polinomiali e $s_i s_{i+1}^2$ step "speciali"
- $3(s_i - 1)s_{i+1}^2 + s_i$ step shifting

La somma del tutto è chiaramente dominata dal monomio $4s_i s_{i+1}^2$, e dunque la complessità temporale del passo standard è $O(s_i s_{i+1}^2)$.

E' chiaro che tutti questi discorsi si applicano solo se $i < \partial(\pi)$. Nel caso in cui il passo standard operi a profondità massima, non possono che esserci tagli lineari, e dunque la complessità diventa $O(s_i)$. $\square$

Bound spaziale

Ci occuperemo ora in sostanza di ripetere il discorso fatto per le sotto-foreste massimali con le foreste contrattive, in base alle quali troveremo un bound all'aumento della size dei livelli superiori della proof-net a cui viene applicato il passo standard. Dimostriamo allora direttamente il seguente teorema:

Teorema 9.7 (Bound spaziale) *Sia π una proof-net di $\overline{\mathbf{LLL}}$, e sia $\sharp_i(\pi) = s_i$, con $0 \leq i \leq \sharp(\pi)$. Allora, dopo l'applicazione del passo standard a profondità i ($i < \partial(\phi)$) si hanno i seguenti bound per le dimensioni parziali della proof-net risultante (che chiamiamo π'):*

$$\sharp_j(\pi') \begin{cases} = s_j & j < j \\ < 3(s_i - 1)s_{i+1}^2 + s_i & j = i \\ < 3(s_i - 1)s_{i+1}^4 s_j + s_i s_{i+1}^2 s_j & j > i \end{cases}$$

Dimostrazione. Il passo standard a profondità i non tocca in alcun modo i livelli strettamente inferiori a i, e dunque è chiaro che resterà $\sharp_j(\pi') = s_j$ per $j < i$.

Alla profondità a cui opera il passo standard, abbiamo già dimostrato nel teorema 9.6 che la dimensione resta sempre inferiore a $3(s_i - 1)s_{i+1}^2 + s_i$; rimane dunque da verificare quanto succede alle profondità maggiori.

Grazie al lemma 9.3, sappiamo che, se π'' è la proof-net i-contrattiva risultante dall'applicazione della prima fase del passo standard, la seconda fase terminerà quando tutti i nodi di $\mathsf{CF}_i(\pi'')$ saranno stati "uccisi" da step [dup]. Poiché i nodi della foresta contrattiva del livello i di una proof-net corrispondono esattamente al numero di !-box presenti a profondità i di tale proof-net, basterà calcolare la dimensione della foresta morta al termine della seconda fase per conoscere il numero di scatole presenti dopo l'attivazione di tutte le contrazioni. Ciascuno step [dup] produce, dopo la sua esecuzione, al massimo k nodi morti, dove k è l'arità della scatola contrattiva associata al nodo cui è applicato lo step. Supponendo di aver maggiorato in qualche modo tale arità, la dimensione della foresta morta è facilmente calcolabile: sappiamo che nel momento in cui la foresta contrattiva muore completamente, la sotto-foresta massimale corrispondente è divenuta vuota; poiché ogni passo [dup] eseguito sulla foresta contrattiva corrisponde ad un passo [prune] o [split] applicato alla corrispondente sotto-foresta massimale, e poiché tale foresta si svuota in un numero di passi al più quadratico nella sua dimensione (teorema 9.5), otteniamo che la dimensione finale della foresta contrattiva morta è al massimo k moltiplicato per il quadrato della sua dimensione iniziale, che maggiora il numero di !-box presenti in π'' a profondità i. In considerazione del fatto che ogni box deve contenere almeno un nodo, il numero di !-box a profondità i (e dunque la dimensione iniziale della foresta) è maggiorato da s_{i+1}; di conseguenza, il numero di scatole presenti in π' (giacché la terza fase non crea né elimina scatole) è limitato da $k \cdot s_{i+1}^2$.

Resta da stabilire quanto grande possa essere k. Sappiamo che nel corso dell'intero passo standard a profondità i la dimensione di tale livello resta sempre inferiore a $3(s_i - 1)s_{i+1}^2 + s_i$; poiché una contrazione non può contrarre più nodi di quanti ce ne siano alla profondità cui appartiene, in ogni dato istante k è sempre limitato dal numero sopra.

Ora, ciascuna scatola a profondità $j > i$ contiene $O(s_j)$ nodi; infatti il contenuto delle scatole o non è stato affatto modificato, oppure è stato modificato in modo trascurabile, in particolare dalla comparsa di alcuni cut a profondità $i + 1$ introdotti dagli step shifting. Le scatole che sono state interessate dalle duplicazioni dei passi polinomiali, che erano inizialmente al massimo s_{i+1}, ora sono divenute al più ks_{i+1}^2; nel caso peggiore, quello cioè in cui tutte le scatole a profondità maggiori di $i + 1$ siano nidificate all'interno delle scatole duplicate, la size del livello $j > i$ è diventata $ks_{i+1}^2 s_j$. Da ciò, segue la tesi. $\Box$

9.3.4 La complessità della strategia standard

Ricapitoliamo i risultati ottenuti nei paragrafi precedenti. Sia π una proof-net di $\overline{\mathbf{LLL}}$, tale che $\sharp_i(\pi) = s_i$, la quale si riduce per mezzo di un passo standard a profondità i nella proof-net π':

- L'esecuzione del passo standard richiede $O(s_i s_{i+1}^2)$ step elementari di riduzione; se $i = \partial(\pi)$, allora gli step richiesti sono semplicemente $O(s_i)$.
- Le dimensioni parziali a profondità strettamente minore di i in π' restano inalterate: $\sharp_j(\pi') = s_j$ per $j < i$.
- A profondità i, si ha $\sharp_i(\pi') = O(s_i s_{i+1}^2)$.
- A profondità maggiore di i, si ha $\sharp_j(\pi') = O(s_i s_{i+1}^4 s_j)$, $j > i$.

In considerazione di questi risultati, dimostriamo un piccolo lemma che ci tornerà parecchio utile per i calcoli che ci accingiamo a fare:

Lemma 9.10 *Sia π una proof-net di $\overline{\mathbf{LLL}}$, tale che $\partial(\pi) = d$ e, per $0 \leq i \leq d$, $\sharp_i(\pi) = s_i$, e sia $\pi_1, \ldots, \pi_d$ una successione di proof-net tale che*

$$\pi \twoheadrightarrow \pi_1 \twoheadrightarrow \ldots \twoheadrightarrow \pi_d$$

dove ogni freccia $\twoheadrightarrow$ rappresenta l'applicazione di un passo standard a profondità via via superiore (a π il passo è applicato a profondità 0, mentre, in generale, a π_j il passo è applicato a profondità j — in sostanza, stiamo normalizzando π mediante la strategia standard). Allora, si avrà

$$\sharp_i(\pi_i) = O(s_0^{6^{i-1}} s_i^5 \prod_{k=1}^{i-1} s_k^{25 \cdot 6^{i-k-1}})$$

$$\sharp_{i+1}(\pi_i) = O(s_0^{6^{i-1}} s_i^4 s_{i+1} \prod_{k=1}^{i-1} s_k^{25 \cdot 6^{i-k-1}})$$

dove, convenzionalmente, una produttoria che va da 1 a 0 vale 1.

Dimostrazione. Sia $\pi_0 = \pi$; poniamo

$$\sigma(i,j) = \sharp_i(\pi_j)$$

Naturalmente, per ipotesi abbiamo $\sigma(i,0) = s_i$. Dal teorema 9.7, abbiamo invece che, per $i \geq j > 0$, la successione σ deve soddisfare la seguente equazione alle differenze finite (a meno di costanti e termini di ordine di grandezza trascurabili):

$$\sigma(i,j) = \sigma(j-1, j-1)(\sigma(j, j-1))^4 \sigma(i, j-1)$$

Dimostreremo ora che la soluzione di tale equazione è

$$\sigma(i,j) = s_0^{6^{j-1}} s_j^4 s_i \prod_{k=1}^{j-1} s_k^{25 \cdot 6^{j-k-1}} \qquad j > 0, i \geq j$$

Effettueremo la dimostrazione per induzione sull'ordine lessicografico delle coppie (i,j). Poiché j è strettamente positivo, e poiché i è maggiore o uguale a j, la coppia più piccola è $(1,1)$. La verifica del passo base dell'induzione è dunque la seguente:

$$\sigma(1,1) = \sigma(0,0)(\sigma(1,0))^4 \sigma(1,0) = s_0 s_1^5$$

che corrisponde al valore trovato dalla nostra soluzione.

Supponiamo ora che la soluzione funzioni per tutte le coppie (i,j) strettamente lessicograficamente inferiori a (m,n). Applicando l'ipotesi d'induzione (indicata con il simbolo $\stackrel{ind}{=}$), otteniamo il seguente risultato:

$$\begin{aligned} \sigma(m,n) &= \sigma(n-1, n-1)(\sigma(n, n-1))^4 \sigma(m, n-1) \stackrel{ind}{=} \\ &\stackrel{ind}{=} s_0^{6^{n-2}} s_{n-1}^4 s_{n-1} \prod_{k=1}^{n-2} s_k^{25 \cdot 6^{n-k-2}} \cdot \left(s_0^{6^{n-2}} s_{n-1}^4 s_n \prod_{k=1}^{n-2} s_k^{25 \cdot 6^{n-k-2}} \right)^4 \cdot \end{aligned}$$

$$
\begin{aligned}
& \cdot s_0^{6^{n-2}} s_{n-1}^4 s_m \prod_{k=1}^{n-2} s_k^{25\cdot 6^{n-k-2}} = \\
= \quad & s_0^{6\cdot 6^{n-2}} s_n^4 s_m s_{n-1}^{25} \prod_{k=1}^{n-2} s_k^{25\cdot 6\cdot 6^{n-k-2}} = \\
= \quad & s_0^{6^{n-1}} s_n^4 s_m \prod_{k=1}^{n-1} s_k^{25\cdot 6^{n-k-1}}
\end{aligned}
$$

che è esattamente la nostra soluzione calcolata in (m, n). La tesi è un caso particolare della soluzione, che si ottiene "diagonalizzandola" e "sovradiagonalizzandola": $\sharp_i(\pi_i) = O(\sigma(i,i))$ e $\sharp_{i+1}(\pi_i) = O(\sigma(i+1,i))$. $\square$

Possiamo a questo punto determinare la complessità totale della strategia standard. Mettendo assieme il teorema 9.6 e il lemma 9.10, otteniamo infatti il risultato annunciato:

Teorema 9.8 (PTIME-correttezza di $\overline{\mathbf{LLL}}$) *Sia π una proof-net di $\overline{\mathbf{LLL}}$, tale che $\sharp(\pi) = s$ e $\partial(\pi) = d$. Allora, π è riducibile in una proof-net π' lazy-cut-free in $O(s^{7^d})$ passi.*

Dimostrazione. Sia, per $0 \le i \le d$, $\sharp_i(\pi) = s_i$. Sappiamo, grazie al teorema 9.6, che ogni passo standard a profondità i impiega un numero di step elementari di riduzione che è, grossomodo, pari alla dimensione del livello i moltiplicata per il quadrato della dimensione del livello $i+1$ (eccetto a profondità massima, dove gli step sono lineari nella dimensione). Le dimensioni di tali livelli ad ogni dato passo della strategia standard ci vengono fornite dal lemma 9.10: al primo passo (profondità 0), abbiamo per ipotesi dimensioni s_0 e s_1; al generico passo che opera a profondità $i \ge 1$, abbiamo circa $\sigma(i,i)$ e $\sigma(i+1,i)$, dove σ è la funzione definita nella dimostrazione del lemma 9.10. Se chiamiamo $\eta(i)$ il numero di step elementari di riduzione eseguiti a profondità i, abbiamo dunque (modulo la notazione "O grande"):

$$
\eta(i) = \begin{cases}
s_0 s_1^2 & i = 0 \\[2ex]
\sigma(i,i)(\sigma(i+1,i))^2 = s_0^{3\cdot 6^{i-1}} s_i^{13} s_{i+1}^2 \prod_{k=1}^{i-1} s_k^{75\cdot 6^{i-k-1}} & 1 \le i \le d-1 \\[2ex]
\sigma(d,d) = s_0^{6^{i-1}} s_d^5 \prod_{k=1}^{d-1} s_k^{25\cdot 6^{d-k-1}} & i = d
\end{cases}
$$

Sfruttando il fatto che le dimensioni parziali sono senz'altro minori o uguali alla dimensione totale, per $1 \le i \le d-1$ si può fare la seguente maggiorazione:

$$
\eta(i) \quad = \quad s_0^{3\cdot 6^{i-1}} s_i^{13} s_{i+1}^2 \prod_{k=1}^{i-1} s_k^{75\cdot 6^{i-k-1}} \le
$$

$$\begin{aligned} &\leq \quad s^{15+3\cdot 6^{i-1}} \prod_{k=1}^{i-1} s^{75\cdot 6^{i-k-1}} = \\ &= \quad \prod_{k=1}^{i-1} s^{15+(3+75\cdot 6^{-k})6^{i-1}} \end{aligned}$$

da cui, essendo entrambi i membri della disuguaglianza maggiori di 1,

$$\begin{aligned} \log(\eta(i)) \quad &\leq \quad \log\left(\prod_{k=1}^{i-1} s^{15+(3+75\cdot 6^{-k})6^{i-1}}\right) \\ &= \quad \sum_{k=1}^{d-1} \log\left(s^{15+(3+75\cdot 6^{-k})6^{i-1}}\right) \\ &= \quad \left(\left(15+\frac{1}{2}6^i\right)(i-1)+\frac{25}{2}6^i\sum_{k=1}^{d-1}\left(\frac{1}{6}\right)^k\right)\log s = \\ &= \quad \left(15(i-1)+\frac{1}{2}6^i(i-1)+\frac{25}{2}6^i\left(\frac{6}{5}\cdot(1-6^{-1})-1\right)\right)\log s = \\ &= \quad \left(\frac{1}{2}(i+4)6^i+15(i+2)\right)\log s \end{aligned}$$

e dunque

$$\eta(i) \leq s^{\frac{1}{2}(i+4)6^i+15(i+2)} \qquad 1 \leq i \leq d-1$$

Mediante un calcolo analogo si ottiene, per le profondità mancanti, $\eta(0) \leq s^3$ e $\eta(d) \leq s^{(6d-151)6^{d-2}+5(d-2)}$. In generale, dunque, possiamo scrivere

$$\eta(i) \leq s^{c_0 i 6^i + c_1 6^i + c_2 i + c_3}$$

per qualche opportuna costante c_0, c_1, c_2, c_3. Abbiamo allora, per ogni i non negativo minore di d,

$$\eta(i) = O(s^{i6^i}) = O(s^{7^i})$$

A questo punto, calcolare la complessità della strategia standard è cosa semplice; il numero di step elementari di riduzione totali sarà infatti la somma di quelli impiegati da ciascun passo. Se indichiamo con τ tale numero, abbiamo (sempre tralasciando le costanti e i termini trascurabili)

$$\tau = \sum_{i=0}^{d} \eta(i) \leq \sum_{i=0}^{d} s^{7^i} \leq \sum_{i=0}^{7^d} s^i = \frac{s^{7^d+1}-1}{s-1} = O(s^{7^d})$$

che dimostra la tesi. $\square$

Il numero di step elementari di riduzione necessari a raggiungere la forma normale di una qualsiasi proof-net di $\overline{\mathbf{LLL}}$ è dunque polinomiale nella size della proof-net; il grado del polinomio è un esponenziale che dipende dalla profondità massima della stessa proof-net. Come vedremo nella prossima sezione, la profondità delle derivazioni di $\overline{\mathbf{LLL}}$ sarà un parametro fissato a priori per tutti i termini che rappresentano costanti di un certo tipo e funzioni da un tipo ad un altro tipo. Di conseguenza, il fatto che il bound sia esponenziale nella profondità non comporta alcun problema: la vera misura della dimensione dell'input di una funzione è la size della proof-net che rappresenta tale input.

Inoltre, è opportuno ricordare che le maggiorazioni eseguite per trovare il bound sono estremamente brutali, il ché giustifica un polinomio dal grado tanto mostruoso. In realtà, in $\overline{\mathbf{LLL}}$ è possibile normalizzare le proof-net in tempi molto più ragionevoli. Ad esempio, sappiamo che per l'eliminazione dei tagli sulle contrazioni esiste una strategia lineare, la quale può senz'altro aiutare a costruire bound più piccoli.

C'è da sottolineare infine una questione non proprio marginale: il teorema appena dimostrato in realtà garantisce la cosiddetta *poly-step*-correttezza del nostro sistema; il bound polinomiale infatti è stato dato sul numero di passi elementari di riduzione, come definiti nella sezione 9.2. Per avere l'inclusione in **PTIME**, in realtà dovremmo far vedere che la normalizzazione delle proof-net di $\overline{\mathbf{LLL}}$ è ottenibile in tempo polinomiale mediante l'esecuzione della strategia standard su una qualche rappresentazione delle proof-net stesse all'interno di una macchina di Turing deterministica. Per far ciò, è sufficiente garantire che ogni passo elementare di riduzione è simulabile su una macchina di Turing in tempo polinomiale. E' chiaro dunque che la questione non è complicatissima: intuitivamente, la scansione di un grafo alla ricerca di un certo nodo (un *cut* ad esempio), qualunque sia la rappresentazione scelta, non può richiedere un tempo più che lineare nel numero di nodi; analogamente, la riscrittura del grafo (operazione che serve a simulare l'esecuzione della regola di eliminazione) non può richiedere un tempo più che polinomiale. Ecco perché, in sostanza, *poly-step*-correttezza e **PTIME**-correttezza possono essere confuse senza troppi problemi.

9.4 PTIME-completezza

Passiamo ora a mostrare la completezza di $\overline{\mathbf{LLL}}$, vale a dire che l'insieme delle funzioni rappresentabili nel nostro sistema contiene la classe **PTIME**.

Dimostreremo la completezza di $\overline{\mathbf{LLL}}$ per gradi; dapprima faremo vedere che in essa si può fornire una rappresentazione dei numeri naturali e delle operazioni fondamentali su di essi e, in particolare, che si può rappresentare qualsiasi polinomio in una variabile per mezzo di una derivazione di $\overline{\mathbf{LLL}}$.

In seguito introdurremo altri tipi di dati fondamentali, quali i caratteri di un alfabeto finito e le stringhe su di esso. Infine mostreremo come sia possibile, data una macchina di Turing deterministica e il polinomio che ne limita la complessità in funzione dell'input, costruire una derivazione di $\overline{\mathbf{LLL}}$ che simuli l'esecuzione della macchina stessa, fornendo in uscita il risultato del calcolo. Ciò completerà il lavoro, dimostrando che $\overline{\mathbf{LLL}}$ corrisponde esattamente alla classe delle funzioni **PTIME**.

La definizione di $\overline{\mathbf{LLL}}$ e tutti i risultati mostrati finora sono stati svolti nell'ambito delle proof-net. Nel seguito si utilizzerà invece il calcolo dei sequenti di $\mathbf{LL}_\S$ (quello di $\overline{\mathbf{LLL}}$ è infatti identico), nella sua versione intuizionista. La scelta è motivata fondamentalmente da due ragioni: la prima è che è più semplice associare alle derivazioni intuizioniste un termine del λ-calcolo tipato, mediante il quale diventa molto più leggibile la funzione rappresentata; la seconda è che scrivere proof-net in LaTeX con gli strumenti attualmente a disposizione è un lavoro che si preferisce fare il meno possibile...

Le regole che utilizzeremo sono in realtà un sottoinsieme di tutte le regole di $\mathbf{LL}_\S$; in particolare, mancano le regole per le costanti logiche (di scarso interesse computazionale) e quelle per il connettivo additivo $\oplus$, del quale non si ha bisogno per dimostrare la completezza:

- **Regole dell'identità**

$$\frac{}{A \vdash A}\ \mathsf{ax} \qquad\qquad \frac{\Gamma \vdash A \qquad A, \Delta \vdash C}{\Gamma, \Delta \vdash C}\ \mathsf{cut}$$

- **Regole strutturali**

$$\frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C}\ \mathsf{X}$$

- **Regole logiche moltiplicative**

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\ \otimes\mathsf{L} \qquad\qquad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\ \otimes\mathsf{R}$$

$$\frac{\Gamma \vdash A \qquad B, \Delta \vdash C}{\Gamma, \Delta, A \multimap B \vdash C}\ \multimap\mathsf{L} \qquad\qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \multimap B}\ \multimap\mathsf{R}$$

- **Regole logiche additive**

$$\frac{\Gamma, A \vdash C}{\Gamma, A \,\&\, B \vdash C}\ \&\,\mathsf{L1} \qquad\qquad \frac{\Gamma, B \vdash C}{\Gamma, A \,\&\, B \vdash C}\ \&\,\mathsf{L2} \qquad\qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\ \&\,\mathsf{R}$$

▶ **Regole logiche esponenziali**

$$\frac{\Gamma, A \vdash C}{\Gamma, !A \vdash C}\,?\mathsf{D} \qquad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A}\,!\mathsf{P}$$

$$\frac{!\Gamma, \Delta \vdash A}{!\Gamma, \S\Delta \vdash \S A}\,\S$$

$$\frac{\Gamma, !A, !A \vdash C}{\Gamma, !A \vdash C}\,?\mathsf{C} \qquad \frac{\Gamma \vdash C}{\Gamma, !A \vdash C}\,?\mathsf{W}$$

▶ **Regole logiche per i quantificatori**

$$\frac{\Gamma, A[B/X] \vdash C}{\Gamma, \forall X.A \vdash C}\,\forall\mathsf{L} \qquad \frac{\Gamma \vdash A[X]}{\Gamma \vdash \forall X.A}\,\forall\mathsf{R} \quad X \text{ non libera in } \Gamma$$

Naturalmente, a ciascuna derivazione intuizionista costruita con queste regole può essere associata una proof-net; le conclusioni della proof-net saranno la formula nella parte destra del sequente e i duali delle formule nella parte sinistra. Ad esempio, ad una derivazione del sequente $A,\ !B \vdash C$ corrisponderà una proof-net con conclusioni $A^{\perp}$, $B^{\perp}$ e C. Le condizioni di appartenenza a $\overline{\mathbf{LLL}}$ sono traducibili in modo ovvio dalle proof-net al calcolo dei sequenti; in particolare, sarà immediato verificare che tutte le derivazioni introdotte nel seguito corrispondono a proof-net che soddisfano le condizioni della definizione 9.5.

Non introdurremo esplicitamente la decorazione del calcolo dei sequenti che fa corrispondere ad ogni derivazione un termine del λ-calcolo tipato, essendo praticamente la versione lineare di quella presentata in ??. Infatti, applicando la cosiddetta "traduzione *forgetful*" alle formule (e dunque ai sequenti) di $\overline{\mathbf{LLL}}$, le derivazioni che presenteremo si possono tradurre senza problemi in termini del Sistema $\mathbb{F}$ con prodotti. Tale traduzione consiste semplicemente nel sostituire tutte le implicazioni lineari $\multimap$ con implicazioni intuizioniste $\Rightarrow$, le congiunzioni $\otimes$ e $\&$ con semplici $\wedge$ e nel cancellare completamente tutti i connettivi esponenziali. Quantificatori e variabili restano invariati. Grazie alla "traduzione *forgetful*" è semplice associare i λ-termini alle derivazioni; in particolare, noi presenteremo le versioni *untyped* dei termini, in modo che la notazione sia il più leggera possibile.

L'unica particolarità dell'assegnazione dei termini alle derivazioni risiede nella presenza della congiunzione moltiplicativa $\otimes$, per la quale viene esplicitamente introdotto un corrispettivo nella sintassi del λ-calcolo. La gestione dei tensori non è del tutto banale; tuttavia noi ci rifaremo alla notazione introdotta da Asperti e Roversi per **ILAL**, ampiamente ben formalizzata e utilizzata, in una qualche variante, praticamente in tutti i lavori sulla logica affine. Per i dettagli sulla definizione della congiunzione moltiplicativa nel λ-calcolo, di veda [2].

Per brevità e semplicità di notazione, adotteremo le seguenti convenzioni:

- Se F è una formula, n un intero non negativo, $F^{(n)} := \underbrace{F, \ldots, F}_{n}$.
- Se F è una formula, $n \geq 2$, allora $F^{\,\&\, n} := (\ldots \underbrace{(F \;\&\; F) \;\&\; \ldots \;\&\; F)}_{n}$.
- La formula $A \multimap (B \multimap C)$ viene abbreviata con $A \multimap B \multimap C$ (l'implicazione è associativa a destra).
- Se $\diamond \in \{!, \S\}$, la formula $\underbrace{\diamond \ldots \diamond}_{n} F$ sarà abbreviata con $\diamond^n F$.
- Una linea di derivazione più spessa indica l'applicazione di più di una regola del tipo indicato; inoltre, la regola di *exchange* è ovviamente lasciata implicita.

9.4.1 Rappresentazione degli interi

Il tipo degli interi in notazione unaria, i cui oggetti corrispondono nel λ-calcolo ai ben noti interi di Church, è rappresentato in $\overline{\mathbf{LLL}}$ dalla medesima formula utilizzata in **LLL** e **LAL**. Posto

$$\mathsf{int}_F := !(F \multimap F) \multimap \S(F \multimap F)$$

dove F è una formula qualsiasi, definiamo il tipo degli interi come

$$\mathsf{int} := \forall X.\mathsf{int}_X$$

Possiamo a questo punto trovare le derivazioni che corrispondono agli interi in notazione unaria. La seguente, che chiamiamo ω_0, rappresenta l'intero 0:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X \vdash X}\,\mathsf{ax}}{\vdash X \multimap X}\,\multimap\mathsf{R}}{\vdash \S(X \multimap X)}\,\S}{!(X \multimap X) \vdash \S(X \multimap X)}\,?\mathsf{W}}{\vdash \mathsf{int}_X}\,\multimap\mathsf{R}}{\vdash \mathsf{int}}\,\forall\mathsf{R}$$

Il λ-termine ad essa corrispondente è $\lambda sz.z$, che è appunto l'intero di Church $\bar{0}$.

Se n è un intero positivo, la seguente derivazione ω_n rappresenta invece il numero n stesso:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\RightLabel{$\multimap$L}
\BinaryInfC{$X, X \multimap X \vdash X$}
\RightLabel{$\multimap$L}
\BinaryInfC{$X, X \multimap X, X \multimap X \vdash X$}
\noLine
\UnaryInfC{$\vdots$}
\noLine
\UnaryInfC{$X, (X \multimap X)^{(n-1)} \vdash X$}
\RightLabel{$\multimap$L}
\BinaryInfC{$X, (X \multimap X)^{(n)} \vdash X$}
\RightLabel{$\multimap$R}
\UnaryInfC{$(X \multimap X)^{(n)} \vdash X \multimap X$}
\doubleLine
\RightLabel{?D}
\UnaryInfC{$(!(X \multimap X))^{(n)} \vdash X \multimap X$}
\RightLabel{$\S$}
\UnaryInfC{$(!(X \multimap X))^{(n)} \vdash \S(X \multimap X)$}
\doubleLine
\RightLabel{?C}
\UnaryInfC{$!(X \multimap X) \vdash \S(X \multimap X)$}
\RightLabel{$\multimap$R}
\UnaryInfC{$\vdash \mathsf{int}_X$}
\RightLabel{$\forall$R}
\UnaryInfC{$\vdash \mathsf{int}$}
\end{prooftree}
$$

Il λ-termine corrispondente a ω_n è $\lambda sz.s(\ldots sz \ldots)$, con n occorrenze della variabile s, che è proprio l'intero di Church $\bar{n}$.

Addizione e successore

Definiamo ora l'addizione sugli interi unari, una delle operazioni fondamentali per poter arrivare alla rappresentazione di generici polinomi in una variabile. La seguente derivazione, che chiamiamo $+$, prende in input due interi ne restituisce la somma:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$!(X \multimap X) \vdash !(X \multimap X)$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$!(X \multimap X) \vdash !(X \multimap X)$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\RightLabel{$\multimap$L}
\BinaryInfC{$X, X \multimap X \vdash X$}
\RightLabel{$\multimap$L}
\BinaryInfC{$X, X \multimap X, X \multimap X \vdash X$}
\RightLabel{$\multimap$R}
\UnaryInfC{$X \multimap X, X \multimap X \vdash X \multimap X$}
\RightLabel{$\S$}
\UnaryInfC{$\S(X \multimap X), \S(X \multimap X) \vdash \S(X \multimap X)$}
\RightLabel{$\multimap$L}
\BinaryInfC{$\S(X \multimap X), !(X \multimap X), \mathsf{int}_X \vdash \S(X \multimap X)$}
\RightLabel{$\multimap$L}
\BinaryInfC{$!(X \multimap X), !(X \multimap X), \mathsf{int}_X, \mathsf{int}_X \vdash \S(X \multimap X)$}
\doubleLine
\RightLabel{$\forall$L}
\UnaryInfC{$!(X \multimap X), !(X \multimap X), \mathsf{int}, \mathsf{int} \vdash \S(X \multimap X)$}
\RightLabel{?C}
\UnaryInfC{$!(X \multimap X), \mathsf{int}, \mathsf{int} \vdash \S(X \multimap X)$}
\RightLabel{$\multimap$R}
\UnaryInfC{$\mathsf{int}, \mathsf{int} \vdash \mathsf{int}_X$}
\RightLabel{$\forall$R}
\UnaryInfC{$\mathsf{int}, \mathsf{int} \vdash \mathsf{int}$}
\end{prooftree}
$$

Il λ-termine corrispondente a $+$ è $\lambda sz.ms(nsz)$, dove m e n sono le due variabili libere associate alle due istanze della formula int a sinistra della conclusione della derivazione. E' chiaro che $+$ rappresenta la consueta addizione per gli interi di Church; se tagliata con ω_1 fornisce, a seconda della formula scelta per il taglio, una derivazione corrispondente al termine $\lambda sz.s(nsz)$ o al termine $\lambda sz.ms(sz)$, che sono sempre le due rappresentazioni standard per il successore sugli interi di Church.

Schemi di iterazione

In $\overline{\mathbf{LLL}}$ è possibile derivare due schemi di iterazione, del tutto analoghi a quelli introdotti in [8]. Il primo, che chiamiamo $\mathsf{It}(\sigma^-, \beta)$, prende una derivazione σ^- di tipo $C, A \vdash A$ (lo "step" dell'iterazione, dove C può anche non esserci), una derivazione β di tipo $\Delta \vdash A$ (la "base" dell'iterazione) e fornisce una derivazione di tipo $!C, !\Delta_1, \S\Delta_2, \mathsf{int} \vdash \S A$ (dove $\Delta_1 \cup \Delta_2 = \Delta$) che è in grado di prendere un intero in input e di iterare la funzione rappresentata da σ^- un numero arbitrario di volte, a partire dalla base β. La derivazione per lo schema $\mathsf{It}(\sigma^-, \beta)$ è la seguente:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\vdots\,\sigma^-}{C, A \vdash A}}{C \vdash A \multimap A}\multimap\mathsf{R}}{!C \vdash A \multimap A}?\mathsf{D}}{!C \vdash !(A \multimap A)}!\mathsf{P}
\qquad
\dfrac{\dfrac{\dfrac{\dfrac{\vdots\,\beta}{\Delta \vdash A} \quad \dfrac{}{A \vdash A}\mathsf{ax}}{\Delta, A \multimap A \vdash A}\multimap\mathsf{L}}{!\Delta_1, \Delta_2, A \multimap A \vdash A}?\mathsf{D}}{!\Delta_1, \S\Delta_2, \S(A \multimap A) \vdash \S A}\S
$$
$$
\dfrac{\dfrac{\cdots}{!C, !\Delta_1, \S\Delta_2, \mathsf{int}_A \vdash \S A}\multimap\mathsf{L}}{!C, !\Delta_1, \S\Delta_2, \mathsf{int} \vdash \S A}\forall\mathsf{L}
$$

Due osservazioni importanti. Anzitutto, è chiaro che, se *step*$[x]$ (x è la variabile libera di tipo A) e *base* sono i λ-termini associati rispettivamente a σ^- e β, a $\mathsf{It}(\sigma^-, \beta)$ viene associato il termine $n(\lambda x.step)base$ (dove n è la variabile libera di tipo int), che altri non è che il solito schema di iterazione sugli interi di Church. In secondo luogo, è interessante notare che, tuttavia, tale schema di iterazione non può essere applicato ad un termine *step* qualunque; la derivazione σ^- infatti deve essere *flat* e, per di più, deve avere al massimo un'altra variabile libera, oltre a quella utilizzata dall'iterazione stessa. Questa limitazione alle funzioni iterabili è presente in tutti i sistemi logici a complessità limitata, e ha un ruolo fondamentale nel controllo dell'espressività di tali sistemi. In particolare, sembra che in qualche modo la *flatness* giochi un ruolo simile a quello delle variabili *safe* nelle caratterizzazioni ricorsive di **PTIME**, come quella di Bellantoni e Cook [4]. Tuttavia, la similitudine è solo apparente, giacché al momento attuale non si è ancora trovato il modo di rappresentare la *safe recursion* nelle logiche *light*, e recentemente sono anche emersi dubbi sul fatto che una tale rappresentazione possa essere mai trovata (si vedano in particolare le considerazioni conclusive di [24]).

Il secondo schema di iterazione prende invece due derivazioni σ ed ε, di tipi rispettivamente $!C \vdash !(A \multimap A)$ ($!C$ può anche non esserci) e $\Delta, \S(A \multimap A) \vdash B$, e itera la funzione *step* rappresentata da σ *all'interno* del termine *exit* rappresentato da ε. La derivazione che rappresenta lo schema, che chiamiamo $\mathsf{It}'(\sigma, \varepsilon)$, è la seguente:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\sigma\\ !C \vdash !(A \multimap A)\end{array} \qquad \begin{array}{c}\vdots\,\varepsilon\\ \S(A \multimap A), \Delta \vdash B\end{array}}{!C, \Delta, \mathsf{int}_A \vdash B}\multimap\mathsf{L}}{!C, \Delta, \mathsf{int} \vdash B}\forall\mathsf{L}$$

Il fatto che l'iterazione avvenga *internamente* al termine associato a ε è chiaro se consideriamo il λ-termine corrispondente a $\mathsf{It}'(\sigma, \varepsilon)$. Sempre chiamando *step* ed *exit* i λ-termini corrispondenti a σ ed ε, osserviamo che *exit* deve contenere necessariamente una variabile libera di tipo $\S(A \multimap A)$, e può dunque essere scritto come $exit[x]$; il λ-termine associato al nostro secondo schema di iterazione è dunque $exit[n\ step/x]$, dove n è la variabile libera di tipo int.

Moltiplicazione

Un'applicazione immediata del primo schema di iterazione introdotto sopra è la moltiplicazione, che può essere definita iterando la somma con base nulla. La derivazione $+$ che rappresenta l'addizione è infatti *flat*, e dunque si può applicare $\mathsf{It}(+, \omega_0)$ ottenendo una derivazione di tipo $!\mathsf{int}, \mathsf{int} \vdash \S\mathsf{int}$, che prende in input due interi e restituisce il loro prodotto. Osserviamo che la derivazione corrispondente alla moltiplicazione non è *flat* e non può dunque essere iterata; ciò è quantomeno ragionevole, visto che l'iterazione del prodotto porterebbe alla definizione della funzione esponenziale, che non è decisamente tra le funzioni **PTIME**!

In caso si voglia controllare che la funzione definita è effettivamente la moltiplicazione, basti ricordare il λ-termine generico associato allo schema It, che in questo caso particolare è $n(\lambda x.add[m])\bar{0}$, dove $add[x, m]$ è il termine corrispondente alla derivazione $+$ e m ed n sono le due variabili libere rispettivamente di tipo $!\mathsf{int}$ e int.

Coercizioni

Un'altra applicazione dello schema di iterazione tradizionale è la definizione di funzioni particolari, dette coercizioni (*coercions*). Tali funzioni servono a modificare il tipo dell'input di una funzione senza modificare la funzione stessa. Si potrà infatti obiettare che la moltiplicazione definita poc'anzi non rappresenta una funzione da interi a interi secondo la definizione da noi adottata. Ciò che disturba è il tipo $!\mathsf{int}$ tra gli input, che ad essere rigorosi dovrebbe essere un semplice int. Le coercizioni servono proprio a questo: rimuovono eventuali esponenziali dai tipi in input senza però modificare in alcun modo il contenuto computazionale della funzione a cui vengono applicate.

Per definire una generica coercizione, consideriamo anzitutto la derivazione S del successore, che abbiamo visto essere ottenibile mediante *cut* dalla derivazione $+$. Il tipo di tale derivazione è $\mathsf{int} \vdash \mathsf{int}$ che, effettuando un numero arbitrario (anche nullo) di dereliction/promotion alternate e successivamente un altro numero qualsiasi (di nuovo anche nullo) di regole per l'introduzione del paragrafo, può essere trasformato in $\S^p!^q\mathsf{int} \vdash \S^p!^q\mathsf{int}$. Analogamente, a partire da ω_0 si può ottenere una derivazione di tipo $\vdash \S^p!^q\mathsf{int}$. Applicando lo schema di iterazione $\mathsf{It}(\sigma^-, \beta)$ (prendendo come σ^- la prima e come β la seconda) otteniamo una funzione di tipo $\mathsf{int} \vdash \S^{p+1}!^q\mathsf{int}$, che si comporta come l'identità sugli interi, ma cambia il tipo dell'output. Per controllare, il λ-termine corrispondente alla generica coercizione è $n(\lambda m.succ)\bar{0}$ che, applicando il successore n volte a partire da zero, restituisce chiaramente il termine $\bar{n}$.

Utilizzando le coercizioni è possibile dare un tipo più "in stile" alla moltiplicazione: da $!\mathsf{int}, \mathsf{int} \vdash \S\mathsf{int}$ otteniamo $\S!\mathsf{int}, \S\mathsf{int} \vdash \S^2\mathsf{int}$ che, per mezzo di due *cut* con le coercizioni adatte, diventa $\mathsf{int}, \mathsf{int} \vdash \S^2\mathsf{int}$.

Predecessore e sottrazione

Il modo più naturale di tipare la funzione predecessore in $\overline{\mathbf{LLL}}$ è l'utilizzo degli additivi. L'idea è la seguente: si costruisce per mezzo di una coppia additiva una sorta di contatore a due registri, inizializzato con valori entrambe nulli. Successivamente, ad ogni passo si copia il contenuto del secondo registro nel primo, e poi si incrementa il secondo registro. In questo modo, il primo registro risulta "sfalsato" di esattamente una unità rispetto al secondo; dopo n iterazioni, il secondo registro conterrà proprio n, mentre il primo conterrà $n-1$. Se n era l'intero di cui si voleva calcolare il predecessore, per ottenere il risultato desiderato basterà a questo punto estrarre il contenuto del primo registro.

Per l'implementazione dell'algoritmo descritto risulta utile il secondo schema di iterazione definito precedentemente. Prendiamo come σ la seguente derivazione:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\RightLabel{$\mathsf{?W}$}
\UnaryInfC{$!(X \multimap X), X \vdash X$}
\RightLabel{$\& \, \mathsf{L2}$}
\UnaryInfC{$!(X \multimap X), X \,\&\, X \vdash X$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\RightLabel{$\multimap\mathsf{L}$}
\BinaryInfC{$X \multimap X, X \vdash X$}
\RightLabel{$\mathsf{?D}$}
\UnaryInfC{$!(X \multimap X), X \vdash X$}
\RightLabel{$\& \, \mathsf{L2}$}
\UnaryInfC{$!(X \multimap X), X \,\&\, X \vdash X$}
\RightLabel{$\& \, \mathsf{R}$}
\BinaryInfC{$!(X \multimap X), X \,\&\, X \vdash X \,\&\, X$}
\RightLabel{$\multimap\mathsf{R}$}
\UnaryInfC{$!(X \multimap X) \vdash X \,\&\, X \multimap X \,\&\, X$}
\RightLabel{$\mathsf{!P}$}
\UnaryInfC{$!(X \multimap X) \vdash !(X \,\&\, X \multimap X \,\&\, X)$}
\end{prooftree}
$$

Il termine ad essa associato è $\lambda x.\langle \mathsf{snd}(x),\ s\ \mathsf{snd}(x)\rangle$, dove s è la variabile libera corrispondente alla formula $!(X \multimap X)$.

Per ε scegliamo invece

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\RightLabel{$\& \, \mathsf{R}$}
\BinaryInfC{$X \vdash X \,\&\, X$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\RightLabel{$\& \, \mathsf{L1}$}
\UnaryInfC{$X \,\&\, X \vdash X$}
\RightLabel{$\multimap\mathsf{L}$}
\BinaryInfC{$X, X \,\&\, X \multimap X \,\&\, X \vdash X$}
\RightLabel{$\multimap\mathsf{R}$}
\UnaryInfC{$X \,\&\, X \multimap X \,\&\, X \vdash X \multimap X$}
\RightLabel{$\S$}
\UnaryInfC{$\S(X \,\&\, X \multimap X \,\&\, X) \vdash \S(X \multimap X)$}
\end{prooftree}
$$

Il λ-termine corrispondente a ε è $\lambda z.\mathsf{fst}(y\langle z,\ z\rangle)$, con y di tipo $\S(X \,\&\, X \multimap X \,\&\, X)$.

Utilizzando il secondo schema di iterazione con σ ed ε appena definite otteniamo

$$
\begin{prooftree}
\AxiomC{$\vdots\ \mathsf{It}'(\sigma, \varepsilon)$}
\noLine
\UnaryInfC{$!(X \multimap X), \mathsf{int} \vdash \S(X \multimap X)$}
\RightLabel{$\multimap\mathsf{R}$}
\UnaryInfC{$\mathsf{int} \vdash \mathsf{int}_X$}
\RightLabel{$\forall\mathsf{R}$}
\UnaryInfC{$\mathsf{int} \vdash \mathsf{int}$}
\end{prooftree}
$$

che è la derivazione che rappresenta il predecessore. Essendo *flat*, può essere iterata tramite It per ottenere la sottrazione, che avrà lo stesso tipo della moltiplicazione.

Il fatto che moltiplicazione e sottrazione abbiano lo stesso tipo non è casuale; esse infatti lavorano entrambe per mezzo dell'iterazione di una funzione "base": l'addizione nel caso della moltiplicazione e il predecessore nel caso della sottrazione. Come osservato in precedenza, in $\overline{\mathbf{LLL}}$ e in tutte le altre logiche *light* esiste una limitazione fortissima all'utilizzo di tale tecnica di programmazione; in pratica, una funzione costruita per mezzo dell'iterazione non può a sua volta essere iterata. Si è già detto che questa limitazione è decisamente fondamentale nel caso della moltiplicazione, giacché l'iterazione del prodotto per una costante porterebbe alla definizione dell'esponenziale. Tuttavia, che la sottrazione debba essere soggetta alla medesima restrizione della moltiplicazione sembra alquanto strano, visto che sottrarre un intero ad un altro non dovrebbe richiedere intuitivamente più di un tempo lineare nell'argomento più piccolo, e dunque l'iterazione del procedimento non dovrebbe causare esplosioni esponenziali della complessità. Nel nostro caso invece non è così, e la causa del problema risiede nella nostra definizione del predecessore; si sarà infatti notato che l'algoritmo descritto sopra per calcolare $n-1$ a partire da n non è proprio il più ovvio che possa venire in mente, ed è banale constatare che la complessità di tale algoritmo è $O(n)$. Poiché per ottenere la sottrazione di due numeri n ed m si ricorre all'iterazione del predecessore, la complessità finale non può che essere $O(nm)$, che è la stessa della moltiplicazione. Una conseguenza peculiare di ciò è che calcolare la differenza tra un numero e sé stesso per trovare zero richiede in $\overline{\mathbf{LLL}}$ un tempo quadratico... che non è certo il massimo dell'efficienza!

Il comportamento poco "furbo" del predecessore non è dovuto ad una nostra mancanza d'ingegno nel riuscire ad escogitare un algoritmo migliore; è stato infatti dimostrato da Parigot che questo fenomeno dipende dalla rappresentazione scelta per i numeri interi, ed è inevitabile: non esiste un termine del λ-calcolo puro (e quindi non ne può certo esistere uno del λ-calcolo tipato di qualsiasi ordine) che calcoli il predecessore di un numero nella cosiddetta rappresentazione "iterativa" (quella scelta da noi) impiegando un numero di passi meno che lineare nella dimensione dell'input. La dimostrazione di questo teorema si può trovare in [20].

Rappresentazione dei polinomi

Passiamo ora ad una classe di funzioni d'importanza fondamentale per il nostro lavoro: i polinomi in una variabile. Naturalmente, poiché stiamo parlando di numeri naturali, la variabile indipendente di tali polinomi sarà senz'altro un intero non negativo; inoltre, il nostro interesse principale è quello di rappresentare i polinomi che siano anche funzioni di complessità proprie, secondo la definizione di [19]. Ciò significa che considereremo solo polinomi a coefficienti anch'essi interi non negativi. Del resto, se il runtime di una macchina di Turing è limitato, ad esempio, dal polinomio n^2-5n+4, dove n è la lunghezza dell'input ed è dunque un numero intero non negativo, avremo che il polinomio a coefficienti non negativi n^2+5n+4 limita anch'esso senza dubbio il runtime della medesima macchina.

Mostreremo dunque come rappresentare in $\overline{\mathbf{LLL}}$ i polinomi in una variabile non negativa a coefficienti interi non negativi nel seguente modo: prima costruiremo il generico monomio kn^e, poi otterremo il polinomio desiderato come somma di tali monomi.

Nel seguito, indicheremo con $add[x, y]$ e $mul[w, y]$ i λ-termini associati rispettivamente alla derivazione $+$ e alla derivazione della moltiplicazione, essendo x e y due variabili libere di tipo int e w una variabile libera di tipo $!\mathsf{int}$. Inoltre, chiameremo $+^p$ e mul^p le derivazioni rispettivamente dell'addizione e della moltiplicazione alle quali sono state aggiunte alla fine p regole per l'introduzione del paragrafo; il tipo di $+^p$ sarà dunque $\S^p\mathsf{int}, \S^p\mathsf{int} \vdash \S^p\mathsf{int}$, mentre quello di mul^p sarà $\S^p!\mathsf{int}, \S^p\mathsf{int} \vdash \S^p\mathsf{int}$.

Supponiamo dunque di voler rappresentare il monomio kn^e, dove k ed e sono due interi non negativi. La seguente derivazione, che chiamiamo $\mathsf{mono}_{k,e}$, itera il prodotto e volte e moltiplica il tutto per k, calcolando dunque il monomio desiderato:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{matrix}\vdots\,\omega_k\\ \vdash \mathsf{int}\end{matrix}\quad\begin{matrix}\vdots\,\mathsf{mul}^0\\ \mathsf{int},!\mathsf{int}\vdash\S\mathsf{int}\end{matrix}}{!\mathsf{int}\vdash\S\mathsf{int}}\,\mathsf{cut}\quad\begin{matrix}\vdots\,\mathsf{mul}^1\\ \S\mathsf{int},\S!\mathsf{int}\vdash\S^2\mathsf{int}\end{matrix}}{\begin{matrix}!\mathsf{int},\S!\mathsf{int}\vdash\S^2\mathsf{int}\\ \vdots\\ !\mathsf{int},\ldots,\S^{e-1}!\mathsf{int}\vdash\S^{e-1}\mathsf{int}\end{matrix}}\,\mathsf{cut}\quad\begin{matrix}\vdots\,\mathsf{mul}^{e-1}\\ \S^{e-1}\mathsf{int},\S^{e-1}!\mathsf{int}\vdash\S^{e}\mathsf{int}\end{matrix}}{!\mathsf{int},\ldots,\S^{e}!\mathsf{int}\vdash\S^{e}\mathsf{int}}\,\mathsf{cut}}{\S!\mathsf{int},\ldots,\S^{e+1}!\mathsf{int}\vdash\S^{e+1}\mathsf{int}}\,\S}{}}{}}{}}{}$$

$$\dfrac{\mathit{coercions}\qquad \S!\mathsf{int},\ldots,\S^{e+1}!\mathsf{int}\vdash\S^{e+1}\mathsf{int}}{\dfrac{\dfrac{\dfrac{\mathsf{int}^{(e)}\vdash\S^{e+1}\mathsf{int}}{(!\mathsf{int})^{(e)}\vdash\S^{e+1}\mathsf{int}}\,?\mathsf{D}}{(!\mathsf{int})^{(e)}\vdash\S^{e+2}\mathsf{int}}\,\S}{!\mathsf{int}\vdash\S^{e+2}\mathsf{int}}\,?\mathsf{C}}\,\mathsf{Cut}$$

Il λ-termine associato a $\mathsf{mono}_{k,e}$, che indicheremo nel seguito con $mono_{k,e}[n]$, è il seguente:

$$mul[n, mul[\ldots mul[n, \bar{k}]\ldots]]$$

con e ripetizioni del termine mul.

In modo simile, iteriamo i monomi un numero arbitrario di volte per ottenere il generico polinomio $\sum_{i=0}^{d} a_i n^i$; in termini (è proprio il caso di dirlo!) di λ-calcolo, la rappresentazione sarà

$$add[mono_{a_d,d}[n], add[mono_{a_{d-1},d-1}[n], add[\ldots add[mono_{a_1,1}[n], \bar{a_0}]$$

dove n è la variabile indipendente del polinomio.

Il seguente è lo schema generale della derivazione associata al polinomio

$\sum_{i=0}^{d} a_i n^i$, che denoteremo con $[\![\sum_{i=0}^{d} a_i n^i]\!]$:

$$
\dfrac{
\mathit{coercions}
\qquad
\dfrac{
\dfrac{
\begin{array}{c}\vdots\,\mathsf{mono}_{a_d,d}\\ !\mathsf{int}\vdash\S^{d+2}\mathsf{int}\end{array}
\qquad
\dfrac{
\dfrac{
\begin{array}{c}
\dfrac{
\dfrac{\begin{array}{c}\vdots\,\mathsf{mono}_{a_1,1}\\ !\mathsf{int}\vdash\S^{3}\mathsf{int}\end{array}
\qquad
\dfrac{\dfrac{\begin{array}{c}\vdots\,\omega_{a_0}\\ \vdash\mathsf{int}\end{array}\quad\begin{array}{c}\vdots\,+^{0}\\ \mathsf{int},\mathsf{int}\vdash\mathsf{int}\end{array}}{\mathsf{int}\vdash\mathsf{int}}\,\mathsf{cut}}{\S^{3}\mathsf{int}\vdash\S^{3}\mathsf{int}}\,\S}
{!\mathsf{int}\vdash\S^{3}\mathsf{int}}\,\mathsf{cut}
}{}
\\ \vdots \\
!\mathsf{int},\ldots,\S^{d-2}!\mathsf{int}\vdash\S^{d+1}\mathsf{int}
\end{array}
\qquad
\begin{array}{c}\vdots\,+^{d+1}\\ \S^{d+1}\mathsf{int},\S^{d+1}\mathsf{int}\vdash\S^{d+1}\mathsf{int}\end{array}
}{\S^{d+1}\mathsf{int},!\mathsf{int},\ldots,\S^{d-2}!\mathsf{int}\vdash\S^{d+1}\mathsf{int}}\,\mathsf{cut}
}{\S^{d+2}\mathsf{int},\S!\mathsf{int},\ldots,\S^{d-1}!\mathsf{int}\vdash\S^{d+2}\mathsf{int}}\,\S
}{!\mathsf{int},\S!\mathsf{int},\ldots,\S^{d-1}!\mathsf{int}\vdash\S^{d+2}\mathsf{int}}\,\mathsf{cut}
}{\S!\mathsf{int},\S^{2}!\mathsf{int},\ldots,\S^{d}!\mathsf{int}\vdash\S^{d+3}\mathsf{int}}\,\S
}{
\dfrac{
\dfrac{
\dfrac{\mathsf{int}^{(d)}\vdash\S^{d+3}\mathsf{int}}{(!\mathsf{int})^{(d)}\vdash\S^{d+3}\mathsf{int}}\,?\mathsf{D}
}{(!\mathsf{int})^{(d)}\vdash\S^{d+4}\mathsf{int}}\,\S
}{!\mathsf{int}\vdash\S^{d+4}\mathsf{int}}\,?\mathsf{C}
}\,\mathsf{Cut}
$$

Applicando una regola § e un *cut* sull'opportuna coercizione, otteniamo una derivazione che fa corrispondere ad un polinomio di grado d il tipo $\mathsf{int}\vdash\S^{d+5}\mathsf{int}$.

9.4.2 Caratteri e stringhe

Al fine di costruire una rappresentazione per la generica macchina di Turing, è fondamentale che nel nostro sistema sia possibile rappresentare i caratteri di un alfabeto finito e le stringhe su tale alfabeto.

Per quanto riguarda i caratteri, se F è una formula qualsiasi, poniamo

$$\mathsf{char}_F^p := F^{\,\&\ p} \multimap F$$

e definiamo il tipo dei caratteri di un alfabeto finito con p simboli come

$$\mathsf{char}^p := \forall X.\mathsf{char}_X^p$$

In particolare, definiamo il tipo $\mathsf{shift} := \mathsf{char}^2$, il quale rappresenta il movimento (destra o sinistra) della testina della nostra macchina di Turing. I λ-termini corrispondenti alle due derivazioni della formula shift sono

$left := \lambda x.\mathsf{fst}(x)$

$right := \lambda x.\mathsf{snd}(x)$

Consideriamo ora la formula

$$\mathsf{str}_F^p := !(F \multimap F) \multimap \ldots \multimap !(F \multimap F) \multimap \S(F \multimap F)$$

con p occorrenze di $!(F \multimap F)$, dove F è una formula qualsiasi. Definiamo allora il tipo delle stringhe di lunghezza finita su un alfabeto di p caratteri come

$$\mathsf{str}^p := \forall X.\mathsf{str}^p_X$$

E' chiaro che str^p non è nient'altro che una generalizzazione degli interi in notazione unaria; infatti, $\mathsf{str}^1 \equiv \mathsf{int}$. Inoltre, essendo gli interi non negativi in notazione binaria un sottoinsieme delle stringhe finite su un alfabeto di 2 simboli, porremo $\mathsf{bint} := \mathsf{str}^2$. Naturalmente, è possibile definire sul tipo bint (o str^p in generale) tutte le funzioni già definite sul tipo int; si tratta semplicemente di estendere le derivazioni presentate nella sezione precedente in modo da costruirne versioni binarie (o p-arie). In particolare, la versione binaria di $+$ sarà una derivazione $\frown$ che, date due stringhe, restituisce la loro concatenazione. A partire da questa, si possono costruire le due funzioni "successore" $\frown_0$ e $\frown_1$, che aggiungono in coda alla stringa rispettivamente il carattere 0 e 1. E' poi possibile definire una versione binaria dei due schemi di iterazione, e quindi definire coercizioni sul tipo bint e una funzione "predecessore" che elimina l'ultimo simbolo della stringa.

Per avere un idea di come funzioni la rappresentazione nel λ-calcolo, ecco ad esempio il termine che rappresenta la stringa binaria "1101":

$$\lambda s_1 s_0 z.s_1(s_0(s_1(s_1 z)))$$

Osserviamo che l'ordine in cui compaiono le "cifre" nel λ-termine è inverso rispetto a quello della notazione usuale da sinistra a destra; questo "trucco" consente di definire in modo più semplice la funzione *predecessore*.

9.4.3 Codifica delle macchine di Turing

Ci occuperemo ora di stabilire il risultato fondamentale di questa sezione, cioè di costruire una rappresentazione in $\overline{\mathbf{LLL}}$ della generica macchina di Turing e di definire una funzione che ne simuli l'esecuzione. La codifica delle macchine di Turing nelle logiche *light* è alquanto delicata, a causa dell'estrema povertà espressiva del linguaggio in cui si deve operare. In particolare, come osservato da Terui in [24], la presenza delle condizioni di stratificazione non consente di programmare una versione *flat* della funzione condizionale del tipo

$$\texttt{if p(x) then } \mathtt{f}_1\texttt{(x) else } \mathtt{f}_2\texttt{(x)}$$

La conseguenza è che non è possibile iterare il condizionale, cosa che sarebbe invece fondamentale per una codifica dell'esecuzione di una macchina di Turing (se lo stato è x, allora fai y, altrimenti fai z, e così via). Occorre dunque ricorrere a rappresentazioni più sofisticate, che lavorino in maniera il più possibile "lineare", cioè utilizzando i connettivi esponenziali nel modo più accorto possibile. La codifica che mostreremo è essenzialmente un adattamento, e per certi aspetti una leggera semplificazione, della codifica trovata da Roversi per **ILAL** ed esposta in [22]. Il funzionamento di tale codifica conferma l'estendibilità dell'approccio di Roversi alle altre logiche *light* (è possibile infatti replicare in **LLL** tutto ciò che verrà mostrato nel seguito per $\overline{\mathbf{LLL}}$), e costituisce tra l'altro la prova definitiva della completezza del sistema di Girard che, sebbene intuitivamente verificabile, era rimasta formalmente "in sospeso" dopo che Roversi stesso aveva individuato un errore nella dimostrazione originale di [12] (si veda sempre [22]).

Nel seguito lavoreremo facendo l'ipotesi semplificativa, ma assolutamente non restrittiva, che le macchine di Turing da rappresentare siano tutte su un alfabeto di 2 caratteri, $\Sigma = \{0, 1\}$, più due caratteri speciali "blank" e "eot" (*end-of-tape*), che indicheremo con $\Box$ e $\star$. Supporremo poi che la funzione di transizione delle macchine di Turing sia totale, cioè sia definita per qualsiasi coppia stato/carattere. Inoltre, assumeremo che il nastro sia finito ma *infinitamente estendibile*. Ciò significa che, inizialmente, l'input della macchina di Turing si presenta come una stringa composta dai caratteri 0 e 1 (dunque un numero in notazione binaria), con la testina posizionata sulla cella immediatamente a sinistra dell'estremo sinistro della stringa; su tale cella, e su quella all'estremo opposto, ci sono i due caratteri delimitatori $\star$ e non sono disponibili altre celle. Durante l'esecuzione, se la macchina di Turing si trova a dover scrivere su una delle due estremità del nastro (riconoscibili appunto per la presenza del carattere $\star$), si preoccuperà di "estendere" il nastro stesso aggiungendo un nuovo delimitatore immediatamente a destra o a sinistra (a seconda di quale estremità si tratti) del vecchio delimitatore, ora rimpiazzato da un altro carattere. Tale operazione di estensione è considerata automatica ed è implicita nella funzione di transizione. Al termine del calcolo, il risultato sarà scritto sempre in notazione binaria, ma non ci sono vincoli sulla posizione finale della testina.

Le configurazioni

Sia dunque **M** una macchina di Turing con le proprietà sopra definite, sull'alfabeto $\Sigma_{\mathsf{x}} = \{0, 1, \Box, \star\}$, con q stati e funzione di transizione δ.

Gli stati saranno rappresentati dalle q derivazioni della formula $\mathsf{state} := \mathsf{char}^q$.

Per brevità, porremo $\mathsf{char} := \mathsf{char}^3$ e $\mathsf{charx} := \mathsf{char}^4$. Il tipo char rappresenta l'alfabeto $\Sigma_{\Box} = \{0, 1, \Box\}$, che è l'alfabeto dei simboli che possono effettivamente occupare le celle del nastro; charx rappresenta invece l'alfabeto Σ_{x} completo del simbolo "eot".

Definiamo poi

$\mathsf{conf}_F := F \otimes F \otimes \mathsf{state}$

$\mathsf{Tur}_F := !(F \multimap F) \multimap !(F \multimap F) \multimap !(F \multimap F) \multimap \S(F \multimap F \multimap \mathsf{conf}_F)$

dove F è una formula qualsiasi.

Il tipo delle configurazioni (contenuto del nastro, posizione della testina e stato corrente) della macchina di Turing $\mathbf{M}$ sarà il seguente:

$$\mathsf{Tur} := \forall X.\mathsf{Tur}_X$$

E' utile fornire un esempio per comprendere come funzioni tale rappresentazione. Supponiamo che $\mathbf{M}$ si trovi nello stato $\mathbf{k}_i$, che il nastro contenga la stringa "10110" e che la testina sia posizionata sullo 0 più a sinistra. Il λ-termine di tipo Tur corrispondente a tale configurazione sarà

$$\lambda s_1 s_0 s_\Box z_l z_r . s_0(s_1 z_l) \otimes s_1(s_1(s_0 z_r)) \otimes \mathbf{k}_i$$

Come si può facilmente constatare, la tripla moltiplicativa nel corpo del λ-termine rappresenta il contenuto in ordine inverso del nastro a sinistra della testina (incluso il carattere corrente), poi il contenuto del nastro a destra della testina e infine lo stato corrente. Un ulteriore esempio: per le convenzioni stabilite sopra, una configurazione iniziale sarà sempre del tipo

$$\lambda s_1 s_0 s_\Box z_l z_r . z_l \otimes s_\diamond(\ldots s_\diamond z_r \ldots) \otimes \mathbf{k}_0$$

dove $\diamond \in \{0, 1\}$ e $\mathbf{k}_0$ è lo stato iniziale.

La funzione step

Cardine della dimostrazione di completezza è la funzione che noi chiameremo *step*, che simula il passo elementare di esecuzione di una macchina di Turing. In sostanza, deve trattarsi di una funzione di tipo $\mathsf{Tur} \vdash \mathsf{Tur}$, che prende in input la configurazione corrente di $\mathbf{M}$ e restituisce quella successiva, ricavandola dalla funzione di transizione δ.

Lo step di una macchina di Turing viene simulato in due fasi che, in analogia a quanto avviene per l'esecuzione di un'istruzione elementare da parte delle CPU reali, chiameremo *fase di fetch* e *fase di decode/execute*. La fase di fetch serve in un qualche modo a "pre-processare" la configurazione corrente della macchina; in sostanza essa estrae testa e coda di entrambe le stringhe che compongono la configurazione stessa. La fase di decode/execute o, più semplicemente, di esecuzione, accede alla funzione di transizione e genera effettivamente la configurazione successiva.

Le due funzioni che intervengono nella fase di fetch sono

$$base_i[z_i] := \langle \square, \lambda q.q\rangle \otimes z_i, \qquad i \in \{l, r\}$$

$$fetch_\diamond := \lambda p \otimes l.\langle \diamond, \mathsf{snd}(p)\rangle \otimes l, \qquad \diamond \in \{0, 1, \square\}$$

Ponendo

$$\mathsf{B} := (\mathsf{charx}\ \ \&\ \ !(X \multimap X)) \otimes X,$$

il tipo di $base_i[z_i]$ è $X \vdash \mathsf{B}$ (X è il tipo della variabile libera z_i), mentre quello di $fetch_\diamond$ è $\vdash !(B \multimap B)$. Le derivazioni corrispondenti a tali termini sono base_i:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\square\\ \vdash \mathsf{charx}\end{array} \qquad \dfrac{\dfrac{\dfrac{}{X \vdash X}\,\mathsf{ax}}{\vdash X \multimap X}\,{\multimap}\mathsf{R}}{\vdash !(X \multimap X)}\,!\mathsf{P}}{\vdash \mathsf{charx}\ \ \&\ \ !(X \multimap X)}\,\&\,\mathsf{R} \qquad \dfrac{}{X \vdash X}\,\mathsf{ax}}{X \vdash \mathsf{B}}\,\otimes\mathsf{R}$$

e $\mathsf{fetch}(\diamond)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\vdots\,\diamond\\ \vdash \mathsf{charx}\end{array}}{!(X \multimap X) \vdash \mathsf{charx}}\,?\mathsf{W} \qquad \dfrac{}{!(X \multimap X) \vdash !(X \multimap X)}\,\mathsf{ax}}{!(X \multimap X) \vdash \mathsf{charx}\ \ \&\ \ !(X \multimap X)}\,\&\,\mathsf{R} \qquad \dfrac{}{X \vdash X}\,\mathsf{ax}}{!(X \multimap X), X \vdash \mathsf{B}}\,\otimes\mathsf{R}}{\mathsf{charx}\ \ \&\ \ !(X \multimap X), X \vdash \mathsf{B}}\,\&\,\mathsf{L2}}{\mathsf{B} \vdash \mathsf{B}}\,\otimes\mathsf{L}}{\vdash \mathsf{B} \multimap \mathsf{B}}\,{\multimap}\mathsf{R}}{\vdash !(\mathsf{B} \multimap \mathsf{B})}\,!\mathsf{P}$$

dove $\diamond$ è una delle tre derivazioni di tipo $\vdash \mathsf{charx}$ corrispondenti ai simboli 0, 1 e $\square$.

La fase di fetch consiste semplicemente nell'applicare la configurazione corrente t (di tipo Tur) ai termini $fetch_\diamond$ e $base_i$. Al termine di tale operazione, la configurazione della macchina di Turing sarà stata elaborata in un oggetto di tipo conf_B. La derivazione corrispondente a questa prima fase di elaborazione, che chiamiamo dofetch, è la seguente:

$$
\begin{prooftree}
\AxiomC{$\vdots\ \mathsf{fetch}_1$}
\noLine
\UnaryInfC{$\vdash\ !(\mathsf{B} \multimap \mathsf{B})$}
\AxiomC{$\vdots\ \mathsf{fetch}_0$}
\noLine
\UnaryInfC{$\vdash\ !(\mathsf{B} \multimap \mathsf{B})$}
\AxiomC{$\vdots\ \mathsf{fetch}_\Box$}
\noLine
\UnaryInfC{$\vdash\ !(\mathsf{B} \multimap \mathsf{B})$}
\AxiomC{$\vdots\ \mathsf{base}_l$}
\noLine
\UnaryInfC{$X \vdash \mathsf{B}$}
\AxiomC{$\vdots\ \mathsf{base}_r$}
\noLine
\UnaryInfC{$X \vdash \mathsf{B}$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$\mathsf{conf}_\mathsf{B} \vdash \mathsf{conf}_\mathsf{B}$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$\mathsf{conf}_X \vdash \mathsf{conf}_X$}
\RightLabel{$\multimap\mathsf{L}$}
\BinaryInfC{$\mathsf{conf}_\mathsf{B}, \mathsf{conf}_\mathsf{B} \multimap \mathsf{conf}_X \vdash \mathsf{conf}_X$}
\RightLabel{$\multimap\mathsf{L}$}
\BinaryInfC{$\mathsf{B} \multimap \mathsf{conf}_\mathsf{B}, X, \mathsf{conf}_\mathsf{B} \multimap \mathsf{conf}_X \vdash \mathsf{conf}_X$}
\RightLabel{$\multimap\mathsf{L}$}
\BinaryInfC{$X, X, \mathsf{B} \multimap \mathsf{B} \multimap \mathsf{conf}_\mathsf{B}, \mathsf{conf}_\mathsf{B} \multimap \mathsf{conf}_X \vdash \mathsf{conf}_X$}
\doubleLine
\RightLabel{$\multimap\mathsf{R}$}
\UnaryInfC{$\mathsf{B} \multimap \mathsf{B} \multimap \mathsf{conf}_\mathsf{B}, \mathsf{conf}_\mathsf{B} \multimap \mathsf{conf}_X \vdash X \multimap X \multimap \mathsf{conf}_X$}
\RightLabel{$\S$}
\UnaryInfC{$\S(\mathsf{B} \multimap \mathsf{B} \multimap \mathsf{conf}_\mathsf{B}), \S(\mathsf{conf}_\mathsf{B} \multimap \mathsf{conf}_X) \vdash \S(X \multimap X \multimap \mathsf{conf}_X)$}
\RightLabel{$\multimap\mathsf{L}$}
\BinaryInfC{$!(\mathsf{B} \multimap \mathsf{B}) \multimap \S(\mathsf{B} \multimap \mathsf{B} \multimap \mathsf{conf}_\mathsf{B}), \S(\mathsf{conf}_\mathsf{B} \multimap \mathsf{conf}_X) \vdash \S(X \multimap X \multimap \mathsf{conf}_X)$}
\RightLabel{$\multimap\mathsf{L}$}
\BinaryInfC{$!(\mathsf{B} \multimap \mathsf{B}) \multimap !(\mathsf{B} \multimap \mathsf{B}) \multimap \S(\mathsf{B} \multimap \mathsf{B} \multimap \mathsf{conf}_\mathsf{B}), \S(\mathsf{conf}_\mathsf{B} \multimap \mathsf{conf}_X) \vdash \S(X \multimap X \multimap \mathsf{conf}_X)$}
\RightLabel{$\multimap\mathsf{L}$}
\BinaryInfC{$\mathsf{Tur}_\mathsf{B}, \S(\mathsf{conf}_\mathsf{B} \multimap \mathsf{conf}_X) \vdash \S(X \multimap X \multimap \mathsf{conf}_X)$}
\RightLabel{$\forall\mathsf{L}$}
\UnaryInfC{$\S(\mathsf{conf}_\mathsf{B} \multimap \mathsf{conf}_X), \mathsf{Tur} \vdash \S(X \multimap X \multimap \mathsf{conf}_X)$}
\end{prooftree}
$$

A questo punto, per ottenere la funzione *step* desiderata, è sufficiente disporre di una funzione di tipo

$$(!(X \multimap X))^{(3)}, \mathsf{conf}_\mathsf{B} \vdash \mathsf{conf}_X$$

Assumendo di avere una derivazione exec che corrisponda a tale funzione, possiamo rappresentare la funzione *step* in questo modo:

$$
\begin{prooftree}
\AxiomC{$\vdots\ \mathsf{exec}$}
\noLine
\UnaryInfC{$(!(X \multimap X))^{(3)} \vdash \S(\mathsf{conf}_\mathsf{B} \multimap \mathsf{conf}_X)$}
\AxiomC{$\vdots\ \mathsf{dofetch}$}
\noLine
\UnaryInfC{$\S(\mathsf{conf}_\mathsf{B} \multimap \mathsf{conf}_X), \mathsf{Tur} \vdash \S(X \multimap X \multimap \mathsf{conf}_X)$}
\RightLabel{cut}
\BinaryInfC{$(!(X \multimap X))^{(3)}, \mathsf{Tur} \vdash \S(X \multimap X \multimap \mathsf{conf}_X)$}
\doubleLine
\RightLabel{$\multimap\mathsf{R}$}
\UnaryInfC{$\mathsf{Tur} \vdash \mathsf{Tur}_X$}
\RightLabel{$\forall\mathsf{R}$}
\UnaryInfC{$\mathsf{Tur} \vdash \mathsf{Tur}$}
\end{prooftree}
$$

Il λ-termine corrispondente a questa derivazione, che chiamiamo step, è

$$\lambda s_1 s_0 s_\Box z_l z_r . exec(t\ fetch_1\ fetch_0\ fetch_\Box\ base_l\ base_r)$$

dove t è la variabile libera di tipo Tur. Come si vede, prima t viene applicata alle varie funzioni di *fetch*, poi il risultato viene dato in pasto alla funzione *exec*.

Codifica della funzione di transizione

Prima di vedere come funziona la *exec*, è opportuno stabilire una codifica per la funzione di transizione δ. Questa può prendere in input caratteri dell'alfabeto completo Σ_x, ma può fornire in uscita solo simboli di $\Sigma_\Box$, poiché il carattere $\star$ non può essere scritto sul nastro all'infuori che sulle estremità. Di conseguenza, posto

$$\mathsf{out} := \mathsf{char} \otimes \mathsf{state} \otimes \mathsf{shift}$$

la derivazione $\hat{\delta}$ che codifica la generica funzione di transizione dovrà essere di tipo

$$\mathsf{charx}, \mathsf{state} \vdash \mathsf{out}$$

Essa prende in input un carattere e uno stato e fornisce in uscita il nuovo carattere, il nuovo stato e lo spostamento della testina.

Un oggetto di tipo out può essere derivato in modo banale a partire dalle derivazioni degli oggetti di tipo char, state e shift che lo compongono; basta applicare due regole $\otimes\mathsf{R}$. La generica funzione di transizione è altrettanto semplice da ricavare; si tratta in pratica di costruire una matrice i cui elementi sono tutte le possibili uscite (oggetti di tipo out) della funzione stessa.

Sappiamo che la nostra macchina di Turing $\mathbf{M}$ ha q stati; poiché δ dev'essere totale, questa deve necessariamente essere definita su $4q$ coppie carattere/stato. Siano allora $\mathsf{out}_{ij}, 1 \leq i \leq q, 1 \leq j \leq 4$ le derivazioni corrispondenti alle uscite in corrispondenza dei $4q$ ingressi possibili, ciascuna di tipo out. Possiamo dapprima costruire le q quadruple che contengono le uscite che ogni stato associa ad un determinato carattere, fornendo la seguente derivazione che chiamiamo ξ_i (sempre con $1 \leq i \leq q$):

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\vdots\ \mathsf{out}_{i1}\\ \vdash \mathsf{out}\end{matrix} \qquad \begin{matrix}\vdots\ \mathsf{out}_{i2}\\ \vdash \mathsf{out}\end{matrix}}{\vdash \mathsf{out}^{\&2}}\ \&\mathsf{R} \qquad \begin{matrix}\vdots\ \mathsf{out}_{i3}\\ \vdash \mathsf{out}\end{matrix}}{\vdash \mathsf{out}^{\&3}}\ \&\mathsf{R} \qquad \begin{matrix}\vdots\ \mathsf{out}_{i4}\\ \vdash \mathsf{out}\end{matrix}}{\vdash \mathsf{out}^{\&4}}\ \&\mathsf{R}$$

A questo punto mettiamo insieme la quadruple (che possono essere viste come vettori) formando quella che possiamo considerare come una matrice $4 \times\ q$, che diviene la tabella di *look-up* che la funzione exec andrà a "consultare" per ricavare il comportamento della macchina di Turing. La derivazione $\hat{\delta}$ che codifica la funzione di transizione è quindi:

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\dfrac{\begin{matrix}\vdots\ \xi_1\\ \vdash \mathsf{out}^{\&4}\end{matrix} \qquad \begin{matrix}\vdots\ \xi_2\\ \vdash \mathsf{out}^{\&4}\end{matrix}}{\vdash (\mathsf{out}^{\&4})^{\&2}}\ \&\mathsf{R}\\ \vdots\\ \vdash (\mathsf{out}^{\&4})^{\&q-1}\end{matrix} \qquad \begin{matrix}\vdots\ \xi_q\\ \vdash \mathsf{out}^{\&4}\end{matrix}}{\vdash (\mathsf{out}^{\&4})^{\&q}}\ \&\mathsf{R} \qquad \dfrac{\dfrac{}{\mathsf{out}^{\&4} \vdash \mathsf{out}^{\&4}}\ \mathsf{ax} \qquad \dfrac{}{\mathsf{out} \vdash \mathsf{out}}\ \mathsf{ax}}{\mathsf{out}^{\&4}, \mathsf{charx}_{\mathsf{out}} \vdash \mathsf{out}}\ \multimap\mathsf{L}}{\mathsf{charx}_{\mathsf{out}}, \mathsf{state}_{\mathsf{out}^{\&4}} \vdash \mathsf{out}}\ \multimap\mathsf{L}}{\mathsf{charx}, \mathsf{state} \vdash \mathsf{out}}\ \forall\mathsf{L}$$

Naturalmente le out_{ij} nelle varie ξ_i sono scelte in modo appropriato secondo il comportamento della funzione di transizione stessa.

La funzione exec

Possiamo ora analizzare la codifica dell'altra funzione chiave nella rappresentazione delle macchine di Turing, cioè la funzione *exec*. Nella definizione di questa funzione interviene un altro tipo, che è

$$\mathsf{C} := \mathsf{charx} \otimes X$$

Il tipo C gioca in *exec* lo stesso ruolo che il tipo B giocava nelle funzioni di fetch; serve semplicemente a processare i dati in ingresso. Prima di presentare la derivazione exec completa, mostriamo il λ-termine che costituisce l'idea alla base della sua struttura, in modo da facilitare la comprensione delle operazioni da essa effettuate. Siano

$$\sigma := \langle\langle\langle s_1, s_0\rangle, s_\square\rangle, \lambda q.q\rangle$$

$$\rho := \langle\langle s_1, s_0\rangle, s_\square\rangle$$

abbiamo

$$\begin{aligned} exec' := \lambda(h_l \otimes t_l).\lambda(h_l \otimes t_l).\lambda j.&\\ (\lambda\chi \otimes \kappa \otimes \mu.&\\ &(\mu\langle t_l \otimes (\chi\ \rho\ (h_r\ \sigma\ t_r)), (h_r\ \sigma\ (\chi\ \rho\ t_l)) \otimes t_r\rangle) \otimes \kappa\\ )\hat{\delta}[h_l, j]& \end{aligned}$$

Questa versione semplificata di *exec* prende in input due oggetti di tipo C (che rappresentano la scomposizione testa/coda delle due sezioni del nastro), lo stato corrente e, a seconda dell'uscita fornita dalla funzione di transizione, restituisce l'oggetto di tipo conf_X appropriato.

La $exec'$ potrebbe facilmente essere trasformata in una funzione di tipo

$$(!(X \multimap X))^{(3)}, \mathsf{conf}_{\mathsf{C}} \vdash \mathsf{conf}_X$$

Tuttavia, come detto in precedenza, la fase di fetch termina con un oggetto di tipo $\mathsf{conf}_{\mathsf{B}}$, e dunque a noi serve un input di quest'ultimo tipo, e non di tipo $\mathsf{conf}_{\mathsf{C}}$. Pertanto, si rivela necessario definire una funzione che consenta di passare dal tipo B al tipo C. La derivazione che rappresenta questa funzione, che chiamiamo conv, è la seguente:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{charx} \vdash \mathsf{charx}}\,\mathsf{ax} \qquad \dfrac{}{X \vdash X}\,\mathsf{ax}}{\mathsf{charx}, X \vdash \mathsf{C}}\,\otimes\mathsf{R}}{\mathsf{charx}\ \&\ \ !(X \multimap X), X \vdash \mathsf{C}}\,\&\ \mathsf{L1}}{\mathsf{B} \vdash \mathsf{C}}\,\otimes\mathsf{L}$$

Come si vede, essa prende un oggetto di tipo B e lo converte nel corrispondente oggetto di tipo C, in modo che *exec* lo possa utilizzare.

Possiamo ora presentare la derivazione exec in modo completo. Anzitutto, sia τ la seguente derivazione:

$$\dfrac{\dfrac{\dfrac{}{X \multimap X \vdash X \multimap X}\,\mathsf{ax}}{!(X \multimap X) \vdash X \multimap X}\,?\mathsf{D}}{(!(X \multimap X))^{(3)} \vdash X \multimap X}\,?\mathsf{W}$$

e siano σ e ρ le seguenti:

$$\dfrac{\dfrac{\dfrac{\overset{\vdots\,\tau}{(!(X \multimap X))^{(3)} \vdash X \multimap X} \quad \overset{\vdots\,\tau}{(!(X \multimap X))^{(3)} \vdash X \multimap X}}{(!(X \multimap X))^{(3)} \vdash (X \multimap X)^{\&\,2}}\,\&\,\mathsf{R} \quad \overset{\vdots\,\tau}{(!(X \multimap X))^{(3)} \vdash X \multimap X}}{(!(X \multimap X))^{(3)} \vdash (X \multimap X)^{\&\,3}}\,\&\,\mathsf{R} \quad \dfrac{\dfrac{\dfrac{}{X \vdash X}\,\mathsf{ax}}{\vdash X \multimap X}\,\multimap\mathsf{R}}{(!(X \multimap X))^{(3)} \vdash X \multimap X}\,?\mathsf{W}}{(!(X \multimap X))^{(3)} \vdash (X \multimap X)^{\&\,4}}\,\&\,\mathsf{R}$$

$$\dfrac{\dfrac{\overset{\vdots\,\tau}{(!(X \multimap X))^{(3)} \vdash X \multimap X} \quad \overset{\vdots\,\tau}{(!(X \multimap X))^{(3)} \vdash X \multimap X}}{(!(X \multimap X))^{(3)} \vdash (X \multimap X)^{\&\,2}}\,\&\,\mathsf{R} \quad \overset{\vdots\,\tau}{(!(X \multimap X))^{(3)} \vdash X \multimap X}}{(!(X \multimap X))^{(3)} \vdash (X \multimap X)^{\&\,3}}\,\&\,\mathsf{R}$$

Chiamiamo poi moveleft e moveright queste due derivazioni, che si occupano di definire il comportamento della macchina in caso lo spostamento sia rispettivamente a sinistra o a destra:

$$\dfrac{\dfrac{}{X \vdash X}\,\mathsf{ax} \quad \dfrac{\dfrac{\overset{\vdots\,\sigma}{(!(X \multimap X))^{(3)} \vdash (X \multimap X)^{\&\,4}} \quad \dfrac{\dfrac{\dfrac{}{X \vdash X}\,\mathsf{ax} \quad \dfrac{\overset{\vdots\,\rho}{(!(X \multimap X))^{(3)} \vdash (X \multimap X)^{\&\,3}} \quad \dfrac{\dfrac{}{X \vdash X}\,\mathsf{ax} \quad \dfrac{}{X \vdash X}\,\mathsf{ax}}{X \multimap X, X \vdash X}\,\multimap\mathsf{L}}{X, (!(X \multimap X))^{(3)}, \mathsf{char}_{X \multimap X} \vdash X}\,\multimap\mathsf{L}}{(!(X \multimap X))^{(3)}, X \multimap X, \mathsf{char}_{X \multimap X}, X \vdash X}\,\multimap\mathsf{L}}{(!(X \multimap X))^{(3)}, X \multimap X, \mathsf{char}, X \vdash X}\,\forall\mathsf{L}}{(!(X \multimap X))^{(6)}, \mathsf{charx}_{X \multimap X}, \mathsf{char}, X \vdash X}\,\multimap\mathsf{L}}{(!(X \multimap X))^{(6)}, \mathsf{charx}, \mathsf{char}, X \vdash X}\,\forall\mathsf{L}}{(!(X \multimap X))^{(6)}, \mathsf{charx}, X, \mathsf{char}, X \vdash X \otimes X}\,\otimes\mathsf{R}$$

$$\dfrac{\dfrac{\dfrac{\overset{\vdots\,\sigma}{(!(X \multimap X))^{(3)} \vdash (X \multimap X)^{\&\,4}} \quad \dfrac{\dfrac{\dfrac{\overset{\vdots\,\rho}{(!(X \multimap X))^{(3)} \vdash (X \multimap X)^{\&\,3}} \quad \dfrac{\dfrac{}{X \vdash X}\,\mathsf{ax} \quad \dfrac{}{X \vdash X}\,\mathsf{ax}}{X \multimap X, X \vdash X}\,\multimap\mathsf{L}}{X, (!(X \multimap X))^{(3)}, \mathsf{char}_{X \multimap X} \vdash X}\,\multimap\mathsf{L} \quad \dfrac{}{X \vdash X}\,\mathsf{ax}}{(!(X \multimap X))^{(3)}, X \multimap X, \mathsf{char}_{X \multimap X}, X \vdash X}\,\multimap\mathsf{L}}{(!(X \multimap X))^{(3)}, X \multimap X, \mathsf{char}, X \vdash X}\,\forall\mathsf{L}}{(!(X \multimap X))^{(6)}, \mathsf{charx}_{X \multimap X}, \mathsf{char}, X \vdash X}\,\multimap\mathsf{L}}{(!(X \multimap X))^{(6)}, \mathsf{charx}, \mathsf{char}, X \vdash X}\,\forall\mathsf{L} \quad \dfrac{}{X \vdash X}\,\mathsf{ax}}{(!(X \multimap X))^{(6)}, \mathsf{charx}, X, \mathsf{char}, X \vdash X \otimes X}\,\otimes\mathsf{R}$$

Utilizzando moveleft e moveright costruiamo la seguente derivazione, che chiamiamo ζ e che introduciamo solamente per convenienza, giacché altrimenti la exec occuperebbe troppo spazio per essere presentata in modo integrale:

$$
\begin{prooftree}
\AxiomC{$\vdots\ \mathsf{moveleft}$}
\noLine
\UnaryInfC{$(!(X \multimap X))^{(6)}, \mathsf{charx}, X, \mathsf{char}, X \vdash X \otimes X$}
\AxiomC{$\vdots\ \mathsf{moveright}$}
\noLine
\UnaryInfC{$(!(X \multimap X))^{(6)}, \mathsf{charx}, X, \mathsf{char}, X \vdash X \otimes X$}
\RightLabel{$\& \mathsf{R}$}
\BinaryInfC{$(!(X \multimap X))^{(6)}, \mathsf{charx}, X, \mathsf{char}, X \vdash (X \otimes X) \,\&\, (X \otimes X)$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \otimes X \vdash X \otimes X$}
\RightLabel{$\multimap\mathsf{L}$}
\BinaryInfC{$(!(X \multimap X))^{(6)}, \mathsf{charx}, X, \mathsf{char}, X, \mathsf{shift}_{X \otimes X} \vdash X \otimes X$}
\RightLabel{$\forall\mathsf{L}$}
\UnaryInfC{$(!(X \multimap X))^{(6)}, \mathsf{charx}, X, \mathsf{char}, X, \mathsf{shift} \vdash X \otimes X$}
\RightLabel{$\otimes\mathsf{L}$}
\UnaryInfC{$\mathsf{char}, \mathsf{shift}, \mathsf{C}, X, (!(X \multimap X))^{(6)} \vdash X \otimes X$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$\mathsf{state} \vdash \mathsf{state}$}
\RightLabel{$\otimes\mathsf{R}$}
\BinaryInfC{$\mathsf{char}, \mathsf{state}, \mathsf{shift}, \mathsf{C}, X, (!(X \multimap X))^{(6)} \vdash \mathsf{conf}_X$}
\RightLabel{$\otimes\mathsf{L}$}
\UnaryInfC{$\mathsf{out}, \mathsf{C}, X, (!(X \multimap X))^{(6)} \vdash \mathsf{conf}_X$}
\end{prooftree}
$$

Un'osservazione: il lettore attento avrà forse notato che τ non è una derivazione di $\overline{\mathbf{LLL}}$, perché non soddisfa la condizione di stratificazione a scendere. Di conseguenza, né σ né ρ sono derivazioni di $\overline{\mathbf{LLL}}$, e quindi non lo sono neanche le appena introdotte moveleft, moveright e ζ. Al contrario di quanto si possa pensare, ciò non è affatto un problema; infatti, tutte quante sono utilizzate come sotto-derivazioni della seguente derivazione, che è proprio la exec, nella quale è possibile vedere come venga alla fine introdotta la scatola che dev'essere obbligatoriamente attraversata dai rami esponenziali:

$$
\begin{prooftree}
\AxiomC{$\vdots\ \mathsf{conv}$}
\noLine
\UnaryInfC{$\mathsf{B} \vdash \mathsf{C}$}
\AxiomC{$\vdots\ \hat{\delta}$}
\noLine
\UnaryInfC{$\mathsf{charx}, \mathsf{state} \vdash \mathsf{out}$}
\AxiomC{$\vdots\ \zeta$}
\noLine
\UnaryInfC{$\mathsf{out}, \mathsf{C}, X, (!(X \multimap X))^{(6)} \vdash \mathsf{conf}_X$}
\RightLabel{cut}
\BinaryInfC{$\mathsf{C}, \mathsf{charx}, X, \mathsf{state}, (!(X \multimap X))^{(6)} \vdash \mathsf{conf}_X$}
\RightLabel{$\otimes\mathsf{L}$}
\UnaryInfC{$\mathsf{C}, \mathsf{C}, \mathsf{state}, (!(X \multimap X))^{(6)} \vdash \mathsf{conf}_X$}
\RightLabel{Cut}
\BinaryInfC{$\mathsf{B}, \mathsf{B}, \mathsf{state}, (!(X \multimap X))^{(6)} \vdash \mathsf{conf}_X$}
\RightLabel{$\otimes\mathsf{L}$}
\UnaryInfC{$\mathsf{conf}_{\mathsf{B}}, (!(X \multimap X))^{(6)} \vdash \mathsf{conf}_X$}
\RightLabel{$\multimap\mathsf{R}$}
\UnaryInfC{$(!(X \multimap X))^{(6)} \vdash \mathsf{conf}_{\mathsf{B}} \multimap \mathsf{conf}_X$}
\RightLabel{$\S$}
\UnaryInfC{$(!(X \multimap X))^{(6)} \vdash \S(\mathsf{conf}_{\mathsf{B}} \multimap \mathsf{conf}_X)$}
\RightLabel{$?\mathsf{C}$}
\UnaryInfC{$(!(X \multimap X))^{(3)} \vdash \S(\mathsf{conf}_{\mathsf{B}} \multimap \mathsf{conf}_X)$}
\end{prooftree}
$$

Osserviamo che la exec è esattamente del tipo desiderato, in modo da poter essere tagliata all'interno di step come voluto.

La funzione startup

Siamo ora in possesso di una funzione che è in grado, data una configurazione di $\mathbf{M}$, di passare alla configurazione successiva. Per poter effettuare un calcolo, serve ora una funzione che ci consenta di inizializzare la macchina di Turing, ovverosia che, data una stringa binaria, produca in output la corrispondente configurazione iniziale, con la medesima stringa sul nastro e la testina posta alla sua sinistra.

Il λ-termine che fa al caso nostro è

$$\lambda s_1 s_0 s_\Box z_l z_r . z_l \otimes (l s_1 s_0 z_r) \otimes \mathbf{k}_0$$

dove $\mathbf{k}_0$ è lo stato iniziale e l è la stringa binaria (di tipo bint) in input. Quella che segue è la derivazione corrispondente, che chiamiamo $\mathsf{startup}(\kappa_0)$ (dove κ_0 è la derivazione associata allo stato iniziale):

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$!(X \multimap X) \vdash !(X \multimap X)$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$!(X \multimap X) \vdash !(X \multimap X)$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \multimap X$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\RightLabel{$\multimap$L}
\BinaryInfC{$X, X \multimap X \vdash X$}
\RightLabel{$\otimes$R}
\BinaryInfC{$X, X, X \multimap X \vdash X \otimes X$}
\AxiomC{$\vdots\ \kappa_0$}
\noLine
\UnaryInfC{$\vdash \mathsf{state}$}
\RightLabel{$\otimes$R}
\BinaryInfC{$X, X, X \multimap X \vdash \mathsf{conf}_X$}
\RightLabel{$\multimap$R}
\UnaryInfC{$X \multimap X \vdash X \multimap X \multimap \mathsf{conf}_X$}
\RightLabel{$\S$}
\UnaryInfC{$\S(X \multimap X) \vdash \S(X \multimap X \multimap \mathsf{conf}_X)$}
\RightLabel{$\multimap$L}
\BinaryInfC{$\mathsf{int}_X, !(X \multimap X) \vdash \S(X \multimap X \multimap \mathsf{conf}_X)$}
\RightLabel{$\multimap$L}
\BinaryInfC{$\mathsf{bint}_X, (!(X \multimap X))^{(2)} \vdash \S(X \multimap X \multimap \mathsf{conf}_X)$}
\RightLabel{$\forall$L}
\UnaryInfC{$(!(X \multimap X))^{(2)}, \mathsf{bint} \vdash \S(X \multimap X \multimap \mathsf{conf}_X)$}
\RightLabel{?W}
\UnaryInfC{$(!(X \multimap X))^{(3)}, \mathsf{bint} \vdash \S(X \multimap X \multimap \mathsf{conf}_X)$}
\RightLabel{$\multimap$R}
\UnaryInfC{$\mathsf{bint} \vdash \mathsf{Tur}_X$}
\RightLabel{$\forall$R}
\UnaryInfC{$\mathsf{bint} \vdash \mathsf{Tur}$}
\end{prooftree}
$$

La funzione getresult

Allo stesso modo, ci serve ora una funzione che estragga il risultato del calcolo dal nastro della macchina di Turing e lo restituisca sotto forma di stringa binaria.

Anzitutto, alcune definizioni preliminari. Sia $!\mathsf{id}$ la seguente derivazione, cui è associato il λ-termine $id := \lambda x.x$:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\RightLabel{$\multimap$R}
\UnaryInfC{$\vdash X \multimap X$}
\RightLabel{!P}
\UnaryInfC{$\vdash !(X \multimap X)$}
\end{prooftree}
$$

Se q, come al solito, è il numero di stati della nostra macchina di Turing, sia ν la seguente derivazione, che è semplicemente una q-upla additiva (un vettore) i cui elementi sono tutti termini id:

$$
\begin{prooftree}
\AxiomC{$\vdots\ !\mathsf{id}$}
\noLine
\UnaryInfC{$\vdash !(X \multimap X)$}
\AxiomC{$\vdots\ !\mathsf{id}$}
\noLine
\UnaryInfC{$\vdash !(X \multimap X)$}
\RightLabel{$\&$ R}
\BinaryInfC{$\vdash (!(X \multimap X))^{\& \ 2}$}
\noLine
\UnaryInfC{$\vdots$}
\noLine
\UnaryInfC{$\vdash (!(X \multimap X))^{\& \ q-1}$}
\AxiomC{$\vdots\ !\mathsf{id}$}
\noLine
\UnaryInfC{$\vdash !(X \multimap X)$}
\RightLabel{$\&$ R}
\BinaryInfC{$\vdash (!(X \multimap X))^{\& \ q}$}
\end{prooftree}
$$

Possiamo ora introdurre la funzione join che, data in ingresso una tripla moltiplicativa costituita da due liste binarie e da uno stato (dunque un oggetto di tipo $\mathsf{conf}_{\mathsf{bint}}$), ignora lo stato e restituisce la stringa binaria ottenuta concatenando le due liste:

$$
\dfrac{\dfrac{\vdots\,\nu}{\vdash (!(X\multimap X))^{\&\,q}}
\qquad
\dfrac{\dfrac{\dfrac{\dfrac{}{!(X\multimap X)\vdash !(X\multimap X)}\,\mathsf{ax}
\quad
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{X\vdash X}\,\mathsf{ax}\quad\dfrac{\dfrac{}{X\vdash X}\,\mathsf{ax}\quad\dfrac{\dfrac{}{X\vdash X}\,\mathsf{ax}\quad\dfrac{}{X\vdash X}\,\mathsf{ax}}{X, X\multimap X\vdash X}\,{\multimap}\mathsf{L}}{X, X\multimap X, X\multimap X\vdash X}\,{\multimap}\mathsf{L}}{X, X\multimap X, X\multimap X, X\multimap X\vdash X}\,{\multimap}\mathsf{L}}{X\multimap X, X\multimap X, X\multimap X\vdash X\multimap X}\,{\multimap}\mathsf{R}}{X\multimap X, X\multimap X, !(X\multimap X)\vdash X\multimap X}\,?\mathsf{D}}{\S(X\multimap X), \S(X\multimap X), !(X\multimap X)\vdash \S(X\multimap X)}\,\S}{\mathsf{bint}_X, \mathsf{bint}_X, (!(X\multimap X))^{(4)}, !(X\multimap X)\vdash \S(X\multimap X)}\,{\multimap}\mathsf{L}}{!(X\multimap X), (!(X\multimap X))^{(4)}, \mathsf{bint}, \mathsf{bint}\vdash \S(X\multimap X)}\,\forall\mathsf{L}}{!(X\multimap X), (!(X\multimap X))^{(2)}, \mathsf{bint}, \mathsf{bint}\vdash \S(X\multimap X)}\,?\mathsf{C}}
{\dfrac{\dfrac{\dfrac{\mathsf{state}_X, (!(X\multimap X))^{(2)}, \mathsf{bint}, \mathsf{bint}\vdash \S(X\multimap X)}{(!(X\multimap X))^{(2)}, \mathsf{bint}, \mathsf{bint}, \mathsf{state}\vdash \S(X\multimap X)}\,\forall\mathsf{L}}{(!(X\multimap X))^{(2)}, \mathsf{conf}_{\mathsf{bint}}\vdash \S(X\multimap X)}\,\otimes\mathsf{L}}{\vdash \mathsf{conf}_{\mathsf{bint}}\multimap \mathsf{bint}}\,{\multimap}\mathsf{R}}\,{\multimap}\mathsf{L}
$$

Il λ-termine corrispondente a join è

$$\lambda r\otimes l\otimes k.\lambda s_1 s_0 z.k\langle\ldots\langle id, id\rangle,\ldots, id\rangle(r s_1 s_0(l s_1 s_0 z))$$

Nel nostro caso, la tripla di tipo $\mathsf{conf}_{\mathsf{bint}}$ da dare in input a join sarà null'altro che la configurazione finale della macchina di Turing, manipolata in modo da presentare le due metà del nastro sotto forma di stringhe binarie. Dobbiamo dunque costruire una funzione che, dato un oggetto di tipo Tur, restituisca la tripla corrispondente; la derivazione che svolge questo compito verrà denominata $\mathsf{extractstr}$:

$$
\dfrac{\dfrac{\dfrac{\dfrac{\vdots\,\frown_0\,/\,\frown_1}{\mathsf{bint}\vdash\mathsf{bint}}}{\vdash \mathsf{bint}\multimap\mathsf{bint}}\,{\multimap}\mathsf{R}}{\vdash !(\mathsf{bint}\multimap\mathsf{bint})}\,!\mathsf{P}
\qquad
\dfrac{\dfrac{\dfrac{\vdots\,\varepsilon}{\vdash\mathsf{bint}}\quad\dfrac{\dfrac{}{\mathsf{conf}_{\mathsf{bint}}\vdash\mathsf{conf}_{\mathsf{bint}}}\,\mathsf{ax}\quad\dfrac{}{\mathsf{bint}\vdash\mathsf{bint}}\,\mathsf{ax}}{\mathsf{conf}_{\mathsf{bint}}, \mathsf{conf}_{\mathsf{bint}}\multimap\mathsf{bin}\vdash\mathsf{bint}}\,{\multimap}\mathsf{L}}{\mathsf{bint}\multimap\mathsf{bint}\multimap\mathsf{conf}_{\mathsf{bint}}, \mathsf{conf}_{\mathsf{bint}}\multimap\mathsf{bin}\vdash\mathsf{bint}}\,{\multimap}\mathsf{L}}{\S(\mathsf{bint}\multimap\mathsf{bint}\multimap\mathsf{conf}_{\mathsf{bint}}), \S(\mathsf{conf}_{\mathsf{bint}}\multimap\mathsf{bin})\vdash\S\mathsf{bint}}\,\S}
{\dfrac{\mathsf{Tur}_{\mathsf{bint}}, \S(\mathsf{conf}_{\mathsf{bint}}\multimap\mathsf{bin})\vdash\S\mathsf{bint}}{\S(\mathsf{conf}_{\mathsf{bint}}\multimap\mathsf{bin}), \mathsf{Tur}\vdash\S\mathsf{bint}}\,\forall\mathsf{L}}\,{\multimap}\mathsf{L}
$$

Nella derivazione si fa riferimento alle due derivazioni $\frown_0$ e $\frown_1$ introdotte sopra e alla derivazione ε che corrisponde alla stringa binaria vuota. Il λ-termine corrispondente a $\mathsf{extractstr}$ è

$$f(t\ succ_1\ succ_0\ succ_0\ nil\ nil)$$

dove f è la variabile libera di tipo $\S(\mathsf{conf}_{\mathsf{bint}} \multimap \mathsf{bint})$, t la variabile libera di tipo Tur, $succ_0$ e $succ_1$ i termini corrispondenti alle derivazioni $\frown_0$ e $\frown_1$ e nil il termine corrispondente alla derivazione ε, ovvero la stringa vuota. Si nota subito che la funzione non fa nient'altro che "appiattire" le due sezioni del nastro, convertendo tutti gli eventuali *blank* in 0.

A questo punto non resta che comporre le due funzioni per mezzo di un *cut*, e ottenere finalmente la $\mathsf{getresult}$:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\ \mathsf{join}\\ \vdash \mathsf{conf}_{\mathsf{bint}} \multimap \mathsf{bin}\end{array}}{\vdash \S(\mathsf{conf}_{\mathsf{bint}} \multimap \mathsf{bin})}\,\S \qquad \begin{array}{c}\vdots\ \mathsf{extractstr}\\ \S(\mathsf{conf}_{\mathsf{bint}} \multimap \mathsf{bin}), \mathsf{Tur} \vdash \S\mathsf{bint}\end{array}}{\mathsf{Tur} \vdash \S\mathsf{bint}}\,\mathsf{cut}$$

La macchina di Turing

Siamo ora in grado di mettere insieme tutti i pezzi del "puzzle", in modo da costruire finalmente la simulazione in $\overline{\mathbf{LLL}}$ dell'esecuzione della nostra macchina di Turing. Tale esecuzione può essere descritta come segue: la macchina $\mathbf{M}$ che stiamo prendendo in considerazione parte con una stringa binaria memorizzata sul nastro, la cui lunghezza indicheremo con n, resta in esecuzione per un numero di passi inferiore a $p(n)$, dove $p(x)$ è un qualche polinomio a coefficienti interi non negativi di grado d, e poi termina lasciando sul nastro un'altra stringa binaria, che costituisce l'output del calcolo. Nel complesso, questo processo di calcolo rappresenta una funzione da stringhe binarie a stringhe binarie, che nel nostro sistema dovrebbe vedersi assegnata una derivazione di tipo $\mathsf{bint} \vdash \S^k\mathsf{bint}$, per qualche $k \geq 0$.

Vediamo dunque come costruire tale derivazione. Anzitutto, la $\mathsf{startup}$ provvede ad inizializzare la macchina con la stringa di input. Un'altra funzione, che non abbiamo ancora considerato, deve prendere lo stesso input e restituirne la lunghezza, sotto forma di intero in notazione unaria. In questo modo, la lunghezza può essere data in ingresso alla derivazione $[\![p(x)]\!]$, che rappresenta il polinomio che limita il *runtime* della macchina; il valore restituito da tale polinomio può essere ora dato in input alla funzione $\mathsf{stepall} := \mathsf{It}(\mathsf{step}, \mathsf{startup})$, la quale itera la funzione step il numero di volte indicato, a partire dalla configurazione iniziale. Una volta terminata l'iterazione, la configurazione risultante può essere data in input alla $\mathsf{getresult}$, che restituisce la stringa binaria che costituisce l'uscita di tutto il processo. Ciò conclude la simulazione: avevamo una stringa binaria in input e abbiamo prodotto una stringa binaria in output.

Manca quindi da definire solo la funzione che restituisce la lunghezza di una stringa binaria. La seguente derivazione, che chiamiamo len, è quella che cerchiamo:

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$!(X \multimap X) \vdash !(X \multimap X)$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$!(X \multimap X) \vdash !(X \multimap X)$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$X \vdash X$}
\RightLabel{$\multimap$L}
\BinaryInfC{$X, X \multimap X \vdash X$}
\RightLabel{$\multimap$R}
\UnaryInfC{$X \multimap X \vdash X \multimap X$}
\RightLabel{§}
\UnaryInfC{$§(X \multimap X) \vdash §(X \multimap X)$}
\RightLabel{$\multimap$L}
\BinaryInfC{$\mathsf{int}_X, !(X \multimap X) \vdash §(X \multimap X)$}
\RightLabel{$\multimap$L}
\BinaryInfC{$\mathsf{bint}_X, !(X \multimap X), !(X \multimap X) \vdash §(X \multimap X)$}
\RightLabel{$\forall$L}
\UnaryInfC{$!(X \multimap X), !(X \multimap X), \mathsf{bint} \vdash §(X \multimap X)$}
\RightLabel{?C}
\UnaryInfC{$!(X \multimap X), \mathsf{bint} \vdash §(X \multimap X)$}
\RightLabel{$\multimap$R}
\UnaryInfC{$\mathsf{bint} \vdash \mathsf{int}_X$}
\RightLabel{$\forall$R}
\UnaryInfC{$\mathsf{bint} \vdash \mathsf{int}$}
\end{prooftree}
$$

Il λ-termine corrispondente a len è $\lambda sz.xssz$, dove x è di tipo bint: è chiaro che len non fa null'altro che "appiattire" una lista binaria in una unaria.

La derivazione che corrisponde alla costruzione descritta per l'esecuzione di una macchina di Turing è dunque la seguente:

$$
\begin{prooftree}
\AxiomC{$\vdots\ \mathsf{len}$}
\noLine
\UnaryInfC{$\mathsf{bint} \vdash \mathsf{int}$}
\RightLabel{?D}
\UnaryInfC{$!\mathsf{bint} \vdash \mathsf{int}$}
\RightLabel{!P}
\UnaryInfC{$!\mathsf{bint} \vdash !\mathsf{int}$}
\AxiomC{$\vdots\ [\![p(x)]\!]$}
\noLine
\UnaryInfC{$!\mathsf{int} \vdash §^{d+4}\mathsf{int}$}
\RightLabel{cut}
\BinaryInfC{$!\mathsf{bint} \vdash §^{d+4}\mathsf{int}$}
\AxiomC{$\vdots\ \mathsf{It(step, startup)}$}
\noLine
\UnaryInfC{$!\mathsf{bintint} \vdash §\mathsf{Tur}$}
\RightLabel{§}
\UnaryInfC{$§^{d+4}\mathsf{int}, §^{d+4}!\mathsf{bint} \vdash §^{d+5}\mathsf{Tur}$}
\RightLabel{cut}
\BinaryInfC{$!\mathsf{bint}, §^{d+4}!\mathsf{bint} \vdash §^{d+5}\mathsf{Tur}$}
\AxiomC{$\vdots\ \mathsf{getresult}$}
\noLine
\UnaryInfC{$\mathsf{Tur} \vdash §\mathsf{bint}$}
\RightLabel{§}
\UnaryInfC{$§^{d+5}\mathsf{Tur} \vdash §^{d+6}\mathsf{bint}$}
\RightLabel{cut}
\BinaryInfC{$!\mathsf{bint}, §^{d+4}!\mathsf{bin} \vdash §^{d+6}\mathsf{bint}$}
\RightLabel{§}
\UnaryInfC{$§!\mathsf{bint}, §^{d+5}!\mathsf{bin} \vdash §^{d+7}\mathsf{bint}$}
\RightLabel{Cuts with coercions}
\UnaryInfC{$\mathsf{bint}, \mathsf{bint} \vdash §^{d+7}\mathsf{bint}$}
\RightLabel{?D}
\UnaryInfC{$!\mathsf{bint},\ !\mathsf{bint} \vdash §^{d+7}\mathsf{bint}$}
\RightLabel{§}
\UnaryInfC{$!\mathsf{bint},\ !\mathsf{bint} \vdash §^{d+8}\mathsf{bint}$}
\RightLabel{?C}
\UnaryInfC{$!\mathsf{bint} \vdash §^{d+8}\mathsf{bint}$}
\RightLabel{§}
\UnaryInfC{$§!\mathsf{bint} \vdash §^{d+9}\mathsf{bint}$}
\RightLabel{Cut with coercion}
\UnaryInfC{$\mathsf{bint} \vdash §^{d+9}\mathsf{bint}$}
\end{prooftree}
$$

Se x è la stringa binaria in ingresso (di tipo bint), $len[x]$ e $getresult[t]$ i λ-termini corrispondenti alle omonime derivazioni, $poly[n]$ il λ-termine associato al polinomio limite e $stepall[n, x]$ il termine che, presa la stringa x, costruisce la configurazione iniziale per mezzo di startup e la itera n volte, allora il λ-termine corrispondente alla derivazione presentata è

$$getresult[stepall[poly[len[x]], x]]$$

che simula l'esecuzione della macchina di Turing esattamente nel modo descritto.

Siamo allora pronti ad enunciare il risultato finale di questa sezione:

Teorema 9.9 (PTIME-completezza) *Se $\boldsymbol{M}$ è una macchina di Turing deterministica che, data in input la stringa binaria x, produce in output la stringa binaria y in un numero di passi $p(|x|)$, dove $|x|$ è la lunghezza di x e p un polinomio di grado d, allora esistono una derivazione ξ del sequente*

$$\vdash \mathsf{bint}$$

e una derivazione π del sequente

$$\mathsf{bint} \vdash \S^{d+9}\mathsf{bint}$$

le cui proof-net corrispondenti $\overline{\xi}$ e $\overline{\pi}$ fanno parte di $\overline{\mathbf{LLL}}$ e:

- *$\overline{\xi}$ è una rappresentazione della stringa x;*
- *la proof-net ottenuta tagliando la conclusione di tipo bint di $\overline{\xi}$ con la conclusione di tipo $\mathsf{bint}^{\perp}$ di $\overline{\pi}$ si riduce ad una proof-net cut-free $\overline{\upsilon}$, con conclusione un'occorrenza della formula $\S^{d+9}\mathsf{bint}$, tale che $\overline{\upsilon}$ è una rappresentazione della stringa y.*

Dimostrazione. Segue da quanto mostrato nel corso della sezione. □

Infine, il teorema di equivalenza tra $\overline{\mathbf{LLL}}$ e la classe **PTIME**:

Teorema 9.10 *Una funzione è calcolabile in tempo polinomiale da una macchina di Turing deterministica se e soltanto se è rappresentabile da una proof-net di $\overline{\mathbf{LLL}}$.*

Dimostrazione. Corollario del teorema 9.9 appena introdotto e del teorema 9.8. □

Bibliografia

[1] A. Asperti. **Light Affine Logic**. In *Proceedings of Symposium on Logic in Computer Science LICS'98*, 1998.

[2] A. Asperti, L. Roversi. **Light Affine Logic: proof-nets, programming notation, P-Time correctness and completeness**. Submitted, 2000.

[3] D. Bechet. **Second Order Connectives and Proof Transformations in Linear Logic**. Presented at the colloquium *Preuves, Réseaux et Types* held in Marseille, France, October 1994, and to appear as a prepublication Université Paris 13, 1999.

[4] S. Bellantoni, S. Cook. **A New Recursion-Theoretic Characterization of the Polytime Functions**. *Computational Complexity*, 2:97-110, 1992.

[5] H. Schwichtenberg, A. S. Troelstra. **Basic Proof Theory**. Cambridge Tracts in Theoretical Computer Science, vol. 43, Cambridge University Press, 2000

[6] A. Cobham, **The intrinsic computational difficulty of functions**. in Y. Bar-Hillel ed., *Proc. of the 1964 International Congress for Logic, Methodology, and the Philosophy of Science*, 24-30, North Holland, Amsterdam, 1964.

[7] R. Cori, D. Lascar. **Logique mathématique**. Axiomes, Masson, 1993.

[8] V. Danos, J.-B. Joinet. **Linear Logic & Elementary Time**. *Proceedings of ICC '99*, 1999.

[9] V. Danos, L. Regnier. **The Structure of Multiplicatives**. *Archives for Mathematical Logic*, 28:181-203 1989.

[10] J.-Y. Girard. **Linear Logic**. *Theoretical Computer Science*, 50:1-102, 1987.

[11] J.-Y. Girard. **Proof-nets: the parallel syntax for proof-theory**. In Ursini and Agliano editors, *Logic and Algebra*, Marcel Dekker, New York, 1995.

[12] J.-Y. Girard. **Light Linear Logic**. *Information and Computation*, 14(3):175-204, 1998.

[13] J.-Y. Girard, P. Taylor, Y. Lafont. **Proofs and Types**. Cambridge Tracts in Theoretical Computer Science, vol. 7, Cambridge University Press, 1989.

[14] S. Guerrini. **Correctness of Multiplicative Proof Nets is Linear**. *14th Annual IEEE Symposium on Logic in Computer Science (LICS '99)*, Trento, Italy, 1999.

[15] H. Herbelin. **Séquents qu'on calcule**. Thèse de doctorat, Paris, 1995.

[16] J.-L. Krivine. **Lambda-calcul. Types et Modèles**. Études et Recherches en Informatique, Masson, 1990.

[17] Y. Lafont. **Soft Linear Logic and Polynomial Time**. To appear in *Theoretical Computer Science*, 2002.

[18] V. P. Orevkov. **Complexity of Proofs and Their Transformations in Axiomatic Theories**. Translations of Mathematical Monographs, vol. 128, American Mathemetical Society, 1991.

[19] C. Papadimitriou. **Computational Complexity**. Addison-Wesley, 1995.

[20] M. Parigot. **On the representation of data in lambda-calculus**. *Proceedings of the third workshop on Computer science logic*, Kaiserslautern, Germany 1990.

[21] H. E. Rose. **Sub-recursion: functions and hierarchy**. Clarendon Press, Oxford, 1999.

[22] L. Roversi. **A P-Time Completeness Proof for Light Logics**. In *Proceedings of CSL'99*, pages 469-483. Springer-Verlag, LNCS 1683, 1999.

[23] R. Statman. **The typed λ-calculus is not elementary recursive**, *Theoretical Computer Science*, 9:73-81, 1979.

[24] K. Terui. **Light Affine Lambda Calculus and Polytime Strong Normalization**. Preprint, 2002.

[25] L. Tortora de Falco. **Réseaux, cohérence et expériences obsessionnelles**. Thèse de doctorat, Paris, 2000.

[26] L. Tortora de Falco. **Additives of Linear Logic and normalization**. To appear in *Theoretical Computer Science*, 2000.