Un Logical Framework Lineare

Iliano Cervesato

Dipartimento di Informatica — Università degli Studi di Torino
Corso Svizzera 182, 10149 Torino — Italy

Tesi di Dottorato di Ricerca in Informatica del VII Ciclo

Supervisore:	Frank Pfenning (*Carnegie Mellon University*)
Revisori:	Alberto Martelli (*Università di Torino*)
	Simona Ronchi della Rocca (*Università di Torino*)
	Mariangiola Dezani (*Università di Torino*)

Questa dissertazione è stata completata durante un'estesa visita dell'autore presso il *Department of Computer Science* della *Carnegie Mellon University*, Pittsburgh, Pennsylvania, USA.

Sommario

Un logical framework è un sistema formale progettato per dare concrete rappresentazioni di sistemi deduttivi e delle loro proprietà. Proposte basate su logiche e teorie dei tipi intuizionistiche, quali il logical framework LF, sono state ampiamente usate per studiare formalismi logici e linguaggi di programmazione. Sfortunatamente, molti costrutti e concetti di uso ricorrente in programmazione non vengono rappresentati in maniera soddisfacente in questi meta-linguaggi. In particolare, costrutti basati sulla nozione di stato, presenti ad esempio nei linguaggi di programmazione imperativi, spesso vengono meno ad un'elegante formalizzazione per mezzo di questi strumenti. Similmente, sistemi logici che, per definizione (quali le logiche substrutturali) o per presentazione (quale il calcolo dei sequenti intuizionistico privo di contrazione di Dyckhoff), sono basati su riduzioni distruttive del contesto richiedono codifiche innaturali in un logical framework intuizionistico. Conseguentemente, l'adeguatezza della rappresentazione risulta difficile da dimostrare, e la meta-teoria formale diventa rapidamente intrattabile.

La logica lineare permette una visione delle formule nel contesto come risorse, la quale può essere sfruttata per modellare la nozione di stato. Le proposte attuali, *Forum* ad esempio, mettono l'accento sul *rappresentare* costrutti imperativi e logiche basate su risorse, ma appaiono inadeguati per *ragionare* concretamente su queste rappresentazioni. D'altra parte, sistemi basati su teorie dei tipi intuizionistiche quali LF rendono semplice la meta-rappresentazione del ragionamento, ma non offrono alcuna nozione di linearità.

Proponiamo un *Logical Framework Lineare*, battezzato LLF, che combina il potere espressivo dei tipi dipendenti e l'abilità della logica lineare di modellare concretamente formalismi con stato. LLF estende la teoria dei tipi sottostante a LF con operatori lineari, visti in questo contesto come costruttori di tipi, e i conseguenti costruttori e distruttori al livello oggetto. Questo permette di raffinare la metodologia di meta-rappresentazione che caratterizza questo formalismo per inglobare sistemi deduttivi con stato. LLF è stato usato per dare concrete rappresentazioni a proprietà di linguaggi di programmazione imperativi, ad aspetti di logiche e λ-calcoli lineari, e ad alcuni giochi. Essendo un'estensione conservativa di LF, LLF eredita inoltre la ricca libreria di esempi codificati con successo per mezzo di questo formalismo. LLF si presenta inoltre adeguato per un'efficiente implementazione come un linguaggio di programmazione logica nello stile di *Elf*.

I principali contributi di questo lavoro sono: (1) la definizione di una teoria dei tipi uniforme che ammetta entità lineari in congiunzione con tipi dipendenti; (2) l'uso di questo sistema come un logical framework per rappresentare e ragionare su problemi che non sono trattati in maniera

soddisfacente da formalismi precedenti, sia lineari che intuizionistici; (3) la descrizione di una semantica operazionale che costituisce il primo passo verso l'implementazione di questo formalismo sotto forma di un linguaggio di programmazione logica lineare.

Tabella dei Contenuti

1 Introduzione **1**

1.1 Prospettiva Storica 7

1.2 Contributi 15

1.3 Organizzazione della Tesi 17

I Nozioni Preliminarie **19**

2 Nozioni di Logica e Teoria dei Tipi **21**

2.1 La Logica Tradizionale 21

2.2 La Logica Lineare 28

2.3 La Teoria dei Tipi Traditionale 30

2.4 La Teoria dei Tipi Lineare 31

3 Programmare con la Logica e la Teoria dei Tipi **33**

3.1 Progettazione di Linguaggi di Programmazione Logica Basata sulla Teoria della Dimostrazione 33

3.1.1 Proof-search Guidata dal Goal 34

3.1.2 Focalizzazione 35

3.1.3 Linguaggi di Programmazione Logica Astratti 35

3.1.4 Linguaggi di Programmazione Logica Concreti 36

3.2 Un Esempio dalla Logica Lineare 36

3.2.1 Le Formule di Harrop Ereditarie Lineari 36

3.2.2 Il Linguaggio *Lolli* 38

3.3 Un Esempio dalla Teoria dei Tipi 38

3.3.1 Il Logical Framework *LF* 38

3.3.2 Il linguaggio *Elf* 40

4 Meta-representazione **43**
4.1 Meta-rappresentazione di Linguaggi Formali 43
4.1.1 Struttura di un Sistema Formale 44
4.1.2 Anatomia di un Sistema Deduttivo 46
4.1.3 Rappresentazione di Linguaggi Formali 48
4.2 Meta-rappresentazione nel Logical Framework *LF* 49
4.3 Il Linguaggio *Mini-ML* 51
4.3.1 Espressioni 51
4.3.2 Tipizzazione 52
4.3.3 Valutazione 52
4.3.4 Preservazione dei Tipi 54
4.4 Rappresentazione di Aspetti di *Mini-ML* in *LF* 55
4.4.1 Sintassi 55
4.4.2 Semantica: Tipizzazione e Valutazione 60
4.4.3 Meta-teoria: Il Teorema di Preservazione dei Tipi 61

II Progettazione di un Logical Framework Basato sulla Logica Lineare **67**

5 La Teoria dei Tipi **69**
5.1 Il Linguaggio 71
5.2 Operazioni di Base 71
5.3 Forme Pre-canoniche 73
5.3.1 Presentazione 74
5.3.2 La Teoria Equazionale 75
5.3.3 Proprietà fondamentali 77
5.3.4 Normalizzazione Forte 80
5.3.5 Il Sistema Algoritmico 87
5.3.6 Decidibilità 89
5.4 Forme Canoniche 91
5.5 Programmazione logica in *LLF* 94
5.5.1 Dimostrazioni canoniche 95
5.5.2 Dimostrabilità Uniforme 96
5.5.3 Risoluzione 97
5.5.4 Il Non-determinismo 100

6 La Metodologia della Meta-rappresentazione Lineare **103**

6.1 Meta-rappresentazione in *LLF* . . . 103
6.2 Sintassi Concreta di *LLF* . . . 105
6.3 *Mini-ML* Imperativo . . . 107
6.3.1 *Mini-ML* con Referenze . . . 107
6.3.2 Rappresentazione in *LLF* . . . 112
6.3.3 Preservazione dei Tipi . . . 115
6.4 Eliminazione del Taglio per le Formule di Harrop Ereditarie Lineari . . . 121
6.5 Mahjongg . . . 127

III Conclusioni **135**

7 Conclusioni e Sviluppi Futuri **137**
7.1 Conclusions . . . 137
7.2 Sviluppi Futuri . . . 138
7.2.1 Aspetti Teorici . . . 138
7.2.2 Implementazione . . . 138
7.2.3 Applicazioni . . . 138
7.2.4 Oltre *LLF* . . . 139

A *Mini-ML* **141**
A.1 *Elf* code for *Mini-ML* . . . 141
A.1.1 Syntax . . . 141
A.1.2 Typing . . . 142
A.1.3 Evaluation . . . 144
A.1.4 Type Preservation . . . 145
A.2 Adequacy Theorems . . . 147
A.2.1 Syntax . . . 147
A.2.2 Typing . . . 148
A.2.3 Evaluation . . . 148
A.2.4 Type Preservation . . . 149

B Proofs from Chapter 5 **151**
B.1 Language . . . 151
B.2 Basic Operations . . . 151
B.3 Pre-canonical Forms . . . 154
B.3.1 Presentation . . . 154

B.3.2 Equational Theory 154
B.3.3 Fundamental Properties 160
B.3.4 Strong Normalization 169
B.3.5 Algorithmic System 178
B.3.6 Decidability 180
B.4 Canonical Forms 184
B.5 Logic Programming in LLF 188
B.5.1 Canonical Proofs 188
B.5.2 Uniform Provability 188
B.5.3 Resolution 189

C *Mini-ML* with References **191**
C.1 *LLF* Source Code 191
C.1.1 Syntax 191
C.1.2 Typing 193
C.1.3 Evaluation 196
C.1.4 Type Preservation 198
C.2 Adequacy Theorems 202
C.2.1 Syntax 202
C.2.2 Typing 203
C.2.3 Evaluation 204
C.2.4 Type Preservation 205

D Linear Hereditary Harrop Formulas **207**
D.1 *LLF* Source Code 207
D.1.1 Formulas 207
D.1.2 Provability 207
D.1.3 Admissibility of `cut` and `cut!` 209
D.1.4 Cut Elimination 211
D.2 Adequacy Theorems 212
D.2.1 Formulas 213
D.2.2 Provability 213
D.2.3 Admissibility of `cut` and `cut!` 213
D.2.4 Cut Elimination 215

Bibliografia **216**

Lista delle Figure

2.1 La Logica Classica 22
2.2 La Logica Minimale 23
2.3 Deduzione Naturale per la Logica Classica 24
2.4 La Logica Lineare Classica, Additivi e Moltiplicativi 25
2.5 La Logica Lineare Classica, Esponenzialie e Quantificatori 26
2.6 La Logica Lineare Minimale, Additivi e Moltiplicativi 27
2.7 La Logica Lineare Minimale, Esponenziali ed Intuizionistici 28
2.8 Deduzione Naturale per la Logica Minimale Lineare, Additivi e Moltiplicativi . . . 29
2.9 Deduzione Naturale per la Logica Minimale Lineare, Esponenziali ed Intuizionistici 30
2.10 Il λ-calcolo Semplicemente Tipato di Church $\lambda^{\rightarrow}$ 30
2.11 Il λ-calcolo Semplicemente Tipato Lineare di Church $\lambda^{\multimap}$ 31

3.1 Formule di Harrop Ereditarie e Clausole di Horn 34
3.2 Formule di Harrop Ereditarie Lineari 37
3.3 Sistema Deduttivo Canonico per *LF* 39

4.1 Organizzazione della Presentazione delle Formule di Harrop Ereditarie 44
4.2 Schemi di Regole d'Inferenza e la loro Rappresentazione in Varie Logiche 47
4.3 Metodologia di Rappresentazione in *LF* 50
4.4 Regole di Tipizzazione per *Mini-ML* 52
4.5 Valutazione Basata su Continuazioni per *Mini-ML* 53
4.6 Regole di Tipizzazione per *Mini-ML*, Istruzioni e Continuazioni 54

5.1 Partizione del Contesto 72
5.2 Sistema Pre-canonico per *LLF*, Kind e Tipi 75
5.3 Sistema Pre-canonico per *LLF*, Oggetti 76
5.4 Riduzione Parallela Nidificata per *LLF* 77
5.5 Riduzione Parallela Nidificata per *LLF*, Chiusura Transitiva 78

5.6 Riduzione Parallela Nidificata per *LLF*, Equivalenza 78
5.7 Riduzione One-step per *LLF* 81
5.8 Riduzione One-step per *LLF*, Chiusura Transitiva 82
5.9 Riduzione One-step per *LLF*, Equivalenza 82
5.10 Il λ-calcolo semplicemente tipato con coppie $\lambda^{\times\rightarrow}$ 84
5.11 Sistema algoritmico per *LLF* 88
5.12 Sistema Canonico per *LLF* 92
5.13 Sviluppo di Sistemi di Programmazione Logica per *LLF* 95
5.14 Sistema di Deduzione Uniforme per *LLF* 97
5.15 Sistema di Risoluzione per *LLF* 98

6.1 Metodologia di Rappresentazione in *LLF* 104
6.2 Regole di Tipizzazione per *MLR*, Espressioni 108
6.3 Regole di Tipizzazione per *MLR*, Istruzioni e Continuazioni 109
6.4 Regole di Tipizzazione per *MLR*: Memoria e Risposte 110
6.5 Valutazione in *MLR*, Espressioni 111
6.6 Valutazione in *MLR*, Valori ed Istruzioni Ausiliarie 112
6.7 Il Frammento Liberamente Generato delle Formule di Harrop Ereditarie Lineari . 122
6.8 Tasselli del Gioco Originale Mahjongg 128
6.9 Una Configurazione Iniziale in Mahjongg 129
6.10 Una Configurazione Iniziale nel Problema dell'Accoppiamento dei Calzini 131

Capitolo 1

Introduzione

I linguaggi formali furono originariamente sviluppati in ambito logico allo scopo di dare rigorosi fondamenti alla matematica e alla logica stessa. La loro popolarità crebbe in maniera significativa con l'avvento dell'Informatica come disciplina scientifica, e degli elaboratori elettronici come realtà economica. Con sporadiche eccezione derivanti dai progressi dell'Intelligenza Artificiale, l'interazione con sistemi informatici avviene infatti attraverso comandi e linguaggi di programmazione caratterizzati da una sintassi rigida la quale dà origine ad operazioni non ambigue.

I sistemi formali sono stati per un lungo tempo descritti informalmente (come quando si insegna un corso introduttivo di linguaggi di programmazione), o semi-formalmente (come nella maggior parte dei libri di logica matematica). Presentazioni semi-formali di linguaggi formali generici spesso assumono la forma di *sistemi deduttivi* (si veda [ML85a, ML85b] per le motivazioni filosofiche e lo sviluppo storico di questa nozione). In queste presentazioni, la nozione di *giudizio*, definito come una relazione tra elementi lessicali del linguaggio, gioca un ruolo centrale. I giudizi permettono di definire aspetti di un linguaggio formale quali la sua *sintassi* (ad esempio *e è un'espressione*, oppure *A è una formula ben formata*) e la sua *semantica* (ad esempio *A è una formula valida*, o *e ha tipo* τ, oppure *e ha valore* v). Possono anche essere usati per esprimere aspetti della *meta-teoria* del linguaggio (ad esempio *i tipi non cambiano durante la valutazione*). I giudizi sono spesso specificati per mezzo di *regole d'inferenza* schematiche (ad esempio $A \wedge B$ *è una formula valida se sia A che B sono formule valide*). Istanze di regole vengono concatenate per fornire evidenza della validità di giudizi sotto forma di *derivazioni*. Si noti che aspetti concettualmente differenti di un linguaggio formale, quali la sua sintassi, la sua semantica e la sua meta-teoria, vengono specificati in maniera uniforme come giudizi in un sistema deduttivo. Similmente, i sistemi deduttivi permettono di dare specifiche uniformi a linguaggi formali possibilmente distanti quali logiche e linguaggi di programmazione di vario tipo.

I primi tentativi di formalizzare sistemi deduttivi avevano motivazioni teoriche: ad esempio, la codifica di Gödel delle formule logiche del prim'ordine come numeri naturali servirono il proposito di dimostrare i suoi famosi teoremi d'incompletessa [Göd31]. Oggi, la formalizzazione di sistemi deduttivi ha scopi molto pratici e, come cercheremo di sostanziare con alcuni esempi, metodi efficaci per raggiungere quest'obiettivo diventeranno presto un'assoluta necessità.

Dal punto di vista dei linguaggi di programmazione, un compilatore trasforma un programma P scritto in un linguaggio sorgente (solitamente di alto livello) in un programma P' in un linguaggio destinazione (normalmente diverso, di livello piú basso). La traduzione attuata dal compilatore è corretta se P e P' manifestano un identico comportamento. Attualmente, la correttezza è accertata empiricamente: non appena una prima implementazione è disponibile, è collaudata su un ampio campione di programmi di prova in cerca di comportamenti anomali, e poi riconsegnato al programmatore per correggerli. Il ciclo di collaudo e correzione procede fintantoché il compilatore non si comporta nella maniera attesa. Chiaramente, questa evidenza statistica di correttezza non garantisce la correttezza reale a causa dell'inevitabile incompletezza del campione di programmi. Metodi concreti di formalizzare i giudizi che descrivono i linguaggi sorgente e destinazione nonché le trasformazione effettuate dal compilatore, sono prerequisiti per dare dimostrazioni formali della correttezza del processo di compilazione. Inoltre, se questi metodi possono essere meccanizzati, la correttezza del compilatore può essere controllata automaticamente. I benefici di quest'approccio sono evidenti poiché non solo eliminerebbe il ciclo di collaudo e correzione e i notevoli costi che ne derivano, ma garantirebbe anche la piena correttezza del compilatore, che non si può ottenere con i metodi attuali.

Simili considerazioni si applicano alla progettazione di componenti hardware quali microprocessori. Le specifiche funzionali di alto livello di un chip vengono "compilate" in un sistema di porte logiche. Un errore durante questa fase può portare a disastri economici, come accadde nel 1994 con il processore *Pentium* difettoso della *Intel*. Invece, la formalizzazione di questo processo garantisce la correttezza del chip fisico rispetto alle specifiche di alto livello. Il sistema *NQTHM*, meglio noto come *dimostratore di teoremi di Boyer-Moore* [BM78, BM88], è stato usato per verificare a posteriori la correttezza dei processori *Motorola MC68020* e *DEC Alpha*, e per specificare da capo un chip *Motorola* [Moo94].

Il compito dei logici moderni è angosciante a causa della complessità che i sistemi logici hanno acquistati per guadagnare in espressività. Uno studente universitario già sperimenta questa realtà in corsi introduttivi di logica matematica quando viene confrontato con i 27 o piú casi (in dipendenza della formulazione) della dimostrazione del teorema di eliminazione del taglio di Gentzen [Pfe95b, Sza70]. L'intera logica lineare raddoppia il numero dei connettivi e quadruplica il numero di casi da analizzare: pochi ricercatori hanno tentato dimostrazioni diretto dell'eliminazione del taglio per questo formalismo [GP94, Pfe94b, Roo91]. La logica generale LU [Gir93] differenza ulteriormente questi connettivi (ad esempio la disgiunzione è sgretolata in *nove* micro-connettivi) e, a nostra conoscenza, nessuno ha intrapreso una dimostrazione di questa proprietà per questa logica.

Se, ad esempio, una dimostrazione di eliminazione del taglio per LU verrà mai fornita, sarà cosí lunga che numerosi errori rischiano di passare inosservati, anche al lettore piú attento. Anche se questa dimostrazione è corretta, non è detto che ci fideremo dei nostri poteri di concentrazione e la rifiuteremo come supporto alla validità del teorema di eliminazione del taglio per LU. Similmente, credere che tutte le dimostrazione in questa tesi siano corrette sarebbe un atto di fede cieca. È infatti probabile che numerosi errori (minori si spera) siano passati inosservati anche dopo le numerose riletture. Abbiamo forse raggiunto il punto nello sviluppo della logica in cui dobbiamo

tristamente discordare con Kreisel quando dichiara (sebbene in un contesto leggermente differente) che "riconosciamo una dimostrazione quando ne vediamo una" [ML73]?

L'abilità di formalizzare sistemi logici porta alla possibilità di meccanizzare il processo di verifica della validità di una dimostrazione. Pertanto, se riusciamo a formalizzare una dimostrazione, eventualmente piú lunga di quanto la mente di una persona normale possa gestire (quale il nostro esempio riguardante LU), possiamo fare verificare la validità di questa dimostrazione ad un computer. Sosteniamo che verificatori automatici di teoremi stanno diventando una necessità. Inoltre, il lavoro dei logici verrebbe grandemente semplificati dalla disponibilità di generatori automatici di teoremi. Ritornando al nostro esempio, è verosimile che molti casi in una dimostrazione di eliminazione del taglio per LU siano immediati e possano essere generati facilmente, mentre solo pochi richiedono un intervento umano. Questa tesi è piena di teoremi e lemmi validi sulla base di semplici induzioni. L'aiuto di un generatore di dimostrazione avrebbe semplificato il nostro compito, permesso un lavoro piú completo e garantito l'assenza di errori. Di nuovo, un prerequisito alla costruzione di strumenti di sviluppo di dimostrazioni è la possibilità di formalizzare sistemi formali.

Simili osservazioni si applicano anche all progettazione di linguaggi di programmazione. Infatti, linguaggi moderni quali *ML* [MTH90] sono formalismi attentamente costruiti che combinano un gran numero di aspetti in un sistema armonico. Le intuizioni iniziali devono essere formalizzate come proprietà del linguaggio e verificate allo scopo di evitare difetti di progettazione. Pertanto il compito del progettista di linguaggi di programmazione assomiglia all'attività del logico, e entrambi hanno bisogno di ambienti di sviluppo simili per migliorare la qualità del loro lavoro. Una notevole parte di questa tesi, ad esempio, è dedicata al dimostrare che il formalismo che proponiamo può essere adeguatamente visto come un linguaggio di programmazione.

La formalizzazione concreta di un sistema deduttivo $\mathcal{L}_o$ consiste di due passi. Innanzitutto, dobbiamo fornire un sistema deduttivo $\mathcal{L}_\mu$ adeguato a *rappresentare* $\mathcal{L}_o$ e a catturare le forme di *ragionamento* che intendiamo modellare. $\mathcal{L}_\mu$ e $\mathcal{L}_o$ vengono convenzionalmente chiamati il *meta-linguaggio* e il *linguaggio oggetto* rispettivamente. Si noti che $\mathcal{L}_\mu$ deve essere in grado di dare una rappresentazione a tutti i giudizi e derivazioni di $\mathcal{L}_o$ che prendono parte alla specifica delle proprietà che desideriamo formalizzare; questi, in generale, non sono limitati alla definizione della sintassi di questo linguaggio.

Inoltre, dobbiamo adottare una *metodologia di meta-rappresentazione* che governi la codifica di derivazioni e giudizi di $\mathcal{L}_o$ in $\mathcal{L}_\mu$. La rappresentazione del sistema oggetto deve essere *adeguata* alla forma di meta-ragionamento che intendiamo fare. Con ciò, vogliamo dire che ogni entità di interesse al livello oggetto deve essere codificata in maniera distinta, e che deve essere possibile decidere se un'espressione del meta-linguaggio è o meno la codifica di un qualche giudizio o derivazione del livello oggetto. Crediamo che ogni formalizzazione di un sistema oggetto deve essere accompagnato da una dimostrazione dell'*adeguatezza* dello schema di meta-rappresentazione su cui si basa. Questo passo viene spesso ignorato, in generale perché il meta-linguaggio e il linguaggio oggetto sono cosí distanti che una dimostrazione semi-formale di questa corrispondenza è troppo laboriosa.

Ritornando ai nostri esempi, un compilatore verrebbe rappresentato da una funzione che mappa operazioni nel linguaggio sorgente in costruzioni semantiche nel linguaggio destinazione. Entrambe sono espresse come (sequenze di) regole di inferenze rispetto alle quali sono stati astratti i dati concreti su cui dovrebbero operare. Verificare la correttezza del compilatore si riduce alla dimostrazione che i singoli schemi di traduzione adottati associano a costrutti sorgenti sequenze di istruzioni nel linguaggio destinazione che sono equivalenti rispetto ad un qualche modello di riferimento. Si noti che programmi sorgente e destinazioni reali non vengono mai presi in considerazione: l'adeguatezza della rappresentazione garantisce che la codifica di ogni costrutto astratto riflette la sua semantica in programmi oggetto concreti.

Similmente, una dimostrazione di eliminazione del taglio per LU verrebbe rappresentata come una relazione tra una derivazione generica e una derivazione priva di tagli. La dimostrazione è corretta se questa relazione associa ad ogni derivazione valida per un sequente dato una derivazione priva di tagli per lo stesso sequente. Di nuovo, ciò che viene effettivamente verificato è che i singoli schemi di eliminazione del taglio siano corretti, e l'adeguatezza della rappresentazione garantisce che questa correttezza si estende a derivazioni concrete.

La scelta di un sistema di rappresentazione $\mathcal{L}_\mu$ e di una metodologia di codifica è importante allo scopo di ottenere formalizzazioni concretamente utilizzabili. Dalla discussione precedente, appare che una proprietà desiderabile per un meta-sistema è la possibilità di *automatizzare* le operazioni che mette a disposizione per fare meta-ragionamento. Questa richiesta porta ad escludere popolari candidati quali la semantica denotazionale [Gun92], in quanto basati su linguaggi, quali logiche di punto fisso e teoria degli insiemi, i cui costrutti sono difficili da meccanizzare per scopi di meta-rappresentazione pratici e per il conseguente meta-ragionamento.

I linguaggi di programmazione general-purpose sono automatizzabili, ma spesso mancano degli strumenti adeguati a rendere agevole il compito di meta-programmazione. Si assuma ad esempio di volere rappresentare una dimostrazione di eliminazione del taglio in C. Dobbiamo incominciare da capo definendo adeguate strutture di dati per rappresentare formule e derivazioni, poi fornire del codice per verificare che sono ben formate, per manipolarle, applicare condizioni collaterali circa le variabili libere e legate, effettuare sostituzioni, ecc. Anche per una logica semplice, il programma risultante è grosso e probabilmente ancora piú complesso della dimostrazione la cui correttezza vogliamo verificare. Se confidiamo nella nostra abilità di scrivere codice corretto e non abbiamo bisogno di convincere formalmente altri, possiamo adottare questo approccio. Tuttavia, se insistiamo sulla possibilità di dimostrare teoremi di adeguatezza, questa non è la strada giusta.

Vorremo invece limitarci a *trascrivere* le dimostrazioni del livello oggetto nel meta-linguaggio senza dover scrivere pagine e pagine di codice ausiliario. Il linguaggio di rappresentazione deve pertanto fornire costrutti in grado di astrarre problemi di basso livello e permettere di concentrarci su aspetti piú vicini al problema in esame. Quest'osservazione ha portato all'idea che, se identifichiamo costrutti ricorrenti in sistemi deduttivi che richiedono complesse procedure ausiliarie in un linguaggio di programmazione general-purpose, potremo *specializzare* il meta-linguaggio per gestirle internamente, alleviando l'onere dall'utente-programmatore. I costrutti che legano variabili (i quantificatori in logica e i parametri formali di funzioni e procedure nei linguaggi di programmazione) esibiscono queste indesiderabili proprietà poiché introducono

complesse regole di visibilità e sono soggette a sostituzioni. Una soluzione a questo problema fu data dall'introduzione degli *indici di de Bruijn* per indicare le variabili [dB72]. Questo metodo viene usato in molti dei linguaggi di meta-rappresentazione descritti qui sotto. Una soluzione alternativa deriva dall'osservazione che i linguaggi contenenti nel loro nucleo un λ-calcolo sono adeguati per rappresentare costrutti che legano variabili [Chu32a, Chu32b]. La λ-astrazione può infatti venire usata per modellare generici costrutti di questo tipo non appena rappresentiamo le variabili oggetto per mezzo di variabili del meta-linguaggi: i problemi riguardanti il nome delle variabili oggetto legate vengono gestiti via α-conversione nel meta-linguaggio e la sostituzione di livello oggetto viene modellata per mezzo dell'applicazione e conseguenti β-riduzione al meta-livello. Questo metodo di rappresentazione è noto come *sintassi astratta di ordine superiore.*

Linguaggi specializzati per dare concrete ed utilizzabili rappresentazioni a sistemi deduttivi vengono chiamati *logical framework* (framework per rappresentare logiche) e cadono in due categorie strettamente correlate. Da un lato, vi sono i formalismi logici basati su una logica di ordine superiore, quale il linguaggio delle *formule di Harrop ereditarie di ordine superiore* [MNPS91]. Dall'altro, varie estensioni verso teorie dei tipi di λ-calcoli [Bar92] offrono potenti strumenti per ragionare su linguaggi oggetto. Fra le numerose proposte che cadono in questa categoria, ricorderemo i linguaggi della famiglia *AUTOMATH* [dB80], le teorie dei tipi costruttive di Martin-Löf [ML80], *LF* [HHP93], il calcolo delle costruzioni [CH88] e le sue estensioni quali il calcolo delle costruzioni induttive [PM93]. Ritorneremo su questi formalismi piú avanti.

Questi meta-linguaggi possiedono differenti proprietà e i ricercatori che li hanno sviluppati erano mossi da diversi obiettivi. La ricca varietà presente nelle loro implementazioni non è pertanto sorprendente. Possiamo comunque distinguere due filoni di ricerca. Da un lato il linguaggio delle formule di Harrop ereditarie di ordine superiore e *LF* sono stati implementati sotto forma dei *linguaggi di programmazione logica* λ*Prolog* [Mil89] e *Elf* [Pfe91, Pfe94a] rispettivamente. D'altro canto, altri formalismi hanno invece portato alla realizzazione di *verificatori di teoremi* (ad esempio *AUTOMATH* [dB80]) oppure di *ambienti interattivi per lo sviluppo di dimostrazioni* con caratteristiche diverse: *Coq* [DFH$^+$93] (calcolo delle costruzioni induttive), *LEGO* [LP92, Pol94] (calcolo delle costruzioni esteso), *ALF* [MN94, Nor93] (teorie dei tipi di Martin-Löf), *NuPrl* [C$^+$86] (prime versioni della teoria dei tipi di Martin-Löf), *Isabelle* [Pau88, Pau93] (formule di Harrop ereditarie di ordine superiore).

Sebbene siano ancora oggetti di intensa ricerca, questi formalismi e le loro implementazioni vengono ormai usati con successo da anni come strumenti di meta-rappresentazione e meta-ragionamento. Numerosi problemi in matematica (ad esempio la formalizzazione del *Foundations of Analysis* di Landin [Jut77]), logica (ad esempio il primo teorema di incompletezza di Gödel [Sha94] e il teorema di eliminazione del taglio per la logica tradizionale [Pfe95b]), teoria dei tipi (ad esempio la proprietà di Church-Rosser per il λ-calcolo non tipato [dB72, Hue94, Nip95, Pfe99, Ras95, Sha88]) e la teoria dei linguaggi di programmazione (ad esempio proprietà del linguaggio di programmazione funzionale *Mini-ML* [MP91]) sono stati dotati di eleganti e utili rappresentazioni in questi formalismi.

Tuttavia, molti costrutti e concetti impiegati nella comune attività di programmazione non possono venire rappresentati in maniera soddisfacente in questi meta-linguaggi. In particolare,

costrutti basati sulla nozione di uno *stato* che evolve con il procedere della computazione, presenti in tutti i linguaggi di programmazione imperativi, sfuggono ad un'elegante formalizzazione per mezzo di questi strumenti. Similmente, sistemi logici che, per definizioni (ad esempio la logica lineare e altre logiche substrutturali) o per presentazione (ad esempio il calcolo dei sequenti intuizionistico di Dyckhoff privo di contrazioni) sono basati su *riduzioni del contesto distruttive*, hanno codifiche poco pratiche in questi logical framework. Queste difficoltà si possono rintracciare nel fatto che i summenzionati meta-linguaggi traggono la loro origine da logiche e teorie dei tipi intuizionistiche tradizionali. Questi formalismi non offrono un supporto diretto alla nozione di uno stato mutabile, come richiesto dai sistemi suddescritti. Una qualche codifica dello stato deve essere passata in giro e manipolata esplicitamente per mezzo di dichiarazioni ausiliarie, ad esempio per accedere o cambiare il valore di una variabile. È difficile mostrare che programmi oggetto vengono codificati correttamente e ragionare su questi oggetti (ossia sulla loro meta-rappresentazione) diventa rapidamente intrattabile [Pfe94b]. Si ha una situazione simile alla codifica dei costrutti che legano variabili nei linguaggi di programmazione general-purpose, come descritto in precedenza.

Queste limitazioni impediscono lo sfruttamento nello sviluppo del software commerciale degli alti potenziali dei logical framework. Infatti, le applicazioni industriali sono scritte per mezzo di efficienti linguaggi imperativi e pertanto richiedono strumenti in grado di manipolare costrutti basati su stato. Lo spettro di applicabilità dei logical framework tradizionali è invece limitato a linguaggi di programmazione funzionali e logici puri. Pertanto, sosteniamo che i logical framework avranno opportunità di circolazione al di fuori del mondo accademico solamente quando verranno attrezzati con la possibilità di modellare in maniera effettiva costrutti di programmazione imperativi.

Diversamente dai formalismi tradizionali, la *logica lineare* [Gir87] offre una visione delle formule logiche come risorse consumabili. È stato notato che questa prospettiva può essere sfruttata per modellare la nozione di stato, come descritto ad esempio in [Chi95, HM94, Mil94, Wad90]. Le proposte attuali mettono l'accento sul problema della *rappresentazione* di costrutti imperativi e logiche basate su risorse, ma appaiono inadeguati per fare *ragionamenti* concreti su queste rappresentazioni. D'altra parte, i framework intuizionisti presentati piú sopra rendono semplice l'effettuazione del meta-ragionamento, ma non includono alcuna nozione di linearità.

Lo scopo di questa tesi è di combinare gli strumenti e le metodologie di meta-ragionamento offerte dai logical framework tradizionali e l'abilità della logica lineare di trattare stati mutevoli in un sistema di meta-rappresentazione uniforme. Il linguaggio di questo *logical framework lineare* (LLF) è una teoria dei tipi lineare che integra pacificamente operatori e concetti propri della logica lineare e costrutti della teoria dei tipi sottostante al logical framework LF. La metodologia di meta-rappresentazione di LLF estende le tecniche disponibili in LF con metodi semplici e diretti per la rappresentazione di uno stato mutevole. Dimostrare risultati di adeguatezza diventa di nuovo fattibile. Similmente a LF, LLF possiede forti proprietà computazionali e può venire implementato come un linguaggio di programmazione logica (lineare).

Dopo questo preambolo piutyosto lungo, collocheremo LLF nello sviluppo storico della logica lineare, dei logical framework, della teoria di tipi e della programmazione logica. Poi, in Sezione

1.2, riassumeremo i risultati contenuti in questa tesi e tenteremo di motivare la loro rilevanza. Infine, presenteremo la struttura di questa tesi in Sezione 1.3.

1.1 Prospettiva Storica

La progettazione di *LLF* mette assieme idee e tecniche provenienti da quattro aree della logica e dell'informatica: la logica lineare, i logical framework, la teoria dei tipi e la programmazione logica. Descriveremo ora brevemente lo sviluppo di ciascuna di queste aree e discuteremo i sistemi correlati piú da vicino a *LLF*. La presentazione è informale, ma definizioni precise verranno date nel corpo di questa tesi.

La Logica Lineare

La *logica lineare* [Gir87] fu presentata nel 1987 da Girard e trae origine da ricerche nell'ambito della semantica formale e della teoria dei tipi [GLT88]. Raffina la logica tradizionale con l'imposizione di rigidi vincoli sull'uso e riuso delle formule logiche presenti nel contesto. Questo porta naturalmente all'interpretazione delle assunzioni logiche come risorse che possono venire *consumate* durante una dimostrazione (mentre la logica tradizionale permette al massimo di *produrle*). Fu presto realizzato che questa proprietà ha numerose applicazioni in varie aree: può essere usata per modellare processi concorrenti [Asp87, GG89, Laf90, MOM91, Mil92, Pra91], per l'elaborazione del linguaggio naturale [Lam95], in teoria della complessità [GSS92], per analizzare programmi funzionali [Abr93, CGR92, Laf88, LM92, RW91, Wad91] e logici [Cer92], ed è stata usata come fondamento per vari linguaggi di programmazione, sia funzionali [Mac91] che logici [AP91, KY93, HM94, Mil94, HW95]. Fu inoltre notato che la logica lineare offre mezzi semplici per modellare uno stato mutevole [Chi95, HM94, Mil94, Wad90] e sistemi deduttivi basati su una gestione non-monotonica delle assunzioni [HM94]. In questa tesi, ci avvantaggiamo di quest'ultima proprietà e sviluppiamo una teoria dei tipi lineari. Descriviamo inoltre come può venire implementata sotto forma di linguaggio di programmazione logica lineare.

I Logical Framework

Nel preambolo, abbiamo definito un *logical framework* come un linguaggio specialmente progettato per permettere rappresentazioni effettive di sistemi deduttivi. L'accento è stato posto sulla possibilità di implementare questi linguaggi allo scopo di permettere un trattamento automatizzato delle rappresentazioni. La richiesta minima consiste nell'abilità di verificare che la rappresentazione di un giudizio o di una derivazione di livello oggetto sia corretta. In addizione a questo, si richiede spesso la possibilità di generare una derivazione per un giudizio dato. Finalmente, può essere desiderabile fornire una deduzione o giudizio incompleto e aspettarsi che l'implementazione lo completi come risultato della computazione.

Il problema del verificare la validità di una derivazione per un giudizio dato è noto come *proof-checking* e viene ridotto a type-checking in linguaggi basati su una teoria dei tipi. Tutte le

realizzazioni concrete di logical framework contengono un *proof-checker* allo scopo di realizzare questo compito fondamentale. Chiaramente, questo problema può venire automatizzato solamente in linguaggi per i quali è possibile decidere se una derivazione è valida. Tutti i framework ai quali siamo interessati possiedono questa proprietà.

La generazione o il completamento di una dimostrazione per un giudizio dato è un problema di *proof-search*. In dipendenza del potere espressivo del meta-linguaggio, può essere piú o meno arduo in pratica. Conseguentemente, un logical framework può venire implementato come un sistema interattivo per lo sviluppo di dimostrazioni oppure come un linguaggio di programmazione logica.

Un *dimostratore di teoremi* è strutturato in un linguaggio di rappresentazione per descrivere derivazioni e giudizi, e un motore inferenziale per fare meta-ragionamento. Il meta-linguaggio è in generale una logica o una teoria dei tipi con le proprietà suddescritte. Il motore inferenziale è dedicato al proof-search. Cablare strategie di ricerca fisse in un dimostratore di teoremi porta a sistemi poco pratici. Ambienti di sviluppo di dimostrazioni moderni offrono invece, in aggiunta a strategie di base, la possibilità di definire le euristiche di ricerca che meglio si adattano al formalismo oggetto considerato. Queste euristiche, comunemente chiamate *tactic* e *tactical*, vengono specificate per mezzo di un linguaggio di programmazione general-purpose. Ad esempio, il dimostratore di teoremi di Boyer-Moore [BM78, BM88] offre un dialetto del *Lisp* per guidare il processo inferenziale. Gli ambienti di sviluppo di dimostrazioni che presenteremo si basano invece su *ML*. Questi linguaggi vengono a volte chiamati 'meta-linguaggi' (da cui *ML* deriva il suo nome; non adottiamo questa terminologia allo scopo di evitare confusione). Si noti che questi sistemi sono formati da due linguaggi distinti, un linguaggio di rappresentazione $\mathcal{L}_\mu$ ottimizzato in modo da descrivere entità del livello oggetto, e un linguaggio di programmazione general-purpose $\mathcal{L}_S$ per specificare la ricerca. Il meta-ragionamento è pertanto svolto in $\mathcal{L}_S$. Questo impedisce che le derivazioni ritrovate vengano rappresentate da termini in $\mathcal{L}_\mu$ e usati per fare ulteriore ragionamento al livello della meta-teoria.

In *programmazione logica*, sia la rappresentazione del formalismo oggetto che il processo di ragionamento vengono svolti nello stesso linguaggio. I giudizi sono rappresentati in generale da formule atomiche e le derivazioni descritte per mezzo di termini che appaiono come argomenti in queste formule. Qualora l'algoritmo di risoluzione, il mechanismo inferenziale di base in programmazione logica, risulta insufficiente, strategie piú complesse si possono codificare direttamente nello stesso linguaggio. In molte proposte, come ad esempio in $\lambda Prolog$ che descriviamo sotto, il meta-ragionamento non produce un termine manipolabile da ulteriori computazioni. Pertanto forme dirette di ragionamento meta-teorico non sono possibili. Altri linguaggi, quali *Elf*, memorizzano i passi del processo di meta-ragionamento in *proof-term* che possono venire immediatamente utilizzati per ragionare su proprietà della meta-teoria.

Un formalismo può essere implementato come un linguaggio di programmazione logica solamente se il *proof-search* può venire ristretto ad un insieme limitato di algoritmi che si possono meccanizzare efficientemente. Molti logical framework basati su linguaggi molto espressivi, quali il calcolo delle costruzioni [CH88], non possiedono questa proprietà, e devono pertanto essere implementati come dimostratori di teoremi interattivi.

Logical framework specifici e le loro implementazioni sono descritti piú sotto. Piuttosto che seguire un rigido sviluppo storico, parliamo prima di programmazione logica e delle proposte che sono state realizzate come linguaggi di programmazione logica. Poi, rivolgeremo la nostra attenzione ai framework che sono stati sviluppati come proof-checker o dimostratori di teoremi, la maggior parte dei quali è basata su una teoria dei tipi.

Sebbene l'importanza di questo problema sia stata riconosciuta [MPP92], il numero dei logical framework che incorporano idee della logica lineare è molto limitato. Oltre alla proposta basata su una teoria dei tipi oggetto di questa tesi, siamo a conoscenza solamente del linguaggio di specifica *Forum* recentemente presentato da Miller.

La Programmazione Logica

L'idea di utilizzare la logica come un linguaggio di programmazione emerse a metà degli anni '60 quando Robinson [Rob65] propose una combinazione della *risoluzione* e dell'*unificazione* (scoperta anni prima da Herbrand [Her71]) come una regola di inferenza per il linguaggio delle clausole di Horn, un ristretto frammento della logica tradizionela. All'inizio degli anni '70, Colmerauer e altri sfruttarono quest'idea nel linguaggio *Prolog*, originariamente progettato per problemi di elaborazione del linguaggio naturale [CKvC73]. L'interpretazione procedurale della risoluzione quale meccanismo computazionale di *Prolog* fu sistematicamente formulato da Kowalski in 1974 [Kow74, Kow79, Kow88]. Lo sviluppo di implementazioni efficienti per questo linguaggio [AK91, War83] accrebbero la sua popolarità al punto che fu adottato per applicazioni industriali in numerosi domini.

Un importante passo nello sviluppo della programmazione logica fu compiuto ad opera di Miller e Nadathur con la presentazione del linguaggio λ*Prolog* [NM88, Mil89]. Oltre che alla presenza di potenti costrutti non disponibili in *Prolog*, quali implicazioni e quantificatori universali nel corpo delle clausole, tipi e termini di ordine superiore, il principale contributo di questi autori è stato metodologico: la progettazione di λ*Prolog* è stata guidata da un'analisi sistematica della teoria della dimostrazione (proof-theory) della logica intuizionistica invece che da considerazioni di tipo semantico. Hanno identificato proprietà che il proof-search in una logica deve possedere allo scopo di essere concretamente meccanizabile come linguaggio di programmazione. L'esistenza di *dimostrazioni uniformi* fu postulato come fondamento proof-theoretic della programmazione logica e usato per dare una definizione di alto livello della nozione generale di *linguaggio di programmazione logica astratto* [MNPS91]. Il metodo di risoluzione come modello operazionale di questi linguaggi fu riformulato in termini proof-theoretic. Il risultato concreto di questa investigazione fu l'identificazione di un grosso frammento della logica intuizionistica, il linguaggio delle *formule di Harrop ereditarie* (di ordine superiore), che è stato implementato nel linguaggio di programmazione logica (concreto) λ*Prolog*.

Il concetto di linguaggio di programmazione logica astratto, proprio per il suo alto livello di astrazione, non tiene conto di alcuni aspetti che devono essere attentamente considerati al momento del dare un'implementazione concreta di un linguaggio di programmazione logica. In particolare, varie *scelte non-deterministiche* dipendenti dalla particolare logica esaminata sono lasciate aperte

in questa definizione. Nel caso della logica intuizionistica, ad esempio, aspetti quali l'ordine in cui i sottogoal devono essere risolti, l'ordine in cui si deve accedere alle clausole nel programma, e il modo in cui si devono istanziare le variabili esistenziali sono tutti non definiti. Tuttavia, sistemi di risoluzione proof-theoretic costituiscono un punto di partenza privo di pregiudizi sopra il quale è possibile formalizzare differenti soluzioni per queste scelte. Adottiamo questo tipo di approccio nello sviluppo di questa tesi.

Questa visione stratificata di un linguaggio di programmazione logica ha anche l'effetto di disaccoppiare la risoluzione come strategia di proof-search e l'unificazione come algoritmo per rendere identici termini e istanziare variabili logiche. La nozione di identità sintattica sottostante all'unificazione del prim'ordine si può infatti generalizzare ad un *teoria equazionale* arbitraria, eventualmente indecidibile quale l'unificazione di ordine superiore. Trovare valori per le variabili logiche che appaiono in queste equazioni viene allora convenientemente espresso come un problema di *risoluzione di vincoli*, e richiede pertanto di essere risolto da algoritmi specializzati alla teoria equazionale adoperata. Ogniqualvolta un'equazione contiene troppe variabili oppure non può venire risolta efficientemente, il risolutore di vincoli pospone la sua risoluzione fintantoché non diventa trattabile; se questo non accade, viene restituita come parte della risposta finale. Questo approccio integra pacificamente il paradigma della *programmazione logica con vincoli* [JL87] e l'approccio proof-theoretic alla programmazione logica. L'unificazione di *Prolog* è un'istanza di questo schema. L'algoritmo di unificazione di ordine superiore originariamente implementato in λ*Prolog* non pospone i vincoli "difficili". Essendo la teoria equazionale di questo linguaggio indecidibile, questa strategia ha l'effetto di causare occasionali divergenze durante l'unificazione, anche se questo è raramente un problema in pratica.

Pym estese l'approccio proof-theoretic di Miller alla programmazione logico dal contesto logica dove fu originariamente concepito alla teoria dei tipi [Pym90, PW90]. Propose una teoria generale del proof-search per questi formalismi, includendo un algoritmo di unificazione per LF. Elliot sviluppò indipendentemente un algoritmo di unificazione per questo tipo di teoria, che fu adattato da Pfenning nell'implementazione di *Elf* [Ell90].

λ*Prolog* è interessante nel contesto di questa tesi in quanto è un primo esempio di logical framework implementato come un linguaggio di programmazione logica. La disponibilità di λ-astrazione al livello dei termini offre la sintassi astratta di ordine superiore come semplice mezzo per codificare gli operatori che legano variabili, presenti in molte logiche e in molti linguaggi di programmazione; abbiamo visto che questi costrutti sono di trattamento problematico in altri linguaggi (fra cui *Prolog*). I giudizi del livello oggetto sono rappresentati in λ*Prolog* sotto forma di formule atomiche, e le regole d'inferenza come clausole nel programma. La disponibilità di uno spettro piú ampio di formule rispetto a *Prolog*, in particolare i goal implicazione e universali, permette una rappresentazione immediata di schemi ricorrenti presenti in regole d'inferenza, quali i giudizi parametrici e ipotetici.

Oltre che al linguaggio di programmazione logica λ*Prolog* [Mil89], il linguaggio delle formule di Harrop ereditarie [MNPS91] è anche stato implementato come il dimostratore automatico di teoremi *Isabelle* [Pau88, Pau93]. Adotta una metodologia di rappresentazione basata su sintassi astratta di ordine superiore simile a λ*Prolog*. È stato usato in numerosi esperimenti, fra cui la

verifica di compilatori [LM94] e di interpreti [BHN+94], per verificare la correttezza di circuiti [Ros90] e per la specifica di sistemi concorrenti [DS89].

Quasi concorrentemente con lo sviluppo di $\lambda Prolog$, Pfenning progettò *Elf* [Pfe91, Pfe94a] quale linguaggio di programmazione logica implementante il logical framework LF. *Elf* incorpora caratteristiche simili al linguaggio di Miller e Nadathur, ossia termini di ordine superiore, tipi, e goal implicazione ed universali. In *Elf*, l'unificazione di ordine superiore è trattata come un problema di risoluzione di vincoli e le equazioni "difficili" sono posposte fino a quando eventuali istanziazioni le rendono trattabili.

Un aspetto nuovo di *Elf* è la sua abilità di memorizzare le proprie computazioni come *proof-term*, un'eredità diretta dalla natura proof-theoretic di LF. Questa caratteristica migliora notevolmente il suo potere espressivo come linguaggio di meta-rappresentazione. In $\lambda Prolog$, le regole d'inferenza del linguaggio oggetto vengono rappresentate come formule nel programma. Pertanto, una derivazione di livello oggetto corrisponde ad una computazione nel meta-linguaggio, rendendo questo formalismo perfettamente adeguato alla rappresentazione della nozione di dimostrabilità del livello oggetto. In *Elf*, le formule nel programma sono etichettate da costanti che appaiono poi nei proof-term quali testimoni dell'applicazione di queste clausole. Pertanto, quando una computazione in *Elf* ricalca una derivazione del livello oggetto, la sequenza delle regole oggetto applicate è memorizzato nel proof-term, che ne costituisce una fedele rappresentazione. Pertanto, *Elf* può rappresentare non solo la nozione di derivabilità, come $\lambda Prolog$, ma anche le stesse derivazioni. I proof-term ottenuti in questa maniera possono essere oggetto di ulteriori computazioni, effettuando in questa maniera meta-ragionamento sulle derivazioni di livello oggetto che rappresentano.

Derivazioni di livello oggetto si possono esplicitamente codificare come parte delle formule di $\lambda Prolog$, assemblandole al momento dell'applicazione delle regole del livello oggetto. Tuttavia, la validità della rappresentazione risultante va dimostrata come una proprietà esterna del sistema. In *Elf* invece, la validità di una derivazione, rappresentata come proof-term, è interiorizzata e deriva dalla semantica operazionale del meta-linguaggio e dai teoremi di adeguatezza per il linguaggio oggetto.

Il primo linguaggio di programmazione logica lineare fu LO [AP91], progettato per incorporare aspetti di programmazione orientata agli oggetti in *Prolog*. Il linguaggio *Lolli* [HM94], ad opera di Hodas e Miller, ebbe invece il proposito esplicito di esplorare le potenzialità della logica lineare come linguaggio di programmazione logica. Fu progettato estendendo il metodo proof-theoretic utilizzato per $\lambda Prolog$ alla logica lineare intuizionistica, e un frammento noto come la logica delle *formule di Harrop ereditarie lineari* fu identificato quale nucleo di questo linguaggio. Un'importante conseguenza della definizione di *Lolli* fu l'identificazione di un nuovo tipo di scelta non-deterministica, la *gestione del contesto lineare*, e di tecniche per eliminarla [CHP96].

Lolli è un'estensione conservativa di $\lambda Prolog$, se non per il fatto che aspetti di ordine superiore non furono mai inclusi in alcuna implementazione concreta di questo linguaggio. Il suo uso per rappresentare derivazioni nello stile di *Elf* o $\lambda Prolog$ è pertanto limitato, ma la sua abilità di codificare in maniera effettiva problemi caratterizzati da uno stato mutevole e logiche basate su

risorse è manifesto [HM94].

Miller ha recentemente proposto il linguaggio di specifica *Forum*, basato sulla logica lineare classica di ordine superiore [Mil94]. Lo scopo di questo formalismo è di incorporare primitive di comunicazione e sincronizzazione per supportare la concorrenza nei linguaggi λ*Prolog* e *Lolli*, i quali sono molto espressivi, ma apparentemente legati ad un modello di esecuzione sequenziale. L'effetto atteso è di fornire mezzi concreti per rappresentare sistemi deduttivi per computazioni concorrenti quali il π-calculo [Mil92]. *Forum* risulta dall'applicazione della metodologia di progettazione di λ*Prolog* alla logica lineare classica. Il risultato è un frammento equivalente, modulo una semplice codifica, all'intera logica lineare classica. Estende λ*Prolog*, *Lolli* e *LO*.

Forum offre una scelta di connettivi piú ampia e un diverso paradigma di programmazione rispetto a *Lolli*, ma ha essenzialmente lo stesso potere espressivo per molti classi di applicazioni. La possibilità di dare codifiche intuizionistiche a linguaggi concorrenti altrettanto elegantemente che in *Forum* non è stata investigata. Tuttavia, grosse specifiche scritte in *Forum* prese da [Chi95] sono state codificate in formalismi intuizionistici in uno stile simile agli esempi descritti nel Capitolo 6. Portarsi sul campo della logica classica non offre nessun beneficio evidente in questo caso. Come già messo in evidenza in [Mil94], la presenza di alcuni connettivi classici, porta invece ad anomalie poco desiderabili nella semantica di proof-search di questo linguaggio, al punto che il suo statuto come linguaggio di programmazione logica è discutibile. Infatti, tentativi preliminari di implementarlo sono risultati in un "sistema [che] è a metà strada tra un linguaggio e un dimostratore di teoremi" [HP95].

Similmente a quanto accade in λ*Prolog*, la presenza di termini di ordine superiore permette a *Forum* i usare la sintassi astratta di ordine superiore come schema di meta-rappresentazione di base. Non disponendo di proof-term, è tuttavia soggetto alle stesse considerazioni discusse nel caso di λ*Prolog*: dà rappresentazioni dirette della dimostrabilità, ma non delle dimostrazioni. Tuttavia, la sua abilità di rappresentare giudizi di logiche e linguaggi di programmazione estende considerevolmente framework intuizionistici quali *Elf* o λ*Prolog*. Ciononostante, presenta gli stessi tipi di limitazioni al momento di *ragionarci* sopra. *Forum* è in grado di simulare computazioni lineari, ma non di rappresentarle. Possiamo rintracciare il problema nel fatto che i termini che offre per costruire formule e rappresentare derivazioni originano da un ordinario λ-calcolo intuizionistico. Le derivazioni lineari, tuttavia, hanno una struttura piú fine e la loro rappresentazione richiede un linguaggio piú ricco, nella fatti specie un λ-*calcolo lineare*.

Come esempio concreto, si consideri il problema di rappresentare una procedura di eliminazione del taglio per la logica lineare. Data una derivazione $\mathcal{D}$ per un sequente lineare $\Delta \longrightarrow \Theta$, questa procedura produce una derivazione $\mathcal{D}'$ equivalente ma priva di tagli. Dato che opera linearmente sulle formule che appaiono in essa, $\mathcal{D}$ dovrebbe essere rappresentata da un λ-termine lineare; lo stesso vale per $\mathcal{D}'$. Questo problema è stato codificato in LF rappresentando i sequenti come tipi e le derivazioni come proof-term [Pfe94b]. LF è intuizionistico e pertanto la linearità di $\ulcorner\mathcal{D}\urcorner$ si è dovuta controllare esplicitamente come una proprietà di $\ulcorner\mathcal{D}\urcorner$, complicando la meta-teoria al punto da renderla non fattibile cosicché solamente l'algoritmo di eliminazione del taglio senza controlli di linearità è stato implementato in *Elf*. D'altra parte, codificare lo stesso problema in *Forum* [Mil94] farebbe corrispondere derivazioni di sequenti lineari con derivazioni lineari nel

meta-linguaggio. Tuttavia, poiché *Forum* è privo di proof-term e di termini lineari, non viene fornita nessuna notazione interna per questi oggetti, sicché l'algoritmo di eliminazione del taglio non è implementabile in questa maniera.

La Teoria dei Tipi

L'inizio della *teoria dei tipi* formale viene solitamente fatta risalire alla definizione da parte di Church nel 1940 del λ-calcolo semplicemente tipato che porta il suo nome [Chu40]. Un parallelo con il frammento implicazionale della logica intuizionistica fu presto notato: le regole di deduzione naturale per l'implicazione in logica intuizionistica sono isomorfe al modo in cui i tipi di oggetti funzionali sono manipolati, e per di piú, gli oggetti stessi possono essere messi in corrispondenza uno a uno con derivazioni delle formule corrispondenti ai loro tipi [CF58]. Howard estese questa scoperta ad un insieme piú ampio di connettivi logici, identificando i corrispondenti operatori su tipi e i costrutti del livello dei termini che potrebbero essere usati per simulare derivazioni in questo frammento esteso. Martin-Löf, nella sua teoria dei tipi intuizionistica, estese queste idee per inglobare i tipi dipendenti [ML80]. Ulteriori estensioni verso logiche di ordine superiore furono contribuite da Girard [GLT88]. Questa proprietà viene chiamata la corrispondenza *formule-tipi* e *dimostrazioni-termini*, o, citando il nome dei suoi principali contributori, l'*isomorfismo di Curry-Howard*. Gli articoli originali di Curry, Howard, cosícome lavori piú recenti di altri ricercatori sono stati raccolti in [dG95].

La ricerca sul progetto *AUTOMATH* portò indipendentemente de Bruijn a presentare una sua corrispondenza formule-tipi tra logica e λ-calcolo. Lo scopo di *AUTOMATH* era "di chiarire e formalizare i principi di base che tutti i matematici usano e su cui concordano. In un certo senso, questa [era] un tentativo di mettere sul palcoscenico la parte della matematiche che viene 'prima della logica', la parte di cui ogni matematico è informalmente consapevole, come ad esempio come usare o dare una definizione" [Geu93]. La teoria dei tipi di *AUTOMATH* era progettata come un'infrastruttura per rappresentare ragionamento matematico generico senza pregiudizi fondazionali. Non era legato a nessuna forma di ragionamento cosicché era possibile e anzi necessario definire all'interno del sistema la logica che meglio si addiceva ad problema matematico trattato.

I linguaggi della famiglia *AUTOMATH* sono stati implementati come un proof-checker con lo stesso nome [dB80]. Gli enunciati del linguaggio oggetto erano rappresentati come tipi e argomentazioni matematiche come λ-termini tipati, con i costrutti che legano variabili modellati per mezzo di indici di de Bruijn. Si verificava la correttezza di una dimostrazione per un enunciato di livello oggetto controllando che il termine che lo codificava era ben formato ed era del tipo corrispondente. Poiché *AUTOMATH* è stato progettato per rappresentare la dimostrabilità di argomentazioni matematiche generiche, l'implementazione risultante non era in grado di generare dimostrazioni, ma solamente di verificarle. L'applicazione piú significativa di questo sistema è stato la specifica e verifica di un libro di testo introduttivo in analisi matematica scritto da Landin [Jut77].

Lo scopo della *Teoria dei Tipi Intuizionistica* di Martin-Löf [ML80], sviluppata quasi contemporaneamente, era radicalmente differente. Dal suo punto di vista, la logica e la teoria dei

tipi costruttive costituivano i fondamenti dell'Informatica. Pertanto, le teorie dei tipi che propose erano di natura fortemente intuizionistica cosicché forme di ragionamento costruttivo si poteva facilmente formalizzare avvantaggiandosi della logica interna del sistema, mentre altre forme di inferenza matematica necessitavano codifiche laboriose.

Varie implementazioni delle teorie dei tipi di Martin-Löf sono disponibili. Le piú note sono *NuPrl* [C$^+$86] e *ALF* [MN94, Nor93]. *NuPrl* è un completo ambiente di sviluppo di dimostrazioni assistito da calcolatore per la logica intuizionistica. L'implementazione contiene numerosi strumenti per automatizzare il processo di sviluppo di dimostrazioni intuizionistiche. In particolare, offre estensione notazionale, costruzione e modifica di dimostrazioni, assistenza automatica, e il mantenimento e riuso di una libreria di risultati precedenti. Questo sistema contribuí a dimostrare l'importanza della teoria dei tipi in Informatica, in particolare come base dello sviluppo interattivo di programmi come estrazione da dimostrazioni. *ALF* adotta una diversa filosofia di rappresentazione, piú vicina in spirito alle implementazioni del calcolo delle costruzioni. I sistemi deduttivi vengono rappresentati come teorie equazionali e fanno pesante uso di definizioni induttive. Il sistema risultante è un ambiente di sviluppo di dimostrazioni generale per la matematica intuizionistica.

Lo sviluppo della teoria dei tipi negli anni successivi al lavoro pionieristico di Martin-Löf e de Bruijn ebbe lo scopo di accrescere il potere di questi formalismi ad esprimere nuovi aspetti della matematica e a modellare concetti di linguaggi di programmazione. Oltre alla teoria dei tipo sottostante ad *LF*, che descriveremo fra breve, una contribuzione importante in questo campo venne dalla definizione del *Calcolo delle Costruzioni* da parte di Coquand e Huet [CH88], il quale estende i linguaggi *AUTOMATH* e le prime teorie dei tipi di Martin-Löf con tipi impredicativi di ordine superiore. Questo formalismo fu successivamente raffinato nel *Calcolo delle Costruzioni Esteso* [Luo89] e piú recentemente nel *Calcolo delle Costruzioni Induttive* [PM93]. Una discussione unitaria di molte di queste proposte può essere rinvenuta in [Bar92].

Vari raffinamenti del calcolo delle costruzioni [CH88, Luo89, PM93] sono stati implementati nei sistemi *Coq* [DFH$^+$93] e *LEGO* [LP92, Pol94]. Si possono vedere come ulteriori sforzi di costruire framework generali per ragionamento formale assistito da calcolatore, basati tuttavia su teorie dei tipi piú potenti dei sistemi a cui si è accennato sopra. *Coq* e *LEGO* sono basati su operazioni primitive di estensione del contesto, sia per mezzo di definizioni che di dichiarazioni. Sono stati utilizzati per formalizzare argomentazioni matematici di vario tipo [CH88] e per la verifica e sintesi di programmi [PM89].

Il *Logical Framework di Edimburgo*, meglio noto sotto in nome di *LF* è stato proposto nel 1987 da Harper, Honsell e Plotkin [HHP93]. La teoria dei tipi su cui si basa è un'estensione del λ-calcolo semplicemente tipato di Church [Chu40] con tipi dipendenti, ed è correlato da vicino alle teorie dei tipi costruttive di Martin-Löf [ML80] e con alcuni dei linguaggi *AUTOMATH* proposti da de Bruijn [dB80]. Costituisce un sottolinguaggio del calcolo delle costruzioni [CH88]. *LF* è stato progettato con l'esplicito scopo di usarlo come un meta-linguaggio per rappresentare sistemi deduttivi ("*A Framework for Defining Logics*" è il titolo dell'articolo originario di Harper, Honsell e Plotkin [HHP93]). L'accento è posto sulla nozione di giudizio a tal punto che

la corrispondenza formule-tipi e dimostrazioni-termini è rimpiazzata dall'equivalenza *giudizi-tipo* e *derivazioni-termini*. La sintassi astratta di ordine superiore è offerta come meccanismo di base per rappresentare derivazioni. La teoria dei tipi di LF è sufficientemente espressiva da permettere la formalizzazione di un'ampia classe di sistemi deduttivi, ma anche abbastanza semplice da permettere dimostrazione di adeguatezza semplici ed efficienti implementazioni del proof-search. È stato infatti osservato che l'introduzione di schemi di induzione porta a complicazioni nei teoremi di adeguatezza. Estensioni verso il calcolo delle costruzioni distruggono l'interpretazione di questo formalismo come un linguaggio di programmazione logica astratto: la proprietà delle dimostrazioni uniformi viene persa e problemi riguardanti l'unificazione diventano piú difficili da trattare in maniera soddisfacente.

Come descritto in precedenza, LF [HHP93] è stato implementato come il linguaggio di programmazione logica *Elf* [Pfe91, Pfe94a]. Ancor prima che questa implementazione fosse disponibile, è stato usato per fornire specifiche effettive di numerosi sistemi deduttivi con varie caratteristiche: logica tradizionale del prim'ordine e di ordine superiore [HHP93], logiche modali da K a $S4$, logica di Hoare e vari λ-calcoli [AHM89, AHMP92], varie teorie dei tipi fra cui lo stesso LF e le proposte di Martin-Löf [Har90], solo per citare i principali esempi. La disponibilità di *Elf* come implementazione di LF portò allo sviluppo di una ricca libreria di ulteriori esempi: fra le numerose codifiche coronate da successo, possiamo ricordare varie specifiche del linguaggio funzionale *Mini-ML* [MP91], una dimostrazione della proprietà di Church-Rosser [Pfe99], e una dimostrazione del teorema di eliminazione del taglio per la logica tradizionale e lineare [Pfe95b].

Estensioni lineari di vari λ-calcoli sono state propose in letteratura [ABCJ94, BBHdP93, LM92, RR94, Sol89]. Sono tutte equivalenti ad un qualche frammento della logica lineare, via il raffinamento naturale del isomorfismo di Curry-Howard alla logica lineare [GdQ92]. Tuttavia, nessuna estensione lineare di teorie dei tipi piú complessa è ancora stata proposta.

1.2 Contributi

In questa tesi, progettiamo il meta-linguaggio LLF quale un'estensione conservativa del logical framework LF [HHP93] con concetti proveniente dalla logica lineare [Gir87]. Facciamo inoltre vedere come LLF si possa usare per fornire effettive rappresentazioni di alto livello di costrutti presenti in linguaggi imperativi e di problematiche nella semantica operazionale di logiche basate su risorse. Questo lavoro è stato ispirato ad un tentativo preliminare di produrre un raffinamento lineare di LF ad opera di Miller, Plotkin e Pym [MPP92].

LLF è concettualmente molto vicino a *Lolli* [HM94] e adotta, quali costruttori di tipi addizionali rispetta a LF e *Elf* [Pfe91], gli stessi connettivi lineari che *Lolli* aggiunge a λ*Prolog* [NM88]. Tuttavia, LLF permette oggetti lineari in tipi di base mentre *Lolli* permette solamente termini intuizionistici nell'equivalente concetto di formula atomica. Inoltre, LLF ha la possibilità di memorizzare le proprie derivazioni come proof-term lineari che possono essere usati in un secondo tempo per fare ragionamento meta-teorico. Similmente alle proposte di cui sopra, LLF si può vedere come un linguaggio di programmazione logica. Ci aspettiamo di essere in grado di

fondere le tecniche utilizzate nelle implementazioni di *Elf* e *Lolli*.

Il contributo principale di questa tesi è la presentazione di un formalismo avente la possibilità di dare rappresentazioni effettive e naturali ai costrutti imperativi presenti in tutti i linguaggi di programmazione di interesse pratico. Abbiamo motivato in precedenza che gli attuali meta-linguaggi non sono adeguati alla rappresentazione di sistemi basati su stato, e pertanto i benefici potenziali che offrono hanno applicazioni pratiche molto limitate nello sviluppo di software commerciale. Sebbene la nostra sperimentazione con LLF sia ancora agli inizi, riteniamo che questo linguaggio e la metodologia di meta-rappresentazione che promulga possano essere sfruttati in varie aree. Innanzitutto, il nostro formalismo ha applicazioni nella fase di progettazione di linguaggi di programmazione general-purpose. Infatti, offre mezzi per rappresentare concretamente i costrutti e le proprietà che ci aspetta siano valide nel linguaggio sviluppato. Inoltre, LLF può essere di aiuto nella progettazione di sistemi logici che, per le loro proprietà sfuggono allo spettro di applicabilità dei sistemi di sviluppo di dimostrazione assistiti da calcolatore tradizionali. Infine, LLF può essere utilizzato come strumento di meta-rappresentazione per la verifica automatica della correttezza di programmi imperativi. È concepibile che, fra qualche anno, i concetti sottostanti al nostro linguaggio possano venire integrati in ambienti di sviluppo del software utilizzati a livello industriale.

A nostra conoscenza, il linguaggio formale alla base di LLF è il primo esempio di una teoria dei tipi uniforme che ammetta entità lineari in congiunzione con tipi dipendenti. Termini lineari del livello oggetto non solo hanno la funzione di memorizzare derivazioni lineari come proof-term, ma possono anche comparire negli indici delle famiglie di tipi (ossia come argomenti di simboli predicativi, in un contesto logico) e possono pertanto essere manipolati come oggetti propri. Sebbene questo formalismo non abbia versioni dipendenti dei costruttori di tipo lineari, il ruolo giocato da queste entità è cosí centrare che ci sentiamo giustificati nel chiamarlo una *teoria dei tipi lineare*. Questo aspetto di LLF è pertanto interessante per se stesso, e offre un fertile terreno di gioco per sperimentare con nuovi costrutti. In particolare, ci ha permesso di approfondire la natura dei tipi dipendenti lineari e dei quantificatori lineari, uno studio che intendiamo riprendere nel futuro.

Questa tesi contribuisce inoltre allo studio della programmazione logica lineare. Facciamo vedere che LLF possiede le proprietà di proof-search che ci autorizzano ad interpretarlo come un linguaggio di programmazione logica lineare astratto. Per di piú, descriviamo come un sistema deduttivo puramente dichiarativo per questo linguaggio viene trasformato passo dopo passo in un equivalente sistema di risoluzione, dove i passi corrispondono a sistemi deduttivi intermedi connessi da rigorose dimostrazione di correttezza e completezza. Inoltre, estendiamo questa tecnica al trattamento formale dei punti di scelta non-deterministici. Ci concentriamo sulla problematica della gestione delle risorse nel contesto lineare, una forma di non-determinismo specifica dei linguaggi di programmazione logica lineari, e risolviamo problemi lasciati aperti da tecniche precedenti.

1.3 Organizzazione della Tesi

Il Capitolo 2 ha lo scopo di fornire al lettore le necessarie nozioni preliminari di logica e teoria dei tipi. Piú precisamente, introduciamo la logica tradizionale e in particolare sul suo frammento intuizionistico. La logica lineare è descritta in dettaglio, con particolare enfasi sugli aspetti su cui *LLF* piú si basa. In questo capitolo, limitiamo la nostra presentazione della teoria dei tipi al frammento semplicemente tipato di Church. Questo ci basta a dare un esempio concreto dell'isomorfismo di Curry-Howard, e a permetterci di presentare una semplice variazione lineare di questo calcolo. Un ulteriore effetto di questo capitolo è di dare definizioni precise a molte delle nozioni usate in questa introduzione.

Nel Capitolo 3, definiamo le nozioni relative ad una presentazione proof-theoretic della programmazione logica, piú precisamente, proof-search guidato dal goal, dimostrabilità uniforme, risoluzione e discutiamo i punti di scelta non-deterministici. Supportiamo queste definizioni con due esempi. Innanzitutto, facciamo vedere che la logica delle formule di Harrop ereditarie lineari costituisce un linguaggio di programmazione logica astratto e discutiamo la sua implementazione concreta nel linguaggio *Lolli*. Poi, volgiamo la nostra attenzione alla teoria dei tipi di *LF* e descriviamo la sua implementazione concreta: il linguaggio di programmazione logica *Elf*.

Il Capitolo 4 completa le nozioni preliminari necessarie ad apprezzare questa tesi. Analizziamo innanzitutto le nozioni di linguaggio formale e di meta-rappresentazione. Poi facciamo vedere come *LF* viene usato in pratica come un logical framework descrivendo, in qualità di esempio la meta-rappresentazione della sintassi, semantica e aspetti della meta-teoria del linguaggio funzionale *Mini-ML*. L'intero esempio è sviluppato in *Elf*.

Nel Capitolo 5, presentiamo la teoria dei tipi di *LLF* nella come un sistema pre-canonico, e dimostriamo una serie di proprietà di questo formalismo, fra cui la decidibilità del type-checking. Inoltre, ripercorriamo il lavoro svolto nel capitolo precedente discutendo forme canoniche, dimostrazioni uniformi, risoluzione e non-determinismo.

Nel capitolo seguente, dimostriamo il potere di meta-rappresentazione di *LLF* su un'estensione di *Mini-ML* con referenze. Forniamo una codifica che estende la descrizione del *Mini-ML* di base presentata nel Capitolo 3 e dimostriamo la sua adeguatezza. Presentiamo inoltre altri esempi, anche se in minor dettaglio. Piú particolareggiatamente, descriviamo una specifica della dimostrazione del teorema di eliminazione del taglio per un frammento della logica lineare e facciamo vedere la codifica di un gioco.

Infine, nel Capitolo 7, ricapitoliamo il lavoro svolto e delineiamo sviluppi futuri di questa ricerca.

Una versione molto piú completa di questo documento è disponibile in lingua Inglese [Cer96]. Invitiamo il lettore interessato a richiederla via posta elettronica direttamente all'autore all'indirizzo *iliano@di.unito.it*.

Parte I

Nozioni Preliminarie

Capitolo 2

Nozioni di Logica e Teoria dei Tipi

In questo capitolo, mostriamo alcuni sistemi logici e teorie dei tipi rilevanti per lo sviluppo della discussione. Ci limitiamo a specificare riferimenti alla letteratura. Invitiamo il lettore interessato a queste nozioni preliminari a consultare i riferimenti presentati, oppure [Cer96] per approfondimenti su questi argomenti.

2.1 La Logica Tradizionale

In questa sezione, ci occupiamo della logica tradizionale [Kle52]. Una logica si può classificare secondo vari criteri:

- La *granularità* dei concetti che si possono esprimere. Distinguiamo allora tra logiche *proposizionali* e logiche dei *predicati*.
- Il *dominio di quantificazione*. Alcune logiche, quale la logica proposizionale del prim'ordine normalmente studiata nei corsi introduttivi di logica matematica, non offrono la possibilità di usare quantificatori. Abbiamo poi la logica del *prim'ordine* quando gli oggetti a cui si può applicare un quantificatore sono individui dei domino. Infine, abbiamo vari tipi di logiche di *ordine superiore* permettendo di quantificare su funzioni di vario ordine, o addirittura su predicati.
- La diversificazione degli oggetti nel dominio. Se non si parla di tipi nella presentazione di una logica, essa è detta non-tipata, o piú accuratemente *monotipata*. Una maggior ricchezza di tipi è possibile.

Una logica può venire presentata in vari modi, in dipendenza di ciò che si vuole farne. I principali sono:

- Le formulazioni *à la Hilbert*, particolarmente adeguata per modellare compilatori per linguaggi funzionali.

Axiom

$$\frac{}{A \longrightarrow A}\ \mathbf{id}$$

Cut

$$\frac{\Gamma \longrightarrow A, \Phi \quad \Gamma, A \longrightarrow \Phi}{\Gamma \longrightarrow \Phi}\ \mathbf{cut}$$

Structural rules

$$\frac{\Gamma, B, A \longrightarrow \Phi}{\Gamma, A, B \longrightarrow \Phi}\ \mathbf{x\_l} \qquad \frac{\Gamma \longrightarrow B, A, \Phi}{\Gamma \longrightarrow A, B, \Phi}\ \mathbf{x\_r}$$

$$\frac{\Gamma \longrightarrow \Phi}{\Gamma, A \longrightarrow \Phi}\ \mathbf{w\_l} \qquad \frac{\Gamma \longrightarrow \Phi}{\Gamma \longrightarrow A, \Phi}\ \mathbf{w\_r}$$

$$\frac{\Gamma, A, A \longrightarrow \Phi}{\Gamma, A \longrightarrow \Phi}\ \mathbf{c\_l} \qquad \frac{\Gamma \longrightarrow A, A, \Phi}{\Gamma \longrightarrow A, \Phi}\ \mathbf{c\_r}$$

Propositional connectives

(No **true_l**)

$$\frac{}{\Gamma \longrightarrow \top, \Phi}\ \mathbf{true\_r}$$

$$\frac{}{\Gamma, \bot \longrightarrow \Phi}\ \mathbf{false\_l}$$

(No **false_r**)

$$\left.\begin{array}{l} \dfrac{\Gamma, A_1 \longrightarrow \Phi}{\Gamma, A_1 \wedge A_2 \longrightarrow \Phi}\ \mathbf{and\_l_1} \\[2ex] \dfrac{\Gamma, A_2 \longrightarrow \Phi}{\Gamma, A_1 \wedge A_2 \longrightarrow \Phi}\ \mathbf{and\_l_2} \end{array}\right\} \qquad \frac{\Gamma \longrightarrow A_1, \Phi \quad \Gamma \longrightarrow A_2, \Phi}{\Gamma \longrightarrow A_1 \wedge A_2, \Phi}\ \mathbf{and\_r}$$

$$\frac{\Gamma, A_1 \longrightarrow \Phi \quad \Gamma, A_2 \longrightarrow \Phi}{\Gamma, A_1 \vee A_2 \longrightarrow \Phi}\ \mathbf{or\_l} \qquad \left\{\begin{array}{l} \dfrac{\Gamma \longrightarrow A_1, \Phi}{\Gamma \longrightarrow A_1 \vee A_2, \Phi}\ \mathbf{or\_r_1} \\[2ex] \dfrac{\Gamma \longrightarrow A_2, \Phi}{\Gamma \longrightarrow A_1 \vee A_2, \Phi}\ \mathbf{or\_r_2} \end{array}\right.$$

$$\frac{\Gamma \longrightarrow A_1, \Phi \quad \Gamma, A_2 \longrightarrow \Phi}{\Gamma, A_1 \to A_2 \longrightarrow \Phi}\ \mathbf{imp\_l} \qquad \frac{\Gamma, A_1 \longrightarrow A_2, \Phi}{\Gamma \longrightarrow A_1 \to A_2, \Phi}\ \mathbf{imp\_r}$$

$$\frac{\Gamma \longrightarrow A, \Phi}{\Gamma, \neg A \longrightarrow \Phi}\ \mathbf{not\_l} \qquad \frac{\Gamma, A \longrightarrow \Phi}{\Gamma \longrightarrow \neg A, \Phi}\ \mathbf{not\_r}$$

Quantifiers

$$\frac{\Gamma, [t/x]A \longrightarrow \Phi}{\Gamma, \forall x.\, A \longrightarrow \Phi}\ \mathbf{all\_l} \qquad \frac{\Gamma \longrightarrow [c/x]A, \Phi}{\Gamma \longrightarrow \forall x.\, A, \Phi}\ \mathbf{all\_r}^{c}$$

$$\frac{\Gamma, [c/x]A \longrightarrow \Phi}{\Gamma, \exists x.\, A \longrightarrow \Phi}\ \mathbf{exists\_l}^{c} \qquad \frac{\Gamma \longrightarrow [t/x]A, \Phi}{\Gamma \longrightarrow \exists x.\, A, \Phi}\ \mathbf{exists\_r}$$

Figura 2.1: La Logica Classica

Axiom

$$\frac{}{\Gamma, A \longrightarrow A}\ \mathbf{id}$$

Cut

$$\frac{\Gamma \longrightarrow A \quad \Gamma, A \longrightarrow C}{\Gamma \longrightarrow C}\ \mathbf{cut}$$

Propositional connectives

(No **true_l**)

$$\frac{}{\Gamma \longrightarrow \top}\ \mathbf{true\_r}$$

$$\frac{}{\Gamma, \bot \longrightarrow C}\ \mathbf{false\_l}$$

(No **zero_r**)

$$\left.\begin{array}{l} \dfrac{\Gamma, A_1 \wedge A_2, A_1 \longrightarrow C}{\Gamma, A_1 \wedge A_2 \longrightarrow C}\ \mathbf{and\_l_1} \\[2ex] \dfrac{\Gamma, A_1 \wedge A_2, A_2 \longrightarrow C}{\Gamma, A_1 \wedge A_2 \longrightarrow C}\ \mathbf{and\_l_2} \end{array}\right\}$$

$$\frac{\Gamma \longrightarrow A_1 \quad \Gamma \longrightarrow A_2}{\Gamma \longrightarrow A_1 \wedge A_2}\ \mathbf{and\_r}$$

$$\frac{\Gamma, A_1 \vee A_2, A_1 \longrightarrow C \quad \Gamma, A_1 \vee A_2, A_2 \longrightarrow C}{\Gamma, A_1 \vee A_2 \longrightarrow C}\ \mathbf{or\_l}$$

$$\left\{\begin{array}{l} \dfrac{\Gamma \longrightarrow A_1}{\Gamma \longrightarrow A_1 \vee A_2}\ \mathbf{or\_r_1} \\[2ex] \dfrac{\Gamma \longrightarrow A_2}{\Gamma \longrightarrow A_1 \vee A_2}\ \mathbf{or\_r_2} \end{array}\right.$$

$$\frac{\Gamma, A_1 \to A_2 \longrightarrow A_1 \quad \Gamma, A_1 \to A_2, A_2 \longrightarrow C}{\Gamma, A_1 \to A_2 \longrightarrow C}\ \mathbf{imp\_l}$$

$$\frac{\Gamma, A_1 \longrightarrow A_2}{\Gamma \longrightarrow A_1 \to A_2}\ \mathbf{imp\_r}$$

Quantifiers

$$\frac{\Gamma, [t/x]A \longrightarrow C}{\Gamma, \forall x.\, A \longrightarrow C}\ \mathbf{all\_l}$$

$$\frac{\Gamma \longrightarrow [c/x]A}{\Gamma \longrightarrow \forall x.\, A}\ \mathbf{all\_r}^{c}$$

$$\frac{\Gamma, [c/x]A \longrightarrow C}{\Gamma, \exists x.\, A \longrightarrow C}\ \mathbf{exists\_l}^{c}$$

$$\frac{\Gamma \longrightarrow [t/x]A}{\Gamma \longrightarrow \exists x.\, A}\ \mathbf{exists\_r}$$

Figura 2.2: La Logica Minimale

- I sistemi di *deduzione naturale* [Sza70] sono molto utili per modellare ragionamento matematico e stanno alla base della specifica logica di linguaggi di programmazione sia logici che funzionali. Permettono inoltre connessioni con λ-calcoli e teorie dei tipi per mezzo di varie estensioni degli isomorfismi di Curry-Howard [dG95].

- I *calcoli dei sequenti* [Sza70], molto adeguati allo studio meta-teorico di formalismi logici.

Le logiche si diversificano anche per l'insieme di formule che ammettono come valide. Le principali sono:

Axioms

$$\frac{}{\Gamma, A \vdash A}\ \mathbf{var}$$

Propositional connectives

(No **true_e**)

$$\frac{}{\Gamma \vdash \top}\ \mathbf{true\_i}$$

$$\frac{\Gamma \longrightarrow \bot}{\Gamma \longrightarrow C}\ \mathbf{false\_e}$$

(No **false_i**)

$$\left.\begin{array}{l} \dfrac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_1}\ \mathbf{and\_e_1} \\[2ex] \dfrac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_2}\ \mathbf{and\_e_2} \end{array}\right\}$$

$$\frac{\Gamma \vdash A_1 \quad \Gamma \vdash A_2}{\Gamma \vdash A_1 \wedge A_2}\ \mathbf{and\_i}$$

$$\frac{\Gamma \vdash A_1 \vee A_2 \quad \Gamma, A_1 \vdash C \quad \Gamma, A_2 \vdash C}{\Gamma \vdash C}\ \mathbf{or\_e}$$

$$\left\{\begin{array}{l} \dfrac{\Gamma \vdash A_1}{\Gamma \vdash A_1 \vee A_2}\ \mathbf{or\_i_1} \\[2ex] \dfrac{\Gamma \vdash A_2}{\Gamma \vdash A_1 \vee A_2}\ \mathbf{or\_i_2} \end{array}\right.$$

$$\frac{\Gamma \vdash A_1 \to A_2 \quad \Gamma \vdash A_1}{\Gamma \vdash A_2}\ \mathbf{imp\_l}$$

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \to A_2}\ \mathbf{imp\_r}$$

Quantifiers

$$\frac{\Gamma \vdash \forall x.\, A}{\Gamma \vdash [t/x]A}\ \mathbf{all\_e}$$

$$\frac{\Gamma \vdash [c/x]A}{\Gamma \vdash \forall x.\, A}\ \mathbf{all\_i}^{c}$$

$$\frac{\Gamma \vdash \exists x.\, A \quad \Gamma, [c/x]A \vdash C}{\Gamma \vdash C}\ \mathbf{exists\_e}^{c}$$

$$\frac{\Gamma \vdash [t/x]A}{\Gamma \vdash \exists x.\, A}\ \mathbf{exists\_i}$$

Figura 2.3: Deduzione Naturale per la Logica Classica

- le logiche *classiche*, basate sulla nozione di verità. Sono state notevolemente rivalutate ultimamente per la rappresentazione di sistemi concorrenti.

- le logiche *intuizionistiche*, che si rifanno alla tradizione costruttivita, sostituendo la nozione di verità classica con quella di evidenza. Queste logiche sono particolarmente vicine alle teorie dei tipi costruttive.

- le logiche *minimali*, che si differenziano da quelle intuizionistiche per il rifiuto dell'uso della negazione come operatore predefinito.

Axiom

$$\frac{}{A \longrightarrow A}\ \mathbf{id} \qquad \frac{\Delta_1 \longrightarrow A, \Theta_1 \quad \Delta_2, A \longrightarrow \Theta_2}{\Delta_1, \Delta_2 \longrightarrow \Theta_1, \Theta_2}\ \mathbf{cut}$$

Cut

Multiplicatives

$$\frac{\Delta \longrightarrow \Theta}{\Delta, \mathbf{1} \longrightarrow \Theta}\ \mathbf{one\_l} \qquad \frac{}{\cdot \longrightarrow \mathbf{1}}\ \mathbf{one\_r}$$

$$\frac{}{\bot \longrightarrow \cdot}\ \mathbf{bot\_l} \qquad \frac{\Delta \longrightarrow \Theta}{\Delta \longrightarrow \bot, \Theta}\ \mathbf{bot\_r}$$

$$\frac{\Delta, A_1, A_2 \longrightarrow \Theta}{\Delta, A_1 \otimes A_2 \longrightarrow \Theta}\ \mathbf{times\_l} \qquad \frac{\Delta_1 \longrightarrow A_1, \Theta_1 \quad \Delta_2 \longrightarrow A_2, \Theta_2}{\Delta_1, \Delta_2 \longrightarrow A_1 \otimes A_2, \Theta_1, \Theta_2}\ \mathbf{times\_r}$$

$$\frac{\Delta_1, A_1 \longrightarrow \Theta_1 \quad \Delta_2, A_2 \longrightarrow \Theta_2}{\Delta_1, \Delta_2, A_1 \parr A_2 \longrightarrow \Theta_1, \Theta_2}\ \mathbf{par\_l} \qquad \frac{\Delta \longrightarrow A_1, A_2, \Theta}{\Delta \longrightarrow A_1 \parr A_2, \Theta}\ \mathbf{par\_r}$$

$$\frac{\Delta_1 \longrightarrow A_1, \Theta_1 \quad \Delta_2, A_2 \longrightarrow \Theta_2}{\Delta_1, \Delta_2, A_1 \multimap A_2 \longrightarrow \Theta_1, \Theta_2}\ \mathbf{lolli\_l} \qquad \frac{\Delta, A_1 \longrightarrow A_2, \Theta}{\Delta \longrightarrow A_1 \multimap A_2, \Theta}\ \mathbf{lolli\_r}$$

Additives

(No **top_l**) $\qquad \dfrac{}{\Delta \longrightarrow \top, \Theta}\ \mathbf{top\_r}$

$$\frac{}{\Delta, \mathbf{0} \longrightarrow \Theta}\ \mathbf{zero\_l} \qquad \text{(No } \mathbf{zero\_r}\text{)}$$

$$\left.\begin{array}{l} \dfrac{\Delta, A_1 \longrightarrow \Theta}{\Delta, A_1 \& A_2 \longrightarrow \Theta}\ \mathbf{with\_l_1} \\[2ex] \dfrac{\Delta, A_2 \longrightarrow \Theta}{\Delta, A_1 \& A_2 \longrightarrow \Theta}\ \mathbf{with\_l_2} \end{array}\right\} \qquad \frac{\Delta \longrightarrow A_1, \Theta \quad \Delta \longrightarrow A_2, \Theta}{\Delta \longrightarrow A_1 \& A_2, \Theta}\ \mathbf{with\_r}$$

$$\frac{\Delta, A_1 \longrightarrow \Theta \quad \Delta, A_2 \longrightarrow \Theta}{\Delta, A_1 \oplus A_2 \longrightarrow \Theta}\ \mathbf{plus\_l} \qquad \left\{\begin{array}{l} \dfrac{\Delta \longrightarrow A_1, \Theta}{\Delta \longrightarrow A_1 \oplus A_2, \Theta}\ \mathbf{plus\_r_1} \\[2ex] \dfrac{\Delta \longrightarrow A_2, \Theta}{\Delta \longrightarrow A_1 \oplus A_2, \Theta}\ \mathbf{plus\_r_2} \end{array}\right.$$

$$\frac{\Delta \longrightarrow A_1, \Theta \quad \Delta, A_2 \longrightarrow \Theta}{\Delta, A_1 \rightsquigarrow A_2 \longrightarrow \Theta}\ \mathbf{wave\_l} \qquad \left\{\begin{array}{l} \dfrac{\Delta, A_1 \longrightarrow \Theta}{\Delta \longrightarrow A_1 \rightsquigarrow A_2, \Theta}\ \mathbf{wave\_r_1} \\[2ex] \dfrac{\Delta \longrightarrow A_2, \Theta}{\Delta \longrightarrow A_1 \rightsquigarrow A_2, \Theta}\ \mathbf{wave\_r_2} \end{array}\right.$$

Figura 2.4: La Logica Lineare Classica, Additivi e Moltiplicativi

Exponentials *Logical rules*

$$\frac{\Delta, A \longrightarrow \Theta}{\Delta, !A \longrightarrow \Theta}\ !\_l \qquad \frac{!\Delta \longrightarrow A, ?\Theta}{!\Delta \longrightarrow !A, ?\Theta}\ !\_r$$

$$\frac{!\Delta, A \longrightarrow ?\Theta}{!\Delta, ?A \longrightarrow ?\Theta}\ ?\_l \qquad \frac{\Delta \longrightarrow A, \Theta}{\Delta \longrightarrow ?A, \Theta}\ ?\_r$$

Structural rules

$$\frac{\Delta \longrightarrow \Theta}{\Delta, !A \longrightarrow \Theta}\ !\_w \qquad \frac{\Delta \longrightarrow \Theta}{\Delta \longrightarrow ?A, \Theta}\ ?\_w$$

$$\frac{\Delta, !A, !A \longrightarrow \Theta}{\Delta, !A \longrightarrow \Theta}\ !\_c \qquad \frac{\Delta \longrightarrow ?A, ?A, \Theta}{\Delta \longrightarrow ?A, \Theta}\ ?\_c$$

Negation

$$\frac{\Delta \longrightarrow A, \Theta}{\Delta, A^{\perp} \longrightarrow \Theta}\ \mathbf{nil\_l} \qquad \frac{\Delta, A \longrightarrow \Theta}{\Delta \longrightarrow A^{\perp}, \Theta}\ \mathbf{nil\_r}$$

Quantifiers

$$\frac{\Delta, [t/x]A \longrightarrow \Theta}{\Delta, \forall x.\, A \longrightarrow \Theta}\ \mathbf{all\_l} \qquad \frac{\Delta \longrightarrow [c/x]A, \Theta}{\Delta \longrightarrow \forall x.\, A, \Theta}\ \mathbf{all\_r}^{c}$$

$$\frac{\Delta, [c/x]A \longrightarrow \Theta}{\Delta, \exists x.\, A \longrightarrow \Theta}\ \mathbf{exists\_l}^{c} \qquad \frac{\Delta \longrightarrow [t/x]A, \Theta}{\Delta \longrightarrow \exists x.\, A, \Theta}\ \mathbf{exists\_r}$$

Figura 2.5: La Logica Lineare Classica, Esponenzialie e Quantificatori

Dati tutti questi criteri, non resta che definire i connettivi che costituiscono queste logiche. In questa sezione discutiamo brevemente la logica tradizionale. Piú specificatamente, parliamo di logica classica e logica minimale.

Il linguaggio della *logica tradizionale classica* del prim'ordine è specificato dalla seguente grammatica:

$$\begin{array}{lll} A ::= & P & \textit{Formule atomiche} \\ \mid & \top \mid \bot \mid \neg A \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid A_1 \to A_2 & \textit{Connettivi proposizionali} \\ \mid & \forall x.\, A \mid \exists x.\, A & \textit{Quantificatori} \end{array}$$

Le formule atomiche sono definita nella solita maniera. Un sistema deduttivo per questa logica, presentato come calcolo dei sequenti, è mostrato in Figura 2.1.

Axiom

$$\frac{}{\Gamma; A \longrightarrow A}\ \mathbf{id}$$

Structural rule

$$\frac{\Gamma, A; \Delta, A \longrightarrow C}{\Gamma, A; \Delta \longrightarrow C}\ \mathbf{clone}$$

Cut rules

$$\frac{\Gamma; \Delta_1 \longrightarrow A \quad \Gamma; \Delta_2, A \longrightarrow C}{\Gamma; \Delta_1, \Delta_2 \longrightarrow C}\ \mathbf{cut} \qquad \frac{\Gamma; \Delta \longrightarrow A \quad \Gamma, A; \cdot \longrightarrow C}{\Gamma; \Delta \longrightarrow C}\ \mathbf{cut!}$$

Multiplicatives

$$\frac{\Gamma; \Delta \longrightarrow C}{\Gamma; \Delta, \mathbf{1} \longrightarrow C}\ \mathbf{one\_l} \qquad \frac{}{\Gamma; \cdot \longrightarrow \mathbf{1}}\ \mathbf{one\_r}$$

$$\frac{\Gamma; \Delta, A_1, A_2 \longrightarrow C}{\Gamma; \Delta, A_1 \otimes A_2 \longrightarrow C}\ \mathbf{times\_l} \qquad \frac{\Gamma; \Delta_1 \longrightarrow A_1 \quad \Gamma; \Delta_2 \longrightarrow A_2}{\Gamma; \Delta_1, \Delta_2 \longrightarrow A_1 \otimes A_2}\ \mathbf{times\_r}$$

$$\frac{\Gamma; \Delta_1 \longrightarrow A_1 \quad \Gamma; \Delta_2, A_2 \longrightarrow C}{\Gamma; \Delta_1, \Delta_2, A_1 \multimap A_2 \longrightarrow C}\ \mathbf{lolli\_l} \qquad \frac{\Gamma; \Delta, A_1 \longrightarrow A_2}{\Gamma; \Delta \longrightarrow A_1 \multimap A_2}\ \mathbf{lolli\_r}$$

Additives

(No **top_l**) $\qquad \dfrac{}{\Gamma; \Delta \longrightarrow \top}\ \mathbf{top\_r}$

$$\frac{}{\Gamma; \Delta, \mathbf{0} \longrightarrow C}\ \mathbf{zero\_l} \qquad \text{(No } \mathbf{zero\_r}\text{)}$$

$$\left.\begin{array}{l} \dfrac{\Gamma; \Delta, A_1 \longrightarrow C}{\Gamma; \Delta, A_1 \,\&\, A_2 \longrightarrow C}\ \mathbf{with\_l_1} \\[2ex] \dfrac{\Gamma; \Delta, A_2 \longrightarrow C}{\Gamma; \Delta, A_1 \,\&\, A_2 \longrightarrow C}\ \mathbf{with\_l_2} \end{array}\right\} \qquad \frac{\Gamma; \Delta \longrightarrow A_1 \quad \Gamma; \Delta \longrightarrow A_2}{\Gamma; \Delta \longrightarrow A_1 \,\&\, A_2}\ \mathbf{with\_r}$$

$$\frac{\Gamma; \Delta, A_1 \longrightarrow C \quad \Gamma; \Delta, A_2 \longrightarrow C}{\Gamma; \Delta, A_1 \oplus A_2 \longrightarrow C}\ \mathbf{plus\_l} \qquad \left\{\begin{array}{l} \dfrac{\Gamma; \Delta \longrightarrow A_1}{\Gamma; \Delta \longrightarrow A_1 \oplus A_2}\ \mathbf{plus\_r_1} \\[2ex] \dfrac{\Gamma; \Delta \longrightarrow A_2}{\Gamma; \Delta \longrightarrow A_1 \oplus A_2}\ \mathbf{plus\_r_2} \end{array}\right.$$

Figura 2.6: La Logica Lineare Minimale, Additivi e Moltiplicativi

Il linguaggio della *logica tradizionale minimale* del prim'ordine è specificato dalla seguente grammatica:

$$\begin{array}{lll} A ::= & P & \textit{Formule atomiche} \\ & \mid \top \mid \bot \mid A_1 \wedge A_2 \mid A_1 \vee A_2 \mid A_1 \to A_2 & \textit{Connettivi proposizionali} \\ & \mid \forall x.\, A \mid \exists x.\, A & \textit{Quantificatori} \end{array}$$

Exponentials

$$\frac{\Gamma;\cdot \longrightarrow A}{\Gamma;\cdot \longrightarrow\, !A}\,{!\_\mathbf{r}} \qquad \frac{\Gamma, A;\Delta \longrightarrow C}{\Gamma;\Delta, !A \longrightarrow C}\,{!\_\mathbf{d}}$$

Intuitionistics

$$\frac{\Gamma;\cdot \longrightarrow A_1 \quad \Gamma;\Delta, A_2 \longrightarrow C}{\Gamma;\Delta, A_1 \to A_2 \longrightarrow C}\,\mathbf{imp\_l} \qquad \frac{\Gamma, A_1;\Delta \longrightarrow A_2}{\Gamma;\Delta \longrightarrow A_1 \to A_2}\,\mathbf{imp\_r}$$

$$\frac{\Gamma;\Delta, [t/x]A \longrightarrow C}{\Gamma;\Delta, \forall x.\, A \longrightarrow C}\,\mathbf{all\_l} \qquad \frac{\Gamma;\Delta \longrightarrow [c/x]A}{\Gamma;\Delta \longrightarrow \forall x.\, A}\,\mathbf{all\_r}^c$$

$$\frac{\Gamma;\Delta, [c/x]A \longrightarrow C}{\Gamma;\Delta, \exists x.\, A \longrightarrow C}\,\mathbf{exists\_l}^c \qquad \frac{\Gamma;\Delta \longrightarrow [t/x]A}{\Gamma;\Delta \longrightarrow \exists x.\, A}\,\mathbf{exists\_r}$$

Figura 2.7: La Logica Lineare Minimale, Esponenziali ed Intuizionistici

Presentiamo questa logica come calcolo dei sequenti in Figura 2.2 e come sistema di deduzione in Figura 2.3.

2.2 La Logica Lineare

La logica lineare [Gir87] raffina la logica tradizionale internalizzando in parte la gestione dei contesti. Si ottiene una logica in cui risulta molto avegole modellare sistemi basati su risorse come le logiche substrutturali e i costrutti di programmazione imperativi. Di nuovo, presentiamo la versione classica e quella minimale. Questa presentazione è basata, in parte, su [Pfe95a].

Il linguaggio della *logica lineare classica* del prim'ordine è specificato dalla seguente grammatica:

$$\begin{array}{rcl}
A ::= & P & \textit{Formule atomiche} \\
\mid & A^{\perp} & \textit{Negazione} \\
\mid & \mathbf{1} \mid \bot \mid A_1 \otimes A_2 \mid A_1 \wp A_2 \mid A_1 \multimap A_2 & \textit{Moltiplicativi} \\
\mid & \top \mid \mathbf{0} \mid A_1 \,\&\, A_2 \mid A_1 \oplus A_2 \mid A_1 \rightsquigarrow A_2 & \textit{Additivi} \\
\mid & !A \mid ?A & \textit{Esponenziali} \\
\mid & \forall x.\, A \mid \exists x.\, A & \textit{Quantificatori}
\end{array}$$

Un calcolo dei sequenti per questo linguaggio è mostrato in Figure 2.4–2.5.

Il linguaggio della *logica lineare minimale* del prim'ordine è specificato dalla seguente gram-

Axioms

$$\frac{}{\Gamma; A \vdash A}\ \textbf{lvar} \qquad \frac{}{\Gamma, A; \cdot \vdash A}\ \textbf{ivar}$$

Multiplicatives

$$\frac{\Gamma; \Delta_1 \vdash \mathbf{1} \quad \Gamma; \Delta_2 \vdash C}{\Gamma; \Delta_1, \Delta_2 \vdash C}\ \textbf{one\_e} \qquad \frac{}{\Gamma; \cdot \vdash \mathbf{1}}\ \textbf{one\_i}$$

$$\frac{\Gamma; \Delta_1 \vdash A_1 \otimes A_2 \quad \Gamma; \Delta_2, A_1, A_2 \vdash C}{\Gamma; \Delta_1, \Delta_2 \vdash C}\ \textbf{times\_e} \qquad \frac{\Gamma; \Delta_1 \vdash A_1 \quad \Gamma; \Delta_2 \vdash A_2}{\Gamma; \Delta_1, \Delta_2 \vdash A_1 \otimes A_2}\ \textbf{times\_i}$$

$$\frac{\Gamma; \Delta_1 \vdash A_1 \multimap A_2 \quad \Gamma; \Delta_2 \vdash A_1}{\Gamma; \Delta_1, \Delta_2 \vdash A_2}\ \textbf{lolli\_l} \qquad \frac{\Gamma; \Delta, A_1 \vdash A_2}{\Gamma; \Delta \vdash A_1 \multimap A_2}\ \textbf{lolli\_r}$$

Additives

(No **top_e**) $\qquad \dfrac{}{\Gamma; \Delta \vdash \top}\ \textbf{top\_i}$

$$\frac{\Gamma; \Delta_1 \longrightarrow \mathbf{0}}{\Gamma; \Delta_1, \Delta_2 \longrightarrow C}\ \textbf{zero\_e} \qquad \text{(No } \textbf{zero\_i}\text{)}$$

$$\left.\begin{array}{l} \dfrac{\Gamma; \Delta \vdash A_1 \,\&\, A_2}{\Gamma; \Delta \vdash, A_1}\ \textbf{with\_e}_1 \\[2ex] \dfrac{\Gamma; \Delta \vdash A_1 \,\&\, A_2}{\Gamma; \Delta \vdash A_2}\ \textbf{with\_e}_2 \end{array}\right\} \qquad \frac{\Gamma; \Delta \vdash A_1 \quad \Gamma; \Delta \vdash A_2}{\Gamma; \Delta \vdash A_1 \,\&\, A_2}\ \textbf{with\_i}$$

$$\frac{\Gamma; \Delta_1 \vdash A_1 \oplus A_2 \quad \Gamma; \Delta_2, A_1 \vdash C \quad \Gamma; \Delta_2, A_2 \vdash C}{\Gamma; \Delta_1, \Delta_2 \vdash C}\ \textbf{plus\_e} \qquad \left\{\begin{array}{l} \dfrac{\Gamma; \Delta \vdash A_1}{\Gamma; \Delta \vdash A_1 \oplus A_2}\ \textbf{plus\_i}_1 \\[2ex] \dfrac{\Gamma; \Delta \vdash A_2}{\Gamma; \Delta \vdash A_1 \oplus A_2}\ \textbf{plus\_i}_2 \end{array}\right.$$

Figura 2.8: Deduzione Naturale per la Logica Minimale Lineare, Additivi e Moltiplicativi

matica:

$$\begin{array}{rlll} A ::= & P & & \textit{Formule atomiche} \\ \mid & \mathbf{1} \mid A_1 \otimes A_2 \mid A_1 \multimap A_2 & & \textit{Moltiplicativi} \\ \mid & \top \mid \mathbf{0} \mid A_1 \,\&\, A_2 \mid A_1 \oplus A_2 & & \textit{Additivi} \\ \mid & !A & & \textit{Esponenziali} \\ \mid & \forall x.\, A \mid \exists x.\, A \mid A_1 \to A_2 & & \textit{Intuizionistici} \end{array}$$

Mostriamo un calcolo dei sequenti in Figure 2.6–2.7, e un sistema di deduzione naturale, che fungerà da base alla discussione nei capitoli successivi, in Figure 2.8–2.9.

Exponentials

$$\frac{\Gamma;\cdot \vdash A}{\Gamma;\cdot \vdash !A}\,!\_\mathbf{i} \qquad \frac{\Gamma;\Delta_1 \vdash !A \quad \Gamma, A;\Delta_2 \vdash C}{\Gamma;\Delta_1,\Delta_2 \vdash C}\,!\_\mathbf{e}$$

Intuitionistics

$$\frac{\Gamma;\Delta \vdash A_1 \to A_2 \quad \Gamma;\cdot \vdash A_1}{\Gamma;\Delta \vdash A_2}\,\mathbf{imp\_e} \qquad \frac{\Gamma, A_1;\Delta \vdash A_2}{\Gamma;\Delta \vdash A_1 \to A_2}\,\mathbf{imp\_i}$$

$$\frac{\Gamma;\Delta \vdash \forall x.\, A}{\Gamma;\Delta \vdash [t/x]A}\,\mathbf{all\_e} \qquad \frac{\Gamma;\Delta \vdash [c/x]A}{\Gamma;\Delta \vdash \forall x.\, A}\,\mathbf{all\_i}^{c}$$

$$\frac{\Gamma;\Delta_1 \vdash \exists x.\, A \quad \Gamma;\Delta_2, [c/x]A \vdash C}{\Gamma;\Delta_1,\Delta_2 \vdash C}\,\mathbf{exists\_e}^{c} \qquad \frac{\Gamma;\Delta \vdash [t/x]A}{\Gamma;\Delta \vdash \exists x.\, A}\,\mathbf{exists\_i}$$

Figura 2.9: Deduzione Naturale per la Logica Minimale Lineare, Esponenziali ed Intuizionistici

$$\frac{}{\Gamma, x:\tau \vdash x:\tau}\,\mathbf{Ch\_var}$$

$$\frac{\Gamma, x:\tau_1 \vdash M:\tau_2}{\Gamma \vdash \lambda x:\tau_1.\, M:\tau_1 \to \tau_2}\,\mathbf{Ch\_lam} \qquad \frac{\Gamma \vdash M_1:\tau_2 \to \tau_1 \quad \Gamma \vdash M_2:\tau_2}{\Gamma \vdash M_1\, M_2:\tau_1}\,\mathbf{Ch\_app}$$

Figura 2.10: Il λ-calcolo Semplicemente Tipato di Church $\lambda^{\to}$

2.3 La Teoria dei Tipi Traditionale

Le teorie dei tipi comunemente utilizzate in Informatica hanno origine nel λ-calcolo semplicemente tipato di Church $\lambda^{\to}$ [Chu40]. In questa sezione, ci concentriamo su questo formalismo. Estensioni nel senso proprio della teoria dei tipi sono definiti in [Bar92] e verrano introdotti nella successiva discussion.

Il linguaggio del λ-*calcolo semplicemente tipato di Church* $\lambda^{\to}$ è specificato dalla seguente grammatica:

$$\begin{array}{rrcl} \textit{Oggetti:} & M & ::= & x \mid \lambda x:\tau.\, M \mid M_1\, M_2 \\ \textit{Tipi:} & \sigma & ::= & b \mid \tau_1 \to \tau_2 \\ \textit{Contesto:} & \Gamma & ::= & \cdot \mid \Gamma, x:\tau \end{array}$$

Le regole che legano gli oggetti e i tipi sono mostrate in Figure 2.10. È stato presto notato che queste regole assomigliano notevolemente alle regole per il frammento intuizionistico della logica tradizionale che comprende solamente l'implicazione. Questo parallelo, poi notevolmente esteso,

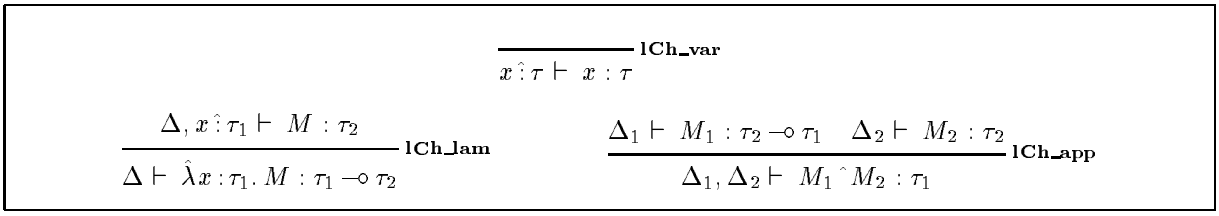


Figura 2.11: Il λ-calcolo Semplicemente Tipato Lineare di Church $\lambda^{\multimap}$

prende il nome di *isomorfismo di Curry-Howard.*

2.4 La Teoria dei Tipi Lineare

Vari λ-calcoli lineari sono stati studiati in letteratura [ABCJ94, Abr93, Tro93]. Presentiamo in questa sezione un semplice raffinamento lineare di $\lambda^{\to}$, che chiamiamo λ-*calcolo lineare semplicemente tipato di Church*, o $\lambda^{\multimap}$ in sigla.

Il linguaggio di $\lambda^{\multimap}$ è specificato dalla seguente grammatica:

$$\begin{array}{rrcl} \textit{Oggetti:} & M & ::= & x \mid \hat{\lambda} x{:}\tau . M \mid M_1 \hat{} M_2 \\ \textit{Tipi:} & \sigma & ::= & b \mid \tau_1 \multimap \tau_2 \\ \textit{Contesto:} & \Gamma & ::= & \cdot \mid \Gamma, x \hat{:} \tau \end{array}$$

Le regole di tipizzazione per questo formalismo sono mostrate in Figura 2.11.

L'isomorfismo di Curry-Howard può essere esteso anche al caso lineare. Lo faremo vedere ampiamente nel corpo di questa tesi. Si veda anche [ABCJ94, GdQ92, Pfe95a].

Capitolo 3

Programmare con la Logica e la Teoria dei Tipi

Lo scopo di questo capitolo è di fornire al lettore un'idea delle questioni di ordine pratico che sorgono quando si tenta di meccanizzare una logica come linguaggio di programmazione. Diamo requisiti affinché un frammento di una logica o di una teoria dei tipi si possa interpretare come un linguaggio di programmazione logica. Motiviamo la nostra discussione con gli esempi classici delle formule di Harrop ereditarie e mostriamo a titolo esemplificativi due formalismi molto vicini al linguaggio che proporremo nei prossimi capitoli: la logica delle formule di Harrop ereditarie lineari e la teoria dei tipi di LF. Parliamo anche degli ulteriori passi necessari all'ottenimento di linguaggi di programmazione logica concreti quali $\lambda Prolog$, *Lolli* ed *Elf* nei casi dei linguaggi presi in considerazione.

Come già nel capitolo precedente, ci limitiamo ad una trattazione sommaria ricca di riferimenti alla letteratura. Invitiamo il lettore interessato a richiedere [Cer96], oppure a consultare direttamente le fonti originarie.

3.1 Progettazione di Linguaggi di Programmazione Logica Basata sulla Teoria della Dimostrazione

Volendo usare un formalismo logico come un linguaggio di programmazione, dobbiamo fornire algoritmi efficienti per scoprire se una formula data è derivabile. Parlare di efficienza in questo contesto è opinabile in quanto per molte logiche di interesse concreto, il problema del proof-search è indecidibile. Tuttavia, in molti casi pratici una procedura di decisione esiste e precisamente in quei casi vogliamo essere in grado di dare una risposta in tempi brevi. La possibilità di raggiungere quest'obiettivo dipende molto dalla logica in esame. Ad esempio, il proof-search è NP-completo già nella logica proposizionale. Pertanto, per raggiungere un'efficienza accettabile, dovremo escludere connettivi. In questa sezione, diamo criteri generali per ottenere frammenti di formalismi logici per cui il proof-search può essere fatto in maniera efficiente. Usiamo la logica

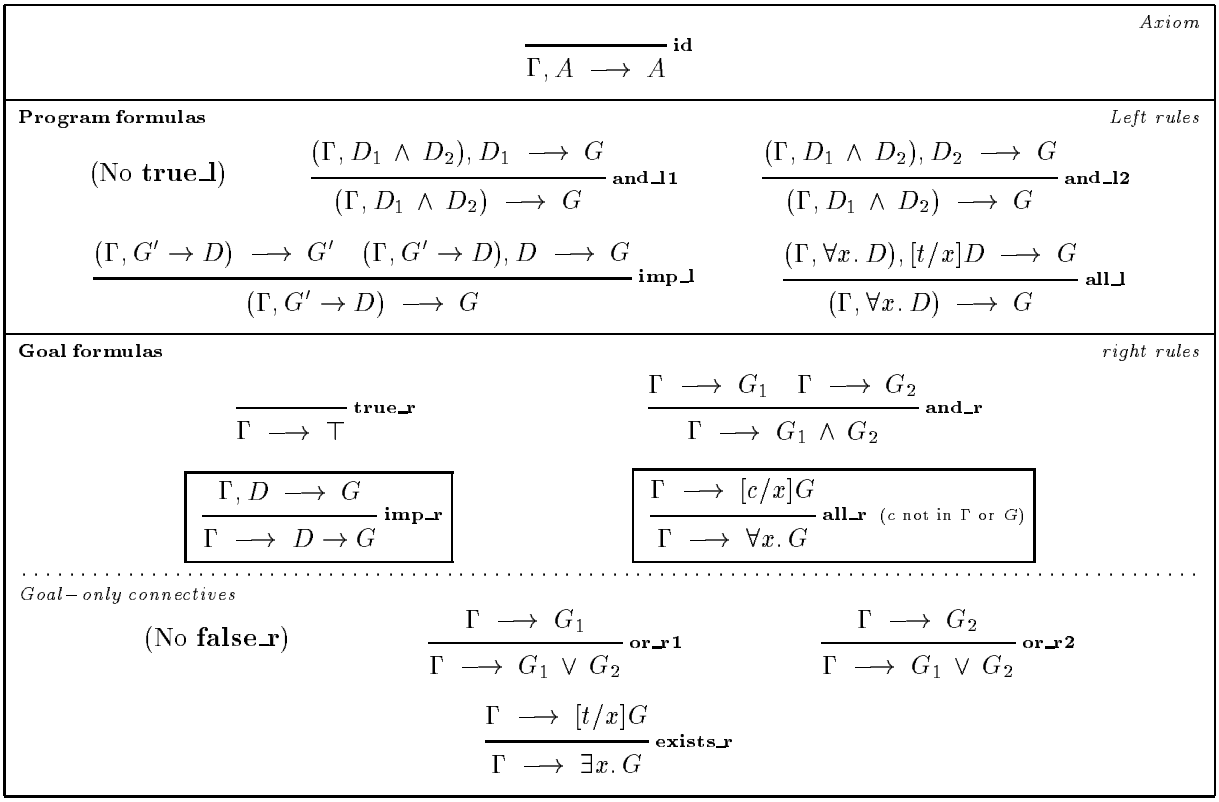


Figura 3.1: Formule di Harrop Ereditarie e Clausole di Horn

tradizionale del prim'ordine come punto di partenza. Si veda [MNPS91] per una trattazione esauriente di questi problemi.

3.1.1 Proof-search Guidata dal Goal

Concentriamoci sul problema di dimostrare la validità di una formula logica G, che chiamiamo *goal*, per mezzo di un insieme di formule Γ, che chiamiamo *programma*. Nella notazione dei sequenti, vogliamo derivare $\Gamma \longrightarrow G$. Grazie al teorema di eliminazione del taglio, sappiamo che per dimostrare questo giudizio, ci basta applicare una regola destra o sinistra alle formule in esso contenute. Nel primo caso, l'unica formula a cui è applicabile è G che determina la regola da usare (ci si può restringere ad usare **id** solamente per formule atomiche). Nel caso di una regola sinistra, dobbiamo scegliere una formula in Γ e poi usare la regola piú appropriata per ridurre il sequente. Quest'opzione è problematica a causa del non-determinismo a cui dà origine. Vorremo invece poter sempre focalizzare la nostra attenzione sul goal, accedendo al programma solamente quando quest'ultimo è atomico. Una logica per cui ogni formula derivabile è derivabile anche

restringendosi a questa strategia è caratterizzata da *proof-search guidato dal goal*. Permettono di interpretare i connettivi logici come direttive di ricerca.

La logica tradizionale non ha questa proprietà. Un'esame dettagliato mostra che gli operatori che la rendono falsa sono il falso $\bot$, la negazione $\neg$, la disgiunzione $\vee$ e il quantificatore esistenziale $\exists$: una loro rimozione porta ad una logica caratterizzata da proof-search guidato dal goal. Possiamo peró distinguere il ruolo delle formule e dei connettivi in dipendenza della posizione dove possono apparire, se nel goal o nel programma. Otteniamo cosí due categorie sintattiche, le *formule goal* e le *formule programma*, che si differenziano per la gamma dei connettivi che ammettono. Fra le prime possiamo reintrodurre gli operatori sopra menzionati (ad eccezione della negazione che è problematica). Fra i secondi non facciamo aggiunte. Queste formule sono definite per mutua ricorsione in quanto solamente l'implicazione permette ad una formula programma di essere usata come goal e viceversa. Il linguaggio risultante è detto *logica delle formule di Harrop ereditarie* ed è definito dalla seguente grammatica:

$$
\begin{array}{lrcl}
\textit{D-formule:} & D & ::= & P \mid \top \mid D_1 \wedge D_2 \mid G \to D \mid \forall x.\, D \\
\textit{G-formule:} & G & ::= & P \mid \top \mid G_1 \wedge G_2 \mid \boxed{D \to G} \mid \boxed{\forall x.\, G} \\
 & & & \quad \mid \bot \mid G_1 \vee G_2 \mid \exists x.\, G
\end{array}
$$

Rimuovendo le produzioni evidenziate dalla definizione di G-formule, otteniamo una delle tante formulazioni della logica delle *clausole di Horn*. Le regole di deduzione per questo formalismo sono mostrate in Figura 3.1. Di nuovo, le regole evidenziate vanno rimosse per dare una semantica alle clausole di Horn.

3.1.2 Focalizzazione

Una volta ridottisi ad una formula atomica dopo aver decomposto un goal, bisogna accedere al programma per andare avanti. Di nuovo abbiamo problemi di non-determinismo. Tuttavia, questo problema viene ridimensionato se ci basta scegliere una formula e poi applicare regole solo ad essa e alle sue sottoformule. Una logica tale che ogni dimostrazione sia equivalente ad una dimostrazione con queste caratteristiche è detta focalizzata nel suo proof-search. Il linguaggio delle formule di Harrop ereditarie e le clausole di Horn hanno questa proprietà. Si osservi che la sua validità dipende dalla formulazione delle regole di deduzione.

3.1.3 Linguaggi di Programmazione Logica Astratti

Una logica caratterizzata da un proof-search guidato dal goal e focalizzata è detta di avere la proprietà delle *dimostrazioni uniformi*. Questa logica è allora un *linguaggio di programmazione logica astratto*. La logica delle clausole di Horn e il linguaggio delle formule di Harrop ereditarie hanno questa caratteristica.

3.1.4 Linguaggi di Programmazione Logica Concreti

L'essere un linguaggio di programmazione logica astratto è un criterio necessario affinché una logica sia dotata di algoritmi di proof-search efficienti, ma non suggerisce esplicitamente quali debbano essere questi algoritmi. Volendo avvicinarsi ad un'implementazione concreta del linguaggio, dobbiamo rendere espliciti questi algoritmi ed eventuali problemi che devono essere gestiti in pratica.

Innanzitutto, si vorrebbe poter interpretare ogni formula di programma come la definizione condizionale di una formula atomica. Sotto quest'ottica, la dimostrazione di una formula atomica si riduce a fare vedere che la sua definizione (o una delle sue definizioni in quanto possono essere piú di una) è derivabile. Pertanto la riduzione del goal ad una formula atomica si dovrebbe interpretare come una *chiamata a procedura*. Una presentazione che mette in rilievo quest'aspetto è detto un sistema di *risoluzione*.

Anche dopo esserci ridotto ad un sistema di risoluzione, l'implementatore si trova a dover risolvere problemi non specificati internamente al formalismo logico. Si tratta questa volta di punti di scelta non-deterministici la cui soluzione dipende dalle caratteristiche che si vogliono dare all'implementazione risultante. Nel caso delle formule di Harrop ereditarie, abbiamo la scelta del sottogoal da risolvere per primo nelle regole binarie (*non-determinismo congiuntivo*), la formula di programma da considerare per un goal atomico o il ramo da tenere per i goal disgiuntivi (*non-determinismo disgiuntivo*) e il termine con cui istanziare i quantificatori (*non-determinismo esistenziale*).

Prese queste decisioni, si ottiene un *linguaggio di programmazione logica concreto* che può venire implementato efficientemente e usato possibilmente per applicazioni non-banali. Molti sviluppatori aumentano questo nucleo logico di base con ulteriori operatori, detti *extra-logici* per guadagnare in espressività. Questo è spesso problematico in quanto fa venir meno l'interpretazione logica dei programmi risultanti.

3.2 Un Esempio dalla Logica Lineare

Come secondo esempio di linguaggio di programmazione logica astratta, consideriamo la logica delle formule di Harrop ereditarie lineari, la cui versione concreta è il linguaggio *Lolli* [Hod94, HM94].

Il linguaggio delle formule di Harrop ereditarie lineari è di particolare interesse per il proseguimento della discussione in quanto è il corrispettivo logico della teoria dei tipi lineare che proponiamo nel corpo di questa tesi.

3.2.1 Le Formule di Harrop Ereditarie Lineari

Il linguaggio delle formule di Harrop ereditarie lineari è un consistente frammento della logica lineare intuizionistica ottenuto come raffinamento lineare dell'omonimo linguaggio intuizionistico.

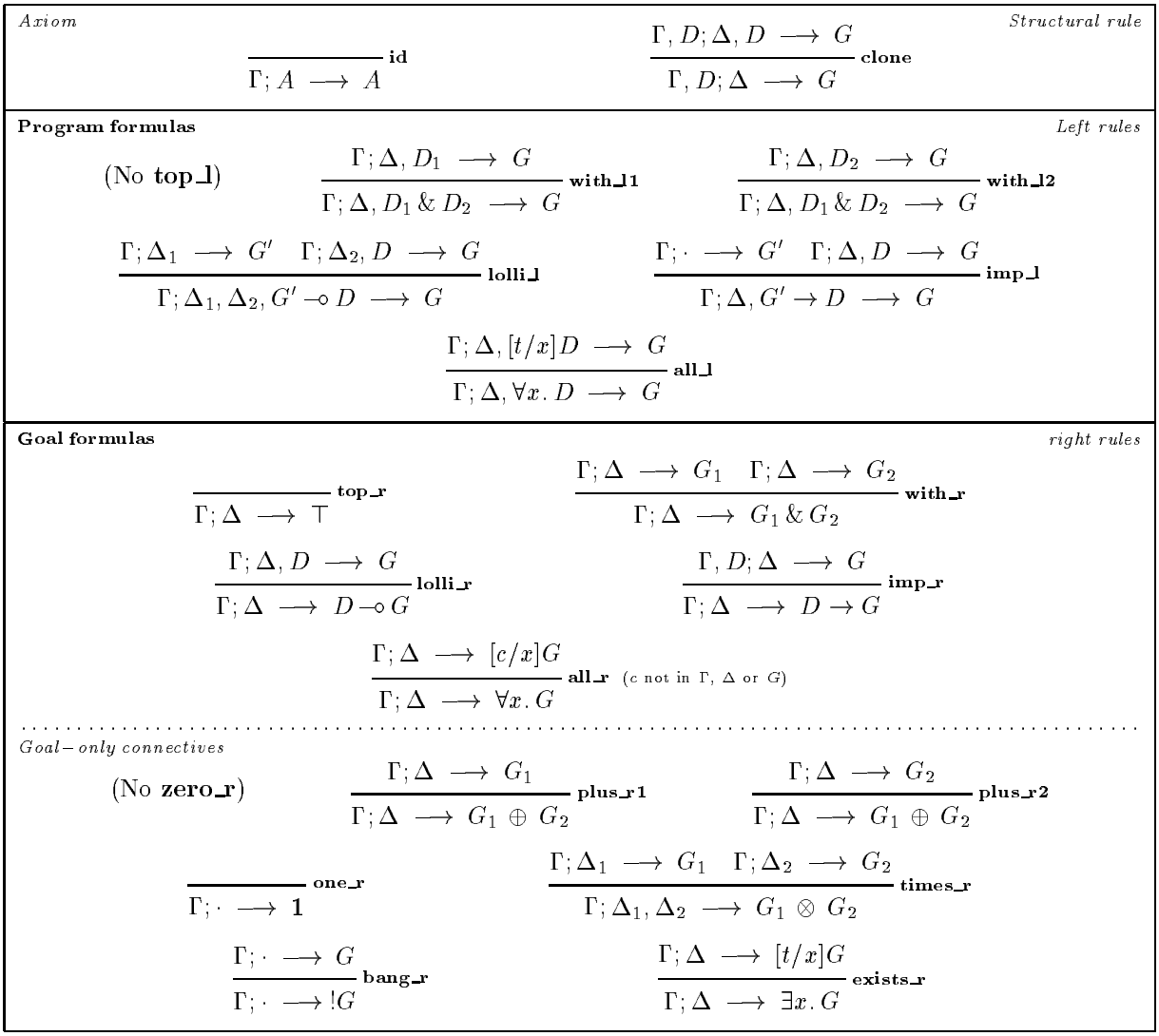


Figura 3.2: Formule di Harrop Ereditarie Lineari

Questa logica è definita dalla seguente grammatica:

$$
\begin{array}{lrcl}
\textit{D-formule:} & D & ::= & P \mid \top \mid D_1 \,\&\, D_2 \mid G \multimap D \mid G \to D \mid \forall x.\, D \\
\textit{G-formule:} & G & ::= & P \mid \top \mid G_1 \,\&\, G_2 \mid D \multimap G \mid D \to G \mid \forall x.\, G \\
 & & & \mid \mathbf{1} \mid \mathbf{0} \mid G_1 \oplus G_2 \mid G_1 \otimes G_2 \mid\ !G\ \mid \exists x.\, G
\end{array}
$$

Un sistema deduttivo per questo linguaggio è mostrato in Figura 3.2.

Come già accade per il caso tradizionale, il linguaggio delle formule di Harrop ereditarie lineari

costituisce un linguaggio di programmazione logica astratta (lineare). La necessità di gestire la linearità non complica la meta-teoria di questo linguaggio se non per l'introduzione di una nuova forma di non-determinismo: la gestione del contesto lineare nelle regole **lolli_l** e **times_r**. Si veda [Hod94, HM94, CHP96] per una trattazione dettagliata di problematiche e soluzioni.

3.2.2 Il Linguaggio *Lolli*

La logica delle formule di Harrop ereditarie lineari è stato implementato come il linguaggio di programmazione logica lineare *Lolli* [Hod94, HM94]. Questo linguaggio si può considerare una estensione lineare di λ*Prolog*.

3.3 Un Esempio dalla Teoria dei Tipi

Il logical framework *LF* è stato progettato da Harper, Honsell e Plotkin [HHP93] come un meta-linguaggio per la rappresentazione ad alto livello di formalismi logici e linguaggi di programmazione. Nel prossimo capitolo, faremo vedere come questo linguaggio viene usato come formalismo di meta-rappresentazione. Parliamo ora invece della teoria dei tipi sottostante e della sua implementabilità come linguaggio di programmazione logica.

LF è l'altro linguaggio da cui prende origine il formalismo che proponiamo in questa tesi. Proponiamo un'estensione di questa teoria dei tipi che include in maniera uniforme tutti i costrutti che caratterizzano la logica delle formule di Harrop ereditarie lineari, o meglio il suo frammento liberamente generato.

3.3.1 Il Logical Framework *LF*

LF [HHP93] è una teoria dei tipi, e piú precisamente un raffinamento del λ-calcolo semplicemente tipato di Church con tipi dipendenti. *LF* è composto da varie entità semantiche, disposte su tre livelli. Vi è innanzitutto il livello degli oggetti. Ogni oggetto appartiene ad un tipo. Sfruttando la presenza dei tipi dipendenti, si possono formare tipi parametrici, detti famiglie di tipi, dove i parametri sono oggetti. Istanziando una famiglia di tipi con oggetti di tipo appropriato, si ottengono dei tipi, che a loro volta classificano oggetti. Tipi e famiglie di tipi costituiscono il livello dei tipi. Infine, le entità del livello dei tipi vengono classificate per mezzo delle cosiddette kind, ottenendo cosí il livello delle kind. Pur non volendo entrare in tecnicismi, è utile dare una visione globale di questi tre livelli mediante la grammatica che li definisce:

$$
\begin{array}{rrcllll}
\textit{Oggetti:} & M & ::= & c & \mid x & \mid \lambda x{:}A.\, M & \mid M_1\, M_2 \\
\textit{Famiglie di tipi:} & P & ::= & a & \mid P\, M & & \\
\textit{Tipi:} & A & ::= & P & \mid \Pi x{:}A_1.\, A_2 & & \\
\textit{Kind:} & K & ::= & \mathrm{TYPE} & \mid \Pi x{:}A.\, K & & \\
\textit{Contesti:} & \Gamma & ::= & \cdot & \mid \Gamma, x{:}A & & \\
\textit{Segnature:} & \Sigma & ::= & \cdot & \mid \Sigma, a{:}K & \mid \Sigma, c{:}A &
\end{array}
$$

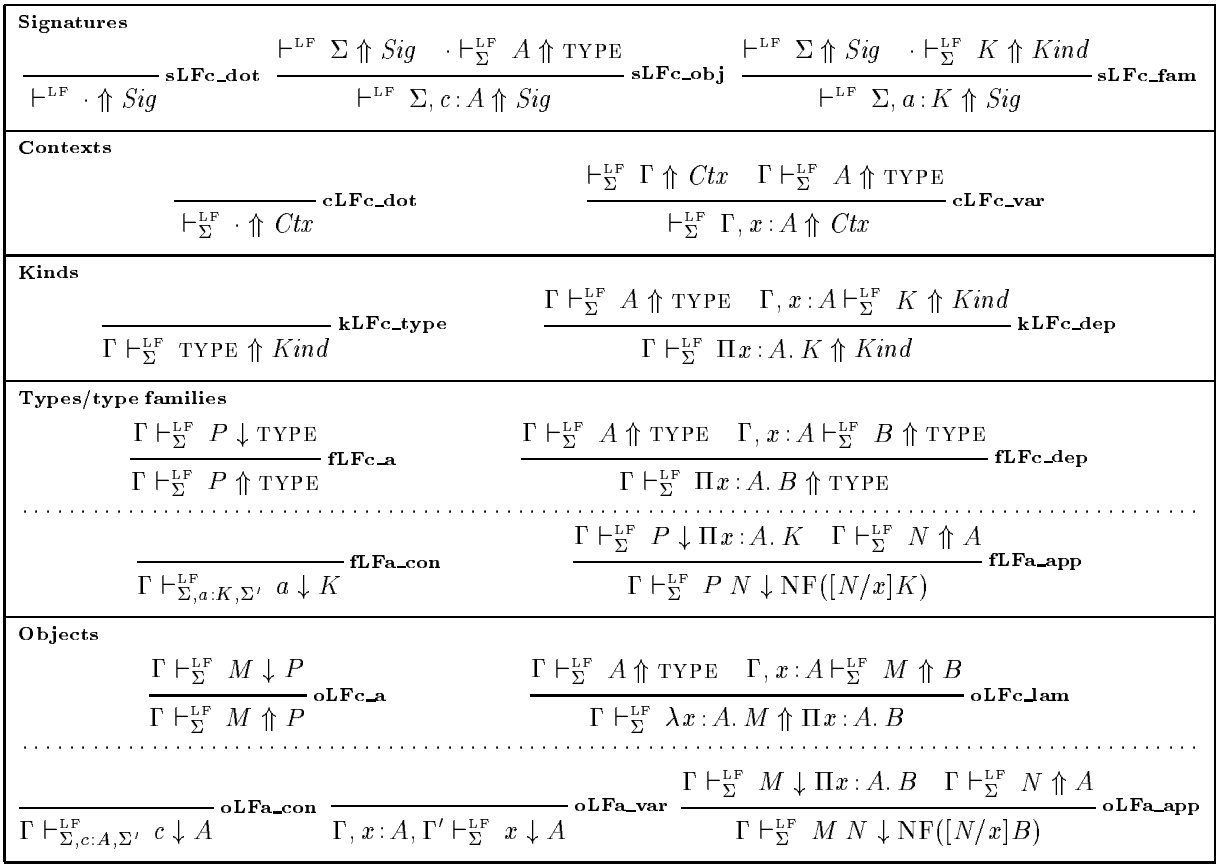


Figura 3.3: Sistema Deduttivo Canonico per LF

x, c ed a fungono da variabili sintattiche che spaziano rispettivamente su variabili e costanti del livello oggetto e costanti di tipo. TYPE è una kind predefinita che classifica tutte le espressioni che sono tipi (contrariamente alle famiglie di tipi). Π è il costruttore di tipi dipendenti. Quando, in $\Pi x:A_1.\, A_2$ ($\Pi x:A.\, K$), x non compare in A_2 (K), la sintassi viene semplificata in $A_1 \to A_2$ ($A \to K$).

LF gode di numerose proprietà desiderabili per un sistema di tipi. In particolare, è possibile definire una nozione di forma canonica per oggetti e tipi. Ogni oggetto ben tipato è riducibile ad un unico oggetto in forma canonica (a meno di α-conversioni). Lo stesso vale al livello di tipi e kind. LF gode pertanto della proprietà di Church-Rosser a tutti i livelli. Entità interriducibili sono equivalenti modulo le regole di $\beta\eta$-conversione. LF è un'estensione molto debole del λ-calcolo semplicemente tipato di Church. Questo fatto, apparentemente limitativo, preserva alcune proprietà fondamentali di quest'ultimo formalismo: la possibilità di usare la sintassi astratta di ordine superiore e quella di vederlo come un linguaggio di programmazione logica. Quali

estensioni di LF preservino la decidibilità del type checking è argomento corrente di ricerca (ad esempio, solo molto recentemente la regola di η-conversione, non considerata in [HHP93], è stata dimostrata accettabile in questo senso).

La teoria dei tipi sottostante ad LF è un linguaggio di programmazione logica astratto [Pfe91]. Si veda anche [Ell90, Pym90, PW90] per analisi del concetto e delle problematiche della programmazione logica nell'ambito delle teorie dei tipi.

3.3.2 Il linguaggio *Elf*

LF è stato implementato come il linguaggio di programmazione logica *Elf* [Pfe91, Pfe94a]. *Elf* è ben piú che un'implementazione di LF. Si tratta infatti di una vera e propria interfaccia interposta tra LF e l'utente che vuole usare questo formalismo per studiare un linguaggio oggetto. Lungi dall'introdurre costrutti spuri, *Elf* mette a disposizione dell'utilizzatore accorgimenti operazionali che semplificano le interazioni con il sistema di tipi. In questa sezione, descriviamo la sintassi concreta di questo linguaggio, che da una parte useremo nel prossimo capitolo per agevolare la scrittura di meta-rappresentazioni scritte in LF. Inoltre, estenderemo la sintassi di questo linguaggio in modo conservativo per includere i costrutti lineare che aggiungeremo a questo formalismo nel corpo di questa tesi.

La seguente tabella associa ogni operatore di LF alla sua rappresentazione concreta. Alcuni operatori ammettono più di una scrittura. In particolare, usiamo la freccia per modellare un costruttore di tipi dipendenti nel cui corpo non compare la variabile legata. Inoltre, offriamo la possibilità di scrivere le implicazioni con gli argomenti rovesciati invertendo il senso della freccia.

	Sintassi astratta	*Sintassi concreta*		
Kind	TYPE	`type`		
	$\Pi x{:}A.\,K$	$\{x:A\}K$	A `->` K	K `<-` A
Tipi	$P\ M$	$P\ \ M$		
	$\Pi x{:}A.\,B$	$\{x:A\}B$	A `->` B	B `<-` A
Oggetti	$\lambda x{:}A.\,M$	$[x:A]M$		
	$M\ N$	$M\ \ N$		

La seguente tabella descrive precedenza e associatività degli operatori introdotti. Ovviamente, si possono usare le parentesi per cambiare la precedenza relativa i questi costrutti. Si noti che il raggio d'azione dei costrutti che legano variabili si estende fino alla fine della dichiarazione corrente, o della prima parentesi chiusa.

Precedenza	*Operatore*	*Posizione*
piú elevata	_ _	associativo a sinistra
	_ -> _	associativo a destra
	_ <- _	associativo a sinistra
piú bassa	{_:_}_ [_:_]_	prefisso a sinistra

Un programma *Elf* rappresenta una segnatura di LF. Ogni dichiarazione $c : A$ viene scritta

$$c: \quad A.$$

e similmente per le dichiarazioni di famiglie di tipo. Le assunzioni presenti nel contesto vengono fatte durante l'esecuzione di un programma *Elf*.

Usiamo `%` per indicare un commento che si estende fino alla fine della riga corrente.

Semplifichiamo la scrittura delle dichiarazioni di LF adottando alcune convenzioni. Innanzitutto, il tipo delle variabili legate dagli operatori di tipo dipendente e λ-astrazione puó essere lasciato implicito ogni volta che è possibile ricostruirlo sulla base delle dichiarazioni circostanti. Il problema della ricostruzione dei tipi è in generale indecidibile, ma ciò è raramente un problema nelle applicazioni concrete. Qualora manteniamo implicito il tipo `A` di una variabile `x` in espressioni di questo tipo, scriviamo `{x}B` e `[x]B` in luogo di `{x:A}B` e `[x:A]B` rispettivamente.

Un'altra utile convenzione è quella di usare identificatori che iniziano con una lettera maiuscola per indicare variabili quantificate da un costruttore di tipo dipendente qualora quest'operatore racchiuda l'intera dichiarazione. In questo modo possiamo omettere questi completamente questi operatori.

Capitolo 4

Meta-representazione

La capacità di fornire rappresentazioni computabili di linguaggi formali, quali le logiche e i linguaggi di programmazione considerati nei capitoli precedenti, è il primo passo nella progettazione di strumenti di sviluppo di dimostrazioni assistiti da computer e verifica automatica di programmi. Infatti, la disponibilità di una maniera concreta per rappresentare inferenze logiche apre le porte alla possibilità di fare verificare loro correttezza da un calcolatore elettronico, il quale può anche aiutarci a ricercarle. Questo è estremamente utile progettando nuove logiche e nuovi linguaggi di programmazione, ma si può anche sfruttare come supporto didattico nell'insegnamento di corsi di logica introduttivi. Similmente, maniere concrete per rappresentare le computazioni effettuate da un linguaggio di programmazione permette la formalizzazione di proprietà di programmi le cui prove possono essere validate o anche parzialmente generate automaticamente. Trasformazioni di programmi quali la compilazione possono venire trattate nella stessa maniera non appena abbiamo a disposizione un modo adeguato per codificare i loro effetti. In particolare la loro correttezza può essere verificata automaticamente.

In questo capitolo, affrontiamo il problema della rappresentazione da una prospettiva tecnica e presentiamo la metodologia adottata nel logical framework LF. Specificamente, analizziamo la struttura di linguaggi formali e sistemi deduttivi in Sezione 4.1, presentiamo astrattamente le tecniche di meta-rappresentazione utilizzate in LF in Sezione 4.2, e le applichiamo alla rappresentazione di alcuni aspetti della meta-teoria del linguaggio di programmazione funzionale *Mini-ML* che usiamo come esempio concreto. Diamo una presentazione informale di questo linguaggio in Sezione 4.3 e mostriamo come la sua sintassi, la sua semantica operazionale e un meta-teorema vengono codificati in LF in Sezione 4.4. Il codice *Elf* completo per quest'esempio si trova in Appendice A.

4.1 Meta-rappresentazione di Linguaggi Formali

In questa sezione, analizziamo come i diversi aspetti di sistemi formali interagiscono e come queste componenti sono strutturate. Tutti questi aspetti si devono tenere in considerazione se si vuole

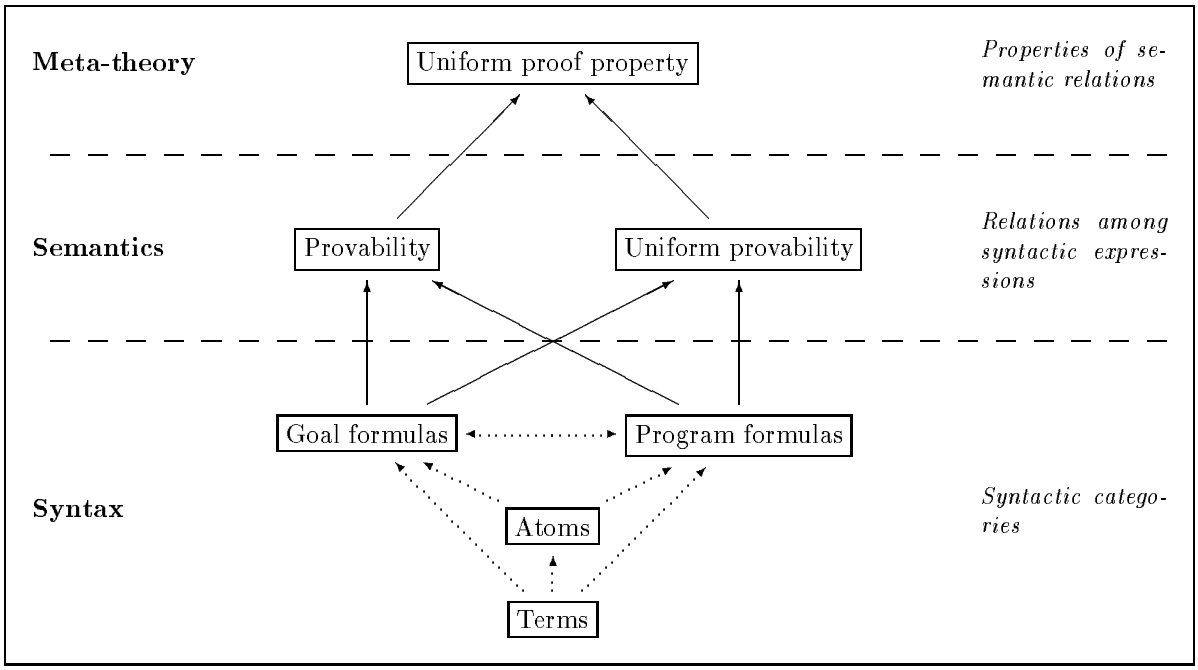


Figura 4.1: Organizzazione della Presentazione delle Formule di Harrop Ereditarie

dare una rappresentazione effettiva di un linguaggio oggetto. In Sottosezione 4.1.1, analizziamo la strutturazione di un sistema formale in livelli e parliamo di giudizi. Poi, in 4.1.2, studiamo la struttura di giudizi e regole d'inferenza. Infine in 4.1.3, parliamo della rappresentazione di linguaggi formali.

4.1.1 Struttura di un Sistema Formale

Un sistema formale è strutturato in una serie di *livelli*. Il livello piú basso è quello della *sintassi*, che descrive quali espressioni sono ammesse nel linguaggio. Subito sopra vi è il livello della *semantica* che descrive il significato dei vari operatori presenti nel linguaggio. Molte applicazioni, quali la programmazione o l'uso di logiche come strumenti di codifica si basano su questo livello. Sopra la semantica, troviamo il livello della *meta-teoria* in cui vengono prese in considerazione proprietà del linguaggio ed operano su oggetti semantici. Questo livello viene preso in considerazione nei libri di logica matematica o progettando un linguaggio di programmazione. Ulteriori livelli possono essere definiti anche se sono raramente utili in pratica. Si veda [Pfe92] per una trattazione completa. Ad esempio, il livello in cui si studiano proprietà di proprietà di linguaggi formali, e delle loro dimostrazioni, prende il nome di *meta-meta-teoria*. Mostriamo in Figura 4.1 la strutturazione in livelli della nostra presentazione delle formule di Harrop ereditarie lineari in

Sezione 3.2.1.

I vari livelli sono strutturati in maniera uniforme come una serie di relazioni dette *giudizi* su entità dei livelli sottostanti. La validità di un giudizio è espressa mediante un *sistema deduttivo* costituito da *regole d'inferenza*. La maniera in cui le regole d'inferenza vengono concatenate a mo' di evidenza della validità di un dato giudizio prende il nome di *derivazione*. Questi concetti, definiti da Martin-Löf, sono discussi in [ML85a, ML85b].

Al livello sintattico, i giudizi sono le relazioni di appartenenza alle categorie sintattiche del linguaggio in esame. Il sistema deduttivo è costituito dalla grammatica che lo definisce, con le singole produzioni come regole d'inferenza. Le derivazioni corrispondono ai parse-tree, o piú comunemente agli elementi della *sintassi astratta* del linguaggio, che si ottengono dai parse-tree eliminando le stringhe concrete che vi compaiono e usando i simboli terminali del linguaggio come nomi per le produzioni.

Al livello semantico, troviamo giudizi quali la derivabilità di una formula, la valutazione di un'espressione o la tipizzazione. Regole d'inferenza, sistemi deduttivi e derivazioni sono le omonime entità che abbiamo già incontrate nei capitoli precedenti.

Ad un primo livello di approssimazione, i giudizi del livello della meta-teoria sono le proprietà stesse che lo compongono, i sistemi deduttivi sono le logiche che si usano per dimostrarli e le derivazioni sono le loro dimostrazioni. Questa prospettiva è tuttavia difficilmente automatizzabile per la sua generalità [Sch95]. Tuttavia, molte di queste proprietà hanno la forma "per ogni x esiste y tale che $P(x, y)$" e per esse esistono dimostrazioni costruttive, ossia dato un valore per x permettono di calcolare uno specifico y per cui $P(x, y)$ risulti valida. La dimostrazione di questa proprietà implementa pertanto una funzione totale $y = P(x)$. Se scriviamo questa funzione come la relazione $P(x, y)$, facciamo rientrare anche questo caso nello schema di Martin-Löf: P è il giudizio, il sistema deduttivo viene estratto dalla dimostrazione della proprietà, e le derivazioni corrispondono al cammino che si deve seguire nella dimostrazione per ottenere $P(x, y)$, o anche per generare un y tale che questa relazione valga [PR96, Roh96]. Si noti che dimostrazioni diverse dànno luogo a giudizi e sistemi deduttivi diversi, che si possono mettere in relazione al livello della meta-meta-teoria. Si osservi inoltre che la costruttività della dimostrazione è cruciale per estrarne un sistema deduttivo. Questa tecnica fu originariamente proposta da Gödel nel suo sistema T [Göd90].

Concludiamo questa sezione con una tabella che riassume l'interpretazione dei concetti introdotti per i livelli della sintassi, semantica, meta-teoria e meta-meta-teoria.

Livello	*Giudizi*	*Variabili schematiche*	*Sistemi deduttivi*	*Derivazioni*
...	...	...	...	...
Meta-meta-teoria	Proprietà di proprietà	Istanze di dimostrazioni, derivazioni, parse-tree	Meta-dimostrazioni	Istanze di meta-dimostrazioni
Meta-teoria	Proprietà di relazioni semantiche	Derivazioni e parse-tree	Dimostrazioni	Istanze di dimostrazioni
Semantica	Relazioni su entità sintattiche	Parse-tree	Sistemi deduttivi	Derivazioni
Sintassi	Categoria sintattica	Stringhe grezze	Grammatiche	Parse-tree

4.1.2 Anatomia di un Sistema Deduttivo

In questa sezione studiamo la struttura di giudizi e sistemi deduttivi comunemente usati in Informatica e li associamo a connettivi logici particolarmente adeguati per rappresentarli. Logiche non contenenti questi connettivi sono spesso incapaci di dare meta-rappresentazioni usabili a questi sistemi formali. Questo studio si focalizza sulla rappresentazione della nozione di derivabilità, non di derivazione che affronteremo nella prossima sezione.

Nella sezione precedente, abbiamo definito un giudizio come una relazione tra derivazioni di livello sottostante. Questa definizione è imprecisa in quanto non comprende espressioni del tipo $\Gamma \longrightarrow G$, che abbiamo usati come giudizi, ma che includono entità, Γ in questo caso, che non sono propriamente derivazioni. Accade spesso che un giudizio menzioni esplicitamente o implicitamente insiemi strutturati che formano il loro *contesto*. In questo, caso, Γ è un insieme semplice, in logica lineare si hanno multiinsiemi, in LF liste. La maniera piú conveniente di rappresentare il contesto di un formalismo oggetto si ha per mezzo di assunzioni nel contesto del linguaggio di meta-rappresentazione, che deve pertanto essere abbastanza ricco da poter codificare la struttura del contesto oggetto. Rappresentazioni dirette come termini di meta-livello non sono adeguate in quanto obbligano a fornire esplicitamente codice per le operazioni necessarie ad accedere e manipolare il contesto oggetto. Questo complica la meta-rappresentazione fino al punto di renderla inutilizzabile. L'idea è invece fare gestire queste operazioni internamente al meta-linguaggio.

Le regole d'inferenza che costituiscono un comune sistema deduttivo presentano una struttura ricorrente che permette di classificarle. Ad ogni classe corrisponde un connettivo logico, o un suo uso, il quale viene utilizzato per dare una meta-rappresentazione a tutte le regole che vi ricadono. Gli schemi di regole piú comuni sono mostrati nelle due colonne piú a sinistra di Figura 4.2. Le altre colonne mostrano la loro rappresentazione in varie logiche.

Una regola d'inferenza viene rappresentato come una formula nel contesto della logica di meta-rappresentazione. Per codificare sistemi deduttivi che non alterano mai la composizione del loro contesto, è sufficiente la logica delle clausole di Horn, su cui *Prolog*, è basato. Questo corrisponde

Issue	Template	Horn clauses	Hereditary Harrop formulas $[LF]$	Linear hereditary Harrop formulas
Rule with no premisses	$\dfrac{}{\Gamma;\Delta \longrightarrow C}$	C	C	$\top \multimap C$
Rule with one premiss	$\dfrac{\Gamma;\Delta \longrightarrow P}{\Gamma;\Delta \longrightarrow C}$	$P \to C$	$P \to C$	$P \multimap C$
Rule with multiple premisses	$\dfrac{\Gamma;\Delta \longrightarrow P_1 \quad \Gamma;\Delta \longrightarrow P_2}{\Gamma;\Delta \longrightarrow C}$	$P_1 \wedge P_2 \to C$	$P_1 \wedge P_2 \to C$ $[P_1 \to P_2 \to C]$	$P_1 \,\&\, P_2 \multimap C$
Access to the context	$\dfrac{\Gamma;\Delta \longrightarrow P}{\Gamma, A;\Delta \longrightarrow C}$	$A \wedge P \to C$	$A \wedge P \to C$ $[P \to A \to C]$	$A \to P \multimap C$
Parametric rules	$\dfrac{\Gamma;\Delta \longrightarrow P^c}{\Gamma;\Delta \longrightarrow C}\,(c)$		$(\forall c.\, P) \to C$ $[(\Pi c : \_.\, P) \to C]$	$(\forall c.\, P) \multimap C$
Hypothetical rules	$\dfrac{\Gamma, A;\Delta \longrightarrow P}{\Gamma;\Delta \longrightarrow C}$		$(A \to P) \to C$	$(A \to P) \multimap C$
Constraints	$\dfrac{\ldots}{\Gamma;\cdot \longrightarrow C}$			$\ldots \to C$
Context splitting	$\dfrac{\Gamma;\Delta_1 \longrightarrow P_1 \quad \Gamma;\Delta_2 \longrightarrow P_2}{\Gamma;\Delta_1, \Delta_2 \longrightarrow C}$			$P_1 \otimes P_2 \multimap C$
Augmentation	$\dfrac{\Gamma;\Delta, A \longrightarrow P}{\Gamma;\Delta \longrightarrow C}$			$(A \multimap P) \multimap C$
Remotion	$\dfrac{\Gamma;\Delta \longrightarrow P}{\Gamma;\Delta, A \longrightarrow C}$			$A \otimes P \multimap C$
Modification	$\dfrac{\Gamma;\Delta, B \longrightarrow P}{\Gamma;\Delta, A \longrightarrow C}$			$A \otimes (B \multimap P) \multimap C$
Extensional rules	[for each t] $\dfrac{\Gamma \longrightarrow P(t)}{\Gamma \longrightarrow C}$			

Figura 4.2: Schemi di Regole d'Inferenza e la loro Rappresentazione in Varie Logiche

ai primi quattro casi in figura, dove si mostrano i connettivi associati. Sistemi deduttivi che fanno assunzioni nel contesto oppure che si basano sulla nozione di parametri necessitano logiche che permettono la risoluzione di goal implicazione ed universali, quali il linguaggio delle formule

di Harrop ereditarie su cui *λProlog* è fondato, oppure *LF* nell'ambito della teoria dei tipi. La maggior parte delle logiche tradizionali e dei linguaggi di programmazione puramente funzionali o logici si possono formulare in modo da ricadere in sistemi deduttivi con regole di questo genere.

Le logiche substrutturali e i linguaggi di programmazione imperativi accedono al contesto per mezzo di operazioni piú complesse quali la rimozione di assunzioni, la loro modifica (ad esempio volendo modellare l'assegnamento), richieste che sia vuoto, eccetera. Possiamo in generale modellare questi sistemi formali mediante giudizi della forma $\Gamma; \Delta \longrightarrow G$ in cui Γ è detto *contesto permanente* e si comporta come illustrato sopra (le assunzioni vengono al piú inserite), mentre Δ viene chiamato *contesto volatile* e permette operazioni arbitrarie. Le logiche lineari, nelle presentazioni in cui il contesto viene partizionato il intuizionistico e lineare, offrono strumenti adeguati per modellare queste situazioni. La metodologia di rappresentazione nel linguaggio delle formule di Harrop ereditarie lineari su cui *Lolli* è basato è mostrato nella colonna piú a destra di Figura 4.2.

In questa tesi, ci baseremo unicamente su regole di questo tipo. Altri schemi sono possibili ed alcuni hanno rilevanza concreta. Mostriamo in figura uno schema di regole estensionali che richiede che una loro premessa valga per un insieme non noto a priori di istanze. Nessuna delle logiche considerata piú sopra è in grado di dare rappresentazioni adeguate a questo schema. Alcune delle loro implementazioni concrete, *Prolog*, *λProlog* e *Lolli* ad esempio, offrono questa possibilità per la presenza di negazione per fallimento finito fra i loro operatori built-in.

4.1.3 Rappresentazione di Linguaggi Formali

I giudizi sono relazioni e vengono pertanto naturalmente rappresentati da predicati in una logica. La rappresentazione degli oggetti che correlano è una questione piú delicata. Abbiamo visto che gli argomenti di un giudizio sono sempre derivazioni. Pertanto, la loro codifica deve appoggiarsi su operatori che riflettono in pieno la ricca varietà di modi in cui le regole di inferenze vengono usate per costruire derivazioni. Questi operatori si possono tuttavia ottenere in maniera sistematica interpretando i connettivi logici identificati nella precedente sottosezione come costruttori di tipi in una teoria dei tipi appropriata. Il λ-calcolo ottenuto considerando i loro costruttori e distruttori ha la giusta potenza espressiva per modellare le derivazioni ottenute con gli schemi di regole corrispondenti. L'estensione dell'isomorfismo di Curry-Howard necessaria dipende dagli schemi che siamo interessati a modellare.

Le derivazioni corrispondenti alle regole d'inferenza che si possono rappresentare nella logica delle clausole di Horn hanno una struttura molto semplice. I termini della logica del prim'ordine, presenti in *Prolog* e *Lolli*, sono sufficienti per questo scopo. Non appena vogliamo rappresentare derivazioni ottenute per mezzo di regole parametriche o ipotetiche, dobbiamo estendere il linguaggio con la λ-astrazione e quindi passare ad un λ-calcolo vero e proprio in luogo del semplice linguaggio di termini che caratterizzava il caso precedente. Linguaggio basati su una logica di ordine superiore quali *λProlog* e *Elf* offrono questa possibilità. Infine derivazioni ottenute per mezzo di regole d'inferenza che operano su un contesto volatile, la cui codifica richiede una logica lineare, devono venire rappresentate in un λ-calcolo lineare. A nostra conoscenza, non vi è

alcun linguaggio di programmazione logica, a parte la proposta di cui questa tesi è l'oggetto, che contenga un tale linguaggio.

Passiamo ora alla formalizzazione della nozione di meta-rappresentazione e alla definizione di criteri di correttezza. Una *meta-rappresentazione* di un insieme J di giudizi da un formalismo oggetto $\mathcal{L}_o$ in un meta-linguaggio $\mathcal{L}_\mu$ è una funzione $\ulcorner\_\urcorner$ che associa ogni giudizio in J ad una formula atomica in $\mathcal{L}_\mu$ e ogni derivazione menzionata in essi ad un termine nel meta-linguaggio. Una definizione piú specifica verrà fornita nel caso di LF.

Per essere di qualche utilità, la funzione di meta-rappresentazione deve codificare giudizi o derivazioni distinti del linguaggio oggetto come entità diverse nel meta-linguaggio. Richiediamo pertanto che questa funzione sia iniettiva. Se il codominio associato ad un giudizio possiede certe caratteristiche di uniformità, è spesso desiderabile che $\ulcorner\_\urcorner$ sia una biiezione. Questi requisiti vengono detti *adeguatezza sintattica.*

Oltre a questa caratterizzazione sintattica, ci aspettiamo che una formula di meta-livello rappresentante un giudizio di J sia dimostrabile se e solo se il giudizio è esso stesso derivabile nel formalismo oggetto. Questa è l'*adeguatezza semantica.* In LF, adeguatezza sintattica e semantica coincidono poiché in questo formalismo la derivabilità è sempre accompagnata da una derivazione che la giustifica, e gli oggetti sono sempre derivazioni di qualche giudizio.

Ogniqualvolta codifichiamo un formalismo oggetto in un meta-linguaggio, dobbiamo fornire evidenza dell'adeguatezza sintattica e semantica della funzione di rappresentazione usata enunciando e dimostrando i *teoremi di adeguatezza* corrispondenti. Questo passo viene spesso omesso o perché la corrispondenza è ovvia, oppure perché è troppo complessa o noiosa da dimostrare. In questa tesi, enunceremo sempre i nostri teoremi di adeguatezza, ma li dimostreremo solamente quando introducono novità sostanziali non apparenti in risultati precedenti.

4.2 Meta-rappresentazione nel Logical Framework *LF*

Il logical framework LF adotta in pieno la metodologia di meta-rappresentazione delineata nella sezione precedente. Consideriamo ad esempio la codifica in questo linguaggio di giudizi oggetto della forma $\Gamma \longrightarrow C$, dove Γ è un contesto permanente e C una formula di un qualche livello sottostante. Applicando ricorsivamente la tecnica che stiamo presentando, codifichiamo i giudizi di cui C è una derivazione per mezzo di un tipo τ di LF. Gli elementi di Γ vengono rappresentati da opportune assunzioni nel contesto del meta-linguaggio. Il giudizio schematico $\Gamma \longrightarrow \_$ viene rappresentato mediante la costante di tipo J avente kind $\tau \to \mathrm{TYPE}$. Pertanto, ogni istanza $\Gamma \longrightarrow C$ di questo schema è un tipo di base della forma $\mathrm{J}\, t_C$ dove t_C è la rappresentazione di C.

Le regole d'inferenza che definiscono il giudizio $\Gamma \longrightarrow C$ sono rappresentate da dichiarazioni $\mathrm{R} : A$ nella segnatura. R identifica la regola, mentre A ne rappresenta la struttura mediante la tecnica vista nella sezione precedente, riproposta schematicamente nel caso di LF in Figura 4.3.

Una derivazione di $\Gamma \longrightarrow C$ è rappresentata da un oggetto M di LF che rende derivabile il giudizio di meta-livello $\Gamma' \vdash^{\mathrm{LF}}_{\Sigma} M \Uparrow \mathrm{J}\, t_C$, dove Σ contiene la codifica del sistema deduttivo di

Issue	Template	Declaration	Derivation
Judgment	$\Gamma \longrightarrow C$	$\mathrm{J} \;:\; \tau \to \mathrm{TYPE}$	$\mathrm{J}\ t_C$
Rule with no premisses	$\dfrac{}{\Gamma \longrightarrow C}\,\mathrm{r}$	$\mathrm{R} \;:\; \Pi X_1{:}\tau_1.\ldots\Pi X_n{:}\tau_n.\ \mathrm{J}\ t_C$	$\mathrm{R}\ t_1\ \ldots\ t_n$
Rule with one premiss	$\dfrac{\Gamma \longrightarrow P}{\Gamma \longrightarrow C}\,\mathrm{r}$	$\mathrm{R} \;:\; \Pi X_1{:}\tau_1.\ldots\Pi X_n{:}\tau_n.\ \mathrm{J}\ t_P \to \mathrm{J}\ t_C$	$\mathrm{R}\ t_1\ \ldots\ t_n\ \mathcal{D}_P$
Rule with multiple premisses	$\dfrac{\Gamma \longrightarrow P_1 \quad \Gamma \longrightarrow P_2}{\Gamma \longrightarrow C}\,\mathrm{r}$	$\mathrm{R} \;:\; \Pi X_1{:}\tau_1.\ldots\Pi X_n{:}\tau_n.\ \mathrm{J}\ t_{P_1} \to \mathrm{J}\ t_{P_2} \to \mathrm{J}\ t_C$	$\mathrm{R}\ t_1\ \ldots\ t_n\ \mathcal{D}_{P_1}\ \mathcal{D}_{P_2}$
Access to the context	$\dfrac{\Gamma \longrightarrow P}{\Gamma, A \longrightarrow C}\,\mathrm{r}$	$\mathrm{R} \;:\; \Pi X_1{:}\tau_1.\ldots\Pi X_n{:}\tau_n.\ \mathrm{J}\ t_P \to \mathrm{J}\ t_A \to \mathrm{J}\ t_C$	$\mathrm{R}\ t_1\ \ldots\ t_n\ \mathcal{D}_P\ \mathcal{D}_A$
Parametric rules	$\dfrac{\Gamma \longrightarrow P^c}{\Gamma \longrightarrow C}\,\mathrm{r}\,(c)$	$\mathrm{R} \;:\; \Pi X_1{:}\tau_1.\ldots\Pi X_n{:}\tau_n.\ (\Pi c{:}\tau_c.\,\mathrm{J}\ t_P) \to \mathrm{J}\ t_C$	$\mathrm{R}\ t_1\ \ldots\ t_n\ (\lambda c{:}\tau_c.\,\mathcal{D}_P)$
Hypothetical rules	$\dfrac{\Gamma, A \longrightarrow P}{\Gamma \longrightarrow C}\,\mathrm{r}$	$\mathrm{R} \;:\; \Pi X_1{:}\tau_1.\ldots\Pi X_n{:}\tau_n.\ (\mathrm{J}\ t_A \to \mathrm{J}\ t_P) \to \mathrm{J}\ t_C$	$\mathrm{R}\ t_1\ \ldots\ t_n\ (\lambda x{:}\mathrm{J}\ t_A.\,\mathcal{D}_P)$

Figura 4.3: Metodologia di Rappresentazione in *LF*

livello oggetto e Γ' la rappresentazione di Γ. Facciamo vedere i termini risultanti nella colonna di destra della figura.

Il trattamento del livello sintattico deve essere specializzato a causa della presenza di variabili, che sono caratterizzate da complesse condizioni di visibilità e da operazioni specifiche quali la sostituzione. Una loro codifica diretta come descritto sopra obbliga a dare una rappresentazione anche a queste condizioni ed operazioni, rendendo l'intera meta-rappresentazione cosí pesante da non permetterne nessun uso concreto. Il livello sintattico viene invece rappresentato mediante la tecnica della *sintassi astratta di ordine superiore*: le variabili oggetto sono rappresentate come variabili di *LF*, i costrutti che le legano mediante operatori di tipo funzionale in modo da usare la λ-astrazione del meta-linguaggio come binder universale, e la sostituzione mediante l'applicazione e conseguente β-riduzione di *LF*.

In fin dei conti, una *meta-rappresentazione* di un insieme di giudizi J di un formalismo oggetto $\mathcal{L}_o$ in *LF* è una funzione $\ulcorner\_\urcorner$ che associa ogni giudizio schematico in J ad un tipo di base e ogni derivazione per questi giudizi ad un oggetto canonico di questo tipo. Giudizi sintattici vengono invece gestiti mediante sintassi astratta di ordine superiore.

Un tipico teorema di adeguatezza per una rappresentazione in LF richiede che la funzione di meta-rappresentazione $\ulcorner \_ \urcorner$ sia una biiezione composizionale tra giudizi derivabili in J e tipi di base abitati di LF, e tra derivazioni di livello oggetto per questi giudizi e oggetti canonici di quei tipi. La composizionalità richiede che la rappresentazione di una sostituzione oggetto coincida con la sostituzione di meta-livello delle rappresentazioni delle entità in gioco. Serve per l'adeguatezza della codifica relativamente alla sintassi astratta di ordine superiore.

4.3 Il Linguaggio *Mini-ML*

In questa sezione e nella prossima mostriamo una rappresentazione concreta in LF. Consideriamo una variante del linguaggio funzionale *Mini-ML* [CDDK86, HM90, MP91, Pfe92] e mostriamo come codificare in LF la sintassi, la semantica e un meta-teorema. Aggiungeremo costrutti imperativi, non trattabili in LF, nel Capitolo 6 e faremo vedere come dare loro una rappresentazione effettiva nell'estensione lineare di LF che proponiamo.

In questa sezione diamo una descrizione informale di *Mini-ML* e della proprietà che vogliamo formalizzare. La rappresentazione in LF è l'oggetto della Sezione 4.4.

4.3.1 Espressioni

Mini-ML è un piccolo linguaggio di programmazione funzionale basato su un λ-calcolo semplicemente tipato à la Curry. Contiene numerali, espressioni condizionali, coppie, definizioni polimorfe, ricorsione e naturalmente espressioni funzionali. Il livello sintattico consiste di espressioni e tipi, ed è specificato dalla seguente grammatica:

Espressioni:	e	$::= x$	(*Variabili*)
		$\mid \mathbf{z} \mid \mathbf{s}\ e \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2$	(*Numeri naturali*)
		$\mid \langle e_1, e_2 \rangle \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e$	(*Coppie*)
		$\mid \mathbf{lam}\ x.\, e \mid e_1\ e_2$	(*Funzioni*)
		$\mid \mathbf{letval}\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{letname}\ x = e_1\ \mathbf{in}\ e_2$	(*Definizioni*)
		$\mid \mathbf{fix}\ x.\, e$	(*Ricorsione*)
Tipi:	τ	$::= \mathbf{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$	

Le nozioni di variabili libere e legate è definito come nel caso del λ-calcolo semplicemente tipato di Church presentato nel Capitolo 2. Al solito, identifichiamo termini α-equivalenti, ossia espressioni che differiscono solamente per il nome delle loro variabili legate. Usiamo $[e'/x]e$ quale nostra sintassi informale per la sostituzione di un'espressione e' in luogo delle occorrenze libere della variabile x in e.

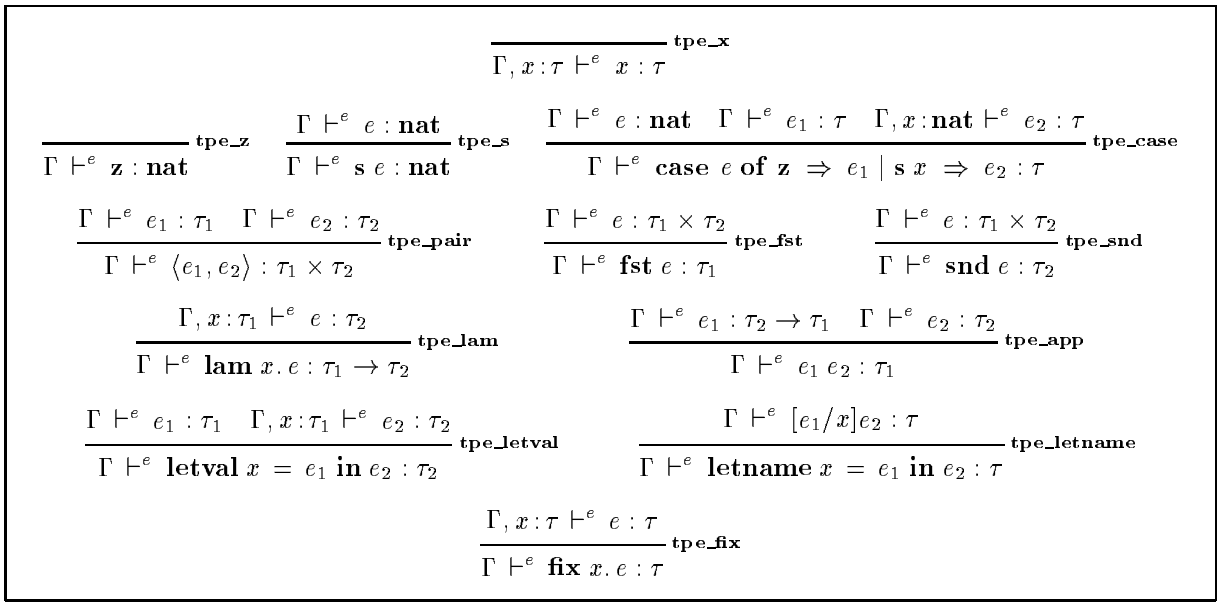


Figura 4.4: Regole di Tipizzazione per *Mini-ML*

4.3.2 Tipizzazione

Le regole di tipizzazione per espressioni *Mini-ML*, specificate in Figura 4.4, definiscono il sotto-linguaggio delle espressioni ben tipate. Esse permettono di discriminare costruzioni sintattiche valide da espressioni prive di senso (**fst z** ad esempio). Questo sistema deduttivo specifica la *semantica statica* di *Mini-ML*.

Per dare un tipo ad un'espressione, dobbiamo assegnare un tipo a tutte le variabili libere che vi compaiono. Ci rifacciamo pertanto ad un contesto definito come segue:

Contesti: $\Gamma ::= \cdot \mid \Gamma, x:\tau$

La semantica statica di *Mini-ML* presenta due caratteristiche interconnesse: innanzitutto esistono espressioni, specie quelle funzionali, che hanno piú di un tipo. Inoltre, l'operatore **levname** è debolmente polimorfo in quanto ad ogni occorrenza di x nella regola **tpe_letname** può essere assegnato un tipo diverso.

4.3.3 Valutazione

La *semantica dinamica* di *Mini-ML* è data dal giudizio di *valutazione* che specifica come calcolare il valore di un'espressione ben formata. In Figura 4.5, presentiamo un sistema deduttivo per la valutazione di espressioni *Mini-ML*. Questo sistema si appoggia su una strategia diversa da quanto solitamente specificato [Pfe92], adottando invece un metodo che si basa su continuazioni

Expressions

(No **ev_x**)

$$\frac{K \vdash \mathbf{return}\,\mathbf{z} \hookrightarrow v}{K \vdash \mathbf{eval}\,\mathbf{z} \hookrightarrow v}\,\mathbf{ev\_z} \qquad \frac{K, \lambda x.\,\mathbf{return}\,\mathbf{s}\;x \vdash \mathbf{eval}\;e \hookrightarrow v}{K \vdash \mathbf{eval}\,\mathbf{s}\;e \hookrightarrow v}\,\mathbf{ev\_s}$$

$$\frac{K, \lambda y.\,\mathbf{case}^*\;y\;\mathbf{of}\;\mathbf{z}\;\Rightarrow\;e_1 \mid \mathbf{s}\;x\;\Rightarrow\;e_2 \vdash \mathbf{eval}\;e \hookrightarrow v}{K \vdash \mathbf{eval}\,\mathbf{case}\;e\;\mathbf{of}\;\mathbf{z}\;\Rightarrow\;e_1 \mid \mathbf{s}\;x\;\Rightarrow\;e_2 \hookrightarrow v}\,\mathbf{ev\_case}$$

$$\frac{K, \lambda x.\,\langle x, e_2\rangle^* \vdash \mathbf{eval}\;e_1 \hookrightarrow v}{K \vdash \mathbf{eval}\,\langle e_1, e_2\rangle \hookrightarrow v}\,\mathbf{ev\_pair}$$

$$\frac{K, \lambda x.\,\mathbf{fst}^*\;x \vdash \mathbf{eval}\;e \hookrightarrow v}{K \vdash \mathbf{eval}\,\mathbf{fst}\;e \hookrightarrow v}\,\mathbf{ev\_fst} \qquad \frac{K, \lambda x.\,\mathbf{snd}^*\;x \vdash \mathbf{eval}\;e \hookrightarrow v}{K \vdash \mathbf{eval}\,\mathbf{snd}\;e \hookrightarrow v}\,\mathbf{ev\_snd}$$

$$\frac{K \vdash \mathbf{return}\,\mathbf{lam}\;x.\,e \hookrightarrow v}{K \vdash \mathbf{eval}\,\mathbf{lam}\;x.\,e \hookrightarrow v}\,\mathbf{ev\_lam} \qquad \frac{K, \lambda x.\,\mathbf{app}^*\;x\;e_2 \vdash \mathbf{eval}\;e_1 \hookrightarrow v}{K \vdash \mathbf{eval}\;e_1\;e_2 \hookrightarrow v}\,\mathbf{ev\_app}$$

$$\frac{K, \lambda x.\,\mathbf{eval}\;e_2 \vdash \mathbf{eval}\;e_1 \hookrightarrow v}{K \vdash \mathbf{eval}\,\mathbf{letval}\;x\;=\;e_1\;\mathbf{in}\;e_2 \hookrightarrow v}\,\mathbf{ev\_letval} \qquad \frac{K \vdash \mathbf{eval}\;[e_1/x]e_2 \hookrightarrow v}{K \vdash \mathbf{eval}\,\mathbf{letname}\;x\;=\;e_1\;\mathbf{in}\;e_2 \hookrightarrow v}\,\mathbf{ev\_letname}$$

$$\frac{K \vdash \mathbf{eval}\;[\mathbf{fix}\;x.\,e/x]e \hookrightarrow v}{K \vdash \mathbf{eval}\,\mathbf{fix}\;x.\,e \hookrightarrow v}\,\mathbf{ev\_fix}$$

Values

$$\frac{}{\mathbf{init} \vdash \mathbf{return}\;v \hookrightarrow v}\,\mathbf{ev\_init} \qquad \frac{K \vdash [v'/x]i \hookrightarrow v}{K, \lambda x.\,i \vdash \mathbf{return}\;v' \hookrightarrow v}\,\mathbf{ev\_cont}$$

Auxiliary instructions

$$\frac{K \vdash \mathbf{eval}\;e_1 \hookrightarrow v}{K \vdash \mathbf{case}^*\;\mathbf{z}\;\mathbf{of}\;\mathbf{z}\;\Rightarrow\;e_1 \mid \mathbf{s}\;x\;\Rightarrow\;e_2 \hookrightarrow v}\,\mathbf{ev\_case}_1^*$$

$$\frac{K \vdash \mathbf{eval}\;[v'/x]e_2 \hookrightarrow v}{K \vdash \mathbf{case}^*\;\mathbf{s}\;v'\;\mathbf{of}\;\mathbf{z}\;\Rightarrow\;e_1 \mid \mathbf{s}\;x\;\Rightarrow\;e_2 \hookrightarrow v}\,\mathbf{ev\_case}_2^*$$

$$\frac{K, \lambda x.\,\mathbf{return}\,\langle v', x\rangle \vdash \mathbf{eval}\;e \hookrightarrow v}{K \vdash \langle v', e\rangle^* \hookrightarrow v}\,\mathbf{ev\_pair}^*$$

$$\frac{K \vdash \mathbf{return}\;v_1 \hookrightarrow v}{K \vdash \mathbf{fst}^*\;\langle v_1, v_2\rangle \hookrightarrow v}\,\mathbf{ev\_fst}^* \qquad \frac{K \vdash \mathbf{return}\;v_2 \hookrightarrow v}{K \vdash \mathbf{snd}^*\;\langle v_1, v_2\rangle \hookrightarrow v}\,\mathbf{ev\_snd}^*$$

$$\frac{K, \lambda x.\,\mathbf{eval}\;e_1 \vdash \mathbf{eval}\;e_2 \hookrightarrow v}{K \vdash \mathbf{app}^*\;(\mathbf{lam}\;x.\,e_1)\;e_2 \hookrightarrow v}\,\mathbf{ev\_app}^*$$

Figura 4.5: Valutazione Basata su Continuazioni per *Mini-ML*

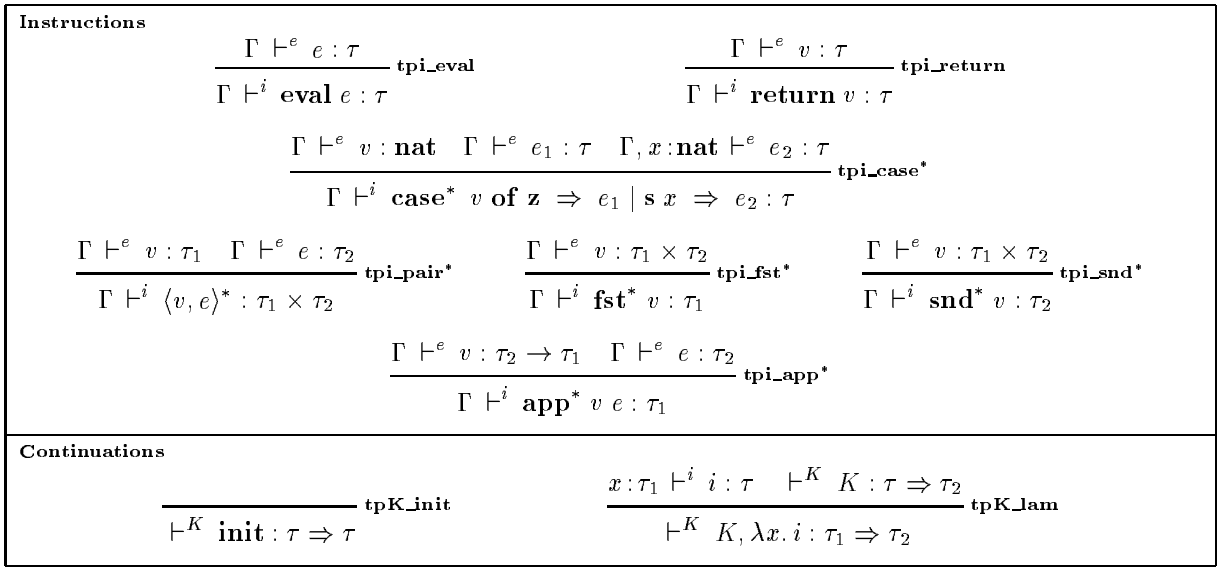


Figura 4.6: Regole di Tipizzazione per *Mini-ML*, Istruzioni e Continuazioni

[DHM91, HP92, Pfe92], particolarmente adeguato alla realizzazione di compilatori ed interpreti.

A questo scopo, dobbiamo aumentare il linguaggio di base di *Mini-ML* con due categorie sintattiche ausiliari: *continuazioni* e *istruzioni*. L'idea è di calcolare il valore di una sottoespressione di un'espressione data per volta, memorizzando le altre in una continuazione a cui si accede non appena si è terminato con la sottoespression corrente. Stadi intermedi di valutazione vengono gestiti mediante istruzioni ausiliari. Abbiamo la seguente grammatica:

$$
\begin{array}{lrcl}
\textit{Istruzioni:} & i & ::= & \mathbf{eval}\ e \mid \mathbf{return}\ v \\
& & \mid & \mathbf{case}^*\ v\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \\
& & \mid & \langle v, e \rangle^* \mid \mathbf{fst}^*\ v \mid \mathbf{snd}^*\ v \\
& & \mid & \mathbf{app}^*\ v\ e \\
\textit{Continuazioni:} & K & ::= & \mathbf{init} \mid K, \lambda x.\, i
\end{array}
$$

Le regole di tipizzazione per queste entità intermedie sono mostrate in Figura 4.6.

4.3.4 Preservazione dei Tipi

Una fondamentale proprietà di *Mini-ML* è che il valore v di un'espressione e ha lo stesso tipo di e. Questo risultato viene chiamato *preservazione dei tipi*. Una dimostrazione di questo fatto è banale e si ottiene per induzione. Essa si basa però sulla validità di due lemmi: weakening e sostituzione.

Lemma 4.3.1 (*Weakening*)

i. Se $\Gamma \vdash^e e : \tau$, *allora* $\Gamma, \Gamma' \vdash^e e : \tau$;

ii. Se $\Gamma \vdash^i i : \tau$, *allora* $\Gamma, \Gamma' \vdash^i i : \tau$. □

Lemma 4.3.2 (*Sostituzione*)

i. Se $\Gamma, x{:}\tau' \vdash^e e : \tau$ *e* $\Gamma \vdash^e e' : \tau'$, *allora* $\Gamma \vdash^e [e'/x]e : \tau$;

ii. Se $\Gamma, x{:}\tau' \vdash^i i : \tau$ *e* $\Gamma \vdash^e e' : \tau'$, *allora* $\Gamma \vdash^i [e'/x]i : \tau$. □

La proprietà di preservazione dei tipi è espressa dal seguente teorema, dove servono assunzioni ausiliari per assicurarsi che la continuazione sia ben tipata. Diamo una dimostrazione dettagliata di questa proprietà nella prossima sezione.

Teorema 4.3.3 (*Preservazione dei tipi*)

Se $K \vdash i \hookrightarrow v$ *con* $\cdot \vdash^i i : \tau$, $e \vdash^K K : \tau \Rightarrow \bar{\tau}$, *allora* $\cdot \vdash^e v : \bar{\tau}$.

Nel caso la continuazione considerata sia iniziale, otteniamo il seguente risultato, a cui siamo interessati.

Corollario 4.3.4 (*Type preservation*)

Se **init** $\vdash$ **eval** $e \hookrightarrow v$ *con* $\cdot \vdash^e e : \tau$, *allora* $\cdot \vdash^e v : \tau$.

4.4 Rappresentazione di Aspetti di *Mini-ML* in *LF*

In questa sezione, formalizziamo in *LF* la discussione informale di *Mini-ML* data nella sezione precedente. L'essenza del materiale mostrato è ispirato da [Pfe92] a cui rinviamo il lettore per approfondimenti. Per semplicità, ci rifacciamo alla sintassi concreta di *Elf* presentata nel capitolo 3. Il codice completo e i teoremi di adeguatezza sono stati raccolti nell'Appendice A.

4.4.1 Sintassi

Le quattro categorie sintattiche di *Mini-ML*, espressioni, tipi, istruzioni e continuazioni, vengono rappresentate mediante le seguenti quattro costanti di tipi:

```
exp : type.        instr : type.
tp  : type.        cont  : type.
```

A scopo illustrativo, ci concentriamo sulle espressioni per le quali sviluppiamo in dettaglio le dimostrazioni dei teoremi di adeguatezza.

Codifichiamo le produzioni della grammatica definente le espressioni di *Mini-ML*, presentata nella sezione precedente, per mezzo della funzione di rappresentazione $\ulcorner\_\urcorner$ definita come segue:

Espressioni : `exp`

$$
\begin{array}{rcl}
\ulcorner x \urcorner & = & x \\
\ulcorner \mathbf{z} \urcorner & = & \texttt{z} \\
\ulcorner \mathbf{s}\ e \urcorner & = & \texttt{s}\ \ulcorner e \urcorner \\
\ulcorner \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \urcorner & = & \texttt{case}\ \ulcorner e \urcorner\ \ulcorner e_1 \urcorner\ \texttt{[}x\texttt{:exp]}\ulcorner e_2 \urcorner \\
\ulcorner \langle e_1, e_2 \rangle \urcorner & = & \texttt{pair}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner \\
\ulcorner \mathbf{fst}\ e \urcorner & = & \texttt{fst}\ \ulcorner e \urcorner \\
\ulcorner \mathbf{snd}\ e \urcorner & = & \texttt{snd}\ \ulcorner e \urcorner \\
\ulcorner \mathbf{lam}\ x.e \urcorner & = & \texttt{lam}\ \texttt{[}x\texttt{:exp]}\ulcorner e \urcorner \\
\ulcorner e_1\ e_2 \urcorner & = & \texttt{app}\ \ulcorner e_1 \urcorner\ \ulcorner e_2 \urcorner \\
\ulcorner \mathbf{letval}\ x = e_1\ \mathbf{in}\ e_2 \urcorner & = & \texttt{letval}\ \ulcorner e_1 \urcorner\ \texttt{[}x\texttt{:exp]}\ulcorner e_2 \urcorner \\
\ulcorner \mathbf{letname}\ x = e_1\ \mathbf{in}\ e_2 \urcorner & = & \texttt{letname}\ \ulcorner e_1 \urcorner\ \texttt{[}x\texttt{:exp]}\ulcorner e_2 \urcorner \\
\ulcorner \mathbf{fix}\ x.e \urcorner & = & \texttt{fix}\ \texttt{[}x\texttt{:exp]}\ulcorner e \urcorner
\end{array}
$$

Dove si hanno le seguenti dichiarazioni per ognuna delle costanti *Elf* utilizzate:

$$
\Sigma_{\texttt{exp}} = (\texttt{exp : type.}) \ + \left\{
\begin{array}{ll}
\texttt{z} & \texttt{: exp.} \\
\texttt{s} & \texttt{: exp -> exp.} \\
\texttt{case} & \texttt{: exp -> exp -> (exp -> exp) -> exp.} \\
\texttt{pair} & \texttt{: exp -> exp -> exp.} \\
\texttt{fst} & \texttt{: exp -> exp.} \\
\texttt{snd} & \texttt{: exp -> exp.} \\
\texttt{lam} & \texttt{: (exp -> exp) -> exp.} \\
\texttt{app} & \texttt{: exp -> exp -> exp.} \\
\texttt{letval} & \texttt{: exp -> (exp -> exp) -> exp.} \\
\texttt{letname} & \texttt{: exp -> (exp -> exp) -> exp.} \\
\texttt{fix} & \texttt{: (exp -> exp) -> exp.}
\end{array}
\right.
$$

Si osservi l'uso della sintassi astratta di ordine superiore per codificare le variabili di *Mini-ML* e i costrutti che le legano: le costanti *Elf* che le rappresentano hanno un tipo funzionale per fare posto alla λ-astrazione usata come binder universale.

Passiamo ora alla dimostrazione dell'adeguatezza di questa rappresentazione. Vogliamo ottenere le seguenti proprietà:

- La rappresentazione di ogni espressione *Mini-ML* ben formata è un oggetto canonico di *LF* di tipo `exp`;
- Ogni oggetto canonico di tipo `exp` è la rappresentazione di una qualche espressione in *Mini-ML*;
- La funzione $\ulcorner\_\urcorner$ è una biiezione;
- Questa funzione è composizionale.

Formalizziamo queste proprietà nella seguente serie di lemmi:

Lemma 4.4.1 (*Correttezza della rappresentazione delle espressioni di* Mini-ML)

Per ogni espressione Mini-ML e con variabili libere $x_1, \ldots, x_n$, il giudizio LF

$$\underbrace{x_1 : \mathtt{exp}, \ldots, x_n : \mathtt{exp}}_{\Gamma} \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} \ulcorner e \urcorner \Uparrow \mathtt{exp}$$

è derivabile.

Dimostrazione.

Si procede per induzione sulla struttura di e. Facciamo vedere i casi piú significativi. Le spiegazioni sono in lingua Inglese.

$\boxed{e = x_i}$

$\mathcal{D}' :: x_1 : \mathtt{exp}, \ldots, x_i : \mathtt{exp}, \ldots, x_n : \mathtt{exp} \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} x_i \downarrow \mathtt{exp}$	by rule **oLFa_var**,
$\mathcal{D} :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} x_i \Uparrow \mathtt{exp}$	by rule **oLFc_a** on $\mathcal{D}'$,
$\mathcal{D} :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} \ulcorner x_i \urcorner \Uparrow \mathtt{exp}$	by definition of $\ulcorner \_ \urcorner$.

$\boxed{e = \mathbf{z}}$

$\mathcal{D}' :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} \mathtt{z} \downarrow \mathtt{exp}$	by rule **oLFa_con**,
$\mathcal{D} :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} \ulcorner \mathtt{z} \urcorner \Uparrow \mathtt{exp}$	by rule **oLFc_a** on $\mathcal{D}'$ and definition of $\ulcorner \_ \urcorner$.

$\boxed{e = \langle e_1, e_2 \rangle}$

$\mathcal{D}'_1 :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} \ulcorner e_1 \urcorner \Uparrow \mathtt{exp}$	by induction hypothesis on e_1,
$\mathcal{D}'_2 :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} \ulcorner e_2 \urcorner \Uparrow \mathtt{exp}$	by induction hypothesis on e_2,
$\mathcal{D}' :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} \mathtt{pair} \downarrow \mathtt{exp} \texttt{ -> } \mathtt{exp} \texttt{ -> } \mathtt{exp}$	by rule **oLFa_con**,
$\mathcal{D}_1 :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} \mathtt{pair} \ulcorner e_1 \urcorner \downarrow \mathtt{exp} \texttt{ -> } \mathtt{exp}$	by rule **oLFa_app** on $\mathcal{D}'$ and $\mathcal{D}'_1$,
$\mathcal{D}_2 :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} \mathtt{pair} \ulcorner e_1 \urcorner \ulcorner e_2 \urcorner \downarrow \mathtt{exp}$	by rule **oLFa_app** on $\mathcal{D}_1$ and $\mathcal{D}'_2$,
$\mathcal{D} :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} \ulcorner \langle e_1, e_2 \rangle \urcorner \Uparrow \mathtt{exp}$	by rule **oLFc_a** on $\mathcal{D}_2$ and definition of $\ulcorner \_ \urcorner$.

$\boxed{e = (\mathbf{letname}\ x\ =\ e_1\ \mathbf{in}\ e_2)}$

$\mathcal{D}'_1 :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} \ulcorner e_1 \urcorner \Uparrow \mathtt{exp}$	by induction hypothesis on e_1,
$\mathcal{D}'_2 :: \Gamma, x : \mathtt{exp} \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} \ulcorner e_2 \urcorner \Uparrow \mathtt{exp}$	by induction hypothesis on e_2 where x is free,
$\mathcal{D}''_2 :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} \lambda x : \mathtt{exp}. \ulcorner e_2 \urcorner \Uparrow \mathtt{exp} \texttt{ -> } \mathtt{exp}$	by rule **oLFc_lam** on $\mathcal{D}_{\mathtt{exp}}$ and $\mathcal{D}'_2$,

$\mathcal{D}'$ :: $\Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}}$ `letname` $\downarrow$ `exp -> (exp -> exp) -> exp`		by rule **oLFa_con**,
$\mathcal{D}_2$ :: $\Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}}$ `letname` $\ulcorner e_1 \urcorner$ $(\lambda x{:}\mathtt{exp}.\ulcorner e_2 \urcorner) \downarrow$ `exp`		by two applications of rule **oLFa_app** on $\mathcal{D}'$ and $\mathcal{D}'_1$, and then $\mathcal{D}''_2$,
$\mathcal{D}$ :: $\Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}}$	$\ulcorner \mathbf{letname}\ x\ =\ e_1\ \mathbf{in}\ e_2 \urcorner \Uparrow \mathtt{exp}$	by rule **oLFc_a** on $\mathcal{D}_2$ and definition of $\ulcorner \_ \urcorner$. ☑

Allo scopo di dimostrare che ogni oggetto *LF* derivabile di tipo `exp` è la rappresentazione di una qualche espressione *Mini-ML*, dobbiamo definire la funzione $\llcorner \_ \lrcorner$, inversa di $\ulcorner \_ \urcorner$.

$$
\begin{array}{rcl}
\llcorner x \lrcorner & = & x \\
\llcorner \mathtt{z} \lrcorner & = & \mathbf{z} \\
\llcorner \mathtt{s}\ M \lrcorner & = & \mathbf{s}\ \llcorner M \lrcorner \\
\llcorner \mathtt{case}\ M\ M_1\ [x{:}\mathtt{exp}]M_2 \lrcorner & = & \mathbf{case}\ \llcorner M \lrcorner\ \mathbf{of}\ \mathbf{z} \Rightarrow \llcorner M_1 \lrcorner \mid \mathbf{s}\ x \Rightarrow \llcorner M_2 \lrcorner \\
\llcorner \mathtt{pair}\ M_1\ M_2 \lrcorner & = & \langle \llcorner M_1 \lrcorner, \llcorner M_2 \lrcorner \rangle \\
\llcorner \mathtt{fst}\ M \lrcorner & = & \mathbf{fst}\ \llcorner M \lrcorner \\
\llcorner \mathtt{snd}\ M \lrcorner & = & \mathbf{snd}\ \llcorner M \lrcorner \\
\llcorner \mathtt{lam}\ [x{:}\mathtt{exp}]M \lrcorner & = & \mathbf{lam}\ x.\llcorner M \lrcorner \\
\llcorner \mathtt{app}\ M_1\ M_2 \lrcorner & = & \llcorner M_1 \lrcorner\ \llcorner M_2 \lrcorner \\
\llcorner \mathtt{letval}\ M_1\ [x{:}\mathtt{exp}]M_2 \lrcorner & = & \mathbf{letval}\ x\ =\ \llcorner M_1 \lrcorner\ \mathbf{in}\ \llcorner M_2 \lrcorner \\
\llcorner \mathtt{letname}\ M_1\ [x{:}\mathtt{exp}]M_2 \lrcorner & = & \mathbf{letname}\ x\ =\ \llcorner M_1 \lrcorner\ \mathbf{in}\ \llcorner M_2 \lrcorner \\
\llcorner \mathtt{fix}\ [x{:}\mathtt{exp}]M \lrcorner & = & \mathbf{fix}\ x.\llcorner M \lrcorner
\end{array}
$$

Si ha allora il seguente lemma, dimostrato per induzione sulla derivazione del giudizio *LF* dato.

Lemma 4.4.2 (*Completezza della rappresentazione delle espressioni Mini-ML*)

Per ogni contesto LF $\Gamma = x_1{:}\mathtt{exp}, \ldots, x_n{:}\mathtt{exp}$ *e oggetto* M*, se il giudizio*

$$\Gamma \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} M \Uparrow \mathtt{exp}$$

è derivabile, allora $\llcorner M \lrcorner$ *è definita ed è un'espressione Mini-ML valida.* □

I seguenti due lemmi si dimostrano per induzione.

Lemma 4.4.3 (*Biiettività della rappresentazione delle espressioni Mini-ML*)

La funzione $\ulcorner \_ \urcorner$ *è una biiezione con inversa* $\llcorner \_ \lrcorner$ *tra espressioni Mini-ML e oggetti LF* M *per i quali* $\llcorner M \lrcorner$ *è definita. Piú precisamente:*

i. Se $\ulcorner e_1 \urcorner = \ulcorner e_2 \urcorner$*, allora* $e_1 = e_2$*;*

ii. Se $\llcorner M \lrcorner$ *è definita, esiste un'espressione Mini-ML e tale che* $\ulcorner e \urcorner = M$*;*

iii. Per ogni espressione Mini-ML e, $\llcorner\ulcorner e\urcorner\lrcorner = e$;

iv. Per ogni oggetto LF M *tale che* $\llcorner M\lrcorner$ *è definito,* $\ulcorner\llcorner M\lrcorner\urcorner = M$. □

Lemma 4.4.4 (*Composizionalità della rappresentazione delle espressioni Mini-ML*)

Per ogni espressione Mini-ML e *ed* e', $\ulcorner[e'/x]e\urcorner = [\ulcorner e'\urcorner/x]\ulcorner e\urcorner$. □

Questi quattro lemmi vengono solitamente espressi in maniera piú sintetica nel seguente teorema, detto appunto adeguatezza della meta-rappresentazione:

Teorema 4.4.5 (*Adeguatezza della rappresentazione delle espressioni Mini-ML*)

La funzione $\ulcorner\_\urcorner$ *è una biiezione composizionale tra espressioni Mini-ML con variabili libere* $x_1, \ldots, x_n$ *e oggetti LF canonici* M *tali che il giudizio*

$$x_1 : \mathtt{exp}, \ldots, x_n : \mathtt{exp} \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{exp}}} M \Uparrow \mathtt{exp}$$

è derivabile. □

Il trattamento dei tipi avviene in maniera del tutto analoga. Poiché i tipi di *Mini-ML* non contengono variabili, non serve parlare di composizionalità . Facciamo vedere la codifica ed enunciamo il teorema di adeguatezza.

Tipi : tp

$$\begin{array}{rcl} \ulcorner \mathbf{nat} \urcorner & = & \mathtt{nat} \\ \ulcorner \tau_1 \times \tau_2 \urcorner & = & \ulcorner \tau_1 \urcorner \ \mathtt{cross} \ \ulcorner \tau_2 \urcorner \\ \ulcorner \tau_1 \to \tau_2 \urcorner & = & \ulcorner \tau_1 \urcorner \ \mathtt{arrow} \ \ulcorner \tau_2 \urcorner \end{array}$$

$$\Sigma_{\mathtt{tp}} = (\mathtt{tp} : \mathtt{type}.) \ + \ \left\{ \begin{array}{l} \mathtt{nat} \ \ : \mathtt{tp}. \\ \mathtt{cross} : \mathtt{tp} \ \mathtt{->} \ \mathtt{tp} \ \mathtt{->} \ \mathtt{tp}. \\ \mathtt{arrow} : \mathtt{tp} \ \mathtt{->} \ \mathtt{tp} \ \mathtt{->} \ \mathtt{tp}. \end{array} \right.$$

Teorema 4.4.6 (*Adeguatezza della rappresentazione dei tipi di Mini-ML*)

La funzione $\ulcorner\_\urcorner$ *è una biiezione tra tipi Mini-ML ed oggetti LF canonici* M *tali che*

$$\cdot \vdash^{\mathrm{LF}}_{\Sigma_{\mathtt{tp}}} M \Uparrow \mathtt{tp}$$

è derivabile. □

Il trattamento di istruzioni e continuazioni segue lo stesso schema. Facciamo vedere solamente le rispettive codifiche.

Istruzioni : `instr`

$$\begin{aligned}
\ulcorner \mathbf{eval}\ e \urcorner &= \texttt{eval}\ \ulcorner e \urcorner \\
\ulcorner \mathbf{return}\ v \urcorner &= \texttt{return}\ \ulcorner v \urcorner \\
\ulcorner \mathbf{case}^*\ v\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2 \urcorner &= \texttt{case*}\ \ulcorner v \urcorner\ \ulcorner e_1 \urcorner\ [x\texttt{:exp}] \ulcorner e_2 \urcorner \\
\ulcorner \langle v, e \rangle^* \urcorner &= \texttt{pair*}\ \ulcorner v \urcorner\ \ulcorner e \urcorner \\
\ulcorner \mathbf{fst}^*\ v \urcorner &= \texttt{fst*}\ \ulcorner v \urcorner \\
\ulcorner \mathbf{snd}^*\ v \urcorner &= \texttt{snd*}\ \ulcorner v \urcorner \\
\ulcorner \mathbf{app}^*\ v\ e \urcorner &= \texttt{app*}\ \ulcorner v \urcorner\ \ulcorner e \urcorner
\end{aligned}$$

Continuazioni : `cont`

$$\begin{aligned}
\ulcorner \mathbf{init} \urcorner &= \texttt{init} \\
\ulcorner K, \lambda x.\, i \urcorner &= \ulcorner K \urcorner\ \texttt{-}\ [x\texttt{:exp}] \ulcorner i \urcorner
\end{aligned}$$

Nei precedenti teoremi di adeguatezza, abbiamo vincolato le segnature a menzionare solamente le costanti necessarie per dare una rappresentazione alle entità in gioco. Queste segnature si possono tuttavia aumentare con ulteriori dichiarazioni purché non interferiscano con queste costanti, mantenendone la validità. Considereremo d'ora in poi la segnatura Σ necessaria per la codifica dell'intero esempio.

4.4.2 Semantica: Tipizzazione e Valutazione

La rappresentazione dei vari giudizi che definisco la tipizzazione delle varie entità in gioco segue lo schema mostrato sopra. Analizziamo in qualche dettaglio il caso delle espressioni.

Il giudizio associa un'espressione *Mini-ML* al suo tipo è rappresentato dalla seguente costante di tipo in *Elf*:

```
tpe :  exp -> tp -> type.
```

Facciamo vedere a titolo esemplificativo la codifica in *Elf* della regola **tpe_letname**:

```
tpe_letname : tpe (letname E1 E2) T
               <- tpe (E2 E1) T.
```

Abbiamo il seguente teorema di adeguatezza:

Teorema 4.4.7 (*Adeguatezza della rappresentazione della tipizzazione in Mini-ML*)

Dati un'espressione Mini-ML e ed un tipo τ, esiste una biiezione composizionale $\ulcorner \_ \urcorner$ Tra derivazioni di

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash^e e : \tau$$

e oggetti LF canonici M tale che

$$\begin{bmatrix} x_1\texttt{:exp}, & \ldots, & x_n\texttt{:exp}, \\ t_1\texttt{:tpe}\ x_1\ \ulcorner \tau_1 \urcorner, & \ldots, & t_n\texttt{:tpe}\ x_n\ \ulcorner \tau_n \urcorner \end{bmatrix} \vdash^{\mathrm{LF}}_{\Sigma} M \Uparrow \texttt{tpe}\ \ulcorner e \urcorner\ \ulcorner \tau \urcorner$$

è derivabile. □

La stessa tecnica si applica alla rappresentazione degli altri due giudizi di tipizzazione di *Mini-ML* presentati nella sezione precedente.

Si procede in maniera simile nel caso della valutazione. Facciamo uso questa volta della seguente dichiarazione di tipi:

```
ev :  cont -> instr -> exp -> type.
```

Si ha il seguente teorema di adeguatezza:

Teorema 4.4.8 (*Adeguatezza della rappresentazione delle valutazioni di Mini-ML*)

Data una continuazione Mini-ML chiusa K, un'istruzione chiusa i, e un'espressione chiusa v, esiste una biiezione composizionale $\ulcorner\_\urcorner$ tra derivazioni di

$$K \vdash i \hookrightarrow v$$

e oggetti LF M tale che

$$\cdot \vdash^{\mathrm{LF}}_{\Sigma} M \Uparrow \mathtt{ev}\ \ulcorner K \urcorner\ \ulcorner i \urcorner\ \ulcorner v \urcorner$$

è derivabile. □

4.4.3 Meta-teoria: Il Teorema di Preservazione dei Tipi

Rappresentiamo il teorema di preservazione dei tipi per *Mini-ML* per mezzo della dichiarazione di tipi:

```
pr :  ev K I A -> tpi I T -> tpK K T T´ -> tpe A T´ -> type.
```

per la quale abbiamo il seguente teorema di adeguatezza:

Teorema 4.4.9 (*Adeguatezza della rappresentazione della preservazione dei tipi per Mini-ML*)

Dati una continuazione Mini-ML chiusa K, un'istruzione chiusa i, un'espressione chiusa v, tipi τ e τ', e oggetti LF E, T_i e T_K tali che i giudizi

$$\begin{array}{l} \cdot \vdash^{\mathrm{LF}}_{\Sigma} E \Uparrow \mathtt{ev}\ \ulcorner K \urcorner\ \ulcorner i \urcorner\ \ulcorner v \urcorner \\ \cdot \vdash^{\mathrm{LF}}_{\Sigma} T_i \Uparrow \mathtt{tpi}\ \ulcorner i \urcorner\ \ulcorner \tau \urcorner \\ \cdot \vdash^{\mathrm{LF}}_{\Sigma} T_K \Uparrow \mathtt{tpK}\ \ulcorner K \urcorner\ \ulcorner \tau \urcorner\ \ulcorner \tau' \urcorner \end{array}$$

siano derivabili, esistono oggetti LF T ed M tali che

$$\cdot \vdash^{\mathrm{LF}}_{\Sigma} T \Uparrow \mathtt{tpe}\ \ulcorner v \urcorner\ \ulcorner \tau' \urcorner$$

e

$$\cdot \vdash^{\mathrm{LF}}_{\Sigma} M \Uparrow \mathtt{pr}\ \ulcorner K \urcorner\ \ulcorner i \urcorner\ \ulcorner \tau \urcorner\ \ulcorner \tau' \urcorner\ \ulcorner v \urcorner\ E\ T_i\ T_K\ T$$

sono derivabili. □

L'oggetto M di cui si parla nell'enunciato di questo teorema viene costruito sulla base della rappresentazione dei vari casi della dimostrazione del teorema di preservazione dei tipi come regole di deduzione. Riproponiamo questo risultato assieme ad una parte della sua dimostrazione. Dopo ogni caso trattato facciamo vedere la dichiarazione *Elf* che lo emula.

Teorema 4.3.3 (*Preservazione dei tipi*)

Se $\mathcal{E} :: K \vdash i \hookrightarrow v$ *con* $\mathcal{T}_i :: \cdot \vdash^i i : \tau$, *e* $\mathcal{T}_K :: \vdash^K K : \tau \Rightarrow \bar{\tau}$, *allora* $\mathcal{T} :: \cdot \vdash^e v : \bar{\tau}$.

Dimostrazione.

Procediamo per induzione sulla struttura di $\mathcal{E}$ e inversione su $\mathcal{T}_i$ e $\mathcal{T}_K$. Facciamo vedere solamente un numero limitato di casi, i rimanenti sono altrettanto semplici. Il codice *Elf* corrispondente a questa dimostrazione si può trovare in Appendice A.

ev_z

$$\mathcal{E} = \frac{\mathcal{E}'}{K \vdash \mathbf{return\ z} \hookrightarrow v}\,\mathbf{ev\_z}$$

$$K \vdash \mathbf{eval\ z} \hookrightarrow v$$

with $i = \mathbf{eval\ z}$.

$\mathcal{T}_e :: \cdot \vdash^e \mathbf{z} : \mathbf{nat}$ and $\tau = \mathbf{nat}$	by inversion on **tpi_eval** and **tpe_z** for $\mathcal{T}_i$,
$\mathcal{T}_i' :: \cdot \vdash^i \mathbf{return\ z} : \mathbf{nat}$	by rule **tpi_return** on $\mathcal{T}_e$,
$\mathcal{T} :: \cdot \vdash^e v : \bar{\tau}$	by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$ and $\mathcal{T}_K$.

```
pr-ev_z        : pr (ev_z E') (tpi_eval tpe_z) TK Ta
                   <- pr E' (tpi_return tpe_z) TK Ta.
```

ev_s

$$\mathcal{E} = \frac{\mathcal{E}'}{K, \lambda x.\,\mathbf{return\ s}\ x \vdash \mathbf{eval}\ e \hookrightarrow v}\,\mathbf{ev\_s}$$

$$K \vdash \mathbf{eval\ s}\ e \hookrightarrow v$$

with $i = \mathbf{eval\ s}\ e$.

$\mathcal{T}_e :: \cdot \vdash^e e : \mathbf{nat}$ and $\tau = \mathbf{nat}$	by inversion on **tpi_eval** and **tpe_s** for $\mathcal{T}_i$,
$\mathcal{T}_i' :: \cdot \vdash^i \mathbf{eval}\ e : \mathbf{nat}$	by rule **tpi_eval** on $\mathcal{T}_e$,
$\mathcal{T}_v :: x : \mathbf{nat} \vdash^e \mathbf{s}\ x : \mathbf{nat}$	by rule **tpe_x** and **tpe_s**,
$\mathcal{T}_i'' :: x : \mathbf{nat} \vdash^i \mathbf{return\ s}\ x : \mathbf{nat}$	by rule **tpi_return** on $\mathcal{T}_v$,
$\mathcal{T}_K' :: \vdash^K K, \lambda x.\,\mathbf{return\ s}\ x : \mathbf{nat} \Rightarrow \bar{\tau}$	by rule **tpK_lam** on $\mathcal{T}_i''$ and $\mathcal{T}_K$,
$\mathcal{T} :: \cdot \vdash^e v : \bar{\tau}$	by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$ and $\mathcal{T}_K'$.

```
pr-ev_s        : pr (ev_s E') (tpi_eval (tpe_s Te)) TK Ta
                   <- pr E'
                         (tpi_eval Te)
                         (tpK_lam TK [x][t:tpe x nat] tpi_return (tpe_s t))
                         Ta.
```

ev_case

$$\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}' \\ K, \lambda y.\,\mathbf{case}^*\ y\ \mathbf{of}\ \mathbf{z}\ \Rightarrow\ e_1 \mid \mathbf{s}\ x\ \Rightarrow\ e_2 \vdash \mathbf{eval}\ e \hookrightarrow v\end{array}}{K \vdash \mathbf{eval\ case}\ e\ \mathbf{of}\ \mathbf{z}\ \Rightarrow\ e_1 \mid \mathbf{s}\ x\ \Rightarrow\ e_2 \hookrightarrow v}\ \mathbf{ev\_case}$$

with $i = \mathbf{eval\ case}\ e\ \mathbf{of}\ \mathbf{z}\ \Rightarrow\ e_1 \mid \mathbf{s}\ x\ \Rightarrow\ e_2$.

$\mathcal{T}_e :: \cdot \vdash^e e : \mathbf{nat},\ \mathcal{T}'_e :: \cdot \vdash^e e_1 : \tau$	and
$\mathcal{T}''_e :: x : \mathbf{nat} \vdash^e e_2 : \tau$	by inversion on **tpi_eval** and **tpe_case** for $\mathcal{T}_i$,
$\mathcal{T}'_i :: \cdot \vdash^i \mathbf{eval}\ e : \mathbf{nat}$	by rule **tpi_eval** on $\mathcal{T}_e$,
$\mathcal{T}''_i :: y : \mathbf{nat} \vdash^i \mathbf{case}^*\ y\ \mathbf{of}\ \mathbf{z}\ \Rightarrow\ e_1 \mid \mathbf{s}\ x\ \Rightarrow\ e_2 : \tau$	by weakening on $\mathcal{T}'_e$ and $\mathcal{T}''_e$, rules **tpe_x** and **tpi_case***,
$\mathcal{T}'_K :: \vdash^K K, \lambda y.\,\mathbf{case}^*\ y\ \mathbf{of}\ \mathbf{z}\ \Rightarrow\ e_1 \mid \mathbf{s}\ x\ \Rightarrow\ e_2 : \mathbf{nat} \Rightarrow \bar{\tau}$	by rule **tpK_lam** on $\mathcal{T}''_i$ and $\mathcal{T}_K$,
$\mathcal{T} :: \cdot \vdash^e v : \bar{\tau}$	by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}'_i$ and $\mathcal{T}'_K$.

```
pr-ev_case     : pr (ev_case E´) (tpi_eval (tpe_case Te´´ Te´ Te)) TK Ta
                   <- pr E´
                         (tpi_eval Te)
                         (tpK_lam TK [x][t:tpe x nat] tpi_case* Te´´ Te´ t)
                         Ta.
```

ev_letval

$$\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}' \\ K, \lambda x.\,\mathbf{eval}\ e_2 \vdash \mathbf{eval}\ e_1 \hookrightarrow v\end{array}}{K \vdash \mathbf{eval\ letval}\ x\ =\ e_1\ \mathbf{in}\ e_2 \hookrightarrow v}\ \mathbf{ev\_letval}$$

with $i = \mathbf{eval\ letval}\ x\ =\ e_1\ \mathbf{in}\ e_2$.

$\mathcal{T}'_e :: \cdot \vdash^e e_1 : \tau'$	and
$\mathcal{T}''_e :: x : \tau' \vdash^e e_2 : \tau$	by inversion on **tpi_eval** and **tpe_letval** for $\mathcal{T}_i$,
$\mathcal{T}'_i :: \cdot \vdash^i \mathbf{eval}\ e_1 : \tau'$	by rule **tpi_eval** on $\mathcal{T}'_e$,
$\mathcal{T}''_i :: x : \tau' \vdash^i \mathbf{eval}\ e_2 : \tau$	by rule **tpi_eval** on $\mathcal{T}''_e$,
$\mathcal{T}'_K :: \vdash^K K, \lambda x.\,\mathbf{eval}\ e_2 : \tau' \Rightarrow \bar{\tau}$	by rule **tpK_lam** on $\mathcal{T}''_i$ and $\mathcal{T}_K$,
$\mathcal{T} :: \cdot \vdash^e v : \bar{\tau}$	by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}'_i$ and $\mathcal{T}'_K$.

```
pr-ev_letval   : pr (ev_letval E´) (tpi_eval (tpe_letval Te´´ Te´)) TK Ta
                   <- pr E´
                         (tpi_eval Te´)
                         (tpK_lam TK [x][t:tpe x T´] tpi_eval (Te´´ x t))
                         Ta.
```

ev_letname

$$\mathcal{E} = \dfrac{\begin{array}{c}\mathcal{E}' \\ K \vdash \mathbf{eval}\ [e_1/x]e_2 \hookrightarrow v\end{array}}{K \vdash \mathbf{eval\ letname}\ x\ =\ e_1\ \mathbf{in}\ e_2 \hookrightarrow v}\ \mathbf{ev\_letname}$$

with $i = \mathbf{eval}\ \mathbf{letname}\ x\ =\ e_1\ \mathbf{in}\ e_2$.

$\mathcal{T}_e\ :: \cdot \vdash^e\ [e_1/x]e_2 : \tau$	by inversion on **tpi_eval** and **tpe_letname** for $\mathcal{T}_i$,
$\mathcal{T}_i'\ :: \cdot \vdash^i\ \mathbf{eval}\ [e_1/x]e_2 : \tau$	by rule **tpi_eval** on $\mathcal{T}_e$,
$\mathcal{T}\ :: \cdot \vdash^e\ v : \bar{\tau}$	by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$ and $\mathcal{T}_K$.

```
pr-ev_letname  : pr (ev_letname E') (tpi_eval (tpe_letname Te)) TK Ta
                  <- pr E' (tpi_eval Te) TK Ta.
```

ev_fix $\quad \mathcal{E} = \dfrac{\overset{\mathcal{E}'}{K \vdash \mathbf{eval}\ [\mathbf{fix}\ x.\, e/x]e \hookrightarrow v}}{K \vdash \mathbf{eval}\ \mathbf{fix}\ x.\, e \hookrightarrow v}\ \mathbf{ev\_fix}$

with $i = \mathbf{eval}\ \mathbf{fix}\ x.\, e$.

$\mathcal{T}_e'\ :: \cdot \vdash^e\ \mathbf{fix}\ x.\, e : \tau$	by inversion on **tpi_eval** for $\mathcal{T}_i$,
$\mathcal{T}_e''\ :: x : \tau \vdash^e\ e : \tau$	by inversion on **tpe_fix** for $\mathcal{T}_e'$,
$\mathcal{T}_e\ :: \cdot \vdash^e\ [\mathbf{fix}\ x.\, e/x]e : \tau$	by the substitution lemma on $\mathcal{T}_e'$ and $\mathcal{T}_e''$,
$\mathcal{T}_i'\ :: \cdot \vdash^i\ \mathbf{eval}\ [\mathbf{fix}\ x.\, e/x]e : \tau$	by rule **tpi_eval** on $\mathcal{T}_e$,
$\mathcal{T}\ :: \cdot \vdash^e\ v : \bar{\tau}$	by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$, $\mathcal{T}_K$.

```
pr-ev_fix      : {Te:{x} tpe x T -> tpe (X x) T}
                 pr (ev_fix E') (tpi_eval (tpe_fix Te'')) TK Ta
                  <- pr E' (tpi_eval (Te'' (fix X) (tpe_fix Te''))) TK Ta.
```

ev_init $\quad \mathcal{E} = \dfrac{}{\mathbf{init} \vdash \mathbf{return}\ v \hookrightarrow v}\ \mathbf{ev\_init}$

with $K = \mathbf{init}$ and $i = \mathbf{return}\ v$.

$\bar{\tau}\ = \tau$	by inversion on **tpK_init** for $\mathcal{T}_K$,
$\mathcal{T}\ :: \cdot \vdash^e\ v : \bar{\tau}$	by inversion on rule **tpi_return** for $\mathcal{T}_i$.

```
pr-ev_init     : pr ev_init (tpi_return Tv) tpK_init Tv.
```

ev_cont $\quad \mathcal{E} = \dfrac{\overset{\mathcal{E}'}{K' \vdash [v/x]i' \hookrightarrow v}}{K', \lambda x.\, i' \vdash \mathbf{return}\ v' \hookrightarrow v}\ \mathbf{ev\_cont}$

with $K = K', \lambda x.\, i'$ and $i = \mathbf{return}\ v'$.

$\mathcal{T}_i''\ :: x : \tau \vdash^i\ i' : \tau'$	and
$\mathcal{T}_K'\ :: \vdash^K\ K : \tau' \Rightarrow \bar{\tau}$	by inversion on **tpK_lam** for $\mathcal{T}_K$,
$\mathcal{T}_v\ :: \cdot \vdash^e\ v' : \tau$	by inversion on **tpi_return** for $\mathcal{T}_i$,
$\mathcal{T}_i'\ :: \cdot \vdash^i\ [v'/x]i' : \tau'$	by the substitution lemma on $\mathcal{T}_i''$ and $\mathcal{T}_v$,

$\mathcal{T}$ $:: \cdot \vdash^e v : \bar{\tau}$ by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$ and $\mathcal{T}_K'$.

```
pr-ev_cont     : {Tv: tpe V´ T}
                 pr (ev_cont E´) (tpi_return Tv) (tpK_lam TK Ti´´) Ta
                  <- pr E´ (Ti´´ V´ Tv) TK Ta.
```

ev_case$_2^*$

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}' \\ K \vdash \mathbf{eval}\ [v/x]e_2 \hookrightarrow v\end{array}}{K \vdash \mathbf{case}^*\ \mathbf{s}\ v'\ \mathbf{of}\ \mathbf{z}\ \Rightarrow\ e_1 \mid \mathbf{s}\ x\ \Rightarrow\ e_2 \hookrightarrow v}\ \mathbf{ev\_case}_2^*$$

with $i = \mathbf{case}^*\ \mathbf{s}\ v'\ \mathbf{of}\ \mathbf{z}\ \Rightarrow\ e_1 \mid \mathbf{s}\ x\ \Rightarrow\ e_2$.

$\mathcal{T}_v$ $:: \cdot \vdash^e v' : \mathbf{nat}$ and
$\mathcal{T}_e''$ $:: x : \mathbf{nat} \vdash^e e_2 : \tau$ by inversion on **tpi_case*** and **tpe_s** for $\mathcal{T}_i$,
$\mathcal{T}_e'$ $:: \cdot \vdash^e [v/x]e_2 : \tau$ by the substitution lemma on $\mathcal{T}_e''$ and $\mathcal{T}_v$,
$\mathcal{T}_i'$ $:: \cdot \vdash^e \mathbf{eval}\ [v'/x]e_2 : \tau$ by rule **tpi_eval** on $\mathcal{T}_e'$,
$\mathcal{T}$ $:: \cdot \vdash^e v : \bar{\tau}$ by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$ and $\mathcal{T}_K$.

```
pr-ev_case*2   : {Tv: tpe V nat}
                 pr (ev_case*2 E´) (tpi_case* Te´´ Te´ (tpe_s Tv)) TK Ta
                  <- pr E´ (tpi_eval (Te´´ V Tv)) TK Ta.
```

ev_app*

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}' \\ K, \lambda x.\, \mathbf{eval}\ e_1 \vdash \mathbf{eval}\ e_2 \hookrightarrow v\end{array}}{K \vdash \mathbf{app}^*\ (\mathbf{lam}\ x.\, e_1)\ e_2 \hookrightarrow v}\ \mathbf{ev\_app}^*$$

with $i = \mathbf{app}^*\ (\mathbf{lam}\ x.\, e_1)\ e_2$.

$\mathcal{T}_e''$ $:: \cdot \vdash^e e_2 : \tau'$ and
$\mathcal{T}_e'$ $:: x : \tau' \vdash^e e_1 : \tau$ by inversion on **tpi_app*** for $\mathcal{T}_i$,
$\mathcal{T}_i'$ $:: \cdot \vdash^i \mathbf{eval}\ e_2 : \tau'$ by rule **tpi_eval** on $\mathcal{T}_e''$,
$\mathcal{T}_i''$ $:: x : \tau' \vdash^i \mathbf{eval}\ e_1 : \tau$ by rule **tpi_eval** on $\mathcal{T}_e'$,
$\mathcal{T}_K'$ $:: \vdash^K K, \lambda x.\, \mathbf{eval}\ e_2 : \tau' \Rightarrow \bar{\tau}$ by rule **tpK_lam** on $\mathcal{T}_i'$ and $\mathcal{T}_K$,
$\mathcal{T}$ $:: \cdot \vdash^e v : \bar{\tau}$ by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$ and $\mathcal{T}_K'$.

```
pr-ev_app*     : pr (ev_app* E´) (tpi_app* Te´´ (tpe_lam Te´)) TK Ta
                  <- pr E´
                        (tpi_eval Te´´)
                        (tpK_lam TK [x][t:tpe x T´] tpi_eval (Te´ x t))
                        Ta.
```

☑

Parte II

Progettazione di un Logical Framework Basato sulla Logica Lineare

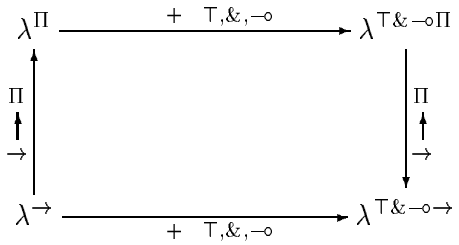
Capitolo 5

La Teoria dei Tipi

La teoria dei tipi λ^{Π} sottostante ad LF raffina il λ-calcolo semplicemente tipato di Church $\lambda^{\to}$ in modo che non solo il valore ma anche il tipo del risultato dell'applicazione di una funzione possa dipendere dal valore dell'argomento. La realizzazione di quest'idea induce vari cambiamenti nella sintassi del linguaggio. Innanzitutto il costruttore di tipi funzionale $\to$ di $\lambda^{\to}$ viene scartato in favore del costruttore di tipi dipendenti Π. Un tipo semplice $A \to B$ viene allora scritto come il tipo fittiziamente dipendente $\Pi x\!:\!A.\, B$ dove x non occorre libera in B. Inoltre, i tipi di base di $\lambda^{\to}$ acquisiscono struttura in λ^{Π}, prendendo la forma di una costante di tipo applicata ad una serie di termini oggetto: $a\ M_1 \ldots M_n$. Infine, la suddivisione dei termini in oggetti e tipi viene arricchita con una terza categoria sintattica, le kind, che permettono di dichiarare il tipo e il numero degli argomenti di una costante di tipo.

La transizione da $\lambda^{\to}$ a λ^{Π} corrisponde vagamente, per mezzo dell'isomorfismo di Curry-Howard, al passaggio dalla logica proposizionale a quella predicativa. Infatti, il costruttore di tipi dipendenti generalizza sia l'implicazione che la quantificazione universale, mentre la struttura delle formule atomiche assomiglia alla costruzione delle famiglie di tipi.

In questo capitolo, proponiamo un simile raffinamento del λ-calcolo semplicemente tipato lineare $\lambda^{\top\&\multimap\to}$ (detto anche $PLLF$) descritto in dettaglio in [Cer96]. Facciamo vedere come promuovere il costruttore di tipi funzionali intuizionistico $\to$ in modo da permettere dipendenze. La teoria dei tipi risultante, $\lambda^{\top\&\multimap\Pi}$, funge da base per il logical framework lineare LLF che promulghiamo in questa tesi. In alternativa, $\lambda^{\top\&\multimap\Pi}$ si può vedere come un'estensione di λ^{Π} con i tre costruttori di tipi lineari che abbiamo introdotto nel capitolo precedente, $\top$, $\&$ e $\multimap$, e dei corrispondenti proof-term. La relazione tra questi quattro sistemi è espressa nel seguente diagramma:



Ad un primo livello di approssimazione, $\lambda^{\top\&\multimap\Pi}$ corrisponde al frammento della logica lineare intuizionistica liberamente generato da $\top$, $\&$, $\multimap$, $\rightarrow$ e $\forall$. Come abbiamo detto, questo frammento è equivalente alla logica delle formule di Harrop ereditarie lineari, su cui il linguaggio di programmazione logica *Lolli* si basa. Tuttavia, un'analisi dettagliata della corrispondenza mostra che il nostro formalismo è ben piú ricco in quanto rende disponibili termini in un λ-calcolo lineare di ordine superiore, in contrapposizione ai termini intuizionistici permessi in *Lolli*.

La promozione di $\rightarrow$ al costruttore di tipi funzionali dipendenti Π porta in maniera naturale all'idea di definire versioni dipendenti di $\&$ e $\multimap$, che chiamiamo $\widetilde{\Sigma}$ e $\widehat{\Pi}$. Non abbiamo fatto questo passo ulteriore per due motivi. Innanzitutto porta a tremende complicazioni nella definizione della teoria dei tipi e conseguentemente nella sua meta-teoria. Inoltre, durante i nostri esperimenti con *LLF*, non abbiamo mai incontrato situazioni in cui $\widetilde{\Sigma}$ risultasse utile, e nelle poche circostanze in cui il costruttore di tipi funzionali lineari dipendenti $\widehat{\Pi}$ avrebbe potuto venire usato, la sua controparte intuizionistica Π è risultato sufficiente per i nostri bisogni di meta-rappresentazione.

Si noti che, sul versante logico dell'isomorfismo di Curry-Howard, $\widetilde{\Sigma}$ e $\widehat{\Pi}$ corrispondono ad un quantificatore esistenziale additivo e ad un quantificatore universale moltiplicativo, rispettivamente. Come appare dalla discussione nel Capitolo 3, i quantificatori sono gli unici costrutti della logica tradizionale che non sono stati raffinati in versioni additive e moltiplicative. Pertanto, la definizione di costruttori di tipi quali $\widetilde{\Sigma}$ e $\widehat{\Pi}$ avrebbe il positivo effetto di contribuire ad eliminare quest'anomalia della logica lineare. Sebbene abbiamo dedicato alcuni sforzi al questo problema, i nostri risultati su questo argomento non sono ancora abbastanza maturi per una loro presentazione.

Il potere espressivo di *LLF* quale logical framework verrà illustrato per mezzo di vari esempi nel Capitolo 6. Dedichiamo invece il presente capitolo allo studio della meta-teoria del sistema $\lambda^{\top\&\multimap\Pi}$ sottostante (che chiameremo *LLF* d'ora in poi). Formalizziamo la sintassi di questo formalismo in Sezione 5.1 e presentiamo le operazioni di base per esso in Sezione 5.2. Definiamo la sua semantica in Sezione 5.3 per mezzo di un sistema pre-canonico e dimostriamo le principali proprietà di questa teoria dei tipi nello stile di [HHP93]. In Sezione 5.4, facciamo vedere un equivalente sistema tale che ogni giudizio derivabile contiene solamente termini canonici. Nella sezione successiva, adattiamo le tecniche discusse nel capitolo precedente e facciamo vedere che *LLF* possiede le proprietà necessarie per interpretarlo e usarlo come un linguaggio di programmazione logica. Le dimostrazioni dei risultati conseguiti in questo capitolo, e dei lemmi di importanza secondaria sono stati raccolti nell'Appendice B (in Inglese). La numerazione è riferita a quest'appendice.

5.1 Il Linguaggio

LLF complica la sintassi del λ-calcolo semplicemente tipato *PLLF* presentato in [Cer96] con gli attributi necessari per accomodare tipi dipendenti. La transizione è simile al passaggio dal λ-calcolo semplicemente tipato di Church ad *LF*. La sintassi di *LLF* è specificata dalla seguente grammatica:

$$
\begin{array}{rrcl}
\textit{Oggetti:} & M & ::= & c \mid x \mid \lambda x{:}A.\,M \mid M_1\,M_2 \\
 & & \| & \langle\rangle \mid \langle M_1, M_2\rangle \mid \textsc{fst}\,M \mid \textsc{snd}\,M \mid \hat{\lambda} x{:}A.\,M \mid M_1 \hat{}\, M_2 \\
\textit{Famiglie di tipi:} & P & ::= & a \mid P\,M \\
\textit{Tipi:} & A & ::= & P \mid \Pi x{:}A_1.\,A_2 \;\|\; \top \mid A_1 \,\&\, A_2 \mid A_1 \multimap A_2 \\
\textit{Kind:} & K & ::= & \textsc{type} \mid \Pi x{:}A.\,K \\
\textit{Contesti:} & \Psi & ::= & \cdot \mid \Psi, x{:}A \;\|\; \Psi, x \,\hat{:}\, A \\
\textit{Segnature:} & \Sigma & ::= & \cdot \mid \Sigma, a{:}K \mid \Sigma, c{:}A
\end{array}
$$

Qui, x, c e a spaziano su variabili del livello oggetto, costanti oggetto e costanti di tipo, rispettivamente. In aggiunta ai nomi mostrati sopra, faremo spesso uso di N e B per oggetti e tipi rispettivamente. Inoltre, chiameremo *termine* un'espressione che è un oggetto, un tipo (o una famiglia di tipi) o un kind. Denotiamo termini con le lettere U e V.

Abbiamo evidenziato i costrutti nuovi rispetto ad *LF* separandoli per mezzo della doppia sbarra $\|$. È da notare la presenza di assunzioni lineari ($x \,\hat{:}\, A$) nel contesto in aggiunta alle normali assunzioni intuizionistiche ($x{:}A$) di *LF*. Infatti, la necessità di mantenere le assunzioni nell'ordine in cui vengono fatte, allo scopo di rispettare le dipendenze, ci impedisce di separare il contesto intuizionistico da quello lineare come nel caso semplicemente tipato [Cer96].

Similmente al caso semplicemente tipato, i nomi delle variabili legate dalle due forme di λ-astrazione o dal costruttore di tipi o kind dipendenti Π sono irrilevanti. Consideriamo uguali termini che differiscono solamente per il nome delle loro variabili legate. Assumiamo la presenza della regola di α-conversione per rinominare implicitamente queste variabili quando necessario.

5.2 Operazioni di Base

La strutturazione di un contesto Ψ di *LLF* come un'unica sequenza complica l'accesso alle varie parti che lo compongono. A questo scopo, definiamo due funzioni, la *parte intuizionistica* $\overline{\Psi}$ e la *parte lineare* $\widehat{\Psi}$, come segue:

$$
\left\{\begin{array}{lcl}
\overline{\cdot} & = & \cdot \\
\overline{\Psi, x{:}A} & = & \overline{\Psi}, x{:}A \\
\overline{\Psi, x \,\hat{:}\, A} & = & \overline{\Psi}
\end{array}\right.
\qquad\qquad
\left\{\begin{array}{lcl}
\widehat{\cdot} & = & \cdot \\
\widehat{\Psi, x{:}A} & = & \widehat{\Psi}, \\
\widehat{\Psi, x \,\hat{:}\, A} & = & \widehat{\Psi}, x \,\hat{:}\, A
\end{array}\right.
$$

Ci permetteranno di accedere a queste parti indipendentemente (la prima in particolare). Useremo questa notazione anche per esprimere il fatto che la porzione lineare del contesto è vincolata ad essere vuota scrivendo un tale contesto $\overline{\Psi}$.

$$\frac{}{\cdot = \cdot \bowtie \cdot}\ \textbf{s\_dot} \qquad \frac{\Psi = \Psi' \bowtie \Psi''}{(\Psi, x \hat{:} A) = (\Psi', x \hat{:} A) \bowtie \Psi''}\ \textbf{s\_lin1}$$

$$\frac{\Psi = \Psi' \bowtie \Psi''}{(\Psi, x : A) = (\Psi', x : A) \bowtie (\Psi'', x : A)}\ \textbf{s\_int} \qquad \frac{\Psi = \Psi' \bowtie \Psi''}{(\Psi, x \hat{:} A) = \Psi' \bowtie (\Psi'', x \hat{:} A)}\ \textbf{s\_lin2}$$

Figura 5.1: Partizione del Contesto

Un altro inconveniente della strutturazione del contesto come lista è l'impossibilità di usare l'operatore di concatenazione (,) per partizionare la parte lineare. Rimediamo a questo inconveniente per mezzo di un giudizio, $\Psi = \Psi_1 \bowtie \Psi_2$, definito in Figura 5.1.

La definizione di *variabile libera* in un termine è data come segue:

Oggetti:

$$\begin{aligned}
\mathrm{FV}(x) &= \{x\} \\
\mathrm{FV}(c) &= \varnothing \\
\mathrm{FV}(\langle\rangle) &= \varnothing \\
\mathrm{FV}(\langle M, N\rangle) &= \mathrm{FV}(M) \cup \mathrm{FV}(N) \\
\mathrm{FV}(\textsc{fst}\ M) &= \mathrm{FV}(M) \\
\mathrm{FV}(\textsc{snd}\ M) &= \mathrm{FV}(M) \\
\mathrm{FV}(\hat{\lambda} x : A.\, M) &= \mathrm{FV}(A) \cup (\mathrm{FV}(M) \setminus \{x\}) \\
\mathrm{FV}(M \hat{}\, N) &= \mathrm{FV}(M) \cup \mathrm{FV}(N) \\
\mathrm{FV}(\lambda x : A.\, M) &= \mathrm{FV}(A) \cup (\mathrm{FV}(M) \setminus \{x\}) \\
\mathrm{FV}(M\ N) &= \mathrm{FV}(M) \cup \mathrm{FV}(N)
\end{aligned}$$

Tipi e kind:

$$\begin{aligned}
\mathrm{FV}(a) &= \varnothing \\
\mathrm{FV}(P\ N) &= \mathrm{FV}(P) \cup \mathrm{FV}(N) \\
\mathrm{FV}(\top) &= \varnothing \\
\mathrm{FV}(A \,\&\, B) &= \mathrm{FV}(A) \cup \mathrm{FV}(B) \\
\mathrm{FV}(A \multimap B) &= \mathrm{FV}(A) \cup \mathrm{FV}(B) \\
\mathrm{FV}(\Pi x : A.\, B) &= \mathrm{FV}(A) \cup (\mathrm{FV}(B) \setminus \{x\}) \\
\\
\mathrm{FV}(\textsc{type}) &= \varnothing \\
\mathrm{FV}(\Pi x : A.\, K) &= \mathrm{FV}(A) \cup (\mathrm{FV}(K) \setminus \{x\})
\end{aligned}$$

Questa definizione si estende ai contesti come segue:

$$\begin{aligned}
\mathrm{FV}(\cdot) &= \varnothing \\
\mathrm{FV}(\Psi, x \hat{:} A) &= \mathrm{FV}(\Psi) \cup \mathrm{FV}(A) \\
\mathrm{FV}(\Psi, x : A) &= \mathrm{FV}(\Psi) \cup \mathrm{FV}(A)
\end{aligned}$$

Il *dominio* di un contesto è dato dall'insieme delle variabili in esso dichiarate. Formalmente,

$$\begin{aligned}
\mathrm{dom}\ \cdot &= \varnothing \\
\mathrm{dom}\ (\Psi, x \hat{:} A) &= \mathrm{dom}\ \Psi \cup \{x\} \\
\mathrm{dom}\ (\Psi, x : A) &= \mathrm{dom}\ \Psi \cup \{x\}
\end{aligned}$$

La restrizione di un contesto ad un insieme di variabili χ dato è definito come segue:

$$
\begin{array}{lcll}
\cdot_{|\chi} & = & \cdot & \\
(\Psi, x \hat{::} A)_{|\chi} & = & \left\{ \begin{array}{ll} \Psi_{|\chi}, x \hat{::} A & \text{se } x \in \chi \\ \Psi_{|\chi} & \text{altrimenti} \end{array} \right. &
\end{array}
$$

Al solito, scriviamo $[M/x]U$ per rappresentare la *sostituzione* priva di capture del termine di livello oggetto M in luogo della variabile x nel termine U:

Oggetti:

$$
\begin{array}{lcl}
[N/x]y & = & \left\{ \begin{array}{ll} N & \text{se } x = y \\ y & \text{altrimenti} \end{array} \right. \\
[N/x]c & = & c \\
[N/x]\langle\rangle & = & \langle\rangle \\
[N/x]\langle M_1, M_2\rangle & = & \langle [N/x]M_1, [N/x]M_2\rangle \\
[N/x](\textsc{fst}\ M) & = & \textsc{fst}\ [N/x]M \\
[N/x](\textsc{snd}\ M) & = & \textsc{snd}\ [N/x]M \\
[N/x](\hat{\lambda} y{:}A.\, M) & = & \hat{\lambda} y{:}[N/x]A.\, [N/x]M \\
[N/x](M_1 \hat{}\, M_2) & = & ([N/x]M_1) \hat{}\, ([N/x]M_2) \\
[N/x](\lambda y{:}A.\, M) & = & \lambda y{:}[N/x]A.\, [N/x]M \\
[N/x](M_1\ M_2) & = & ([N/x]M_1)\ ([N/x]M_2)
\end{array}
$$

Tipi e kind:

$$
\begin{array}{lcl}
[N/x]a & = & a \\
[N/x](P\ M) & = & ([N/x]P)\ ([N/x]M) \\
[N/x]\top & = & \top \\
[N/x](A \,\&\, B) & = & [N/x]A \,\&\, [N/x]B \\
[N/x](A \multimap B) & = & [N/x]A \multimap [N/x]B \\
[N/x](\Pi y{:}A.\, B) & = & \Pi y{:}[N/x]A.\, [N/x]B \\
[N/x]\textsc{type} & = & \textsc{type} \\
[N/x](\Pi y{:}A.\, K) & = & \Pi y{:}[N/x]A.\, [N/x]K
\end{array}
$$

Questa nozione di estende naturalmente ai contesti:

$$
\begin{array}{lcl}
[N/x]\cdot & = & \cdot \\
[N/x](\Psi, y \hat{:} A) & = & [N/x]\Psi,\ y \hat{:} [N/x]A \\
[N/x](\Psi, y{:}A) & = & [N/x]\Psi,\ y : [N/x]A
\end{array}
$$

5.3 Forme Pre-canoniche

Il significato delle entità sintattiche di un linguaggio si possono presentare sotto varie forme, la scelta essendo dettata dagli aspetti che vogliamo mettere in evidenza. In questa sezione, definiamo la semantica di *LLF* per mezzo di un sistema deduttivo che chiamiamo *pre-canonico*. Questo sistema si basa su una nozione di uguaglianza definizionale che incorpora solamente la β-equivalenza. Tuttavia, il sistema è costruito in maniera tale che ogni oggetto derivabile è in forma η-espansa.

Quest'ultima caratteristica differenzia questa presentazione dal trattamento originario di LF in [HHP93]. Lí, Harper, Honsell e Plotkin presentano un calcolo che si basa solamente sulle regole di β-conversione, ma in grado di derivare anche termini in forma non η-espansa. L'assenza di regole estensionali semplifica notevolmente la meta-teoria di questi formalismi. Tuttavia, l'uso di LF, cosí come di LLF, sia per fare meta-rappresentazione che per proof-search, si basa su termini in forma espansa. Questo difetto nella presentazione di LF fu successivamente corretto da vari autori [Coq91, Geu93, Sal90], al costo di non poche complicazioni. Il nostro approccio mantiene la semplicità di [HHP93], senza però dare origine ad una discrepanza tra il linguaggio usato in pratica e il linguaggio studiato teoricamente. Ripetiamo infatti che la presentazione che proponiamo permette di derivare solamente entità in forma η-espansa.

Procediamo con la discussione come segue: dopo una presentazione generale del sistema deduttivo pre-canonico per LLF in Sottosezione 5.3.1, ci concentriamo sulla teoria equazionale in 5.3.2 e poi presentiamo alcune proprietà di base in 5.3.3. Dimostriamo la normalizzazione forte per questo sistema in 5.3.4 e ne diamo una formulazione algoritmica in 5.3.5 che ci serve a dimostrare la decidibilità del type-checking per LLF in Sottosezione 5.3.6.

5.3.1 Presentazione

Le regole che compongono il sistema pre-canonico sono mostrate in Figure 5.2 e 5.3. La regola **opa_lapp** si basa sul giudizio di partizionamento del contesto mostrato in Figura 5.1. Le regole che definiscono la nozione di uguaglianza definizionale di LLF sono racchiuse in Figure 5.4–5.6. Allo scopo di abbreviare la trattazione e renderla piú leggibile, conglobiamo le espressioni $\Psi \vdash_\Sigma U \Uparrow V$ e $\Psi \vdash_\Sigma U \downarrow V$ scrivendo $\Psi \vdash_\Sigma U \Uparrow\!\!\downarrow V$ in proprietà in cui entrambe le versioni sono valide.

Le regole in Figure 5.2–5.3 spiccano per due caratteristiche. Innanzitutto, differiscono da una presentazione canonica simile a quanto descritto nel precedente capitolo solamente per la presenza della regola **opa_c**, la quale è responsabile per la possibilità di formazione di β-redex durante il processo di derivazione. Altre regole, quali **fpa_eq**, **opc_eq** e **opa_eq** permettono l'applicazione delle regole definenti l'uguaglianza definizionale a tipi e kind; le abbiamo già incontrate nel caso di LF. Il secondo aspetto che caratterizza questo calcolo è l'ampio uso della funzione $\overline{\ldots}$. Viene usata nelle regole che definiscono il livello oggetto, dove serve a modellare la nozione di linearità in maniera quasi diretta rispetto a quanto visto nel capitolo precedente. Inoltre, il suo uso è sistematico nelle regole per tipi e kind. Infatti, non permettiamo a entità di questo tipo di dipendere da variabili lineari. Diciamo che esse sono *linearmente chiuse*. Una liberalizzazione di questo vincolo avrebbe l'effetto di richiedere la presenza di tipi dipendenti lineari, senza però aumentare significativamente la potenza espressiva del calcolo risultante.

La nozione di *uguaglianza definizionale* che consideriamo in Figure 5.4–5.6 è la relazione di equivalenza $\equiv$ costruita sulla relazione di congruenza $\longrightarrow$. Questa presentazione usa la strategia di *riduzione parallela nidificata*. Ignorando alcuni dettagli, questa congruenza si basa sulle seguenti regole di *β-riduzione*:

$$\beta_{fst}: \quad \mathrm{FST}\,\langle M, N\rangle \longrightarrow M \qquad \beta_{lapp}: \quad (\hat{\lambda} x{:}A.\, M)\,\hat{}\,N \longrightarrow [N/x]M$$
$$\beta_{snd}: \quad \mathrm{SND}\,\langle M, N\rangle \longrightarrow N \qquad \beta_{iapp}: \quad (\lambda x{:}A.\, M)\,N \longrightarrow [N/x]M.$$

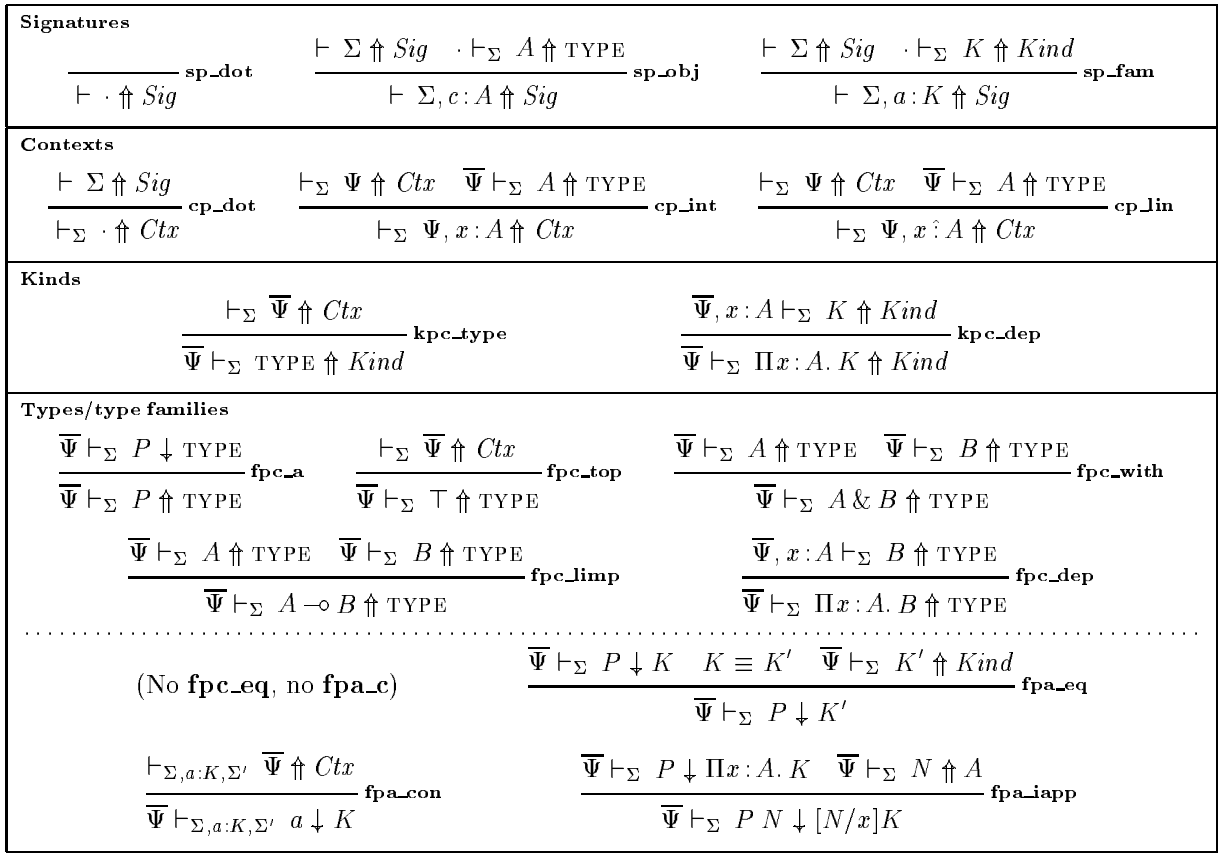


Figura 5.2: Sistema Pre-canonico per *LLF*, Kind e Tipi

Le espressioni che appaiono sulla sinistra sono dette *β-redex*. Un termine è in *forma normale* se non contiene alcun β-redex. Dimostreremo che ogni termine derivabile ammette una forma normale. Si noti che questa proprietà non vale in generale per termini non derivabili.

5.3.2 La Teoria Equazionale

La nostra nozione di uguaglianza definizionale si avvale di numerose proprietà che risulteranno utili nel dimostrare successivi risultati. Fra le piú importanti, vi è la confluenza, espressa come segue:

Objects

$$\frac{\Psi \vdash_\Sigma M \downarrow P}{\Psi \vdash_\Sigma M \Uparrow P}\ \textbf{opc\_a} \qquad \frac{\Psi \vdash_\Sigma M \Uparrow B \quad A \equiv B \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE}}{\Psi \vdash_\Sigma M \Uparrow A}\ \textbf{opc\_eq}$$

$$\frac{\vdash_\Sigma \Psi \Uparrow Ctx}{\Psi \vdash_\Sigma \langle\rangle \Uparrow \top}\ \textbf{opc\_unit} \qquad \frac{\Psi \vdash_\Sigma M \Uparrow A \quad \Psi \vdash_\Sigma N \Uparrow B}{\Psi \vdash_\Sigma \langle M, N\rangle \Uparrow A \,\&\, B}\ \textbf{opc\_pair}$$

$$\frac{\Psi, x \hat{:} A \vdash_\Sigma M \Uparrow B}{\Psi \vdash_\Sigma \hat{\lambda} x : A.\, M \Uparrow A \multimap B}\ \textbf{opc\_llam} \qquad \frac{\Psi, x : A \vdash_\Sigma M \Uparrow B}{\Psi \vdash_\Sigma \lambda x : A.\, M \Uparrow \Pi x : A.\, B}\ \textbf{opc\_ilam}$$

$$\frac{\Psi \vdash_\Sigma M \Uparrow A}{\Psi \vdash_\Sigma M \downarrow A}\ \textbf{opa\_c} \qquad \frac{\Psi \vdash_\Sigma M \downarrow B \quad A \equiv B \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE}}{\Psi \vdash_\Sigma M \downarrow A}\ \textbf{opa\_eq}$$

$$\frac{\vdash_{\Sigma, c:A, \Sigma'} \overline{\Psi} \Uparrow Ctx}{\overline{\Psi} \vdash_{\Sigma, c:A, \Sigma'} c \downarrow A}\ \textbf{opa\_con} \qquad \frac{\vdash_\Sigma \overline{\Psi}, x \hat{:} A, \overline{\Psi'} \Uparrow Ctx}{\overline{\Psi}, x \hat{:} A, \overline{\Psi'} \vdash_\Sigma x \downarrow A}\ \textbf{opa\_lvar} \qquad \frac{\vdash_\Sigma \overline{\Psi}, x : A, \overline{\Psi'} \Uparrow Ctx}{\overline{\Psi}, x : A, \overline{\Psi'} \vdash_\Sigma x \downarrow A}\ \textbf{opa\_ivar}$$

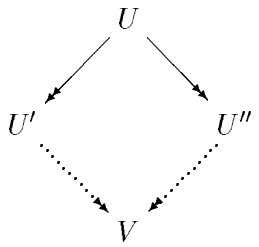
(No rule for $\top$)

$$\frac{\Psi \vdash_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash_\Sigma \text{FST}\ M \downarrow A}\ \textbf{opa\_fst} \qquad \frac{\Psi \vdash_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash_\Sigma \text{SND}\ M \downarrow B}\ \textbf{opa\_snd}$$

$$\frac{\Psi' \vdash_\Sigma M \downarrow A \multimap B \quad \Psi'' \vdash_\Sigma N \Uparrow A \quad \Psi = \Psi' \bowtie \Psi''}{\Psi \vdash_\Sigma M \hat{} N \downarrow B}\ \textbf{opa\_lapp} \qquad \frac{\Psi \vdash_\Sigma M \downarrow \Pi x : A.\, B \quad \overline{\Psi} \vdash_\Sigma N \Uparrow A}{\Psi \vdash_\Sigma M\ N \downarrow [N/x]B}\ \textbf{opa\_iapp}$$

Figura 5.3: Sistema Pre-canonico per *LLF*, Oggetti

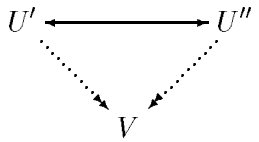
Lemma 5.3.10 (*Confluenza*)



Se $U \longrightarrow^* U'$ *e* $U \longrightarrow^* U''$, *allora esiste un termine* V *tale che* $U' \longrightarrow^* V$ *e* $U'' \longrightarrow^* V$. □

Un'importante conseguenza di questo risultato è la proprietà di Church-Rosser, che riduce termini equivalenti ad una comune radice.

Teorema 5.3.11 (*Church-Rosser*)



Se $U' \equiv U''$, *allora esiste un termine* V *tale che* $U' \longrightarrow^* V$ *e* $U'' \longrightarrow^* V$. □

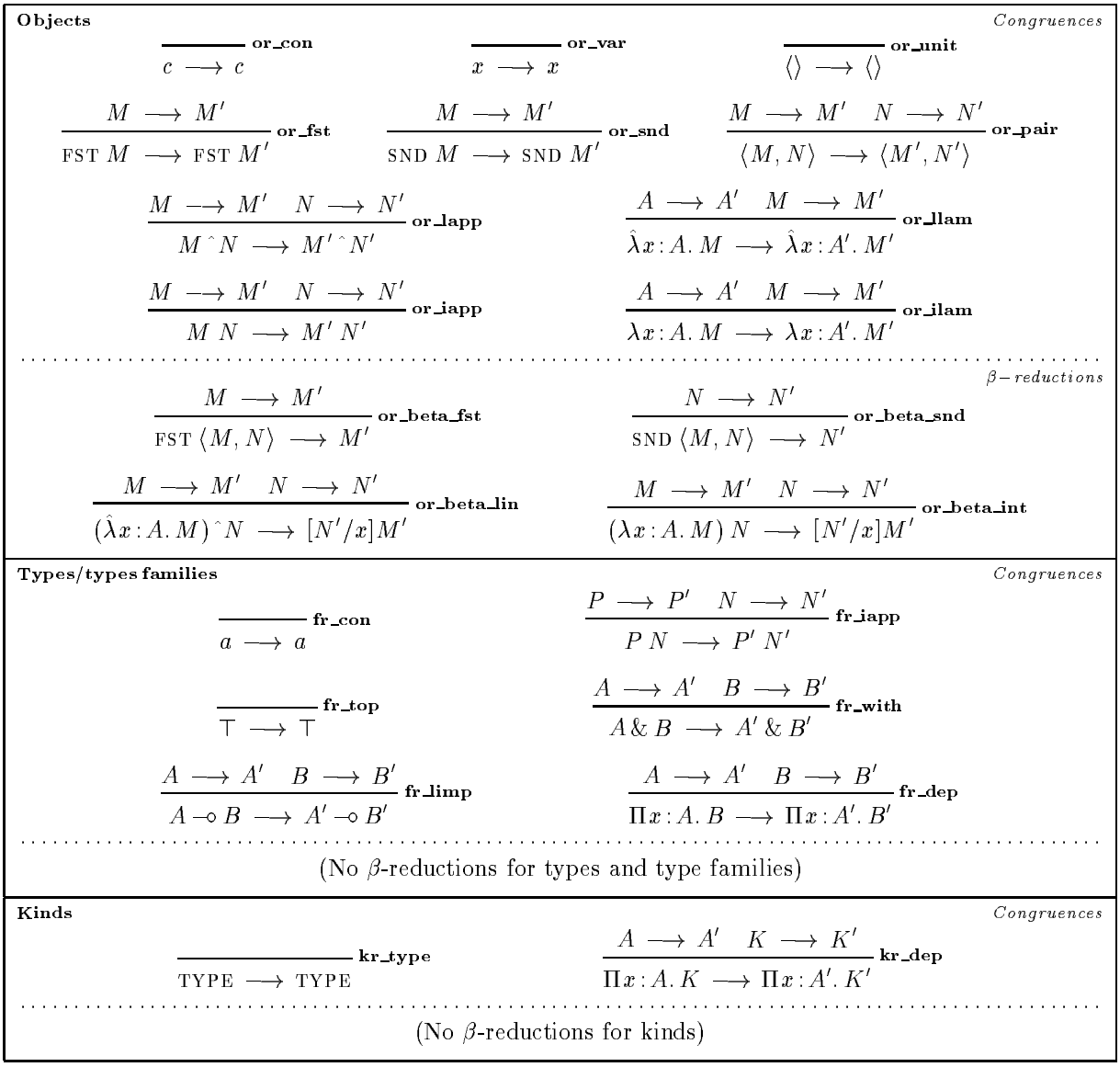


Figura 5.4: Riduzione Parallela Nidificata per *LLF*

5.3.3 Proprietà fondamentali

Passiamo ora alla presentazione di alcune proprietà fondamentali della teoria dei tipi di *LLF*. Incominciamo con lemma che permette di capire come vengono usate le variabili, specie quelle lineari, nel nostro linguaggio.

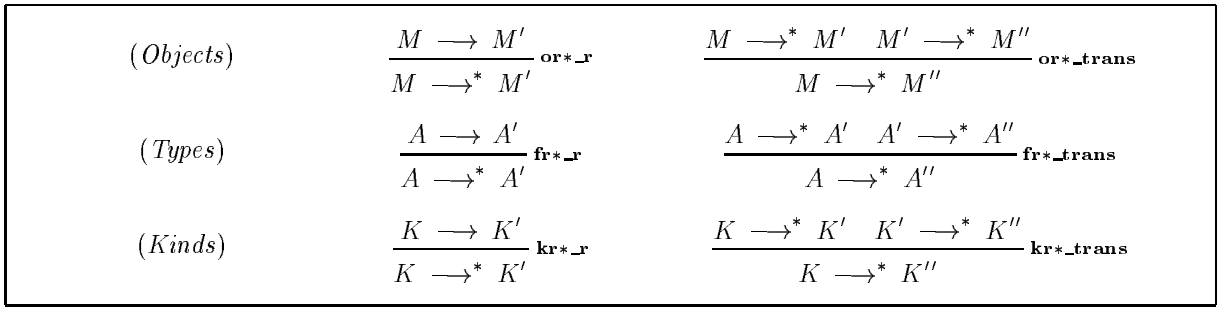


Figura 5.5: Riduzione Parallela Nidificata per LLF, Chiusura Transitiva

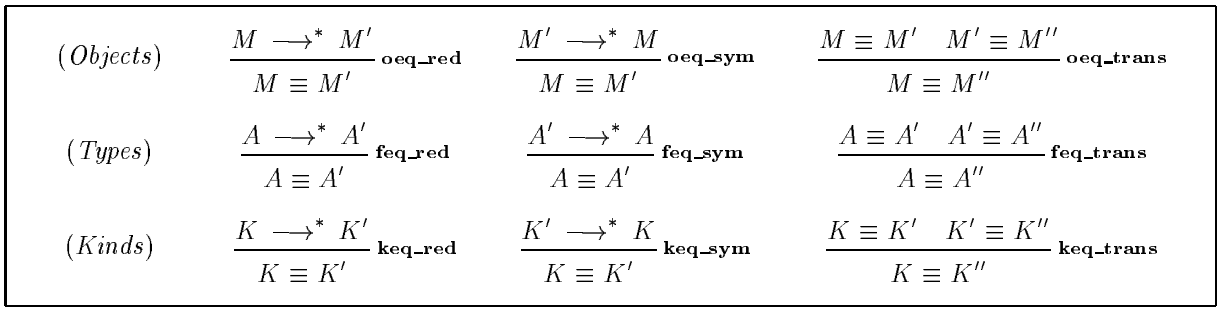


Figura 5.6: Riduzione Parallela Nidificata per LLF, Equivalenza

Lemma 5.3.12 (*Variabili libere*)

i. *Se* $\Psi \vdash_\Sigma U \Uparrow\!\downarrow V$, *allora* $\mathrm{FV}(U) \subseteq \mathrm{dom}\,\Psi$ *e* $\mathrm{FV}(V) \subseteq \mathrm{dom}\,\overline{\Psi}$;

ii. *Se* $\Psi \vdash_\Sigma U \Uparrow\!\downarrow V$ *o* $\vdash_\Sigma \Psi \Uparrow Ctx$, *allora* $\mathrm{FV}(\Psi) \subseteq \mathrm{dom}\,\overline{\Psi}$;

iii. *Se* $\Psi \vdash_\Sigma U \Uparrow\!\downarrow V$ *o* $\vdash_\Sigma \Psi \Uparrow Ctx$ *o* $\vdash \Sigma \Uparrow Sig$, *allora* $\mathrm{FV}(\Sigma) = \varnothing$;

iv. *Se* $\Psi, x :\!\!: A, \Psi' \vdash_\Sigma U \Uparrow\!\downarrow V$ *o* $\vdash_\Sigma \Psi, x :\!\!: A, \Psi' \Uparrow Ctx$, *allora* $\mathrm{FV}(A) \cup \mathrm{FV}(\Psi) \subseteq \mathrm{dom}\,\overline{\Psi}$. □

Si noti in particolare che il tipi, kind, contesto e segnatura non possono contenere variabili lineari libere. Esprimiamo questa proprietà dicendo che sono *linearmente chiusi*. Si noti inoltre che le segnature non possono menzionare variabili libere di alcun genere.

La prossima proprietà che presentiamo riguarda le regole strutturali di permutazione e weakening. La prima afferma che possono venire liberamente scambiate a patto di non violare dipendenze, la seconda specifica che si possono aggiungere assunzioni intuizionistiche senza invalidare alcuna proprietà.

Lemma 5.3.15 (*Proprietà strutturali*)

Permutazione:

i. Se $\Psi, \Psi', x \hat{::} A, \Psi'' \vdash_\Sigma U \Uparrow\!\downarrow V$ *e* $\overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}$, *allora* $\Psi, x \hat{::} A, \Psi', \Psi'' \vdash_\Sigma U \Uparrow\!\downarrow V$;

ii. Se $\vdash_\Sigma \Psi, \Psi', x \hat{::} A, \Psi'' \Uparrow Ctx$ *e* $\overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}$, *allora* $\vdash_\Sigma \Psi, x \hat{::} A, \Psi', \Psi'' \Uparrow Ctx$.

Weakening:

Se $\Psi \vdash_\Sigma U \Uparrow\!\downarrow V$ *e* $\vdash_\Sigma \Psi, \overline{\Psi'} \Uparrow Ctx$, *allora* $\Psi, \overline{\Psi'} \vdash_\Sigma U \Uparrow\!\downarrow V$. □

Ci si aspetterebbe una terza proprietà strutturale, il strengthening, il quale specifica che un'assunzione che non viene usata in alcun luogo in un giudizio può essere eliminata. Sebbene valida, non abbiamo la possibilità di dimostrarla al momento in quanto l'uso delle regole di β-riduzione come regole di espansione potrebbe farla comparire nella derivazione di questo giudizio, anche se non appare nel giudizio stesso. Saremo in grado di dimostrare questa proprietà solo dopo avere provato il teorema di normalizzazione forte.

Il lemma che segue è di estrema importanza al momento di usare *LLF* come linguaggio di meta-rappresentazione. Infatti, essa giustifica l'uso di tipi dipendenti intuizionistici in quelle occasioni in cui le loro versioni lineari sarebbero necessarie.

Lemma 5.3.17 (*Appiattimento*)

Se $\Psi, x \hat{:} A, \Psi' \vdash_\Sigma M \Uparrow\!\downarrow B$, *allora* $\Psi, x : A, \Psi' \vdash_\Sigma M \Uparrow\!\downarrow B$. □

I risultati che qui chiamiamo *transitività* hanno un elevato numero di significati nei vari contesti in una teoria dei tipi viene considerata. Qui, li vediamo come un'ulteriore proprietà strutturale che specifica la transitività, in senso lato, della nostra nozione di derivabilità. Si possono però anche interpretare come principi di sostituzione e, modulo l'uso dell'isomorfismo di Curry-Howard, come teoremi di eliminazione del taglio.

Lemma 5.3.18 (*Transitività intuizionistica*)

i. Se $\overline{\Psi} \vdash_\Sigma N \Uparrow A$ *e* $\Psi, x : A, \Psi' \vdash_\Sigma U \Uparrow\!\downarrow V$, *allora* $\Psi, [N/x]\Psi' \vdash_\Sigma [N/x]U \Uparrow\!\downarrow [N/x]V$;

ii. Se $\overline{\Psi} \vdash_\Sigma N \Uparrow A$ *e* $\vdash_\Sigma \Psi, x : A, \Psi' \Uparrow Ctx$, *allora* $\vdash_\Sigma \Psi, [N/x]\Psi' \Uparrow Ctx$. □

Lemma 5.3.20 (*Transitività lineare*)

i. Se $\Psi \vdash_\Sigma N \Uparrow A$ *e* $\overline{\Psi}, x \hat{:} A, \Psi' \vdash_\Sigma M \Uparrow\!\downarrow B$, *allora* $\Psi, \Psi' \vdash_\Sigma [N/x]M \Uparrow\!\downarrow B$;

ii. Se $\Psi' \vdash_\Sigma N \Uparrow A$ *e* $\vdash_\Sigma \overline{\Psi'}, x \hat{:} A, \Psi' \Uparrow Ctx$, *allora* $\vdash_\Sigma \Psi, \Psi' \Uparrow Ctx$. □

Un'altra importante proprietà del nostro formalismo è che i tipi e i kind che possono apparire nella parte destra di giudizi principali costituenti il sistema pre-canonico sono unici, modulo uguaglianza definizionale.

Lemma 5.3.21 (*Unicità di tipi e kind*)

i. Se $\Psi' \vdash_\Sigma M \Uparrow\!\downarrow A'$ *e* $\Psi'' \vdash_\Sigma M \Uparrow\!\downarrow A''$ *con* $\Psi'_{|\mathrm{FV}(M)} = \Psi''_{|\mathrm{FV}(M)}$ *e dove le frecce possono non coincidere, allora* $A' \equiv A''$;

ii. Se $\overline{\Psi} \vdash_\Sigma A \Uparrow\!\downarrow K'$ *e* $\overline{\Psi} \vdash_\Sigma A \Uparrow\!\downarrow K''$ *dove le frecce possono non coincidere, allora* $K' \equiv K''$. □

Concludiamo questa rassegna di proprietà di base facendo vedere che il nostro calcolo è estensionale. Una conseguenza di questo lemma è che tutte le espressioni che compaiono in un giudizio derivabile sono in forma η-espansa, tranne al piú il termine U in giudizi del tipo $\Psi \vdash_\Sigma U \downarrow V$, che servono precisamente a costruire oggetti in forma η-espansa.

Lemma 5.3.24 (*Estensionalità*)

i. Se $\Psi \vdash_\Sigma M \Uparrow \top$, *allora* $M = \langle\rangle$;

ii. Se $\Psi \vdash_\Sigma M \Uparrow A \,\&\, B$, *allora* $M = \langle M_1, M_2\rangle$;

iii. Se $\Psi \vdash_\Sigma M \Uparrow A \multimap B$, *allora* $M = \hat{\lambda} x{:}A'.\, M_1$ *con* $A \equiv A'$;

iv. Se $\Psi \vdash_\Sigma M \Uparrow \Pi x{:}A.\, B$, *allora* $M = \lambda x{:}A'.\, M_1$ *con* $A \equiv A'$. □

5.3.4 Normalizzazione Forte

In *LLF*, cosí come in *LF*, esistono termini che non sono riducibili ad una forma normale. Il problema se un termine sia normalizzabile o meno è in generale indecidibile, in quanto equivalente all'*halting problem* per i linguaggi funzionali, la cui semantica operazionale è basata sull'operazione di riduzione. Tuttavia, la forte tipizzazione di *LF* e *LLF* fa sí che ogni termine derivabile è riducibile ad un'unica forma normale. Per di piú, questa forma normale si può ottenere applicando riduzioni agli eventuali β-redex che compaiono in un termine in ordine arbitrario. Questa è l'essenza del teorema di normalizzazione forte per *LLF*.

La dimostrazione di questa proprietà ricalca la tecnica originariamente proposta per *LF* in [HHP93] e richiede un pò di lavoro preliminare. L'idea è di ridurre la dimostrazione della normalizzazione forte per *LLF* all'analoga proprietà del λ-calcolo semplicemente tipato con coppie $\lambda^{\times\to}$. Proponiamo una funzione di codifica che mantiene sequenze di riduzione e derivabilità.

Prima di fare ciò, è preferibile cambiare la strategia di riduzione su cui ci basiamo. Le regole in Figure 5.7–5.9 specificano la strategia nota come *one-step*, che ha la proprietà di operare un'unica

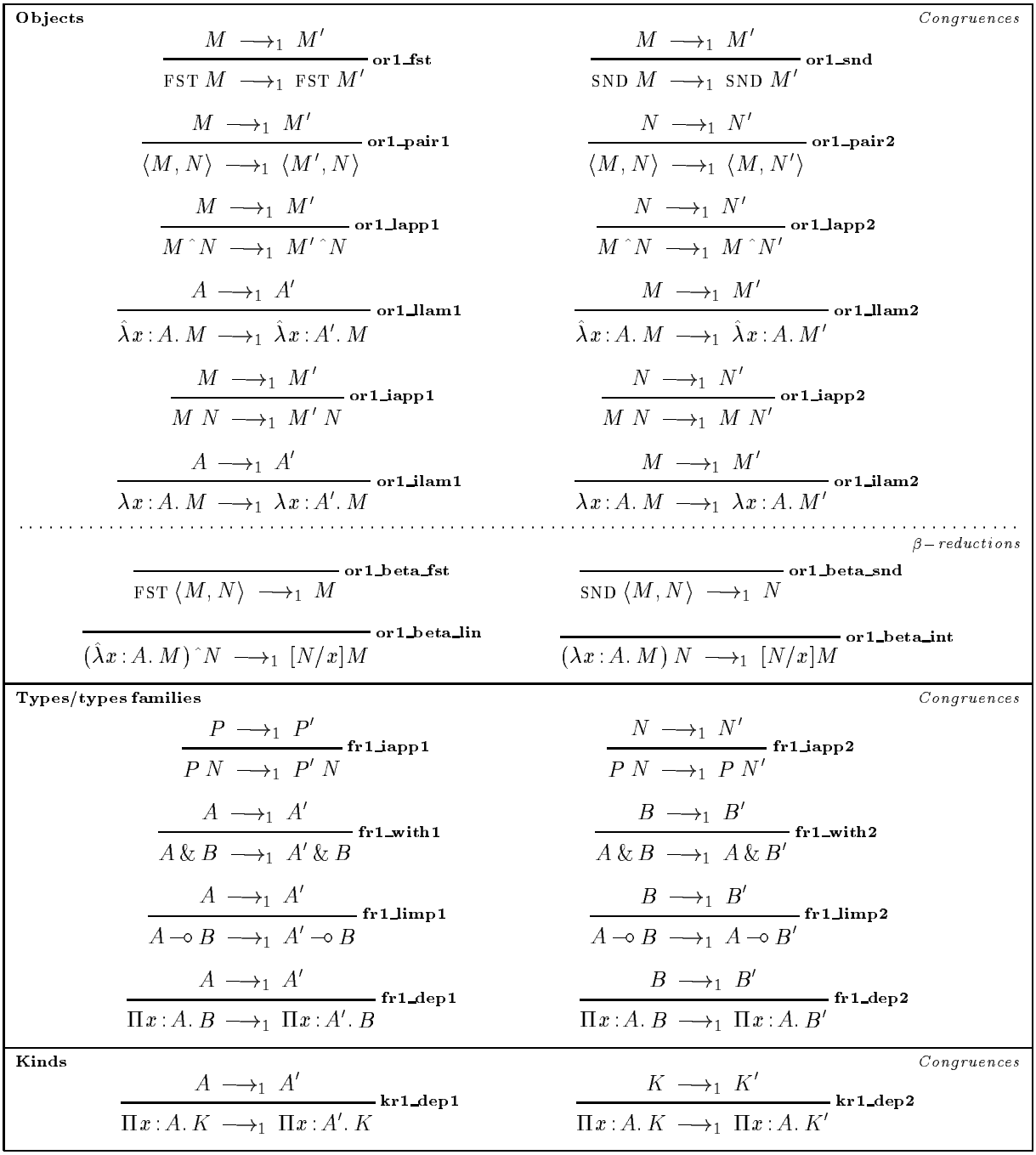


Figura 5.7: Riduzione One-step per *LLF*

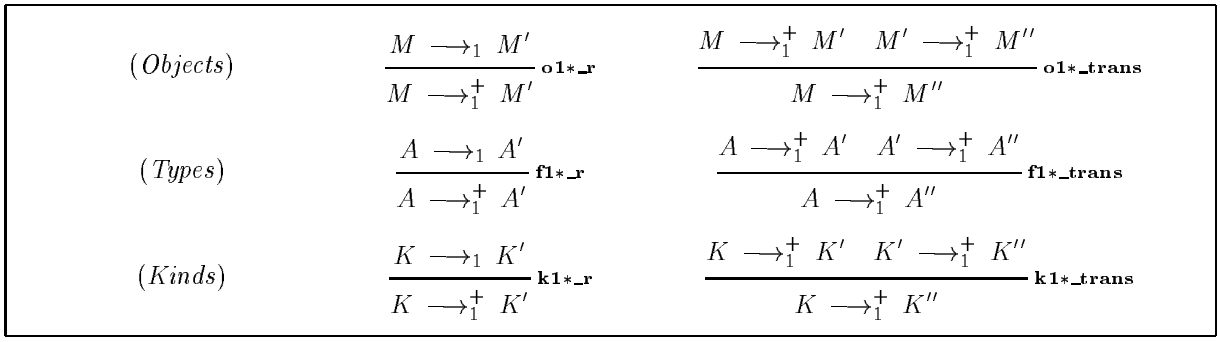


Figura 5.8: Riduzione One-step per *LLF*, Chiusura Transitiva

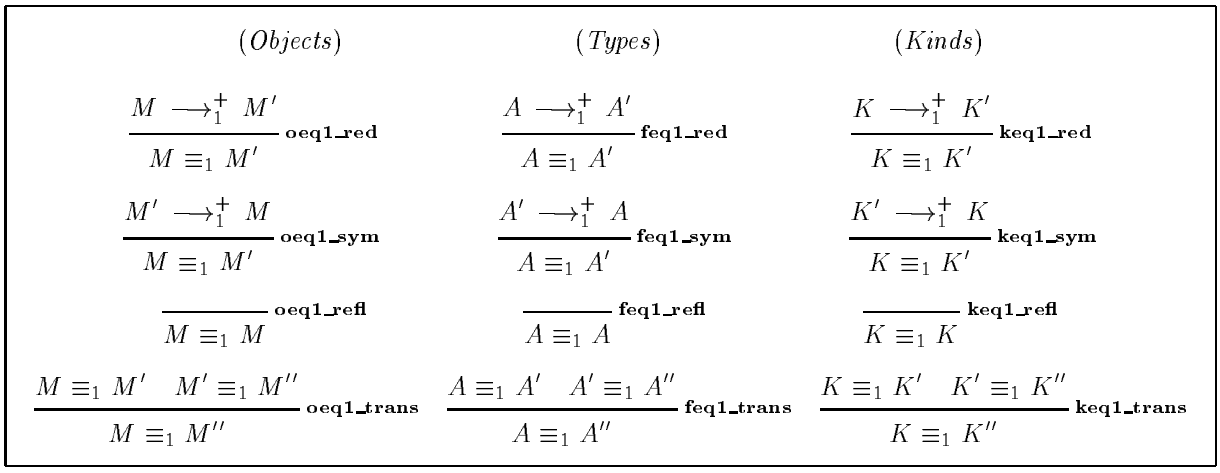


Figura 5.9: Riduzione One-step per *LLF*, Equivalenza

riduzione in un termine dato, a differenza della strategia originariamente adottata, che può operare zero o piú passi simultaneamente. Questa presentazione è tuttavia equivalente alla strategia di riduzione parallela nidificata, come espresso dal seguente lemma:

Lemma 5.3.25 (*Correttezza e completezza di* $\equiv_1$ *rispetto a* $\equiv$)

i. *Se* $U \longrightarrow U'$, *allora o* $U = U'$ *oppure* $U \longrightarrow_1^+ U'$;

ii. *Se* $U \longrightarrow^* U'$, *allora o* $U = U'$ *oppure* $U \longrightarrow_1^+ U'$;

iii. *Se* $U \longrightarrow_1 U'$, *allora* $U \longrightarrow U'$;

iv. *Se* $U \longrightarrow_1^+ U'$, *allora* $U \longrightarrow^* U'$;

v. $U \equiv_1 U'$ *se e solo se* $U \equiv U'$. □

Il lemma che segue afferma che la relazione di derivabilità per LLF è chiusa rispetto all'operazione di riduzione. È di fondamentale importanza allo scopo di dimostrare il teorema di normalizzazione forte.

Lemma 5.3.28 (*Subject reduction*)

Se $\Psi \vdash_\Sigma U \Uparrow\!\downarrow V$ *e* $U \longrightarrow_1 U'$, *allora* $\Psi \vdash_\Sigma U' \Uparrow\!\downarrow V$. □

Il linguaggio del λ-calcolo semplicemente tipato con coppie $\lambda^{\times\to}$ è specificato dalla seguente grammatica:

$$\begin{array}{rrcl}
\textit{Oggetti:} & M & ::= & c \mid x \mid \langle M_1, M_2\rangle^s \mid \textsc{fst}^s\, M \mid \textsc{snd}^s\, M \mid \lambda^s x{:}\sigma.\, M \mid M_1\, M_2 \\
\textit{Tipi:} & \sigma & ::= & b \mid \sigma_1 \times^s \sigma_2 \mid \sigma_1 \to^s \sigma_2 \\
\textit{Contesti:} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\sigma \\
\textit{Segnature:} & \Sigma & ::= & \cdot \mid \Sigma, c{:}\sigma
\end{array}$$

Esso differisce dal λ-calcolo semplicemente tipato di Church solamente per l'aggiunta del costruttore di prodotti cartesiani e dei relativi operatori al livello oggetto. Le regole di tipizzazione e di riduzione vengono mostrate in Figura 5.10.

La codifica che proponiamo trasforma giudizi di LLF $\Psi \vdash_\Sigma U \Uparrow\!\downarrow V$ in espressioni di $\lambda^{\times\to}$ della forma $\Gamma \vdash_{\Sigma'} M : \sigma$. Associa il termine generico U all'oggetto M in $\lambda^{\times\to}$, V al tipo semplice σ, Σ alla segnatura Σ' e Ψ al contesto Γ. Usiamo la funzione $\tau(\_)$ per rappresentare tipi e kind di LLF come tipi di $\lambda^{\times\to}$. Usiamo invece la famiglia di funzioni $|\_|$ per termini, contesti e segnature. Questa codifica è specificata come segue:

Tipi:

$$\begin{array}{lcl}
\tau(P) & = & \omega \\
\tau(\top) & = & \omega \\
\tau(A \,\&\, B) & = & \tau(A) \times^s \tau(B) \\
\tau(A \multimap B) & = & \tau(A) \to^s \tau(B) \\
\tau(\Pi x{:}A.\, B) & = & \tau(A) \to^s \tau(B)
\end{array}$$

Kind:

$$\begin{array}{lcl}
\tau(\textsc{type}) & = & \omega \\
\tau(\Pi x{:}A.\, K) & = & \tau(A) \to^s \tau(K)
\end{array}$$

Typing rules

$$\frac{}{\Gamma \vdash_{\Sigma,c:\sigma} \; c : \sigma}\ \textbf{st\_con} \qquad \frac{}{\Gamma, x:\sigma \vdash_\Sigma \; x : \sigma}\ \textbf{st\_var}$$

$$\frac{\Gamma \vdash_\Sigma \; M_1 : \sigma_1 \quad \Gamma \vdash_\Sigma \; M_2 : \sigma_2}{\Gamma \vdash_\Sigma \; \langle M_1, M_2\rangle^s : \sigma_1 \times^s \sigma_2}\ \textbf{st\_pair} \qquad \frac{\Gamma \vdash_\Sigma \; M : \sigma \times^s \sigma'}{\Gamma \vdash_\Sigma \; \mathrm{FST}^s \, M : \sigma}\ \textbf{st\_fst} \qquad \frac{\Gamma \vdash_\Sigma \; M : \sigma' \times^s \sigma}{\Gamma \vdash_\Sigma \; \mathrm{SND}^s \, M : \sigma}\ \textbf{st\_snd}$$

$$\frac{\Gamma, x:\sigma_1 \vdash_\Sigma \; M : \sigma_2}{\Gamma \vdash_\Sigma \; \lambda^s x{:}\sigma_1.\, M : \sigma_1 \to^s \sigma_2}\ \textbf{st\_lam} \qquad \frac{\Gamma \vdash_\Sigma \; M_1 : \sigma_2 \to^s \sigma_1 \quad \Gamma \vdash_\Sigma \; M_2 : \sigma_2}{\Gamma \vdash_\Sigma \; M_1 \, M_2 : \sigma_1}\ \textbf{st\_app}$$

One–step reduction *Congruences and reductions*

$$\frac{M \xrightarrow{s}_1 M'}{\mathrm{FST}^s \, M \xrightarrow{s}_1 \mathrm{FST}^s \, M'}\ \textbf{str1\_fst} \qquad \frac{M \xrightarrow{s}_1 M'}{\mathrm{SND}^s \, M \xrightarrow{s}_1 \mathrm{SND}^s \, M'}\ \textbf{str1\_snd}$$

$$\frac{M \xrightarrow{s}_1 M'}{\langle M, N\rangle^s \xrightarrow{s}_1 \langle M', N\rangle^s}\ \textbf{str1\_pair1} \qquad \frac{N \xrightarrow{s}_1 N'}{\langle M, N\rangle^s \xrightarrow{s}_1 \langle M, N'\rangle^s}\ \textbf{str1\_pair2}$$

$$\frac{M \xrightarrow{s}_1 M'}{M \; N \xrightarrow{s}_1 M' \; N}\ \textbf{str1\_app1} \qquad \frac{N \xrightarrow{s}_1 N'}{M \; N \xrightarrow{s}_1 M \; N'}\ \textbf{str1\_iapp2}$$

$$\frac{M \xrightarrow{s}_1 M'}{\lambda^s x{:}\sigma.\, M \xrightarrow{s}_1 \lambda^s x{:}\sigma.\, M'}\ \textbf{str1\_lam2} \qquad \frac{}{(\lambda^s x{:}\sigma.\, M) \, N \xrightarrow{s}_1 [N/x]M}\ \textbf{str1\_beta}$$

$$\frac{}{\mathrm{FST}^s \, \langle M, N\rangle^s \xrightarrow{s}_1 M}\ \textbf{str1\_beta\_fst} \qquad \frac{}{\mathrm{SND}^s \, \langle M, N\rangle^s \xrightarrow{s}_1 N}\ \textbf{str1\_beta\_snd}$$

Transitive closure

$$\frac{M \xrightarrow{s}_1 M'}{M \xrightarrow{s}_1{}^+ M'}\ \textbf{st1*\_r} \qquad \frac{M \xrightarrow{s}_1{}^+ M' \quad M' \xrightarrow{s}_1{}^+ M''}{M \xrightarrow{s}_1{}^+ M''}\ \textbf{st1*\_trans}$$

Figura 5.10: Il λ-calcolo semplicemente tipato con coppie $\lambda^{\times\to}$

Oggetti:

$$\begin{aligned}
|x| &= x \\
|c| &= \pi_c \\
|\langle\rangle| &= \pi_{\langle\rangle} \\
|\langle M, N\rangle| &= \langle |M|, |N|\rangle^s \\
|\mathrm{FST}\, M| &= \mathrm{FST}^s \, |M| \\
|\mathrm{SND}\, M| &= \mathrm{SND}^s \, |M| \\
|\hat{\lambda} x{:}A.\, M| &= (\lambda^s y{:}\omega.\, \lambda^s x{:}\tau(A).\, |M|)\, |A| \\
|M \hat{} N| &= |M| \, |N| \\
|\lambda x{:}A.\, M| &= (\lambda^s y{:}\omega.\, \lambda^s x{:}\tau(A).\, |M|)\, |A| \\
|M \; N| &= |M| \, |N|
\end{aligned}$$

Tipi e kind:

$$\begin{aligned}
|a| &= \pi_a \\
|P \; N| &= |P| \, |N| \\
|\top| &= \pi_\top \\
|A \,\&\, B| &= \pi_\& \, |A| \, |B| \\
|A \multimap B| &= \pi_\multimap \, |A| \, |B| \\
|\Pi x{:}A.\, B| &= \pi_{\tau(A)} \, |A| \, (\lambda^s x{:}\tau(A).\, |B|) \\
\\
|\mathrm{TYPE}| &= \pi_{\mathrm{TYPE}} \\
|\Pi x{:}A.\, K| &= \pi_{\tau(A)} \, |A| \, (\lambda^s x{:}\tau(A).\, |K|)
\end{aligned}$$

Contesto:

$$\begin{array}{lcl} \tau(\cdot) & = & \cdot \\ \tau(\Psi, x \hat{:} A) & = & \tau(\Psi), x{:}\tau(A) \\ \tau(\Psi, x{:}A) & = & \tau(\Psi), x{:}\tau(A) \end{array}$$

Segnatura:

$$\begin{array}{lcll} \tau(\Sigma) = \pi_c & : & \tau(A), & \textit{per } c{:}A \textit{ in } \Sigma \\ \quad \pi_a & : & \tau(K), & \textit{per } a{:}K \textit{ in } \Sigma \\ \quad \pi_\sigma & : & \omega \to^s (\sigma \to^s \omega) \to^s \omega, & \\ \quad \pi_{\langle\rangle} & : & \omega, & \\ \quad \pi_\top & : & \omega, & \\ \quad \pi_\& & : & \omega \to^s \omega \to^s \omega, & \\ \quad \pi_{\multimap} & : & \omega \to^s \omega \to^s \omega, & \\ \quad \pi_{\mathrm{TYPE}} & : & \omega & \end{array}$$

Questa codifica mantiene la derivabilità nel senso che un giudizio derivabile in LLF viene sempre mappato su un giudizio derivabile in $\lambda^{\times\to}$.

Lemma 5.3.33 (*Adeguatezza della codifica*)

i. Se $\Psi \vdash_\Sigma M \Uparrow\Downarrow A$, *allora* $\tau(\Psi) \vdash_{\tau(\Sigma)} |M| : \tau(A)$;

ii. Se $\overline{\Psi} \vdash_\Sigma A \Uparrow\Downarrow K$, *allora* $\tau(\overline{\Psi}) \vdash_{\tau(\Sigma)} |A| : \tau(K)$;

iii. Se $\overline{\Psi} \vdash_\Sigma K \Uparrow Kind$, *allora* $\tau(\overline{\Psi}) \vdash_{\tau(\Sigma)} |K| : \omega$. □

La codifica proposta ha l'ulteriore proprietà di mantenere le riduzioni. Pertanto, ogniqualvolta un termine U è riducibile a U' in LLF, si ha che $|U|$ è riducibile a $|U'|$ in $\lambda^{\times\to}$ in almeno altrettanti passi.

Lemma 5.3.34 (*Mantenimento delle sequenze di riduzione*)

Se $U \longrightarrow_1 V$, *allora* $|U| \xrightarrow{s}{}^+_1 |V|$. □

Il teorema di normalizzazione forte afferma che ogni termine derivabile è fortemente normalizzabile. Questa proprietà vale in $\lambda^{\times\to}$, come dimostrato ad esempio in [Gan80, Tro86]. Usiamo questo fatto per dimostrare che è valida anche in LLF.

Teorema 5.3.35 (*Normalizzazione forte*)

Se $\Psi \vdash_\Sigma U \Uparrow\Downarrow V$, *allora* U *è fortemente normalizzabile.* □

La validità della normalizzazione forte ci permette di derivare numerose ulteriori proprietà per il nostro linguaggio. Innanzitutto un termine normalizzabile ammette un'unica forma normale:

Corollario 5.3.36 (*Unicità delle forme normali*)

Se $\Psi \vdash_\Sigma U \Uparrow\!\downarrow V$, $U \longrightarrow^* U'$ *e* $U \longrightarrow^* U''$ *con sia* U' *che* U'' *in forma normale, allora* $U' = U''$. □

Questa proprietà ci permette di definire la funzione $\mathrm{NF}(\_)$ per denotare *la* forma normale di un termine derivabile U. $\mathrm{NF}(U)$ viene calcolata a partire da U applicando β-riduzioni in ordine arbitrario fintanto che un termine in forma normale non viene prodotto. La normalizzazione forte ci garantisce che questo processo termina dopo un numero finito di passi e che il termine ottenuto è unico.

Un'ulteriore importante conseguenza del teorema di normalizzazione forte è che la teoria equazionale di LLF è decidibile.

Corollario 5.3.38 (*Decidabilità della teoria equazionale*)

Se $\Psi \vdash_\Sigma U' \Uparrow\!\downarrow V$ *e* $\Psi \vdash_\Sigma U'' \Uparrow\!\downarrow V$, *allora è possibile decidere ricorsivamente se* $U' \equiv U''$ *è derivabile.* □

Un'altra conseguenza di questo risultato è che ogni giudizio derivabile in LLF si può vincolare a menzionare solamente oggetti in forma normale.

Corollario 5.3.40 (*Forme normali*)

i. Se $\Psi \vdash_\Sigma U \Uparrow\!\downarrow V$, *allora* $\mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(U) \Uparrow\!\downarrow \mathrm{NF}(V)$;

ii. Se $\vdash_\Sigma \Psi \Uparrow Ctx$, *allora* $\vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(\Psi) \Uparrow Ctx$

iii. Se $\vdash \Sigma \Uparrow Sig$, *allora* $\vdash \mathrm{NF}(\Sigma) \Uparrow Sig$. □

Un'ultima conseguenza del teorema di normalizzazione forte è che la regola **opa_c** può essere eliminata. Come suddescritto, solamente la presenza di questa regola permette la formazione di β-redex in termini derivabili. La sua eliminazione ha l'effetto di eliminare la possibilità di derivare termini non normali senza per ciò impedire la derivabilità di alcun termine normale.

Lemma 5.3.41 (*Ammissibilità di* **opa_c**)

Siano Σ^*, Ψ^*, M^*, A^* *e* K^* *in forma normale.*

i. Se $\mathcal{P} :: \vdash \Sigma^* \Uparrow Sig$, *allora* $\mathcal{P}' :: \vdash \Sigma^* \Uparrow Sig$

ii. Se $\mathcal{P} :: \vdash_{\Sigma^*} \Psi^* \Uparrow Ctx$, *allora* $\mathcal{P}' :: \vdash_{\Sigma^*} \Psi^* \Uparrow Ctx$

iii. Se $\mathcal{P} :: \overline{\Psi^*} \vdash_{\Sigma^*} K^* \Uparrow Kind$, *allora* $\mathcal{P}' :: \overline{\Psi^*} \vdash_{\Sigma^*} K^* \Uparrow Kind$

iv. Se $\mathcal{P} :: \overline{\Psi^*} \vdash_{\Sigma^*} A^* \Uparrow\!\downarrow K$, *allora* $\mathcal{P}' :: \overline{\Psi^*} \vdash_{\Sigma^*} A^* \Uparrow\!\downarrow K$

v. Se $\mathcal{P} :: \Psi^* \vdash_{\Sigma^*} M^* \Uparrow\!\!\downarrow A$, *allora*

- *se* A *è una famiglia di tipi oppure* $M^* = \top, \langle M, N\rangle, \hat{\lambda}x{:}A.\, M$ *o* $\lambda x{:}A.\, M$, *allora* $\mathcal{P}' :: \Psi^* \vdash_{\Sigma^*} M^* \Uparrow A$
- *se* A *è una famiglia di tipi oppure* $M^* = c, x, \mathrm{FST}\, M, \mathrm{SND}\, M, M \hat{} N$ *o* $M\, N$, *allora* $\mathcal{P}' :: \Psi^* \vdash_{\Sigma^*} M^* \downarrow A$

dove $\mathcal{P}'$ *non contiene occorrenze di* **opa_c**. *Per di piú, dovunque* $\Psi' \vdash_{\Sigma^*} M' \downarrow A'$ *occorre in* $\mathcal{P}'$, M' *è una costante, una variabile, una proiezione oppure una delle due applicazioni.* □

5.3.5 Il Sistema Algoritmico

La dimostrazione della decidibilità del problema del type-checking per *LLF* è difficile da raggiungere nel sistema pre-canonico. Infatti, è difficile predire la taglia di una derivazione a causa delle regole di equivalenza e delle regole **opa_c** e **opc_a**, che possono dare luogo al ripetersi di giudizi già incontrati in precedenza. Sempre seguendo la traccia di [HHP93], eliminiamo il problema dovuto all'equivalenza definizionale grazie ad un sistema che definiamo *algoritmico*. Questo ci permetterà di legare la dimensione di almeno una derivazione per un giudizio dato ad una misura dei termini che lo compongono.

Il sistema algoritmico è presentato in Figura 5.11 esso si differenzia dal sistema pre-canonico per l'eliminazione delle regole riguardanti l'uguaglianza definizionale in favore di applicazioni puntuali della funzione di normalizzazione $\mathrm{NF}(\_)$ nei luoghi dove vengono operate sostituzioni. Usiamo questa funzione anche in tutte le regole che accedono al contesto o alla segnatura. In questa maniera, otteniamo la proprietà che il termine che appare sulla destra della freccia dei giudizi di *LLF* è sempre normale, e quindi in forma canonica.

La corrispondenza tra il sistema algoritmico e quello pre-canonico è formalizzata dai seguenti teoremi di correttezza e completezza:

Teorema 5.3.42 (*Correttezza del sistema algoritmico*)

i. Se $\Psi \vdash^a_\Sigma U \Uparrow\!\!\downarrow V$, *allora* $\Psi \vdash_\Sigma U \Uparrow\!\!\downarrow V$ *e* V *è in forma normale;*

ii. Se $\vdash^a_\Sigma \Psi \Uparrow Ctx$, *allora* $\vdash_\Sigma \Psi \Uparrow Ctx$;

iii. Se $\vdash^a \Sigma \Uparrow Sig$, *allora* $\vdash \Sigma \Uparrow Sig$. □

Teorema 5.3.43 (*Completezza del sistema algoritmico*)

i. Se $\Psi \vdash_\Sigma U \Uparrow\!\!\downarrow V$, *allora* $\Psi \vdash^a_\Sigma U \Uparrow\!\!\downarrow \mathrm{NF}(V)$;

ii. Se $\vdash_\Sigma \Psi \Uparrow Ctx$, *allora* $\vdash^a_\Sigma \Psi \Uparrow Ctx$;

Signatures

$$\frac{}{\vdash^a \cdot \Uparrow Sig}\ \textbf{sa\_dot} \qquad \frac{\vdash^a \Sigma \Uparrow Sig \quad \cdot \vdash^a_\Sigma A \Uparrow \text{TYPE}}{\vdash^a \Sigma, c : A \Uparrow Sig}\ \textbf{sa\_obj} \qquad \frac{\vdash^a \Sigma \Uparrow Sig \quad \cdot \vdash^a_\Sigma K \Uparrow Kind}{\vdash^a \Sigma, a : K \Uparrow Sig}\ \textbf{sa\_fam}$$

Contexts

$$\frac{\vdash^a \Sigma \Uparrow Sig}{\vdash^a_\Sigma \cdot \Uparrow Ctx}\ \textbf{ca\_dot} \qquad \frac{\vdash^a_\Sigma \Psi \Uparrow Ctx \quad \overline{\Psi} \vdash^a_\Sigma A \Uparrow \text{TYPE}}{\vdash^a_\Sigma \Psi, x : A \Uparrow Ctx}\ \textbf{ca\_int} \qquad \frac{\vdash^a_\Sigma \Psi \Uparrow Ctx \quad \overline{\Psi} \vdash^a_\Sigma A \Uparrow \text{TYPE}}{\vdash^a_\Sigma \Psi, x \hat{:} A \Uparrow Ctx}\ \textbf{ca\_lin}$$

Kinds

$$\frac{\vdash^a_\Sigma \overline{\Psi} \Uparrow Ctx}{\overline{\Psi} \vdash^a_\Sigma \text{TYPE} \Uparrow Kind}\ \textbf{kca\_type} \qquad \frac{\overline{\Psi}, x : A \vdash^a_\Sigma K \Uparrow Kind}{\overline{\Psi} \vdash^a_\Sigma \Pi x : A.\, K \Uparrow Kind}\ \textbf{kca\_dep}$$

Types/type families

$$\frac{\overline{\Psi} \vdash^a_\Sigma P \downarrow \text{TYPE}}{\overline{\Psi} \vdash^a_\Sigma P \Uparrow \text{TYPE}}\ \textbf{fca\_a} \qquad \frac{\vdash^a_\Sigma \overline{\Psi} \Uparrow Ctx}{\overline{\Psi} \vdash^a_\Sigma \top \Uparrow \text{TYPE}}\ \textbf{fca\_top} \qquad \frac{\overline{\Psi} \vdash^a_\Sigma A \Uparrow \text{TYPE} \quad \overline{\Psi} \vdash^a_\Sigma B \Uparrow \text{TYPE}}{\overline{\Psi} \vdash^a_\Sigma A \,\&\, B \Uparrow \text{TYPE}}\ \textbf{fca\_with}$$

$$\frac{\overline{\Psi} \vdash^a_\Sigma A \Uparrow \text{TYPE} \quad \overline{\Psi} \vdash^a_\Sigma B \Uparrow \text{TYPE}}{\overline{\Psi} \vdash^a_\Sigma A \multimap B \Uparrow \text{TYPE}}\ \textbf{fca\_limp} \qquad \frac{\overline{\Psi}, x : A \vdash^a_\Sigma B \Uparrow \text{TYPE}}{\overline{\Psi} \vdash^a_\Sigma \Pi x : A.\, B \Uparrow \text{TYPE}}\ \textbf{fca\_dep}$$

$$\frac{\vdash^a_{\Sigma, a:K, \Sigma'} \overline{\Psi} \Uparrow Ctx}{\overline{\Psi} \vdash^a_{\Sigma, a:K, \Sigma'} a \downarrow \text{NF}(K)}\ \textbf{faa\_con} \qquad \frac{\overline{\Psi} \vdash^a_\Sigma P \downarrow \Pi x : A.\, K \quad \overline{\Psi} \vdash^a_\Sigma N \Uparrow A}{\overline{\Psi} \vdash^a_\Sigma P\ N \downarrow \text{NF}([N/x]K)}\ \textbf{faa\_iapp}$$

Objects

$$\frac{\Psi \vdash^a_\Sigma M \downarrow P}{\Psi \vdash^a_\Sigma M \Uparrow P}\ \textbf{oca\_a} \qquad \frac{\vdash^a_\Sigma \Psi \Uparrow Ctx}{\Psi \vdash^a_\Sigma \langle\rangle \Uparrow \top}\ \textbf{oca\_unit} \qquad \frac{\Psi \vdash^a_\Sigma M \Uparrow A \quad \Psi \vdash^a_\Sigma N \Uparrow B}{\Psi \vdash^a_\Sigma \langle M, N \rangle \Uparrow A \,\&\, B}\ \textbf{oca\_pair}$$

$$\frac{\Psi, x \hat{:} A \vdash^a_\Sigma M \Uparrow B}{\Psi \vdash^a_\Sigma \hat{\lambda} x : A.\, M \Uparrow \text{NF}(A) \multimap B}\ \textbf{oca\_llam} \qquad \frac{\Psi, x : A \vdash^a_\Sigma M \Uparrow B}{\Psi \vdash^a_\Sigma \lambda x : A.\, M \Uparrow \Pi x : \text{NF}(A).\, B}\ \textbf{oca\_ilam}$$

$$\frac{\Psi \vdash^a_\Sigma M \Uparrow A}{\Psi \vdash^a_\Sigma M \downarrow A}\ \textbf{oaa\_c} \qquad \frac{\vdash^a_{\Sigma, c:A, \Sigma'} \overline{\Psi} \Uparrow Ctx}{\overline{\Psi} \vdash^a_{\Sigma, c:A, \Sigma'} c \downarrow \text{NF}(A)}\ \textbf{oaa\_con}$$

$$\frac{\vdash^a_\Sigma \overline{\Psi}, x \hat{:} A, \overline{\Psi'} \Uparrow Ctx}{\overline{\Psi}, x \hat{:} A, \overline{\Psi'} \vdash^a_\Sigma x \downarrow \text{NF}(A)}\ \textbf{oaa\_lvar} \qquad \frac{\vdash^a_\Sigma \overline{\Psi}, x : A, \overline{\Psi'} \Uparrow Ctx}{\overline{\Psi}, x : A, \overline{\Psi'} \vdash^a_\Sigma x \downarrow \text{NF}(A)}\ \textbf{oaa\_ivar}$$

(No rule for $\top$) $\qquad \frac{\Psi \vdash^a_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash^a_\Sigma \text{FST}\ M \downarrow A}\ \textbf{oaa\_fst} \qquad \frac{\Psi \vdash^a_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash^a_\Sigma \text{SND}\ M \downarrow B}\ \textbf{oaa\_snd}$

$$\frac{\Psi' \vdash^a_\Sigma M \downarrow A \multimap B \quad \Psi'' \vdash^a_\Sigma N \Uparrow A \quad \Psi = \Psi' \bowtie \Psi''}{\Psi \vdash^a_\Sigma M \hat{}\, N \downarrow B}\ \textbf{oaa\_lapp} \qquad \frac{\Psi \vdash^a_\Sigma M \downarrow \Pi x : A.\, B \quad \overline{\Psi} \vdash^a_\Sigma N \Uparrow A}{\Psi \vdash^a_\Sigma M\ N \downarrow \text{NF}([N/x]B)}\ \textbf{oaa\_iapp}$$

Figura 5.11: Sistema algoritmico per *LLF*

iii. Se $\vdash \Sigma \Uparrow \mathit{Sig}$, *allora* $\vdash^a \Sigma \Uparrow \mathit{Sig}$. □

Le dimostrazioni di queste cosí come di numerose proprietà viste in precedenza hanno un carattere costruttivo.

Il rigido accesso all'uguaglianza definizionale, e in particolare l'impossibilità di usarla per β-espansioni, permette una dimostrazione diretta del lemma di strengthening nel sistema algoritmico. I risultati di correttezza e completezza di cui sopra ci permettono di riportare questi risultati nel sistema pre-canonico.

Lemma 5.3.45 (*Strengthening*)

i. Se $\Psi, x \hat{::} A, \Psi' \vdash_\Sigma U \Uparrow\!\downarrow V$ *e* $x \notin \mathrm{FV}(\Psi') \cup \mathrm{FV}(U) \cup \mathrm{FV}(V)$, *allora* $\Psi, \Psi' \vdash_\Sigma U \Uparrow\!\downarrow V$;

ii. Se $\vdash_\Sigma \Psi, x \hat{::} A, \Psi' \Uparrow \mathit{Ctx}$ *e* $x \notin \mathrm{FV}(\Psi')$, *allora* $\vdash_\Sigma \Psi, \Psi' \Uparrow \mathit{Ctx}$. □

5.3.6 Decidibilità

L'assenza di equivalenze esplicite nel sistema algoritmico limita considerevolmente la scelta delle regole d'inferenza che si possono usare ad ogni passo di derivazione. In questa sezione, ci avvantaggiamo di questa proprietà per dimostrare che i problemi di type-checking e type-synthesis sono decidibili in *LLF*. Entrambi questi problemi devono venire affrontati simultaneamente nel nostro linguaggio. La dimostrazione di questi risulti dà origine ad una procedura di decisione in grado di verificare se un dato giudizio di *LLF* sia derivabile o meno. Questa procedura ha anche la capacità di calcolare un tipo o un kind per un giudizio il cui argomento piú a destra è lasciato non specificato, o di dichiarare che non esiste alcun termine che renda derivabile quel giudizio.

A questo scopo, definiamo una nozione di misura per un giudizio algoritmico. Questo numero ci permette di derivare un limite superiore alla taglia di almeno una delle sue derivazioni. Questa misura è definita per mezzo di una famiglia di *funzioni di taglia* $\|\_\|$. La taglia di termini, contesti e segnature è definita come segue:

Oggetti:

$$
\begin{aligned}
\|x\| &= 1 \\
\|c\| &= 1 \\
\|\langle\rangle\| &= 1 \\
\|\langle M, N\rangle\| &= 1 + \|M\| + \|N\| \\
\|\textsc{fst}\ M\| &= 1 + \|M\| \\
\|\textsc{snd}\ M\| &= 1 + \|M\| \\
\|\hat{\lambda} x\!:\!A.\, M\| &= 2 + \|A\| + \|M\| \\
\|M\,\hat{}\,N\| &= 1 + \|M\| + \|N\| \\
\|\lambda x\!:\!A.\, M\| &= 2 + \|A\| + \|M\| \\
\|M\ N\| &= 1 + \|M\| + \|N\|
\end{aligned}
$$

Tipi e kind:

$$
\begin{aligned}
\|a\| &= 1 \\
\|P\ N\| &= 1 + \|P\| + \|N\| \\
\|\top\| &= 1 \\
\|A \,\&\, B\| &= 1 + \|A\| + \|B\| \\
\|A \multimap B\| &= 1 + \|A\| + \|B\| \\
\|\Pi x\!:\!A.\, B\| &= 2 + \|A\| + \|B\| \\
\\
\|\textsc{type}\| &= 1 \\
\|\Pi x\!:\!A.\, K\| &= 2 + \|A\| + \|K\|
\end{aligned}
$$

Contesto:

$$
\begin{aligned}
\|\cdot\| &= 1 \\
\|\Psi, x \hat{:} A\| &= 1 + \|\Psi\| + \|A\| \\
\|\Psi, x\!:\!A\| &= 1 + \|\Psi\| + \|A\|
\end{aligned}
$$

Segnatura:

$$
\begin{aligned}
\|\cdot\| &= 1 \\
\|\Sigma, c \hat{:} A\| &= 1 + \|\Sigma\| + \|A\| \\
\|\Sigma, a\!:\!K\| &= 1 + \|\Sigma\| + \|K\|
\end{aligned}
$$

Sulla base di queste funzioni, definiamo la taglia di un giudizio:

Giudizi:

$$
\begin{aligned}
\|\vdash^a \Sigma \Uparrow \mathit{Sig}\| &= \|\Sigma\| \\
\|\vdash^a_\Sigma \Psi \Uparrow \mathit{Ctx}\| &= \|\Sigma\| + \|\Psi\| \\
\|\overline{\Psi} \vdash^a_\Sigma K \Uparrow \mathit{Kind}\| &= \|\Sigma\| + \|\Psi\| + \|K\| \\
\|\overline{\Psi} \vdash^a_\Sigma A \Uparrow \textsc{type}\| &= \|\Sigma\| + \|\Psi\| + \|A\| \\
\|\overline{\Psi} \vdash^a_\Sigma A \downarrow K\| &= \|\Sigma\| + \|\Psi\| + \|A\| \\
\|\Psi \vdash^a_\Sigma M \Uparrow A\| &= \|\Sigma\| + \|\Psi\| + \|M\| \\
\|\Psi \vdash^a_\Sigma M \downarrow A\| &= \|\Sigma\| + \|\Psi\| + \|M\| \\
\|\Psi = \Psi' \bowtie \Psi''\| &= \|\Psi\|
\end{aligned}
$$

Si noti che la taglia di un giudizio non fa mai riferimento al termine che appare sulla destra della freccia. Questo è necessario per i nostri scopi poiché la taglia di questo termine nelle premesse regole di eliminazione può in generale essere piú grande che nella conclusione.

Abbiamo progettato queste funzioni in modo che la taglia di un giudizio caratterizzi il limite superiore dell'altezza di almeno una delle sue derivazioni:

Lemma 5.3.46 (*Limite superiore sulla taglia di una derivazione*)

i. *Sia* $h = \|\Psi \vdash^a_\Sigma U \Uparrow\!\!\downarrow V\|$. *Se* $\mathcal{A} :: \Psi \vdash^a_\Sigma U \Uparrow\!\!\downarrow V$, *allora* $\mathcal{A}' :: \Psi \vdash^a_\Sigma U \Uparrow\!\!\downarrow V$,

ii. *Sia* $h = \| \vdash^a_\Sigma \Psi \Uparrow Ctx\|$. *Se* $\mathcal{A} :: \vdash^a_\Sigma \Psi \Uparrow Ctx$, *allora* $\mathcal{A}' :: \vdash^a_\Sigma \Psi \Uparrow Ctx$,

iii. *Sia* $h = \| \vdash^a \Sigma \Uparrow Sig\|$. *Se* $\mathcal{A} :: \vdash^a \Sigma \Uparrow Sig$, *allora* $\mathcal{A}' :: \vdash^a \Sigma \Uparrow Sig$,

dove $\mathcal{A}'$ *ha altezza inferiore a* $2h$ *e contiene al piú* 3^{2h} *nodi.* □

La disponibilità di una tale misura ci permette di dimostrare facilmente che il problema della derivabilità di un giudizio algoritmico dato, ossia del type-checking, è decidibile in *LLF*. Questo problema va affrontato simultaneamente con la questione della decidibilità del type-synthesis. I teoremi di correttezza e completezza ottenuti nella sezione precedente ci permettono di riportare queste proprietà al sistema pre-canonico.

Teorema 5.3.49 (*Decidibilità del type-checking*)

i. (*Type-checking*)

È possibile decidere ricorsivamente se $\Psi \vdash_\Sigma U \Uparrow\!\!\downarrow V$, $\vdash_\Sigma \Psi \Uparrow Ctx$ *e* $\vdash \Sigma \Uparrow Sig$ *sono derivabili.*

ii. (*Type-synthesis*)

Dati una segnatura Σ, *un contesto* Ψ *e un termine* U, *esiste una procedura ricorsiva che calcola un termine* V *tale che il giudizio* $\Psi \vdash_\Sigma U \Uparrow\!\!\downarrow V$ *sia derivabile, o determina che nessun tale* V *esiste.* □

La decidibilità del type-checking è una proprietà fondamentale per usare un formalismo come un linguaggio di meta-rappresentazione. Infatti, similmente a *LF*, *LLF* codifica giudizi del linguaggio oggetto come tipi e le loro derivazioni come termini di livello oggetto. La decidibilità del type-checking permette di determinare se un dato termine rappresenta una derivazione valida per un formalismo oggetto (eventualmente lineare). Come descritto nell'introduzione, questa possibilità apre le porte alla verifica formale della correttezza di programmi e alla validazione automatica di dimostrazioni matematiche.

5.4 Forme Canoniche

Il sistema algoritmico descritto nella precedente sezione presenta due svantaggi volendo usare *LLF* come linguaggio di meta-rappresentazione e linguaggio di programmazione logica. Innanzitutto, permette di derivare termini non in forma normale: codificando un formalismo di livello oggetto, siamo interessati unicamente a termini normali, anzi canonici, inoltre, derivazioni in questa forma sono molto piú facili da ricercare che derivazioni generiche, le quali si possono generare automaticamente a partire da un qualsiasi termine canonico. Inoltre, il sistema algoritmico verifica

Signatures

$$\frac{}{\vdash \cdot \Uparrow Sig}\ \textbf{sc\_dot} \qquad \frac{\vdash \Sigma \Uparrow Sig \quad \cdot \vdash_\Sigma A \Uparrow \text{TYPE}}{\vdash \Sigma, c : A \Uparrow Sig}\ \textbf{sc\_obj} \qquad \frac{\vdash \Sigma \Uparrow Sig \quad \cdot \vdash_\Sigma K \Uparrow Kind}{\vdash \Sigma, a : K \Uparrow Sig}\ \textbf{sc\_fam}$$

Contexts

$$\frac{}{\vdash_\Sigma \cdot \Uparrow Ctx}\ \textbf{cc\_dot} \qquad \frac{\vdash_\Sigma \Psi \Uparrow Ctx \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE}}{\vdash_\Sigma \Psi, x : A \Uparrow Ctx}\ \textbf{cc\_int} \qquad \frac{\vdash_\Sigma \Psi \Uparrow Ctx \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE}}{\vdash_\Sigma \Psi, x \hat{:} A \Uparrow Ctx}\ \textbf{cc\_lin}$$

Kinds

$$\frac{}{\overline{\Psi} \vdash_\Sigma \text{TYPE} \Uparrow Kind}\ \textbf{kc\_type} \qquad \frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE} \quad \overline{\Psi}, x : A \vdash_\Sigma K \Uparrow Kind}{\overline{\Psi} \vdash_\Sigma \Pi x : A.\, K \Uparrow Kind}\ \textbf{kc\_dep}$$

Types/type families

$$\frac{\overline{\Psi} \vdash_\Sigma P \downarrow \text{TYPE}}{\overline{\Psi} \vdash_\Sigma P \Uparrow \text{TYPE}}\ \textbf{fc\_a} \qquad \frac{}{\overline{\Psi} \vdash_\Sigma \top \Uparrow \text{TYPE}}\ \textbf{fc\_top} \qquad \frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE} \quad \overline{\Psi} \vdash_\Sigma B \Uparrow \text{TYPE}}{\overline{\Psi} \vdash_\Sigma A \,\&\, B \Uparrow \text{TYPE}}\ \textbf{fc\_with}$$

$$\frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE} \quad \overline{\Psi} \vdash_\Sigma B \Uparrow \text{TYPE}}{\overline{\Psi} \vdash_\Sigma A \multimap B \Uparrow \text{TYPE}}\ \textbf{fc\_limp} \qquad \frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE} \quad \overline{\Psi}, x : A \vdash_\Sigma B \Uparrow \text{TYPE}}{\overline{\Psi} \vdash_\Sigma \Pi x : A.\, B \Uparrow \text{TYPE}}\ \textbf{fc\_dep}$$

$$\frac{}{\overline{\Psi} \vdash_{\Sigma, a:K, \Sigma'} a \downarrow K}\ \textbf{fa\_con} \qquad \frac{\overline{\Psi} \vdash_\Sigma P \downarrow \Pi x : A.\, K \quad \overline{\Psi} \vdash_\Sigma N \Uparrow A}{\overline{\Psi} \vdash_\Sigma P\ N \downarrow \text{NF}([N/x]K)}\ \textbf{fa\_iapp}$$

Objects

$$\frac{\Psi \vdash_\Sigma M \downarrow P}{\Psi \vdash_\Sigma M \Uparrow P}\ \textbf{oc\_a} \qquad \frac{}{\Psi \vdash_\Sigma \langle\rangle \Uparrow \top}\ \textbf{oc\_unit} \qquad \frac{\Psi \vdash_\Sigma M \Uparrow A \quad \Psi \vdash_\Sigma N \Uparrow B}{\Psi \vdash_\Sigma \langle M, N \rangle \Uparrow A \,\&\, B}\ \textbf{oc\_pair}$$

$$\frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE} \quad \Psi, x \hat{:} A \vdash_\Sigma M \Uparrow B}{\Psi \vdash_\Sigma \hat{\lambda} x : A.\, M \Uparrow A \multimap B}\ \textbf{oc\_llam} \qquad \frac{\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE} \quad \Psi, x : A \vdash_\Sigma M \Uparrow B}{\Psi \vdash_\Sigma \lambda x : A.\, M \Uparrow \Pi x : A.\, B}\ \textbf{oc\_ilam}$$

$$\frac{}{\overline{\Psi} \vdash_{\Sigma, c:A, \Sigma'} c \downarrow A}\ \textbf{oa\_con} \qquad \frac{}{\overline{\Psi}, x \hat{:} A, \overline{\Psi'} \vdash_\Sigma x \downarrow A}\ \textbf{oa\_lvar} \qquad \frac{}{\overline{\Psi}, x : A, \overline{\Psi'} \vdash_\Sigma x \downarrow A}\ \textbf{oa\_ivar}$$

(No rule for $\top$) $\qquad$ $\dfrac{\Psi \vdash_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash_\Sigma \text{FST}\ M \downarrow A}\ \textbf{oa\_fst}$ $\qquad$ $\dfrac{\Psi \vdash_\Sigma M \downarrow A \,\&\, B}{\Psi \vdash_\Sigma \text{SND}\ M \downarrow B}\ \textbf{oa\_snd}$

$$\frac{\Psi' \vdash_\Sigma M \downarrow A \multimap B \quad \Psi'' \vdash_\Sigma N \Uparrow A \quad \Psi = \Psi' \bowtie \Psi''}{\Psi \vdash_\Sigma M \hat{} N \downarrow B}\ \textbf{oa\_lapp} \qquad \frac{\Psi \vdash_\Sigma M \downarrow \Pi x : A.\, B \quad \overline{\Psi} \vdash_\Sigma N \Uparrow A}{\Psi \vdash_\Sigma M\ N \downarrow \text{NF}([N/x]B)}\ \textbf{oa\_iapp}$$

Figura 5.12: Sistema Canonico per *LLF*

la validità di contesto e segnatura in ogni ramo di una derivazione. Questo è poco efficiente in quanto la segnatura non viene alterata e il contesto cambia in maniera prevedibile e per lo piú incrementale.

Il *sistema canonico* descritto in Figura 5.12 risolve entrambi questi problemi. La derivazione di termini rigorosamente in forma normale avviene rimuovendo la regola **oaa_c**. Si elimina la verifica di segnatura e contesto assumendo di partire da una segnatura e un contesto valido e verificando la validità di ogni assunzione successivamente inserita nel contesto. La partizione di un contesto valido risulta sempre in due contesti validi nella regola **oa_lapp**.

Questo sistema, su cui ci baseremo successivamente per fare meta-rappresentazione e proof-search, è correlato al sistema algoritmico e quindi a quello pre-canonico dai seguenti teoremi di correttezza e completezza:

Teorema 5.4.1 (*Correttezza del sistema canonico*)

i. *Se* $\vdash \Sigma \Uparrow \mathit{Sig}$, $\vdash_\Sigma \Psi \Uparrow \mathit{Ctx}$ *e* $\Psi \vdash_\Sigma U \Uparrow\!\!\downarrow V$, *allora* $\Psi \vdash^a_\Sigma U \Uparrow\!\!\downarrow V$;

ii. *Se* $\vdash \Sigma \Uparrow \mathit{Sig}$ *e* $\vdash_\Sigma \Psi \Uparrow \mathit{Ctx}$, *allora* $\vdash^a_\Sigma \Psi \Uparrow \mathit{Ctx}$;

iii. *Se* $\vdash \Sigma \Uparrow \mathit{Sig}$, *allora* $\vdash^a \Sigma \Uparrow \mathit{Sig}$.

Inoltre, Σ, Ψ, U *e* V *sono in forma normale.* □

Teorema 5.4.2 (*Completezza del sistema canonico*)

i. *Se* $\Psi \vdash^a_\Sigma M \Uparrow\!\!\downarrow A$, *allora*

- *se* A *è una famiglia di tipi, allora* $\mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(M) \Uparrow\!\!\downarrow A$,
- *se* $\mathrm{NF}(M) = \langle\rangle, \langle M', M''\rangle, \hat{\lambda}x{:}A'.\, M'$ *o* $\lambda x{:}A'.\, M'$, *allora* $\mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(M) \Uparrow A$
- *se* $\mathrm{NF}(M) = x, c, \mathrm{FST}\ M', \mathrm{SND}\ M', M'\hat{}M''$ *o* $M'\,M''$, *allora* $\mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(M) \downarrow A$

ii. *Se* $\overline{\Psi} \vdash^a_\Sigma A \Uparrow\!\!\downarrow K$, *allora* $\mathrm{NF}(\overline{\Psi}) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(A) \Uparrow\!\!\downarrow K$;

iii. *Se* $\overline{\Psi} \vdash^a_\Sigma K \Uparrow \mathit{Kind}$, *allora* $\mathrm{NF}(\overline{\Psi}) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(K) \Uparrow \mathit{Kind}$;

iv. *Se* $\vdash^a_\Sigma \Psi \Uparrow \mathit{Ctx}$, *allora* $\vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(\Psi) \Uparrow \mathit{Ctx}$;

v. *Se* $\vdash^a \Sigma \Uparrow \mathit{Sig}$, *allora* $\vdash \mathrm{NF}(\Sigma) \Uparrow \mathit{Sig}$. □

Avendo presentato un sistema canonico per LF in Sezione 3.3.1, il sistema canonico per LLF appena introdotto ci dà la possibilità di mettere in relazione questi due formalismi. Il risultato è che LLF è un'estensione conservativa di LF.

Teorema 5.4.5 (*Estensione rispetto a LF*)

i. *Se* $\Gamma \vdash^{\mathrm{LF}}_{\Sigma} U \Uparrow\!\Downarrow V$, *allora* $\Gamma \vdash_{\Sigma} U \Uparrow\!\Downarrow V$;

ii. *Se* $\vdash^{\mathrm{LF}}_{\Sigma} \Gamma \Uparrow Ctx$, *allora* $\vdash_{\Sigma} \Gamma \Uparrow Ctx$;

iii. *Se* $\vdash^{\mathrm{LF}} \Sigma \Uparrow Sig$, *allora* $\vdash \Sigma \Uparrow Sig$. □

Teorema 5.4.6 (*Conservatività rispetto a LF*)

Siano Σ, Γ, U *e* V *una segnatura, un contesto, e due termini di LF, rispettivamente, allora*

i. *Se* $\Gamma \vdash_{\Sigma} U \Uparrow\!\Downarrow V$, *allora* $\Gamma \vdash^{\mathrm{LF}}_{\Sigma} U \Uparrow\!\Downarrow V$;

ii. *Se* $\vdash_{\Sigma} \Gamma \Uparrow Ctx$, *allora* $\vdash^{\mathrm{LF}}_{\Sigma} \Gamma \Uparrow Ctx$;

iii. *Se* $\vdash \Sigma \Uparrow Sig$, *allora* $\vdash^{\mathrm{LF}} \Sigma \Uparrow Sig$.

Queste proprietà hanno conseguenze importanti. Infatti non solo ogni giudizio derivabile in *LF* è derivabile in *LLF*, ma inoltre tutte le tecniche di rappresentazione, teoremi di adeguatezza ed esempi sviluppati per *LF* rimangono validi in *LLF*. Questa eredità dota questo formalismo appena nato con una ricca libreria di applicazioni concrete. Inoltre, vedendo *LLF* come un linguaggio di programmazione logica, la conservatività rispetto ad *LF* fa sí che ogni programma *LF* sia un programma *LLF* corretto e quindi si comporti nella stessa identica maniera in un'ipotetica implementazione dl nostro linguaggio.

5.5 Programmazione logica in *LLF*

Avendo risolto con successo il problema del type-checking in *LLF*, ci accingiamo a studiare il problema del *type-inhabitance*, ossia di trovare metodi efficienti per generare un termine M tale che il giudizio $\Psi \vdash_{\Sigma} M \Uparrow\!\Downarrow A$ sia derivabile. Questo problema è in generale indecidibile. Visto sotto la luce dell'isomorfismo di Curry-Howard, questa questione è un problema di *proof-search*: Ψ e Σ formano un insieme di formule detto *programma*, A è una formula detta *goal*, e M costituisce una fedele rappresentazione di una derivazione di A a partire da Ψ e Σ. M viene chiamato *proof-term*.

La presenza di tipi dipendenti non invalida nessuno dei risultati riguardanti il proof-search ottenuti in *PLLF* [Cer96]. In particolare, come vedremo, *LLF* si può interpretare come un linguaggio di programmazione logica astratto e faremo vedere un sistema di risoluzione per questo linguaggio. La presenza di dipendenze aggiunge una nuova forma di non-determinismo al caso semplicemente tipato [Cer96]: il non-determinismo esistenziale, ossia la determinazione di dei termini con cui vanno istanziate le variabili che appaiono in un tipo atomico. Lo sviluppo di *LLF* come linguaggio di programmazione è riassunto in Figura 5.13. Il riquadro che racchiude il sistema canonico nonché tutti quelli sottostanti sta ad indicare che questo sistema deve essere sempre presente allo scopo di verificare la correttezza di tipi e kind.

La discussione prosegue nel seguente modo: innanzitutto guardiamo al sistema canonico mettendo in evidenza le sue proprietà dal punto di vista del proof-search. In Sottosezione 5.5.2,

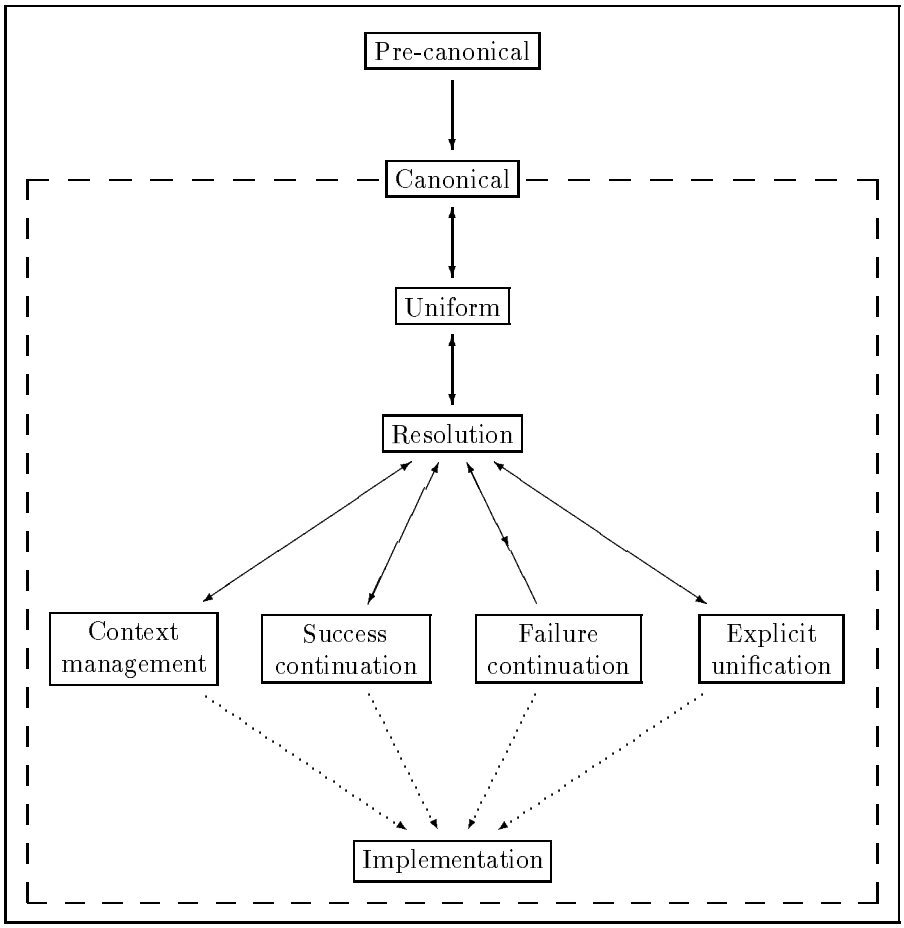


Figura 5.13: Sviluppo di Sistemi di Programmazione Logica per *LLF*

presentiamo un sistema di derivazioni uniformi per *LLF*, che raffiniamo in un calcolo di risoluzione in 5.5.3. Concludiamo con un breve cenno al non-determinismo rimanente in questo sistema.

5.5.1 Dimostrazioni canoniche

Dal punto di vista della teoria della dimostrazione, le derivazioni ottenibili con il sistema canonico presentato nella sezione precedente hanno varie proprietá interessanti.

La richiesta di operare solamente su termini in forma η-espansa corrisponde, dal punto di vista logico, a richiedere che gli assiomi di un calcolo dei sequenti operino esclusivamente su formule atomiche, o, in un calcolo di deduzione naturale, che le regole d'introduzione e di eliminazione s'incontrino solamente quando il goal è atomico. In *LLF* come in molti altri sistemi formali, ogni

giudizio derivabile ha una derivazione con queste caratteristiche. Derivazioni in forma η-espansa sono semplici da rinvenire in quanto sono *dirette dal goal* (goal-oriented): il goal viene decomposto fintanto che non dà origine ad una formula atomica, e solamente allora si sceglie una formula nel programma al fine di proseguire la computazione. Questo fatto è uno dei due requisiti per potere interpretare una logica o una teoria dei tipi come un linguaggio di programmazione logica astratta.

In un calcolo sei sequenti, le dimostrazioni corrispondenti a proof-term in forma normale sono quelle che non usano la regola di taglio. Pertanto il teorema di normalizzazione forte è il corrispettivo del teorema di eliminazione del taglio, in versione forte. Nella pratica matematica comune, ciò corrisponde ad espandere tutti i lemmi che ne verrebbero altrimenti a fare parte.

5.5.2 Dimostrabilità Uniforme

Una derivazione logica è *focalizzata* (focused) se ogni qual volta si deve dimostrare un goal atomico P, viene selezionata un'unica assunzione A nel programma e viene processata fintantoché non produce un atomo che coincide con P, eventualmente lanciando la risoluzione di sottogoal nel frattempo. Una logica in cui ogni derivazione è equivalente ad una qualche derivazione che è sia guidata dal goal che focalizzata viene detta *uniforme*. Questa logica si può allora interpretare come un linguaggio di programmazione logica astratto.

Le derivazioni ottenibili in *LLF* sono focalizzate sebbene ciò non risulti evidente dal sistema canonico. In Figura 5.14, mostriamo un sistema deduttivo equivalente alla presentazione canonica di questo linguaggio, ma in cui la formula usata per risolvere un goal atomico viene separata dal resto del programma, e decomposta fino a quando produce una formula atomica che coincide con questo goal.

Le regole di *immediate entailment* nella parte bassa di Figura 5.14 sono ottenute in maniera sistematica dalle regole di risoluzione per formule atomiche. Si noti che la regola **r_iapp** è fatta corrispondere a due regole nel sistema uniforme. In **i_iapp_static**, il costruttore di tipi dipendenti è interpretato come un quantificatore universale e A come il tipo della variabile quantificata. Viene pertanto risolto mediante istanziazione. Nel caso di **i_iapp_dynamic**, A è visto come un goal e viene risolto mediante proof-search. Questo accade quando la dipendenza è fittizia e il tipo dipendente rappresenta un'implicazione, oppure quando si vuole ordinare la risoluzione di un goal rispetto ad un altro (staging).

Il sistema di risoluzione e quello di dimostrabilità uniforme sono correlati dai seguenti teoremi di correttezza e completezza:

Teorema 5.5.1 (*Correttezza della derivabilità uniforme*)

i. *Se* $\Psi \xrightarrow{u}_\Sigma M : A$, *allora* $\Psi \vdash_\Sigma M \Uparrow A$;

ii. *Se* $\Psi' \xrightarrow{u}_\Sigma M : A \gg N : P$ *e* $\Psi'' \vdash_\Sigma M \downarrow A$ *con* $\Psi = \Psi' \bowtie \Psi''$, *allora* $\Psi \vdash_\Sigma N \downarrow P$. □

Teorema 5.5.2 (*Completezza della derivabilità uniforme*)

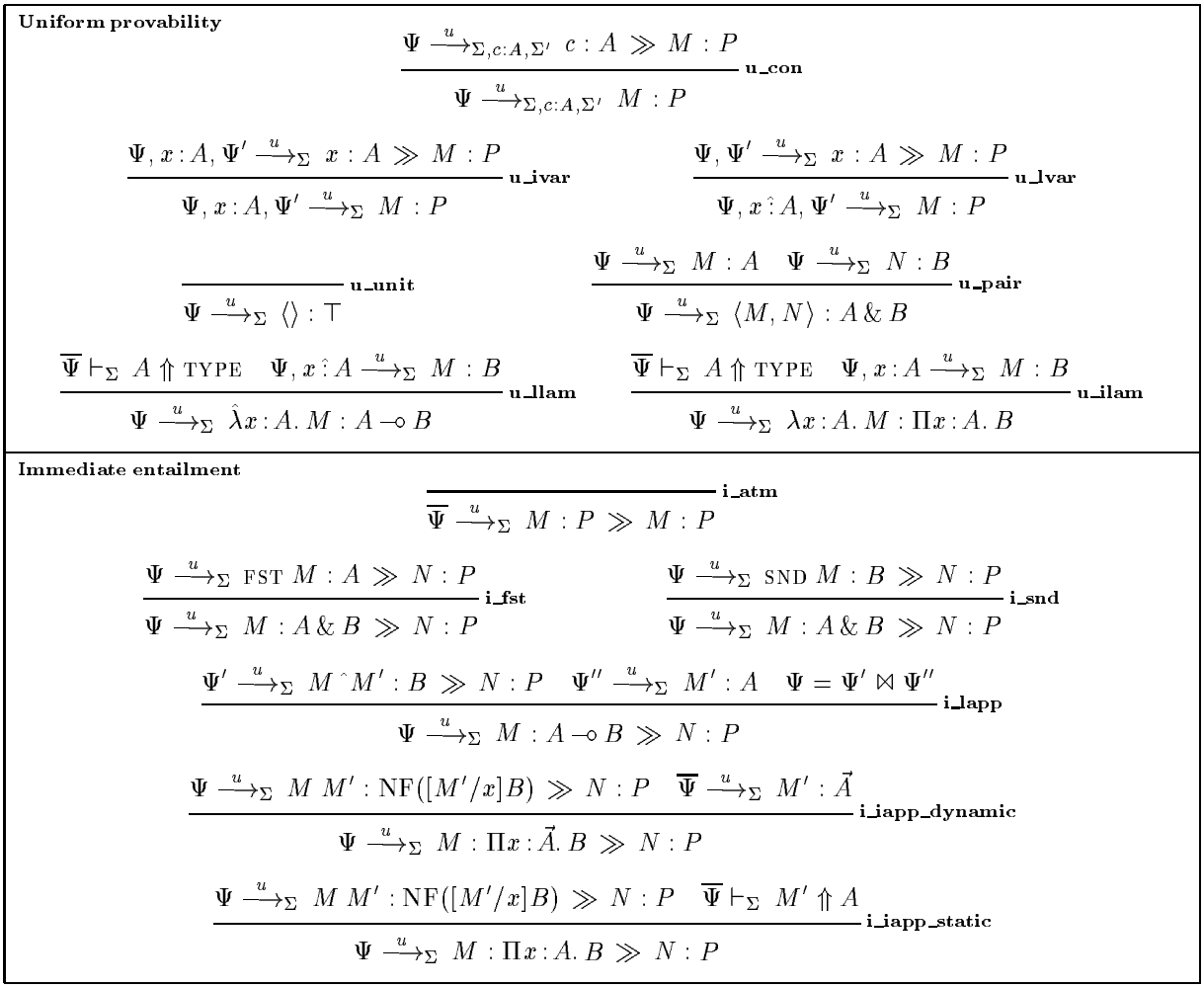


Figura 5.14: Sistema di Deduzione Uniforme per *LLF*

i. Se $\Psi \vdash_{\Sigma} M \Uparrow A$, *allora* $\Psi \xrightarrow{u}_{\Sigma} M : A$;

ii. Se $\Psi' \vdash_{\Sigma} M \downarrow A$ *e* $\Psi'' \xrightarrow{u}_{\Sigma} M : A \gg N : P$ *con* $\Psi = \Psi' \bowtie \Psi''$, *allora* $\Psi \xrightarrow{u}_{\Sigma} N : P$. □

5.5.3 Risoluzione

In un linguaggio di programmazione logica, la risoluzione di un goal atomico viene interpretata come una *chiamata a procedura*. La procedura in questione corrisponde con una o piú clausole nel programma che hanno questo atomo come testa. Il sistema di derivabilità uniforme presentato

Resolution

$$\frac{c : A \gg M : P \setminus O : G \quad \Psi \xrightarrow{r}_{\Sigma, c:A, \Sigma'} O : G}{\Psi \xrightarrow{r}_{\Sigma, c:A, \Sigma'} M : P}\ \textbf{r\_con}$$

$$\frac{x : A \gg M : P \setminus O : G \quad \Psi, x : A, \Psi' \xrightarrow{r}_{\Sigma} O : G}{\Psi, x : A, \Psi' \xrightarrow{r}_{\Sigma} M : P}\ \textbf{r\_ivar} \qquad \frac{}{\Psi \xrightarrow{r}_{\Sigma} \langle\rangle : \top}\ \textbf{r\_unit}$$

$$\frac{x : A \gg M : P \setminus O : G \quad \Psi, \Psi' \xrightarrow{r}_{\Sigma} O : G}{\Psi, x \hat{:} A, \Psi' \xrightarrow{r}_{\Sigma} M : P}\ \textbf{r\_lvar} \qquad \frac{\Psi \xrightarrow{r}_{\Sigma} M : A \quad \Psi \xrightarrow{r}_{\Sigma} N : B}{\Psi \xrightarrow{r}_{\Sigma} \langle M, N\rangle : A \,\&\, B}\ \textbf{r\_pair}$$

$$\frac{\overline{\Psi} \vdash_{\Sigma} A \Uparrow \text{TYPE} \quad \Psi, x \hat{:} A \xrightarrow{r}_{\Sigma} M : B}{\Psi \xrightarrow{r}_{\Sigma} \hat{\lambda} x : A.\, M : A \multimap B}\ \textbf{r\_llam} \qquad \frac{\overline{\Psi} \vdash_{\Sigma} A \Uparrow \text{TYPE} \quad \Psi, x : A \xrightarrow{r}_{\Sigma} M : B}{\Psi \xrightarrow{r}_{\Sigma} \lambda x : A.\, M : \Pi x : A.\, B}\ \textbf{r\_ilam}$$

$$\frac{}{\overline{\Psi} \xrightarrow{r}_{\Sigma} M ? M : P \doteq P}\ \textbf{r\_eq} \qquad \text{(No rule for } \mathbf{0}\text{)}$$

$$\frac{\Psi \xrightarrow{r}_{\Sigma} O_1 : G_1}{\Psi \xrightarrow{r}_{\Sigma} [\![O_1, O_2]\!] : G_1 \oplus G_2}\ \textbf{r\_pl1} \qquad \frac{\Psi \xrightarrow{r}_{\Sigma} O_2 : G_2}{\Psi \xrightarrow{r}_{\Sigma} [\![O_1, O_2]\!] : G_1 \oplus G_2}\ \textbf{r\_pl2}$$

$$\frac{\Psi' \xrightarrow{r}_{\Sigma} M : A \quad \Psi'' \xrightarrow{r}_{\Sigma} O : G \quad \Psi = \Psi' \bowtie \Psi''}{\Psi \xrightarrow{r}_{\Sigma} \langle\!\langle M, O\rangle\!\rangle : A \otimes G}\ \textbf{r\_times}$$

$$\frac{\overline{\Psi} \xrightarrow{r}_{\Sigma} M : \vec{A} \quad \Psi \xrightarrow{r}_{\Sigma} O : \mathrm{NF}([M/x]G)}{\Psi \xrightarrow{r}_{\Sigma} (M, O) : \Sigma x : \vec{A}.\, G}\ \textbf{r\_isum\_dynamic}$$

$$\frac{\overline{\Psi} \vdash_{\Sigma} M \Uparrow A \quad \Psi \xrightarrow{r}_{\Sigma} O : \mathrm{NF}([M/x]G)}{\Psi \xrightarrow{r}_{\Sigma} (M, O) : \Sigma x : A.\, G}\ \textbf{r\_isum\_static}$$

Formula decomposition

$$\frac{}{M : P' \gg N : P \setminus M ? N : P' \doteq P}\ \textbf{dec\_atm} \qquad \frac{}{M : \top \gg N : P \setminus \text{ABORT } M : \mathbf{0}}\ \textbf{dec\_top}$$

$$\frac{\text{FST } M : A_1 \gg N : P \setminus O_1 : G_1 \quad \text{SND } M : A_2 \gg N : P \setminus O_2 : G_2}{M : A_1 \,\&\, A_2 \gg N : P \setminus [\![O_1, O_2]\!] : G_1 \oplus G_2}\ \textbf{dec\_with}$$

$$\frac{M \hat{} M' : B \gg N : P \setminus O : G}{M : A \multimap B \gg N : P \setminus \langle\!\langle M', O\rangle\!\rangle : A \otimes G}\ \textbf{dec\_limp}$$

$$\frac{M\ M' : \mathrm{NF}([M'/x]B) \gg N : P \setminus O : \mathrm{NF}([M'/x]G)}{M : \Pi x : A.\, B \gg N : P \setminus (M', O) : \Sigma x : A.\, G}\ \textbf{dec\_idep}$$

Figura 5.15: Sistema di Risoluzione per *LLF*

nella precedente sottosezione non rende questa visione esplicita. Infatti, un goal atomico è risolto selezionando una formula nel programma e scomponendola nei suoi costituenti elementari. Vorremo invece trasformare questa clausola in un goal equivalente e ridurci a cercare una derivazione per esso.

Una trasformazione di questo genere non è direttamente possibile in LLF. Dobbiamo infatti arricchire il nostro linguaggio con costrutti ausiliari il cui scopo è precisamente quello di costruire formule equivalenti alla decomposizione di una clausola nel sistema uniforme. Queste nuove formule e gli operatori del livello oggetto corrispondenti sono specificati dalla seguente grammatica:

	(G-*formule*)		(G-*termini*)	
(P)	G	$::= P_1 \doteq P_2$	O	$::= M_1?M_2$
$(\top)$		$\mid \mathbf{0}$		$\mid \text{ABORT}\ M$
$(\&)$		$\mid G_1 \oplus G_2$		$\mid [\![O_1, O_2]\!]$
$(\multimap)$		$\mid A \otimes G$		$\mid \langle\!\langle M, O \rangle\!\rangle$
(Π)		$\mid \Sigma x{:}A.\, G$		$\mid (M, O)$

L'unica novità rispetto a quanto visto nel capitolo precedente è data dall'operatore Σ, che viene interpretato di volta in volta come un quantificatore esistenziale, come la congiunzione intuizionistica a sinistra $\overset{!}{\otimes}$ o come un operatore di staging.

Il *sistema di risoluzione* viene mostrato in Figura 5.15. La trasformazione di una clausola A in una G-formula G per risolvere un goal atomico P viene gestita dal giudizio di *decomposizione* $M : A \gg N : P \setminus O : G$, il quale si prende carico anche di trasformare il proof-term O risultante dalla risoluzione di G in un proof-term equivalente N per dimostrare P grazie ad A nel sistema uniforme. M viene usato come un'accumulatore.

Ammettendo le G-formule nel linguaggio dei goal di LLF (ma non nel linguaggio dei programmi), otteniamo un linguaggio in qualche modo equivalente alla logica delle formule di Harrop ereditarie lineari su cui è basato il linguaggio di programmazione logica *Lolli*. Il risultante linguaggio nell'ambito della nostra teoria dei tipi è dato dalla seguente grammatica:

D-formule:	$D ::=$	P	$\mid \top$	$\mid D_1 \,\&\, D_2$	$\mid G \multimap D$	$\mid \Pi x{:}\vec{G}.\, D$	$\mid \Pi x{:}A.\, D$	
G-formule:	$G ::=$	P	$\mid \top$	$\mid G_1 \,\&\, G_2$	$\mid D \multimap G$	$\mid \Pi x{:}\vec{D}.\, G$	$\mid \Pi x{:}A.\, G$	
		$\mid P_1 \doteq P_2$	$\mid \mathbf{0}$	$\mid G_1 \oplus G_2$	$\mid G_1 \otimes G_2$	$\mid \Sigma x{:}\vec{D}.\, G$	$\mid \Sigma x{:}A.\, G$	

Il sistema di risoluzione corrisponde strettamente al sistema uniforme e transitivamente a quello canonico. Si hanno infatti i seguenti teoremi di correttezza e completezza.

Teorema 5.5.4 (*Correttezza del sistema di risoluzione*)

i. Se $\Psi \xrightarrow{r}_{\Sigma} M : A$, *allora* $\Psi \xrightarrow{u}_{\Sigma} M : A$;

ii. Se $M : A \gg N : P \setminus O : G$ *e* $\Psi \xrightarrow{r}_{\Sigma} O : G$, *allora* $\Psi \xrightarrow{u}_{\Sigma} M : A \gg N : P$. □

Teorema 5.5.5 (*Completezza del sistema di risoluzione*)

i. Se $\Psi \xrightarrow{u}_{\Sigma} M : A$, *allora* $\Psi \xrightarrow{r}_{\Sigma} M : A$;

ii. Se $\Psi \xrightarrow{u}_{\Sigma} M : A \gg N : P$, *allora* $M : A \gg N : P \setminus O : G$ *e* $\Psi \xrightarrow{r}_{\Sigma} O : G$. □

5.5.4 Il Non-determinismo

Il sistema di risoluzione presentato nella sottosezione precedente ha qualità operazionali ben piú forti del sistema canonico da cui siamo partiti. Tuttavia lascia quattro punti di scelta aperti all'implementatore. Dal punto di vista logico, corrispondono a quattro sorgenti di non-determinismo presenti nella nostra specifica:

Non-determinismo legato alla distribuzione delle risorse

Il contesto Ψ presente nella conclusione della regola **r_times** deve essere partizionato in Ψ' e Ψ'' per risolvere i sottogoal A e G di una congiunzione moltiplicativa. L'effetto del giudizio di partizione del contesto è quello di distribuire le assunzioni lineari presenti in Ψ fra Ψ' e Ψ''. Vi sono in generale molti modi di partizionare il contesto, ma soli pochi permettono di risolvere A e G. La selezione non è guidata da alcun criterio in questa regola e pertanto una partizione corretta deve essere indovinata per dimostrare questi sottogoal.

Non-determinismo congiuntivo

L'ordine in cui i sottogoal si devono risolvere nelle regole binarie **r_pair**, **r_times** e **r_isum_dynamic** è lasciato non-specificato. Possono infatti essere dimostrati in parallelo, ma una implementazione sequenziale richiede di risolverli uno dopo l'altro.

Non-determinismo disgiuntivo

Similmente, non viene dato alcun criterio per decidere l'ordine in cui le assunzioni nel programma devono venire selezionate per fare risoluzione (regole **r_con**, **r_ivar** e **r_lvar**), né quale sottogoal di una formula disgiuntiva va tentato per primo (regole **r_plus1** e **r_plus2**). Un'implementazione parallela li può provare in simultaneamente, ma un motore inferenziale sequenziale li deve ordinare.

Non-determinismo esistenziale

Non viene data nessuna strategia per calcolare il termine di livello oggetto usato per istanziare una variabile esistenziale in regola **r_isum_static**. Si assume invece che questo termine venga fornito esternamente, mentre il sistema si limita a verificare la sua validità.

Abbiamo incontrato le prime tre forme di non-determinismo nel capitolo precedente. Le soluzioni proposte allora si applicano anche nel contesto dipendente di *LLF*. Infatti, anche qui le risorse lineari vengono distribuite in maniera lazy dando tutto il contesto ad uno dei sottogoal, lasciandolo consumare le risorse di cui ha bisogno e passando il resto all'altra sottoformula

[CHP96]. Per risolvere il non-determinismo congiuntivo, bisogna predisporre azioni per ripristinare il contesto in modo da potere sequenzializzare la risoluzione dei due sottogoal delle regole binarie. Infine il non-determinismo disgiuntivo va risolto con la tecnica del backtracking.

Il non-determinismo esistenziale richiede invece di adattare i classici metodi di *unificazione* [Her71, Rob65, MM77], specie quelli che prendono in considerazioni termini di ordine superiore e teorie dei tipi [Hue76, Sny91, Ell90, Dug93]. Come già avviene nell'implementazione corrente di *Elf*, è possibile e conveniente vedere l'unificazione di ordine superiore in LLF come un problema di risoluzione di vincoli [JL87, Pfe91].

Capitolo 6

La Metodologia della Meta-rappresentazione Lineare

In questo capitolo, facciamo vedere come *LLF* possa venire usato come un linguaggio di meta-rappresentazione per codificare problemi dotati di stato, basati su risorse o che comunque si rifanno ad un contesto di tipo volatile. Presentiamo tre esempi di meta-rappresentazione presi da aree diverse: sono caratterizzati dal fatto che non sono codificabili in maniera effettiva in *LF*, ma si possono facilmente modellare in *LLF*. Ognuno mette in luce aspetti diversi del nostro linguaggio. Innanzitutto, riprendiamo in Sezione 6.3 il linguaggio *Mini-ML* descritto nel Capitolo 4 inserendovi una nozione di memoria e istruzioni imperative quali l'assegnamento. Ci muoviamo poi sul campo logico in Sezione 6.4 facendo vedere come viene codificata una dimostrazione di eliminazione del taglio per un frammento della logica lineare. Infine, facciamo vedere come si possa codificare in *LLF* un popolare gioco, *Mahjongg*, in Sezione 6.5. Iniziamo tuttavia con una discussione generale della metodologia di meta-rappresentazione di cui si avvale *LLF*. In Sezione 6.2 presentiamo un'estensione conservativa della sintassi concreta di *LF* disponibile in *Elf* introducendovi notazioni per i costrutti propri di *LLF*. Useremo infatti questa sintassi e convenzioni simili a quelle disponibili in *Elf* per alleggerire la notazione.

6.1 Meta-rappresentazione in *LLF*

La differenza principale tra *LF* ed *LLF* è che quest'ultimo linguaggio ammette assunzioni lineari nel contesto Ψ di un giudizio $\Psi \vdash_\Sigma M \Uparrow A$. La porzione lineare del contesto di *LLF* offre la possibilità di modellare il contesto volatile di quei sistemi deduttivi che necessitano un accesso non-monotonico al loro contesto. Gli operatori lineari del nostro linguaggio permettono di manipolare in maniera efficiente e flessibile queste assunzioni e di rappresentare derivazioni lineari.

Mostriamo in Figura 6.1 la rappresentazione in *LLF* di regole d'inferenza e derivazioni per un sistema deduttivo che includa il giudizio $\Gamma; \Delta \longrightarrow C$ dove Δ è la porzione volatile del contesto, mentre Γ ne è la parte permanente. Questa figura riprende la descrizione astratta

Issue	Template	Declaration	Derivation
Judgment	$\Gamma;\Delta \longrightarrow C$	$\mathrm{J} : \tau \to \mathrm{TYPE}$	$\mathrm{J}\ t_C$
Rule with no premisses	$\dfrac{}{\Gamma;\Delta \longrightarrow C}\,\mathbf{r}$	$\mathrm{R} : \Pi X_1{:}\tau_1. \ldots \Pi X_n{:}\tau_n.$ $\top \multimap \mathrm{J}\ t_C$	$\mathrm{R}\ t_1 \ \ldots\ t_n \mathbin{\hat{}} \langle\rangle$
Rule with one premiss	$\dfrac{\Gamma;\Delta \longrightarrow P}{\Gamma;\Delta \longrightarrow C}\,\mathbf{r}$	$\mathrm{R} : \Pi X_1{:}\tau_1. \ldots \Pi X_n{:}\tau_n.$ $\mathrm{J}\ t_P \multimap \mathrm{J}\ t_C$	$\mathrm{R}\ t_1 \ \ldots\ t_n \mathbin{\hat{}} \mathcal{D}_P$
Rule with multiple premisses	$\dfrac{\Gamma;\Delta \longrightarrow P_1 \quad \Gamma;\Delta \longrightarrow P_2}{\Gamma;\Delta \longrightarrow C}\,\mathbf{r}$	$\mathrm{R} : \Pi X_1{:}\tau_1. \ldots \Pi X_n{:}\tau_n.$ $\mathrm{J}\ t_{P_1} \,\&\, \mathrm{J}\ t_{P_2} \multimap \mathrm{J}\ t_C$	$\mathrm{R}\ t_1 \ \ldots\ t_n \mathbin{\hat{}} \langle \mathcal{D}_{P_1}, \mathcal{D}_{P_2} \rangle$
Access to the context	$\dfrac{\Gamma;\Delta \longrightarrow P}{\Gamma, A;\Delta \longrightarrow C}\,\mathbf{r}$	$\mathrm{R} : \Pi X_1{:}\tau_1. \ldots \Pi X_n{:}\tau_n.$ $\mathrm{J}\ t_P \to \mathrm{J}\ t_A \multimap \mathrm{J}\ t_C$	$\mathrm{R}\ t_1 \ \ldots\ t_n\ \mathcal{D}_P \mathbin{\hat{}} \mathcal{D}_A$
Parametric rules	$\dfrac{\Gamma;\Delta \longrightarrow P^c}{\Gamma;\Delta \longrightarrow C}\,\mathbf{r}\,(c)$	$\mathrm{R} : \Pi X_1{:}\tau_1. \ldots \Pi X_n{:}\tau_n.$ $(\Pi c{:}\tau_c.\, \mathrm{J}\ t_P) \multimap \mathrm{J}\ t_C$	$\mathrm{R}\ t_1 \ \ldots\ t_n \mathbin{\hat{}} (\lambda c{:}\tau_c.\, \mathcal{D}_P)$
Hypothetical rules	$\dfrac{\Gamma, A;\Delta \longrightarrow P}{\Gamma;\Delta \longrightarrow C}\,\mathbf{r}$	$\mathrm{R} : \Pi X_1{:}\tau_1. \ldots \Pi X_n{:}\tau_n.$ $(\mathrm{J}\ t_A \to \mathrm{J}\ t_P) \multimap \mathrm{J}\ t_C$	$\mathrm{R}\ t_1 \ \ldots\ t_n \mathbin{\hat{}} (\lambda x{:}\mathrm{J}\ t_A.\, \mathcal{D}_P)$
Constraints	$\dfrac{\Gamma;\cdot \longrightarrow P}{\Gamma;\cdot \longrightarrow C}\,\mathbf{r}$	$\mathrm{R} : \Pi X_1{:}\tau_1. \ldots \Pi X_n{:}\tau_n.$ $\mathrm{J}\ t_P \to \mathrm{J}\ t_C$	$\mathrm{R}\ t_1 \ \ldots\ t_n\ \mathcal{D}_P$
Context splitting	$\dfrac{\Gamma;\Delta_1 \longrightarrow P_1 \quad \Gamma;\Delta_2 \longrightarrow P_2}{\Gamma;\Delta_1,\Delta_2 \longrightarrow C}\,\mathbf{r}$	$\mathrm{R} : \Pi X_1{:}\tau_1. \ldots \Pi X_n{:}\tau_n.$ $\mathrm{J}\ t_{P_1} \multimap \mathrm{J}\ t_{P_2} \multimap \mathrm{J}\ t_C$	$\mathrm{R}\ t_1 \ \ldots\ t_n \mathbin{\hat{}} \mathcal{D}_{P_1} \mathbin{\hat{}} \mathcal{D}_{P_2}$
Augmentation	$\dfrac{\Gamma;\Delta, A \longrightarrow P}{\Gamma;\Delta \longrightarrow C}\,\mathbf{r}$	$\mathrm{R} : \Pi X_1{:}\tau_1. \ldots \Pi X_n{:}\tau_n.$ $(\mathrm{J}\ t_A \multimap \mathrm{J}\ t_P) \multimap \mathrm{J}\ t_C$	$\mathrm{R}\ t_1 \ \ldots\ t_n \mathbin{\hat{}} (\hat{\lambda} x{:}\mathrm{J}\ t_A.\, \mathcal{D}_P)$
Remotion	$\dfrac{\Gamma;\Delta \longrightarrow P}{\Gamma;\Delta, A \longrightarrow C}\,\mathbf{r}$	$\mathrm{R} : \Pi X_1{:}\tau_1. \ldots \Pi X_n{:}\tau_n.$ $\mathrm{J}\ t_P \multimap \mathrm{J}\ t_A \multimap \mathrm{J}\ t_C$	$\mathrm{R}\ t_1 \ \ldots\ t_n \mathbin{\hat{}} \mathcal{D}_P \mathbin{\hat{}} \mathcal{D}_A$
Modification	$\dfrac{\Gamma;\Delta, B \longrightarrow P}{\Gamma;\Delta, A \longrightarrow C}\,\mathbf{r}$	$\mathrm{R} : \Pi X_1{:}\tau_1. \ldots \Pi X_n{:}\tau_n.$ $(\mathrm{J}\ t_B \multimap \mathrm{J}\ t_P) \multimap \mathrm{J}\ t_A \multimap \mathrm{J}\ t_C$	$\mathrm{R}\ t_1 \ \ldots\ t_n \mathbin{\hat{}}$ $(\hat{\lambda} x{:}\mathrm{J}\ t_B.\, \mathcal{D}_P) \mathbin{\hat{}} \mathcal{D}_A$

Figura 6.1: Metodologia di Rappresentazione in *LLF*

presentata in Figura 4.2 ed estende il trattamento mostrato in Figura 4.3 nel caso in cui il meta-linguaggio fosse *LF* alla gestione di assunzioni volatili. Si noti l'uso dell'implicazione lineare in luogo dell'implicazione intuizionistica per la rappresentazione di regole generiche. Differentemente da quest'ultima, essa permette il libero flusso delle assunzioni lineari tra premesse e conclusioni delle regole. Nel caso in cui il sistema deduttivo in esame non si avvalga di una porzione lineare, entrambe le implicazioni hanno lo stesso effetto. Questo è importante volendo eseguire in *LLF* gli esempi accumulati per *LF*.

Il concetto di meta-rappresentazione e di adeguatezza estende in maniera naturale gli analoghi concetti presenti nel caso di *LF*. Infatti, una *meta-rappresentazione* in *LLF* di un insieme di giudizi J di un formalismo oggetto $\mathcal{L}_o$ è una funzione $\ulcorner \_ \urcorner$ che associa ogni giudizio schematico in J ad un tipo di base e ogni derivazione per questi giudizi ad un oggetto canonico di questo tipo. Giudizi sintattici vengono invece gestiti mediante sintassi astratta di ordine superiore, possibilmente lineare se il linguaggio oggetto include operatori sintattici che legano variabili in maniera lineare.

I teoremi di adeguatezza per una rappresentazione in *LLF* vengono enunciati e dimostrati in maniera identica a quanto mostrato nel caso di *LF* nel Capitolo 4. Come già accade in *LF*, bisogna fare vedere che il contesto di *LLF* evolve nella stessa maniera del contesto del giudizio oggetto che si sta modellando, questo si applica in particolare al contesto lineare. Una piccola sorgente di complicazione emerge nella dimostrazione della correttezza della rappresentazione in quanto i termini disponibili in *LLF* hanno una struttura piú complicata di quanto rinvenibile in *LF*.

6.2 Sintassi Concreta di *LLF*

In questa sezione facciamo vedere come estendere la sintassi concreta di *LF* su cui si basa *Elf* per includere gli operatori di *LLF*. Nel fare ciò, siamo stati guidati da due obiettivi: innanzitutto, un programma *Elf* non deve venire alterato sintatticamente volendolo eseguire in un'eventuale implementazione di *LLF*. Detto in altro modo, l'estensione che proponiamo deve essere conservativa rispetto a *Elf*. Inoltre, vogliamo evitare il proliferare di operatori, mantenendone il numero il piú limitato possibile, in modo da agevolare ulteriori estensioni quando risulteranno necessarie.

La seguente tabella associa ogni operatore di *LLF* alla sua rappresentazione concreta. Come già accade in *Elf*, alcuni operatori possono essere scritti in maniera diversa. In particolare, usiamo la freccia per modellare un costruttore di tipi dipendenti nel cui corpo non compare la variabile legata. Inoltre, offriamo la possibilità di scrivere le implicazioni con gli argomenti rovesciati invertendo il senso della freccia.

	Sintassi astratta	*Sintassi concreta*		
Kind	TYPE	`type`		
	$\Pi x{:}A.\,K$	$\{x{:}A\}K$	A `->` K	K `<-` A
Tipi	$P\ M$	$P\ \ M$		
	$\top$	`<T>`		
	$A \,\&\, B$	A `&` B		
	$A \multimap B$	A `-o` B	B `o-` A	
	$\Pi x{:}A.\,B$	$\{x{:}A\}B$	A `->` B	B `<-` A
Oggetti	$\langle\rangle$	`()`		
	$\langle M, N\rangle$	M`,`N		
	FST M	`<fst>` M		
	snd M	`<snd>` M		
	$\hat{\lambda} x{:}A.\,M$	`[`x`^`A`]`M		
	$M \hat{}\, N$	M `^` N		
	$\lambda x{:}A.\,M$	`[`x`:`A`]`M		
	$M\ N$	$M\ \ N$		

La seguente tabella descrive precedenza e associatività degli operatori introdotti. Ovviamente, si possono usare le parentesi per cambiare la precedenza relativa i questi costrutti. Si noti che non è piú valida la proprietà di *LF* secondo la quale il raggio d'azione dei costrutti che legano variabili si estende fino alla fine della dichiarazione corrente, o della prima parentesi chiusa.

Precedenza	*Operatore*	*Posizione*
piú elevata	`_ _` `_ ^ _`	associativo a sinistra
	`_ & _`	associativo a destra
	`_ -o _` `_ -> _`	associativo a destra
	`_ o- _` `_ <- _`	associativo a sinistra
	`{_:_}_` `[_:_]_` `[_^_]_` `<fst> _` `<snd> _`	prefisso a sinistra
piú bassa	`_,_`	associativo a destra

Un programma *Elf* rappresenta una segnatura di *LF*. In *LLF*, conviene alterare questa convenzione permettendo di specificare nel programma assunzioni lineari: infatti qualora il formalismo da rappresentare si avvalga di numerose assunzioni lineari, è piú comodo scriverlo nel programma che avere una clausola che le assuma nel contesto durante l'esecuzione. A questo scopo, permettiamo dichiarazioni della forma

`<linear>` c: A.

dove la parola chiave `<linear>` sta ad indicare che la dichiarazione che segue va inserita nel contesto come assunzione lineare, e non nella segnatura.

Come già accade in *LF*, usiamo **%** per indicare un commento che si estende fino alla fine della riga corrente.

Adottiamo le stesse convenzioni di *Elf* per semplificare la scrittura di dichiarazioni in *LLF*. Il tipo delle variabili legate dagli operatori di tipo dipendente e λ-astrazione può essere lasciato implicito ogni volta che è possibile ricostruirlo sulla base delle dichiarazioni circostanti. Come in *LF*, il problema della ricostruzione dei tipi è in generale indecidibile, ma ciò è raramente un problema nelle applicazioni concrete. Qualora manteniamo implicito il tipo `A` di una variabile `x` in espressioni di questo tipo, scriviamo `{x}B`, `[x]B` e `[x^]B` in luogo di `{x:A}B`, `[x:A]B` e `[x^A]B` rispettivamente.

Un'altra utile convenzione è quella di usare identificatori che iniziano con una lettera maiuscola per indicare variabili quantificate da un costruttore di tipo dipendente qualora quest'operatore racchiuda l'intera dichiarazione. In questo modo possiamo omettere completamente questi operatori.

Una novità riguarda la possibilità di usare la giustapposizione per rappresentare anche le applicazioni lineari quando sia possibile ricostruire la natura esatta di questo operatore in base alle dichiarazioni circostanti. Qualora questa ricostruzione non sia possibile, dobbiamo usare esplicitamente l'operatore `^`.

6.3 *Mini-ML* Imperativo

Come nostro primo esempio, riprendiamo il linguaggio *Mini-ML* presentato nel Capitolo 4 estendendolo con una memoria e istruzioni imperative per accedervi e modificarla. Costrutti di questo tipo non possono venire modellati in maniera effettiva in *LF* in quanto non è possibile rappresentare in maniera efficiente la memoria. Le assunzioni lineari di *LLF* si possono usare per codificare le celle di memoria e gli operatori lineari del nostro linguaggio offrono strumenti per modellare le operazioni che vi operano.

Definiamo la sintassi e la semantica del linguaggio, che chiameremo *MLR*, in Sottosezione 6.3.1. Rappresentiamo questi costrutti in *LLF* nella successiva Sottosezione 6.3.2. Infine, proponiamo il teorema di preservazione dei tipi per *MLR* e la sua rappresentazione in *LLF* in Sottosezione 6.3.3.

6.3.1 *Mini-ML* con Referenze

L'estensione imperativa di *Mini-ML* richiede vari arricchimenti alla sintassi di questo linguaggio. Innanzitutto, dobbiamo introdurre una nozione di *memoria*, costituita da celle che possono assumere un valore. Definiamo una nuova categoria sintattica per modellare questa nozione. Ci serve inoltre la nuova categoria lessicale delle *celle di memoria* come costrutto intrinseco del linguaggio.

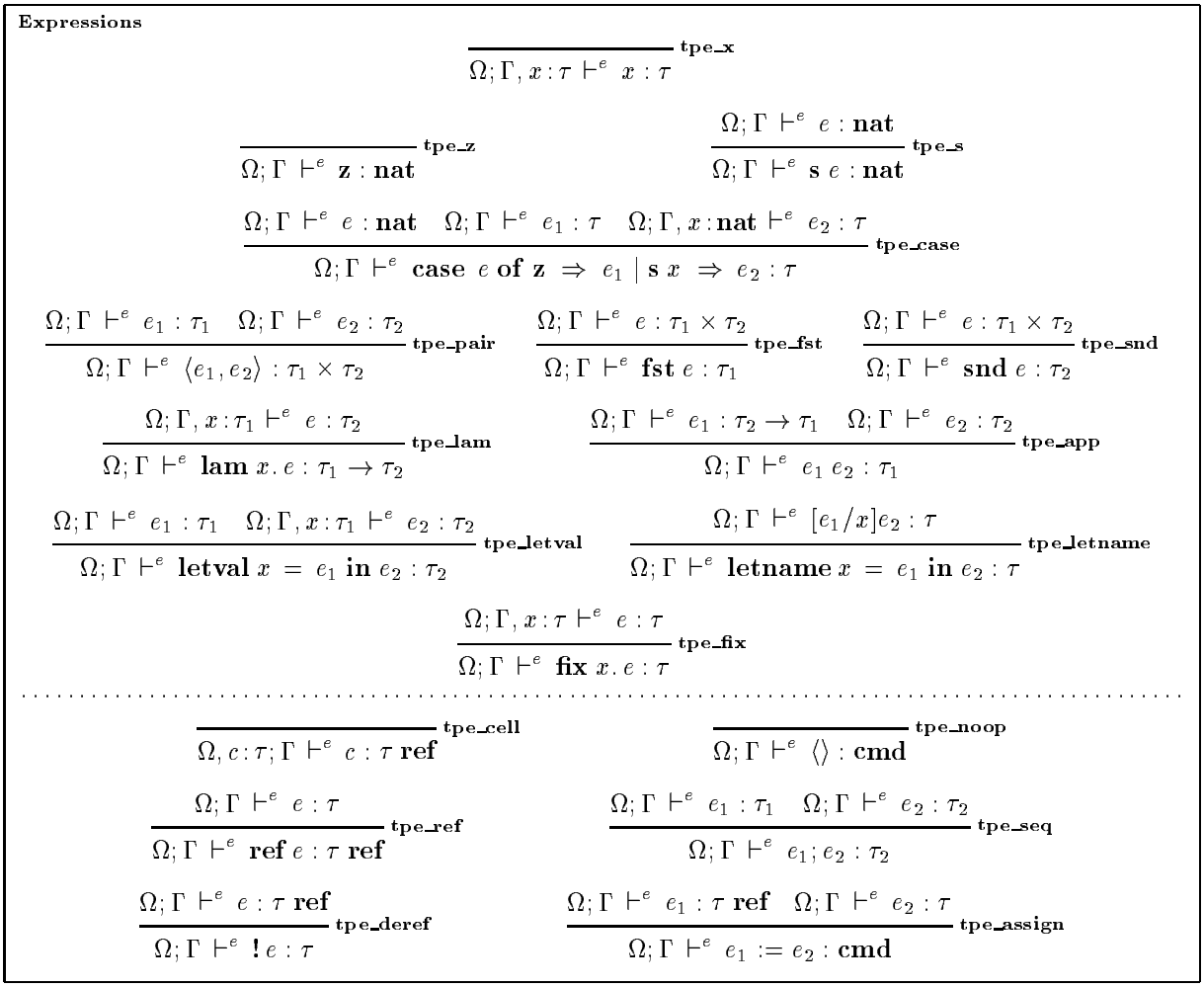


Figura 6.2: Regole di Tipizzazione per *MLR*, Espressioni

Il linguaggio delle espressioni si arricchisce con sei nuovi costrutti: oltre alle celle di memoria, abbiamo un'operazione di *referenziazione* che serve a definire puntatori, un'operazione di *dereferenziazione* e un'operazione di *assegnamento*. Aggiungiamo inoltre il costrutto di *operazione nulla*, corrispondente al valore di un assegnamento e la possibilità di sequenzializzare espressioni in modo da concatenare l'effetto di piú istruzioni imperative.

Il linguaggio dei tipi viene accresciuto con due nuovi costrutti: un tipo di base corrispondente ai *comandi* del linguaggio (operazione nulla e assegnamento) e, per ogni tipo, il tipo referenza, ossia puntatore ad un valore di quel tipo. I nuovi costrutti del linguaggio sono mostrati nella grammatica sottostante, separandoli dagli operatori originari di *Mini-ML* con una doppia sbarra

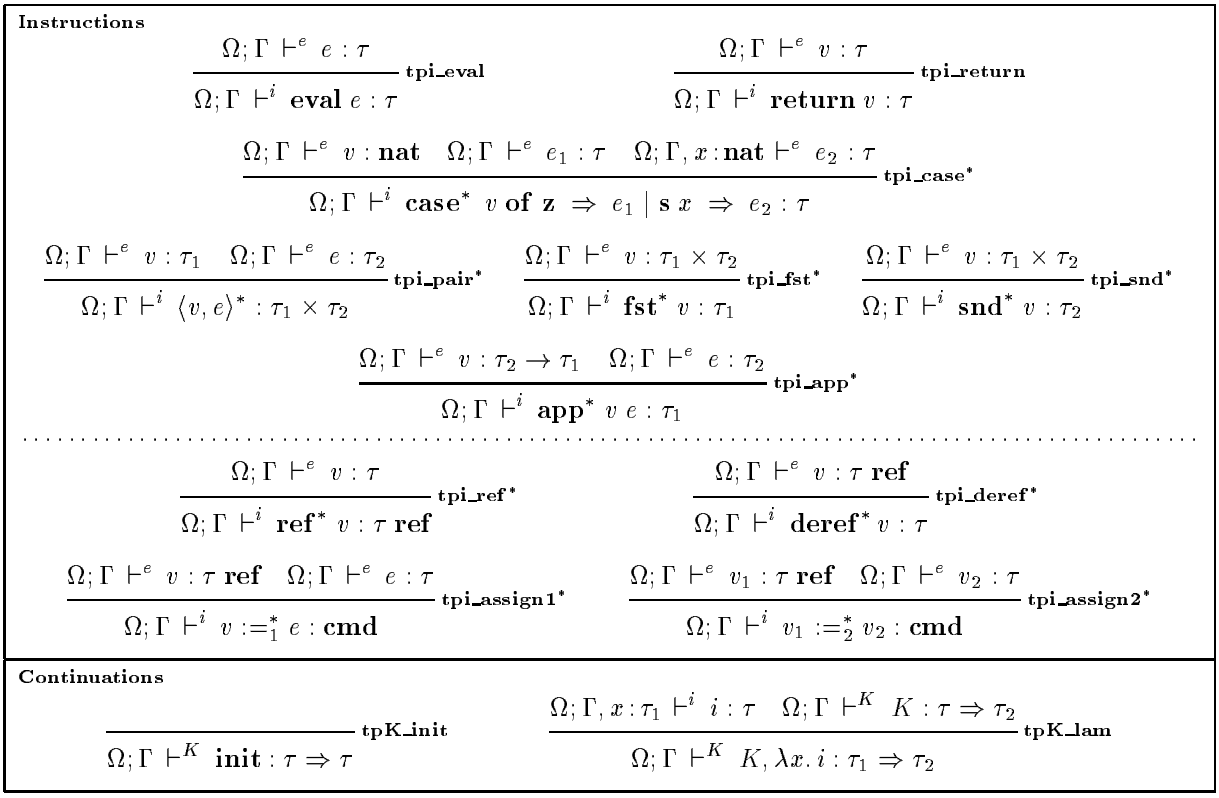


Figura 6.3: Regole di Tipizzazione per *MLR*, Istruzioni e Continuazioni

($\|$).

Espressioni:	e	$::= x$	(*Variabili*)
		$\mid \mathbf{z} \mid \mathbf{s}\ e \mid \mathbf{case}\ e\ \mathbf{of}\ \mathbf{z} \Rightarrow e_1 \mid \mathbf{s}\ x \Rightarrow e_2$	(*Numeri naturali*)
		$\mid \langle e_1, e_2\rangle \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e$	(*Coppie*)
		$\mid \mathbf{lam}\ x.\, e \mid e_1\ e_2$	(*Funzioni*)
		$\mid \mathbf{letval}\ x = e_1\ \mathbf{in}\ e_2 \mid \mathbf{letname}\ x = e_1\ \mathbf{in}\ e_2$	(*Definizioni*)
		$\mid \mathbf{fix}\ x.\, e$	(*Ricorsione*)
		$\| \ c \mid \mathbf{ref}\ e \mid\ !\, e$	(*Referenze*)
		$\mid \langle\rangle \mid e_1 := e_2 \mid e_1; e_2$	(*Comandi*)
Tipi:	τ	$::= \mathbf{nat} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2$	
		$\| \ \tau\ \mathbf{ref} \mid \mathbf{cmd}$	
Memoria:	S	$::= \cdot \mid S, c = v$	

La semantica statica di *MLR* estende in maniera naturale il sistema di tipizzazione che abbiamo visto per *Mini-ML*. La possibilità di menzionare celle di memoria nelle espressioni richiede la

Store

$$\frac{}{\Omega \vdash^S \cdot : \cdot}\ \textbf{tpS\_empty} \qquad \frac{\Omega \vdash^S S : \Omega' \quad \Omega; \cdot \vdash^e v : \tau}{\Omega \vdash^S (S, c = v) : (\Omega', c{:}\tau)}\ \textbf{tpS\_cell}$$

Answers

$$\frac{\Omega \vdash^S S : \Omega \quad \Omega; \cdot \vdash^e v : \tau}{\Omega \vdash^a (S, v) : \tau}\ \textbf{tpa\_val} \qquad \frac{\Omega, c{:}\tau' \vdash^a a : \tau}{\Omega \vdash^a \textbf{new}\ c.\, a : \tau}\ \textbf{tpa\_new}$$

Figura 6.4: Regole di Tipizzazione per *MLR*: Memoria e Risposte

presenza di un nuovo contesto, il *contesto di memoria*, che assegna un tipo ad ogni cella. Abbiamo la seguente grammatica:

$$\begin{array}{rrcl} \textit{Contesti:} & \Gamma & ::= & \cdot \mid \Gamma, x{:}\tau \\ \textit{Contesti di memoria:} & \Omega & ::= & \cdot \mid \Omega, c{:}\tau \end{array}$$

Le regole di tipizzazione per le espressioni di *MLR* sono mostrate in Figura 6.2. I casi relativi ai costrutti nuovamente introdotti sono evidenziati nella parte bassa della figura.

La semantica dinamica del linguaggio arricchisce notevolmente la nozione di valutazione presentata per il caso puramente funzionale. Innanzitutto dobbiamo generalizzare la nozione di risultato di una valutazione. In *Mini-ML*, un'espressione produceva un valore quando valutata. In *MLR*, dobbiamo appaiare il valore prodotto dalla valutazione di un'espressione con il contenuto della memoria al termine della valutazione, inoltre, dobbiamo astrarre alcune celle di memoria in essa contenute. Otteniamo cosí la nozione di *risposta*. I nuovi costrutti presenti in *MLR* dànno origine a nuove istruzioni ausiliari. La seguente grammatica definisce istruzioni, continuazioni e risposte:

$$\begin{array}{rrcl} \textit{Istruzioni:} & i & ::= & \textbf{eval}\ e \mid \textbf{return}\ v \\ & & & \mid \textbf{case}^*\ v\ \textbf{of}\ \textbf{z} \Rightarrow e_1 \mid \textbf{s}\ x \Rightarrow e_2 \\ & & & \mid \langle v, e\rangle^* \mid \textbf{fst}^*\ v \mid \textbf{snd}^*\ v \\ & & & \mid \textbf{app}^*\ v\ e \\ & & & \parallel \textbf{ref}^*\ v \mid \textbf{deref}^*\ v \mid v :=_1^*\ e \mid v_1 :=_2^*\ v_2 \\ \textit{Continuazioni:} & K & ::= & \textbf{init} \mid K, \lambda x.\, i \\ \textit{Risposte:} & a & ::= & (S, v) \mid \textbf{new}\ c.\, a \end{array}$$

Le regole di tipizzazione per istruzioni, continuazioni, memoria e risposte sono riportate nelle Figure 6.3 e 6.2.

Le regole di valutazione, sempre secondo lo schema basato su continuazioni, sono presentate nelle Figure 6.6 e 6.5. Di nuovo, abbiamo evidenziato nella parte bassa di queste figure il trattamento dei nuovi costrutti del linguaggio. Si noti come viene valutata una referenza ad un'espressione (regola **ev_ref***), ossia un oggetto che si rende disponibile tramite un'indirezione:

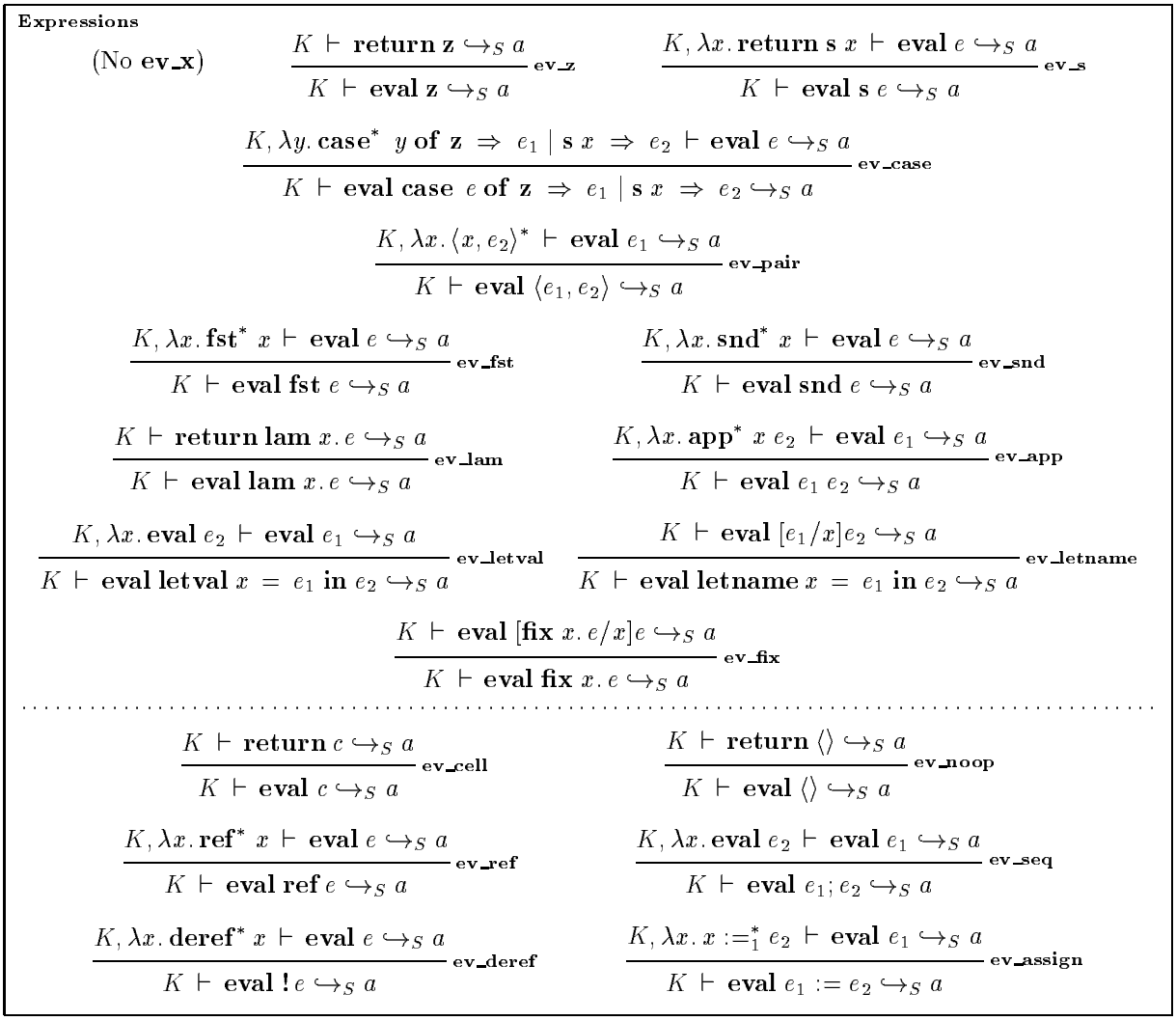


Figura 6.5: Valutazione in *MLR*, Espressioni

viene creata una nuova cella di memoria dove viene inserito il valore referenziato e viene ritornata questa cella. Poiché questa cella è nuova, dobbiamo astrarla nella risposta. È a causa di questa possibilità che dobbiamo impacchettare l'intera memoria in una risposta. La regola per la dereferenziazione non fa altro che leggere il contenuto di una cella di memoria. La regola che tratta l'assegnamento cambia il valore di una cella di memoria.

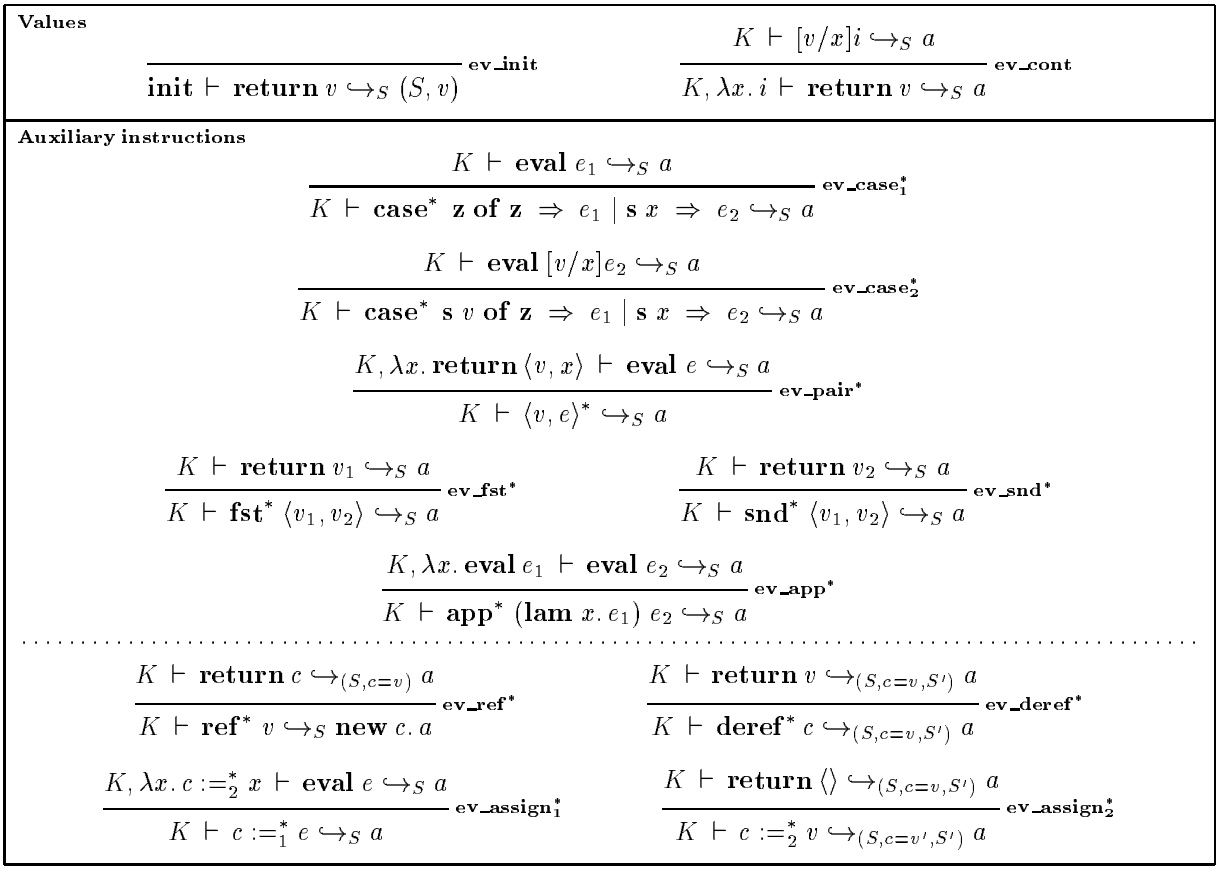


Figura 6.6: Valutazione in *MLR*, Valori ed Istruzioni Ausiliarie

6.3.2 Rappresentazione in *LLF*

La rappresentazione del livello sintattico di *MLR* non richiede l'accesso alla parte lineare del contesto di *LLF*. Pertanto non ci dilungheremo, dando solamente la codifica dei costrutti nuovi rispetto a *Mini-ML*. Le dichiarazioni delle famiglie di tipo necessarie per codificare la sintassi astratta di *MLR* sono date dalle seguenti dichiarazioni:

```
exp  : type.          instr  : type.
tp   : type.          cont   : type.

item : type.          store  : type.
cell : type.          answer : type.
```

Le quattro ultime dichiarazioni sono nuove. Le prime due servono a rappresentare la memoria. Le ultime due codificano celle di memoria e risposte.

Le seguenti dichiarazioni permettono di rappresentare i nuovi costrutti per espressioni e tipi:

Espressioni : `exp` ...

$$
\begin{array}{rcl}
\ulcorner c \urcorner & = & \texttt{rf } c \\
\ulcorner \mathbf{ref}\, e \urcorner & = & \texttt{ref } \ulcorner e \urcorner \\
\ulcorner \mathbf{!}\, e \urcorner & = & \texttt{! } \ulcorner e \urcorner \\
\ulcorner \langle\rangle \urcorner & = & \texttt{noop} \\
\ulcorner e_1 := e_2 \urcorner & = & \ulcorner e_1 \urcorner \texttt{ is } \ulcorner e_2 \urcorner \\
\ulcorner e_1; e_2 \urcorner & = & \ulcorner e_1 \urcorner \texttt{ ; } \ulcorner e_2 \urcorner
\end{array}
$$

Tipi : `tp` ...

$$
\begin{array}{rcl}
\ulcorner \tau\ \mathbf{ref} \urcorner & = & \ulcorner \tau \urcorner \texttt{ tref} \\
\ulcorner \mathbf{cmd} \urcorner & = & \texttt{cmd}
\end{array}
$$

Le dichiarazioni per modellare le continuazioni non cambiano. Si hanno invece le seguenti definizioni per quanto riguarda le nuove istruzioni ausiliarie:

Istruzioni : `instr` ...

$$
\begin{array}{rcl}
\ulcorner \mathbf{ref}^*\ v \urcorner & = & \texttt{ref* } \ulcorner v \urcorner \\
\ulcorner \mathbf{deref}^*\ v \urcorner & = & \texttt{deref* } \ulcorner v \urcorner \\
\ulcorner v :=^*_1 e \urcorner & = & \ulcorner v \urcorner \texttt{ is*1 } \ulcorner e \urcorner \\
\ulcorner v_1 :=^*_2 v_2 \urcorner & = & \ulcorner v_1 \urcorner \texttt{ is*2 } \ulcorner v_2 \urcorner
\end{array}
$$

Completamente nuove sono le dichiarazioni che permettono di trattare la memoria e le risposte:

Memoria : `item` *and* `store`

$$
\begin{array}{rcl}
\ulcorner \cdot \urcorner & = & \texttt{estore} \\
\ulcorner S, c = v \urcorner & = & \ulcorner S \urcorner \texttt{ and (} c \texttt{ = } \ulcorner v \urcorner \texttt{)}
\end{array}
$$

Risposte : `answer`

$$
\begin{array}{rcl}
\ulcorner (S, v) \urcorner & = & \texttt{close } \ulcorner S \urcorner\ \ulcorner v \urcorner \\
\ulcorner \mathbf{new}\ c.\, a \urcorner & = & \texttt{new [}c\texttt{:cell]} \ulcorner a \urcorner
\end{array}
$$

I teoremi di adeguatezza di questa meta-rappresentazione si possono trovare in Appendice C.

Anche la rappresentazione della semantica statica di *MLR* non introduce sostanziali novità rispetto a quanto visto nel Capitolo 4 per *Mini-ML*. Facciamo vedere solamente la dichiarazione delle costanti di tipo:

```
tpc :  cell -> tp -> type.
tpe :  exp -> tp -> type.
tpi :  instr -> tp -> type.
tpK :  cont -> tp -> tp -> type.
tpS :  store -> type.
tpa :  answer -> tp -> type.
```

La prima di queste dichiarazioni serve ad assegnare un tipo ad ogni cella di memoria. Non corrisponde ad un giudizio vero e proprio di *MLR*. Viene invece inserita nel contesto intuizionistico come assunzione sul tipo delle celle.

La modellazione del contenuto della memoria e delle operazioni che permettono di accedervi durante la valutazione richiede l'uso degli aspetti tipicamente lineari di *LLF*. La semantica dinamica di *MLR* si appoggia su tre famiglie di tipi:

```
collect:  store -> type.
read:  cell -> exp -> type.
ev :  cont -> instr -> answer -> type.
contains:  cell -> exp -> type.
```

Le prime due sono ausiliari e la loro funzione verrà spiegata fra poco. La terza è la dichiarazione che rappresenta il giudizio di valutazione basato su continuazioni presentato nelle Figure 6.6–6.5. L'ultima dichiarazione ha come unico scopo la modellazione del contenuto di una cella di memoria. Appare solamente nella parte lineare del contesto.

Il contenuto della memoria viene rappresentato in maniera distribuita nella codifica che proponiamo. Per ogni cella c costituente la memoria, vi è una dichiarazione c: `cell` nella parte *intuizionistica* del contesto di *LLF*. Il nome di una cella può infatti apparire a piú riprese in un'espressione *MLR*. Il contenuto corrente v di c è invece rappresentato dall'assunzione *lineare* h: `contains` c v. Piú precisamente, abbiamo la seguente codifica per un giudizio di valutazione in *MLR*, dove si assume che la memoria S abbia la forma $S = (c_1 = v_1, \ldots, c_n = v_n)$

$$\ulcorner K \vdash i \hookrightarrow_S a \urcorner = \begin{bmatrix} c_1\!:\!\texttt{cell}, & \ldots, & c_n\!:\!\texttt{cell}, \\ h_1\,\hat{:}\,\texttt{contains}\ c_1\ \ulcorner v_1 \urcorner, & \ldots, & h_n\,\hat{:}\,\texttt{contains}\ c_n\ \ulcorner v_n \urcorner \end{bmatrix} \vdash_\Sigma ? \Uparrow \texttt{ev}\ \ulcorner K \urcorner\ \ulcorner i \urcorner\ \ulcorner a \urcorner$$

dove, come vedremo fra poco nel teorema di adeguatezza, ? rappresenta una derivazione per questo giudizio nel caso sia derivabile.

Il teorema di adeguatezza della rappresentazione della valutazione per *MLR* è presentato qui sotto. La sua dimostrazione segue gli schemi mostrati nel caso di *LF*. Viene leggermente complicato il trattamento della completezza a causa della maggior varietà di costrutti che possono comparire in un termine in *LLF*.

Teorema 6.3.1 (*Adeguatezza della valutazione per MLR*)

Dati una continuazione chiusa K, un'istruzione chiusa i, una memoria $S = (c_1 = v_1, \ldots, c_n = v_n)$ e una risposta a in MLR, esiste una biiezione composizionale tra derivazioni di

$$K \vdash i \hookrightarrow_S a$$

e oggetti LLF M tali che

$$\begin{bmatrix} c_1\!:\!\texttt{cell}, & \ldots, & c_n\!:\!\texttt{cell}, \\ h_1\,\hat{:}\,\texttt{contains}\ c_1\ \ulcorner v_1 \urcorner, & \ldots, & h_n\,\hat{:}\,\texttt{contains}\ c_n\ \ulcorner v_n \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \texttt{ev}\ \ulcorner K \urcorner\ \ulcorner i \urcorner\ \ulcorner a \urcorner$$

è derivabile. □

Ritorniamo alle dichiarazioni ausiliari presentate sopra. Come abbiamo detto, la memoria di *MLR* è rappresentata in maniera distribuita nel contesto di *LLF*. Tuttavia, essa viene a far parte della risposta che viene restituita al termine della valutazione. La dichiarazione `collect` permette di passare dalla rappresentazione distribuita usata dinamicamente alla rappresentazione come oggetto di tipo `store` che appare nella rappresentazione delle risposte. Abbiamo la seguente definizione:

```
col_empty : collect estore.
col_item  : collect (S and (C = V))
              o- contains C V
              o- collect S.
```

La famiglia di tipi `read` implementa l'operazione di lettura di un valore presente in memoria. Non è strettamente necessaria in quanto la stessa operazione si poteva emulare alternativamente ricorrendo a soli operatori moltiplicativi senza la necessità di dichiarazioni ausiliari. Illustra tuttavia una tecnica ricorrente: l'uso di operatori additivi per accedere ad uno o piú valori senza modificare il contesto volatile. Abbiamo la seguente definizione:

```
read_val  : read C V
              o- contains C V
              o- <T>.
```

6.3.3 Preservazione dei Tipi

Anche per *MLR* vale il teorema di preservazione dei tipi. Come per *Mini-ML*, esso afferma che il tipo del risultato della valutazione di un'espressione coincide con il tipo dell'espressione stessa. Nuovamente, l'enunciato di questo risultato è complicato dalla necessità di tener conto dei vari ingredienti ausiliari: il tipo della continuazione e della memoria che compaiono nel testo del teorema.

Come abbiamo già fatto nel caso puramente funzionale, facciamo vedere la rappresentazione in *LLF* di questo meta-teorema contemporaneamente alla sua dimostrazione. Prima di procedere con questo passo, enunciamo alcuni lemmi ausiliari. Come per *Mini-ML*, ci servono le proprietà di weakening e sostituzione. La prima viene estesa permettendo l'aggiunta di assunzioni sulle celle di memoria senza cambiare la validità delle derivazioni di tipi. Questi risultati si dimostrano per induzione.

Lemma 6.3.2 (*Weakening*)

i. *Se* $\Omega;\Gamma \vdash^e e : \tau$, *allora* $\Omega,\Omega';\Gamma,\Gamma' \vdash^e e : \tau$;

ii. *Se* $\Omega;\Gamma \vdash^i i : \tau$, *allora* $\Omega,\Omega';\Gamma,\Gamma' \vdash^i i : \tau$;

iii. *Se* $\Omega;\Gamma \vdash^K K : \tau_1 \Rightarrow \tau_2$, *allora* $\Omega,\Omega';\Gamma,\Gamma' \vdash^K K : \tau_1 \Rightarrow \tau_2$;

iv. *Se* $\Omega \vdash^S S : \overline{\Omega}$, *allora* $\Omega, \Omega' \vdash^S S : \overline{\Omega}$;

v. *Se* $\Omega \vdash^a a : \tau$, *allora* $\Omega, \Omega' \vdash^a a : \tau$. □

Lemma 6.3.3 (*Sostituzione*)

i. *Se* $\Omega; \Gamma, x{:}\tau' \vdash^e e : \tau$ *e* $\Omega; \Gamma \vdash^e e' : \tau'$, *allora* $\Omega; \Gamma \vdash^e [e'/x]e : \tau$;

ii. *Se* $\Omega; \Gamma, x{:}\tau' \vdash^i i : \tau$ *e* $\Omega; \Gamma \vdash^e e' : \tau'$, *allora* $\Omega; \Gamma \vdash^i [e'/x]i : \tau$. □

Ci serve inoltre un piccolo lemma che ci permette di accedere ad elementi intermedi nella memoria. Anche questo risultato viene provato per induzione.

Lemma 6.3.4 (*Inversione sulla tipizzazione della memoria*)

i. *Se* $\Omega \vdash^S (S', c = v, S'') : \overline{\Omega}$, *allora* $\overline{\Omega} = \Omega', c{:}\tau, \Omega''$ *con* $\Omega \vdash^S S' : \Omega'$, $\Omega; \cdot \vdash^e v : \tau$ *e* $\Omega \vdash^S S'' : \Omega''$;

ii. *Se* $\Omega \vdash^S S' : \Omega'$, $\Omega; \cdot \vdash^e v : \tau$ *e* $\Omega \vdash^S S'' : \Omega''$,
allora $\Omega \vdash^S (S', c = v, S'') : (\Omega', c{:}\tau, \Omega'')$. □

Sebbene vengano usate nella dimostrazione, nessuna di queste proprietà dà origine a dichiarazioni ausiliari nella rappresentazione del teorema di preservazione dei tipi. Le prime due sono infatti gestite internamente dalla meta-teoria di *LLF*, la terza viene by-passata grazie alla rappresentazione distribuita della memoria nel contesto del meta-linguaggio.

Prima di presentare l'enunciato e la dimostrazione del teorema di preservazione dei tipi per *MLR*, mostriamo le dichiarazioni *LLF* necessarie alla sua rappresentazione:

```
prRead :  read C V -> tpc C T -> tpe V T -> type.
prCollect :  collect S -> tpS S -> type.
pr :  ev K I A -> tpi I T -> tpK K T T´ -> tpa A T´ -> type.
prCell :  contains C V -> tpc C T -> tpe V T -> type.
```

Le prime due hanno a che fare con la gestione al livello meta-teorico delle dichiarazioni ausiliari `collect` e `read`. La terza serve a rappresentare il teorema di preservazione dei tipi vero e proprio. La quarta dichiarazione associa dinamicamente il contenuto di una cella di memoria al suo tipo. Essa non corrisponde ad alcun giudizio del linguaggio oggetto, fa invece solamente parte della parte lineare del contesto della meta-rappresentazione. Abbiamo infatti il seguente teorema di adeguatezza, il quale illustra anche come viene effettuata la meta-rappresentazione.

Teorema 6.3.5 (*Adeguatezza della rappresentazione della presentazione dei tipi per MLR*)

Dati una continuazione chiusa K, un'istruzione chiusa i, una risposta chiusa a, una memoria chiusa $S = (c_1 = v_1, \ldots, c_n = v_n)$, un contesto di memoria $\Omega = (c_1 : \tau_1, \ldots, c_n : \tau_n)$, tipi τ e τ', e oggetti LLF *E, T_i, T_K e T_S tali che i giudizi*

$$\begin{bmatrix} c_1\!:\!\mathtt{cell}, & \ldots, & c_n\!:\!\mathtt{cell}, \\ h_1\,\hat{:}\,\mathtt{contains}\ c_1\ \ulcorner v_1\urcorner, & \ldots, & h_n\,\hat{:}\,\mathtt{contains}\ c_n\ \ulcorner v_n\urcorner \end{bmatrix} \vdash_\Sigma E \Uparrow \mathtt{ev}\ \ulcorner K\urcorner\ \ulcorner i\urcorner\ \ulcorner a\urcorner$$

$$\begin{bmatrix} c_1\!:\!\mathtt{cell}, & \ldots, & c_p\!:\!\mathtt{cell}, \\ t_1\!:\!\mathtt{tpc}\ c_1\ \ulcorner\tau_1\urcorner, & \ldots, & t_p\!:\!\mathtt{tpc}\ c_p\ \ulcorner\tau_p\urcorner \end{bmatrix} \vdash_\Sigma T_i \Uparrow \mathtt{tpi}\ \ulcorner i\urcorner\ \ulcorner\tau\urcorner$$

$$\begin{bmatrix} c_1\!:\!\mathtt{cell}, & \ldots, & c_p\!:\!\mathtt{cell}, \\ t_1\!:\!\mathtt{tpc}\ c_1\ \ulcorner\tau_1\urcorner, & \ldots, & t_p\!:\!\mathtt{tpc}\ c_p\ \ulcorner\tau_p\urcorner \end{bmatrix} \vdash_\Sigma T_K \Uparrow \mathtt{tpK}\ \ulcorner i\urcorner\ \ulcorner\tau\urcorner\ \ulcorner\tau'\urcorner$$

$$\begin{bmatrix} c_1\!:\!\mathtt{cell}, & \ldots, & c_p\!:\!\mathtt{cell}, \\ t_1\!:\!\mathtt{tpc}\ c_1\ \ulcorner\tau_1\urcorner, & \ldots, & t_p\!:\!\mathtt{tpc}\ c_p\ \ulcorner\tau_p\urcorner \end{bmatrix} \vdash_\Sigma T_S \Uparrow \mathtt{tpS}\ \ulcorner S\urcorner$$

siano derivabili, esistono oggetti LLF T_a e M tali che

$$\begin{bmatrix} c_1\!:\!\mathtt{cell}, & \ldots, & c_p\!:\!\mathtt{cell}, \\ t_1\!:\!\mathtt{tpc}\ c_1\ \ulcorner\tau_1\urcorner, & \ldots, & t_p\!:\!\mathtt{tpc}\ c_p\ \ulcorner\tau_p\urcorner \end{bmatrix} \vdash_\Sigma T_a \Uparrow \mathtt{tpA}\ \ulcorner a\urcorner\ \ulcorner\tau'\urcorner$$

e

$$\begin{bmatrix} c_1\!:\!\mathtt{cell}, & \ldots, & c_n\!:\!\mathtt{cell}, \\ h_1\!:\!\mathtt{contains}\ c_1\ \ulcorner v_1\urcorner, & \ldots, & h_n\!:\!\mathtt{contains}\ c_n\ \ulcorner v_n\urcorner, \\ t_1\!:\!\mathtt{tpc}\ c_1\ \ulcorner\tau_1\urcorner, & \ldots, & t_n\!:\!\mathtt{tpc}\ c_n\ \ulcorner\tau_n\urcorner, \\ p_1\,\hat{:}\,\mathtt{prCell}\ h_1\ t_1, & \ldots, & p_n\,\hat{:}\,\mathtt{prCell}\ h_n\ t_n \end{bmatrix}$$

$$\vdash_\Sigma M \Uparrow \mathtt{pr}\ \ulcorner K\urcorner\ \ulcorner i\urcorner\ \ulcorner\tau\urcorner\ \ulcorner\tau'\urcorner\ \ulcorner a\urcorner\ E\ T_i\ T_K\ T_a$$

sono derivabili. □

Si noti che si fa uso del contesto lineare anche per la rappresentazione del teorema di preservazione dei tipi in *MLR*. Infatti, questo è un esempio di *meta-teoria lineare*. Si noti che le assunzioni sul contenuto della memoria ($\mathtt{contains}\ c_i\ \ v_i$), pur essendo di per sè lineari, sono inserite nella parte intuizionistica del contesto. Non le manteniamo nel contesto lineare in quanto *LLF* non ammette tipi dipendenti lineari.

Passiamo ora all'enunciato e alla dimostrazione del teorema di preservazione dei tipi per *MLR*. Come già fatto nel caso puramente funzionale, presentiamo la codifica in *LLF* dei casi considerati.

Teorema 6.3.6 (*Preservazione dei tipi*)

Se $\mathcal{E} :: K \vdash i \hookrightarrow_S a$ *con* $\mathcal{T}_i :: \Omega; \cdot \vdash^i i : \tau$, $\mathcal{T}_K :: \Omega; \cdot \vdash^K K : \tau \Rightarrow \bar{\tau}$ *e* $\mathcal{T}_S :: \Omega \vdash^S S : \Omega$, *allora* $\mathcal{T}_a :: \Omega \vdash^a a : \bar{\tau}$.

Dimostrazione.

Procediamo per induzione sulla struttura di $\mathcal{E}$ e inversione su $\mathcal{T}_i$, $\mathcal{T}_K$ e $\mathcal{T}_S$. Facciamo vedere solamente alcuni casi particolarmente significativi assieme alla loro implementazione in *LLF*. Gli altri casi sono altrettanto semplici; la loro implementazione si può trovare in Appendice C.

ev_s

$$\mathcal{E} = \frac{\mathcal{E}'}{\dfrac{K, \lambda x.\,\mathbf{return\ s}\ x \vdash \mathbf{eval}\ e \hookrightarrow_S a}{K \vdash \mathbf{eval\ s}\ e \hookrightarrow_S a}\ \mathbf{ev\_s}}$$

with $i = \mathbf{eval\ s}\ e$.

$\mathcal{T}_e :: \Omega;\cdot \vdash^e e : \mathbf{nat}$ and $\tau = \mathbf{nat}$	by inversion on **tpi_eval** and **tpe_s** for $\mathcal{T}_i$,
$\mathcal{T}_i' :: \Omega;\cdot \vdash^i \mathbf{eval}\ e : \mathbf{nat}$	by rule **tpi_eval** on $\mathcal{T}_e$,
$\mathcal{T}_v :: \Omega; x{:}\mathbf{nat} \vdash^e \mathbf{s}\ x : \mathbf{nat}$	by rule **tpe_x** and **tpe_s**,
$\mathcal{T}_i'' :: \Omega; x{:}\mathbf{nat} \vdash^i \mathbf{return\ s}\ x : \mathbf{nat}$	by rule **tpi_return** on $\mathcal{T}_v$,
$\mathcal{T}_K' :: \Omega;\cdot \vdash^K K, \lambda x.\,\mathbf{return\ s}\ x : \mathbf{nat} \Rightarrow \bar{\tau}$	by rule **tpK_lam** on $\mathcal{T}_i''$ and $\mathcal{T}_K$,
$\mathcal{T}_a :: \Omega \vdash^a a : \bar{\tau}$	by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$, $\mathcal{T}_K'$ and $\mathcal{T}_S$.

```
pr-ev_s        : pr (ev_s E) (tpi_eval (tpe_s Te)) TK Ta
                   o- pr E
                         (tpi_eval Te)
                         (tpK_lam TK [x][t:tpe x nat] tpi_return (tpe_s t))
                         Ta.
```

ev_cell

$$\mathcal{E} = \frac{\mathcal{E}'}{\dfrac{K \vdash \mathbf{return}\ c \hookrightarrow_S a}{K \vdash \mathbf{eval}\ c \hookrightarrow_S a}\ \mathbf{ev\_cell}}$$

with $i = \mathbf{eval}\ c$.

$\mathcal{T}_e :: \Omega;\cdot \vdash^e c : \tau'\ \mathbf{ref}$ and $\tau = \tau'\ \mathbf{ref}$	by inversion on **tpi_eval** and **tpe_cell** for $\mathcal{T}_i$,
$\mathcal{T}_i' :: \Omega;\cdot \vdash^i \mathbf{return}\ c : \tau'\ \mathbf{ref}$	by rule **tpi_return** on $\mathcal{T}_e$,
$\mathcal{T}_a :: \Omega \vdash^a a : \bar{\tau}$	by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$, $\mathcal{T}_K$ and $\mathcal{T}_S$.

```
pr-ev_cell     : pr (ev_cell E) (tpi_eval (tpe_cell Tc)) TK Ta
                   o- pr E (tpi_return (tpe_cell Tc)) TK Ta.
```

ev_ref

$$\mathcal{E} = \frac{\mathcal{E}'}{\dfrac{K, \lambda x.\,\mathbf{ref^*}\ x \vdash \mathbf{eval}\ e \hookrightarrow_S a}{K \vdash \mathbf{eval\ ref}\ e \hookrightarrow_S a}\ \mathbf{ev\_ref}}$$

with $i = \mathbf{eval\ ref}\ e$.

$\mathcal{T}_e :: \Omega;\cdot \vdash^e e : \tau'$ and $\tau = \tau'\ \mathbf{ref}$	by inversion on **tpi_eval** and **tpe_ref** for $\mathcal{T}_i$,

$\mathcal{T}_i' :: \Omega; \cdot \vdash^i \mathbf{eval}\, e : \tau'$ by rule **tpi_eval** on $\mathcal{T}_e$,
$\mathcal{T}_i'' :: \Omega; x : \tau' \vdash^i \mathbf{ref}^*\, x : \tau'\, \mathbf{ref}$ by rules **tpe_x** and **tpi_ref***,
$\mathcal{T}_K' :: \Omega; \cdot \vdash^K K, \lambda x.\, \mathbf{ref}^*\, x : \tau' \Rightarrow \bar{\tau}$ by rule **tpK_lam** on $\mathcal{T}_i''$ and $\mathcal{T}_K$,
$\mathcal{T}_a :: \Omega \vdash^a a : \bar{\tau}$ by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$, $\mathcal{T}_K'$ and $\mathcal{T}_S$.

```
pr-ev_ref      : pr (ev_ref E) (tpi_eval (tpe_ref Te)) TK Ta
                   o- pr E
                         (tpi_eval Te)
                         (tpK_lam TK [x][t:tpe x T] tpi_ref* t)
                         Ta.
```

ev_seq
$$\mathcal{E} = \frac{\mathcal{E}'}{K, \lambda x.\, \mathbf{eval}\, e_2 \vdash \mathbf{eval}\, e_1 \hookrightarrow_S a}{K \vdash \mathbf{eval}\, e_1; e_2 \hookrightarrow_S a}\ \mathbf{ev\_seq}$$

with $i = \mathbf{eval}\, e_1; e_2$.

$\mathcal{T}_e' :: \Omega; \cdot \vdash^e e_1 : \tau'$ and
$\mathcal{T}_e'' :: \Omega; \cdot \vdash^e e_2 : \tau$ by inversion on **tpi_eval** and **tpe_seq** for $\mathcal{T}_i$,
$\mathcal{T}_i' :: \Omega; \cdot \vdash^i \mathbf{eval}\, e_1 : \tau'$ by rule **tpi_eval** on $\mathcal{T}_e'$,
$\mathcal{T}_i'' :: \Omega; \cdot \vdash^i \mathbf{eval}\, e_2 : \tau$ by rule **tpi_eval** on $\mathcal{T}_e''$,
$\mathcal{T}_i''' :: \Omega; x : \tau' \vdash^i \mathbf{eval}\, e_2 : \tau$ by weakening on $\mathcal{T}_i''$,
$\mathcal{T}_K' :: \Omega; \cdot \vdash^K K, \lambda x.\, \mathbf{eval}\, e_2 : \tau' \Rightarrow \bar{\tau}$ by rule **tpK_lam** on $\mathcal{T}_i'''$ and $\mathcal{T}_K$,
$\mathcal{T}_a :: \Omega \vdash^a a : \bar{\tau}$ by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$, $\mathcal{T}_K'$ and $\mathcal{T}_S$.

```
pr-ev_seq      : pr (ev_seq E) (tpi_eval (tpe_seq Te'' Te')) TK Ta
                   o- pr E
                         (tpi_eval Te'')
                         (tpK_lam TK [x][t:tpe x T'] tpi_eval Te')
                         Ta.
```

ev_init
$$\mathcal{E} = \frac{}{\mathbf{init} \vdash \mathbf{return}\, v \hookrightarrow_S (S, v)}\ \mathbf{ev\_init}$$

with $K = \mathbf{init}$, $i = \mathbf{return}\, v$ and $a = (S, v)$.

$\bar{\tau} = \tau$ by inversion on **tpK_init** for $\mathcal{T}_K$,
$\mathcal{T}_v :: \Omega; \cdot \vdash^e v : \tau$ by inversion on rule **tpi_return** for $\mathcal{T}_i$,
$\mathcal{T}_a :: \Omega \vdash^a (S, v) : \bar{\tau}$ by rule **tpa_val** on $\mathcal{T}_S$ and $\mathcal{T}_v$.

```
pr-ev_init     : pr (ev_init C) (tpi_return Tv) tpK_init (tpa_close Tv TS)
                   o- prCollect C TS.
```

ev_cont

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}' \\ K' \vdash [v/x]i' \hookrightarrow_S a\end{array}}{K', \lambda x.\, i' \vdash \mathbf{return}\ v \hookrightarrow_S a}\ \mathbf{ev\_cont}$$

with $K = K', \lambda x.\, i'$ and $i = \mathbf{return}\ v$.

$\mathcal{T}_i'' :: \Omega; x{:}\tau \vdash^i i' : \tau'$ and

$\mathcal{T}_K' :: \Omega; \cdot \vdash^K K : \tau' \Rightarrow \bar{\tau}$ by inversion on **tpK_lam** for $\mathcal{T}_K$,

$\mathcal{T}_v :: \Omega; \cdot \vdash^e v : \tau$ by inversion on **tpi_return** for $\mathcal{T}_i$,

$\mathcal{T}_i' :: \Omega; \cdot \vdash^i [v/x]i' : \tau'$ by the substitution lemma on $\mathcal{T}_i''$ and $\mathcal{T}_v$,

$\mathcal{T}_a :: \Omega \vdash^a a : \bar{\tau}$ by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$, $\mathcal{T}_K'$ and $\mathcal{T}_S$.

```
pr-ev_cont     : {Tv: tpe V T}
                 pr (ev_cont E) (tpi_return Tv) (tpK_lam TK Ti) Ta
                  o- pr E (Ti V Tv) TK Ta.
```

ev_ref*

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}' \\ K \vdash \mathbf{return}\ c \hookrightarrow_{(S, c=v)} a'\end{array}}{K \vdash \mathbf{ref}^*\ v \hookrightarrow_S \mathbf{new}\ c.\, a'}\ \mathbf{ev\_ref^*}$$

with $i = \mathbf{ref}^*\ v$ and $a = \mathbf{new}\ c.\, a'$.

$\mathcal{T}_v :: \Omega; \cdot \vdash^e v : \tau'$ and $\tau = \tau'\ \mathbf{ref}$ by inversion on **tpi_ref*** for $\mathcal{T}_i$,

$\mathcal{T}_v' :: \Omega, c{:}\tau'; \cdot \vdash^e v : \tau'$ by weakening on $\mathcal{T}_v$,

$\mathcal{T}_i' :: \Omega, c{:}\tau'; \cdot \vdash^i \mathbf{return}\ c : \tau'\ \mathbf{ref}$ by rules **tpe_cell** and **tpi_return**,

$\mathcal{T}_S'' :: \Omega, c{:}\tau' \vdash^S S : \Omega$ by weakening on $\mathcal{T}_S$,

$\mathcal{T}_S' :: \Omega, c{:}\tau' \vdash^S (S, c = v) : (\Omega, c{:}\tau')$ by rule **tpS_cell** on $\mathcal{T}_S''$ and $\mathcal{T}_v'$,

$\mathcal{T}_K' :: \Omega, c{:}\tau'; \cdot \vdash^K K : \tau \Rightarrow \bar{\tau}$ by weakening on $\mathcal{T}_K$,

$\mathcal{T}_a' :: \Omega, c{:}\tau' \vdash^a a' : \bar{\tau}$ by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$, $\mathcal{T}_K'$ and $\mathcal{T}_S'$,

$\mathcal{T}_a :: \Omega \vdash^a \mathbf{new}\ c.\, a : \bar{\tau}$ by rule **tpS_new** on $\mathcal{T}_a'$.

```
pr-ev_ref*     : pr (ev_ref* E) (tpi_ref* Tv) TK (tpa_new Ta)
                  o- ({c:cell}{Cn:contains c V}{Tc:tpc c T}
                      prCell Cn Tc Tv
                      -o pr (E c Cn) (tpi_return (tpe_cell Tc)) TK (Ta c Tc)).
```

ev_deref*

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}' \\ K \vdash \mathbf{return}\ v \hookrightarrow_{(S', c=v, S'')} a\end{array}}{K \vdash \mathbf{deref}^*\ c \hookrightarrow_{(S', c=v, S'')} a}\ \mathbf{ev\_deref^*}$$

with $S = S', c = v, S''$ and $i = \mathbf{deref}^*\ v$.

$\mathcal{T}_v :: \Omega; \cdot \vdash^e v : \tau$ by the store inversion lemma on $\mathcal{T}_S$,

$\mathcal{T}_i' :: \Omega; \cdot \vdash^i \mathbf{return}\, v : \tau$ by rule **tpi_return** on $\mathcal{T}_v$,
$\mathcal{T}_a :: \Omega \vdash^a a : \bar{\tau}$ by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$, $\mathcal{T}_K$ and $\mathcal{T}_S$.

```
pr-ev_deref*   : pr (ev_deref* (R , E)) (tpi_deref* (tpe_cell Tc)) Tk Ta
                   o- prRead R Tc Tv
                   &  pr E (tpi_return Tv) TK Ta.
```

ev_assign$_2^*$

$$\mathcal{E} = \frac{K \vdash \mathbf{return}\,\langle\rangle \hookrightarrow_{(S', c=v, S'')} a}{K \vdash c :=_2^* v \hookrightarrow_{(S', c=v', S'')} a}\,\mathbf{ev\_assign_2^*}$$

with $S = S', c = v', S''$ and $i = c :=_2^* v$.

$\mathcal{T}_c :: \Omega; \cdot \vdash^e c : \tau',\ \tau = \mathbf{cmd}$ and
$\mathcal{T}_v :: \Omega; \cdot \vdash^e v : \tau'$ by inversion on **tpi_assign$_2^*$** for $\mathcal{T}_i$,
$\mathcal{T}_i' :: \Omega; \cdot \vdash^i \mathbf{return}\,\langle\rangle : \mathbf{cmd}$ by rules **tpe_noop** and **tpi_return**,
$\Omega = \Omega', c : \tau', \Omega'',\ \mathcal{T}_S^* :: \Omega \vdash^S S' : \Omega'$ and
$\mathcal{T}_S^{**} :: \Omega \vdash^S S'' : \Omega''$ by the store inversion lemma on $\mathcal{T}_S$
$\mathcal{T}_S' :: \Omega \vdash^S (S', c = v, S'') : \Omega$ by the store inversion lemma on $\mathcal{T}_S^*$, $\mathcal{T}_v$ and $\mathcal{T}_S^{**}$,
$\mathcal{T}_a :: \Omega \vdash^a a : \bar{\tau}$ by induction hypothesis on $\mathcal{E}'$, $\mathcal{T}_i'$, $\mathcal{T}_K$ and $\mathcal{T}_S'$.

```
pr-ev_assign*2 : pr (ev_assign*2 E Cn´) (tpi_assign*2 Tv (tpe_cell Tc)) Tk Ta
                   o- prCell Cn´ Tc Tv´
                   o- (prCell Cn Tc Tv
                      -o pr (E Cn) (tpi_return (tpe_noop)) Tk Ta).
```

☑

6.4 Eliminazione del Taglio per le Formule di Harrop Ereditarie Lineari

Diamo ora un secondo esempio di meta-rappresentazione in *LLF*. Questa volta consideriamo un frammento della logica lineare, il frammento liberamente generato del linguaggio delle formule di Harrop ereditarie lineari presentato nel Capitolo 3, di cui facciamo vedere la codifica del livello della sintassi, delle regole di deduzione e del teorema di eliminazione del taglio quale meta-teorema. Ci riferiamo a questo linguaggio perché lo abbiamo già incontrato in veste appena piú ricca, perché è un frammento piccolo ma non banale della logica lineare, e infine perché la considerazione di linguaggi piú estesi non porta a sostanziali complicazioni. La codifica che proponiamo si rifà a [Pfe94b]. Il codice *LLF* completo nonché gli enunciati dei teoremi di adeguatezza sono stati rilegati in Appendice D.

Axiom

$$\frac{}{\Gamma; A \longrightarrow A}\ \mathbf{id}$$

Structural rule

$$\frac{\Gamma, A; \Delta, A \longrightarrow C}{\Gamma, A; \Delta \longrightarrow C}\ \mathbf{clone}$$

Right rules

$$\frac{}{\Gamma; \Delta \longrightarrow \top}\ \mathbf{top\_r}$$

$$\frac{\Gamma; \Delta \longrightarrow A \quad \Gamma; \Delta \longrightarrow B}{\Gamma; \Delta \longrightarrow A \,\&\, B}\ \mathbf{with\_r}$$

$$\frac{\Gamma; \Delta, A \longrightarrow B}{\Gamma; \Delta \longrightarrow A \multimap B}\ \mathbf{lolli\_r}$$

$$\frac{\Gamma, A; \Delta \longrightarrow B}{\Gamma; \Delta \longrightarrow A \to B}\ \mathbf{imp\_r}$$

$$\frac{\Gamma; \Delta \longrightarrow [c/x]A}{\Gamma; \Delta \longrightarrow \forall x.\, A}\ \mathbf{all\_r}$$

Left rules

(No **top_l**)

$$\frac{\Gamma; \Delta, A \longrightarrow C}{\Gamma; \Delta, A \,\&\, B \longrightarrow C}\ \mathbf{with\_l1} \qquad \frac{\Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta, A \,\&\, B \longrightarrow C}\ \mathbf{with\_l2}$$

$$\frac{\Gamma; \Delta_1 \longrightarrow A \quad \Gamma; \Delta_2, B \longrightarrow C}{\Gamma; \Delta_1, \Delta_2, A \multimap B \longrightarrow C}\ \mathbf{lolli\_l}$$

$$\frac{\Gamma; \cdot \longrightarrow A \quad \Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta, A \to B \longrightarrow C}\ \mathbf{imp\_l}$$

$$\frac{\Gamma; \Delta, [t/x]A \longrightarrow C}{\Gamma; \Delta, \forall x.\, A \longrightarrow C}\ \mathbf{all\_l}$$

cut rules

$$\frac{\Gamma; \Delta_1 \longrightarrow A \quad \Gamma; \Delta_2, A \longrightarrow C}{\Gamma; \Delta_1, \Delta_2 \longrightarrow C}\ \mathbf{cut} \qquad \frac{\Gamma; \cdot \longrightarrow A \quad \Gamma, A; \Delta \longrightarrow C}{\Gamma; \Delta \longrightarrow C}\ \mathbf{cut!}$$

Figura 6.7: Il Frammento Liberamente Generato delle Formule di Harrop Ereditarie Lineari

Il frammento liberamente generato della logica delle formula di Harrop ereditarie lineari è definito dalla seguente grammatica:

$$A ::= P \mid \top \mid A_1 \,\&\, A_2 \mid A_1 \multimap A_2 \mid A_1 \to A_2 \mid \forall x.\, A$$

dove P spazia sulle formule atomiche, costruite al solito su termini del prim'ordine.

Usiamo le famiglie di tipi `o` ed `i` per rappresentare la categoria sintattica delle formule del nostro linguaggio e dei termini, rispettivamente. Piú precisamente, abbiamo le seguenti dichiarazioni:

```
o : type.
i : type.
```

La codifica delle formule non necessita l'uso della parte lineare del contesto di *LLF*, anzi viene effettuata ricorrendo solamente al frammento del nostro linguaggio corrispondente ad *LF*.

Le regole che specificano la validità di una formula di Harrop ereditaria lineare sono mostrate in Figura 6.7. Codifichiamo un sequente $\Gamma; \Delta \longrightarrow A$ di questo linguaggio rappresentando ogni

formula A_i in Γ mediante un'assunzione intuizionistica `int` A_i nel contesto di LLF e ogni formula B_j in Δ mediante un'assunzione lineare `lin` B_j. Il sequente stesso viene invece rappresentato dalla famiglia di tipi `pr` se si considera il calcolo che non comprende le regole di taglio, e `pr+` se includiamo anche queste regole; entrambe queste costanti assumono come argomento la codifica di A. Le dichiarazioni per queste costanti di tipo sono:

```
int : o -> type.
lin : o -> type.
pr  : o -> type.
pr+ : o -> type.
```

La rappresentazione delle singole regole d'inferenza si appoggia direttamente sui costrutti disponibili in LLF. Facciamo vedere alcuni casi tipici:

$$\ulcorner\ \frac{}{\Gamma; A \longrightarrow A}\,\mathbf{id}\ \urcorner \quad = \quad \texttt{id} \ : \ \texttt{(lin A -o pr A).}$$

$$\ulcorner\ \frac{\Gamma, A; \Delta, A \longrightarrow C}{\Gamma, A; \Delta \longrightarrow C}\,\mathbf{clone}\ \urcorner \quad = \quad \begin{cases} \texttt{clone} \ : \ \texttt{(int A -> lin A -o pr C)} \\ \qquad\quad \texttt{-o (int A -> pr C).} \end{cases}$$

$$\ulcorner\ \frac{\Gamma; \Delta \longrightarrow A \quad \Gamma; \Delta \longrightarrow B}{\Gamma; \Delta \longrightarrow A \,\&\, B}\,\mathbf{with\_r}\ \urcorner \quad = \quad \begin{cases} \texttt{with\_r} \ : \ \texttt{pr A \& pr B} \\ \qquad\quad \texttt{-o pr (A with B).} \end{cases}$$

$$\ulcorner\ \frac{\Gamma; \cdot \longrightarrow A \quad \Gamma; \Delta, B \longrightarrow C}{\Gamma; \Delta, A \to B \longrightarrow C}\,\mathbf{imp\_l}\ \urcorner \quad = \quad \begin{cases} \texttt{lolli\_l} \ : \ \texttt{pr A -o (lin B -o pr C)} \\ \qquad\quad \texttt{-o (lin (A lolli B) -o pr C).} \end{cases}$$

Si noti l'uso delle due forme di implicazioni al meta-livello, perfettamente in accordo con quanto descritto nella sezione iniziale di questo capitolo. L'implicazione lineare permette di operare sul contesto lineare partizionandolo e aggiungendovi assunzioni. L'uso dell'implicazione intuizionistica permette invece di vincolare il contenuto del contesto lineare, oppure di inserire assunzioni nella porzione intuizionistica.

Abbiamo il seguente teorema di adeguatezza.

Teorema 6.4.1 (*Adeguatezza delle derivazioni prive di tagli*)

Date formule $A_1, \ldots, A_i$, $B_1, \ldots, B_l$ *e* C *le quali menzionano costanti individuali* $c_1, \ldots, c_n$, *esiste una biiezione composizionale* $\ulcorner\_\urcorner$ *tra derivazioni prive di tagli di*

$$A_1, \ldots, A_i;\ B_1, \ldots, B_l \longrightarrow C$$

e oggetti LLF canonici M *tale che il giudizio*

$$\begin{bmatrix} c_1 : \texttt{i}, & \ldots, & c_n : \texttt{i}, \\ x_1 : \texttt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \texttt{int}\ \ulcorner A_i \urcorner, \\ y_1 \,\hat{:}\, \texttt{lin}\ \ulcorner B_1 \urcorner, & \ldots, & y_l \,\hat{:}\, \texttt{lin}\ \ulcorner B_l \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \texttt{pr}\ \ulcorner C \urcorner$$

è derivabile. □

Le dichiarazioni per `pr+` sono identiche a quelle presentate per `pr` se non per l'aggiunta delle due clausole che codificano le regole di taglio. Il codice completo si può trovare in Appendice D.

$$\ulcorner \frac{\Gamma;\Delta_1 \longrightarrow A \quad \Gamma;\Delta_2, A \longrightarrow C}{\Gamma;\Delta_1,\Delta_2 \longrightarrow C}\,\mathbf{cut} \urcorner = \left\{ \begin{array}{l} \texttt{cut : pr+ A -o (lin A -o pr+ C)} \\ \qquad\qquad \texttt{-o pr+ C.} \end{array} \right.$$

$$\ulcorner \frac{\Gamma;\cdot \longrightarrow A \quad \Gamma, A;\Delta \longrightarrow C}{\Gamma;\Delta \longrightarrow C}\,\mathbf{cut!} \urcorner = \left\{ \begin{array}{l} \texttt{cut! : pr+ A -> (int A -> pr+ C)} \\ \qquad\qquad \texttt{-o pr+ C.} \end{array} \right.$$

Considerazioni simili a quanto descritto sopra valgono per queste dichiarazioni. Vale inoltre un simile teorema di adeguatezza.

Passiamo ora alla rappresentazione della meta-teoria del nostro esempio. Vogliamo codificare il teorema di dimostrazione del taglio. Ricordiamo che questo risultato afferma che ogni derivazione, eventualmente facente uso delle regole di taglio, può essere trasformata in una derivazione equivalente priva di queste regole. Il primo passo verso la dimostrazione di questo risultato consiste nel far vedere che entrambe le regole di taglio sono ammissibili nel sistema deduttivo considerato. Al solito, facciamo vedere la rappresentazione di questo risultato contemporaneamente alla sua dimostrazione. Facciamo uso delle seguenti due famiglie di tipi per codificare l'ammissibilità delle due regole di taglio:

```
adm  : pr A -> (lin A -> pr C) -> pr C -> type.
adm! : pr A -> (int A -> pr C) -> pr C -> type.
```

La rappresentazione di questa dimostrazione non fa uso della porzione lineare del contesto di *LLF*. La dimostrazione di questo risultato si rifà a [Pfe94b].

Lemma 6.4.2 (*Ammissibilità di* `cut` e `cut!`)

i. Date derivazioni prive di tagli $\mathcal{D}_1 :: \Gamma;\Delta_1 \longrightarrow A$ e $\mathcal{D}_2 :: \Gamma;\Delta_2, A \longrightarrow C$, esiste una derivazione priva di tagli $\mathcal{D}$ di $\Gamma;\Delta_1,\Delta_2 \longrightarrow C$.

ii. Date derivazioni prive di tagli $\mathcal{D}_1 :: \Gamma;\cdot \longrightarrow A$ e $\mathcal{D}_2 :: \Gamma, A;\Delta \longrightarrow C$, esiste una derivazione priva di tagli $\mathcal{D}$ di $\Gamma;\Delta \longrightarrow C$.

Dimostrazione.

Dimostriamo le due parti del lemma simultaneamente per induzione sulla struttura di A, sulla convenzione che (i) è piú piccolo di (ii) e sulla struttura di $\mathcal{D}_1$ e $\mathcal{D}_2$. Facciamo vedere solamente alcuni casi significativi assieme al codice *LLF* risultante.

(id, _) $\quad \mathcal{D}_1 = \dfrac{}{\Gamma; A \longrightarrow A}\,\mathbf{id} \quad \mathbf{X} \quad \begin{array}{c}\mathcal{D}_2 \\ \Gamma;\Delta_2, A \longrightarrow C\end{array}$

with $\Delta_1 = A$.

Set $\mathcal{D}$ to $\mathcal{D}_2$, but first replace the occurrence of A in $\mathcal{D}_2$ with the one in $\mathcal{D}_1$.

```
adm_id-*      : adm (id X) D2 (D2 X).
```

(imp$_\!\multimap$, $\_$) $\mathcal{D}_1 = \dfrac{\begin{array}{cc}\mathcal{D}_1' & \mathcal{D}_1'' \\ \Gamma;\cdot \longrightarrow B_1 & \Gamma;\Delta_1', B_2 \longrightarrow A\end{array}}{\Gamma;\Delta_1', B_1 \to B_2 \longrightarrow A}\,\mathbf{imp\_\!\multimap}$ X $\begin{array}{c}\mathcal{D}_2 \\ \Gamma;\Delta_2, A \longrightarrow C\end{array}$

with $\Delta_1 = \Delta_1', B_1 \to B_2$.

$\mathcal{D}'' :: \Gamma;\Delta_1', B_2, \Delta_2 \longrightarrow C$ by induction hypothesis on $\mathcal{D}_1''$ and $\mathcal{D}_2$,
$\mathcal{D} :: \Gamma;\Delta_1', B_1 \to B_2, \Delta_2 \longrightarrow C$ by rule **imp$_\!\multimap$** on $\mathcal{D}''$.

```
adm_imp_l-*   : adm (imp_l D11 D12 Y) D2 (imp_l D11 D' Y)
                  <- ({y:lin B2} adm (D12 y) D2 (D' y)).
```

($\_$, id) $\begin{array}{c}\mathcal{D}_1 \\ \Gamma;\Delta_1 \longrightarrow A\end{array}$ X $\mathcal{D}_2 = \dfrac{}{\Gamma; A \longrightarrow A}\,\mathbf{id}$

with $\Delta_2 = \cdot$ and $C = A$.

We simply set $\mathcal{D}$ to $\mathcal{D}_1$ since this derivation is itself a proof of $\Gamma;\Delta_1,\Delta_2 \longrightarrow A$.

```
adm_*-id      : adm D1 id D1.
```

($\_$, lolli$_\!\multimap$), not on A We must distinguish two cases depending on the premiss of **lolli$_\!\multimap$** A is propagated to.

- *Left premiss*: $\begin{array}{c}\mathcal{D}_1 \\ \Gamma;\Delta_1 \longrightarrow A\end{array}$ X $\mathcal{D}_2 = \dfrac{\begin{array}{cc}\mathcal{D}_2' & \mathcal{D}_2'' \\ \Gamma;\Delta_2', A \longrightarrow B_1 & \Gamma;\Delta_2'', B_2 \longrightarrow C\end{array}}{\Gamma;\Delta_2', B_1 \multimap B_2, \Delta_2'', A \longrightarrow C}\,\mathbf{lolli\_\!\multimap}$

  with $\Delta_2 = \Delta_2', B_1 \multimap B_2, \Delta_2''$.

  $\mathcal{D}' :: \Gamma;\Delta_1,\Delta_2' \longrightarrow B_1$ by induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}_2'$,
  $\mathcal{D} :: \Gamma;\Delta_1,\Delta_2', B_1 \multimap B_2, \Delta_2'' \longrightarrow C$ by rule **lolli$_\!\multimap$** on $\mathcal{D}'$.

```
adm_*-lolli_l1: adm D1 ([x:lin A] lolli_l (D21 x) D22 Y) (lolli_l D' D22 Y)
                  <- adm D1 D21 D'.
```

- *Right premiss*: $\begin{array}{c}\mathcal{D}_1 \\ \Gamma;\Delta_1 \longrightarrow A\end{array}$ X $\mathcal{D}_2 = \dfrac{\begin{array}{cc}\mathcal{D}_2' & \mathcal{D}_2'' \\ \Gamma;\Delta_2' \longrightarrow B_1 & \Gamma;\Delta_2'', B_2, A \longrightarrow C\end{array}}{\Gamma;\Delta_2', B_1 \multimap B_2, \Delta_2'', A \longrightarrow C}\,\mathbf{lolli\_\!\multimap}$

  with $\Delta_2 = \Delta_2', B_1 \multimap B_2, \Delta_2''$.

  $\mathcal{D}'' :: \Gamma;\Delta_1,\Delta_2'', B_2 \longrightarrow C$ by induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}_2''$,
  $\mathcal{D} :: \Gamma;\Delta_1,\Delta_2', B_1 \multimap B_2, \Delta_2'' \longrightarrow C$ by rule **lolli$_\!\multimap$** on $\mathcal{D}''$.

```
adm_*-lolli_l2: adm D1 ([x:lin A] lolli_l D21 (D22 x) Y) (lolli_l D21 D' Y)
                  <- ({y: lin B2} adm D1 ([x:lin A] D22 x y) (D' y)).
```

(lolli_r, lolli_l) on A

$$\mathcal{D}_1 = \dfrac{\mathcal{D}_1'}{\dfrac{\Gamma;\Delta_1, A_1 \longrightarrow A_2}{\Gamma;\Delta_1 \longrightarrow A_1 \multimap A_2}} \quad \mathbf{X} \quad \mathcal{D}_2 = \dfrac{\mathcal{D}_2' \qquad \mathcal{D}_2''}{\dfrac{\Gamma;\Delta_2' \longrightarrow A_1 \quad \Gamma;\Delta_2'', A_2 \longrightarrow C}{\Gamma;\Delta_2', \Delta_2'', A_1 \multimap A_2 \longrightarrow C}}$$

with $A = A_1 \multimap A_2$ and $\Delta_2 = \Delta_2', \Delta_2''$.

$\mathcal{D}'' :: \Gamma;\Delta_1, \Delta_2'', A_1 \longrightarrow C$ by induction hypothesis on $\mathcal{D}_1'$ and $\mathcal{D}_2''$,
$\mathcal{D} :: \Gamma;\Delta_1, \Delta_2', \Delta_2'' \longrightarrow C$ by induction hypothesis on $\mathcal{D}''$ and $\mathcal{D}_2'$.

```
adm_lolli_r+lolli_l : adm (lolli_r D11) (lolli_l D21 D22) D
                        <- ({x:lin A1} adm (D11 x) D22 (D' x))
                        <- adm D21 D' D.
```

(imp_r, imp_l) on A

$$\mathcal{D}_1 = \dfrac{\mathcal{D}_1'}{\dfrac{\Gamma, A_1;\Delta_1 \longrightarrow A_2}{\Gamma;\Delta_1 \longrightarrow A_1 \to A_2}} \quad \mathbf{X} \quad \mathcal{D}_2 = \dfrac{\mathcal{D}_2' \qquad \mathcal{D}_2''}{\dfrac{\Gamma;\cdot \longrightarrow A_1 \quad \Gamma;\Delta_2, A_2 \longrightarrow C}{\Gamma;\Delta_2, A_1 \to A_2 \longrightarrow C}}$$

with $A = A_1 \to A_2$.

$\mathcal{D}'' :: \Gamma, A_1;\Delta_1, \Delta_2 \longrightarrow C$ by induction hypothesis on $\mathcal{D}_1'$ and $\mathcal{D}_2''$,
$\mathcal{D} :: \Gamma;\Delta_1, \Delta_2 \longrightarrow C$ by induction hypothesis (ii) on $\mathcal{D}''$ and $\mathcal{D}_2'$.

```
adm_imp_r+imp_l     : adm (imp_r D11) (imp_l D21 D22) D
                        <- ({x:int A1} adm (D11 x) D22 (D' x))
                        <- adm! D21 D' D.
```

with_r

$$\dfrac{\mathcal{D}_1}{\Gamma;\cdot \longrightarrow A} \quad \mathbf{X} \quad \mathcal{D}_2 = \dfrac{\mathcal{D}_2' \qquad \mathcal{D}_2''}{\dfrac{\Gamma, A;\Delta \longrightarrow C_1 \quad \Gamma, A;\Delta \longrightarrow C_2}{\Gamma, A;\Delta \longrightarrow C_1 \,\&\, C_2}\,\mathbf{with\_r}}$$

with $C = C_1 \,\&\, C_2$.

$\mathcal{D}' :: \Gamma;\Delta \longrightarrow C_1$ by induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}_2'$,
$\mathcal{D}'' :: \Gamma;\Delta \longrightarrow C_2$ by induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}_2''$,
$\mathcal{D} :: \Gamma;\Delta \longrightarrow C_1 \,\&\, C_2$ by rule **with_r** on $\mathcal{D}'$ and $\mathcal{D}''$.

```
adm!_*-with_r : adm! D1 ([x:int A] with_r (D21 x , D22 x)) (with_r (D' , D''))
                  <- adm! D1 D21 D'
                  <- adm! D1 D22 D''.
```

lolli_l

$$\dfrac{\mathcal{D}_1}{\Gamma;\cdot \longrightarrow A} \quad \mathbf{X} \quad \mathcal{D}_2 = \dfrac{\mathcal{D}_2' \qquad \mathcal{D}_2''}{\dfrac{\Gamma, A;\Delta' \longrightarrow B_1 \quad \Gamma, A;\Delta'', B_2 \longrightarrow C}{\Gamma, A;\Delta', \Delta'', B_1 \multimap B_2 \longrightarrow C}\,\mathbf{lolli\_l}}$$

with $C = C_1 \,\&\, C_2$.

$\mathcal{D}' :: \Gamma; \Delta' \longrightarrow B_1$ by induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}'_2$,
$\mathcal{D}'' :: \Gamma; \Delta'', B_2 \longrightarrow C$ by induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}''_2$,
$\mathcal{D} :: \Gamma; \Delta', \Delta'', B_1 \multimap B_2 \longrightarrow C$ by rule **lolli_l** on $\mathcal{D}'$ and $\mathcal{D}''$.

```
adm!_*-lolli_l: adm! D1 ([x:int A] lolli_l (D21 x) (D22 x) Y) (lolli_l D' D'' Y)
                <- adm! D1 D21 D'
                <- ({y: lin B2} adm! D1 ([x:int A] D22 x y) (D'' y)).
```

clone on A

$$\frac{\mathcal{D}_1}{\Gamma; \cdot \longrightarrow A} \quad \mathbf{X} \quad \mathcal{D}_2 = \frac{\begin{array}{c}\mathcal{D}'_2 \\ \Gamma, A; \Delta, A \longrightarrow C\end{array}}{\Gamma, A; \Delta \longrightarrow C}\,\mathbf{clone}$$

$\mathcal{D}' :: \Gamma; \Delta, A \longrightarrow C$ by induction hypothesis on $\mathcal{D}_1$ and $\mathcal{D}'_2$,
$\mathcal{D} :: \Gamma; \Delta \longrightarrow C$ by induction hypothesis (i) on $\mathcal{D}_1$ and $\mathcal{D}'$.

```
adm!_*+clone  : adm! D1 (clone D21) D
                 <- ({x:lin A} adm! D1 ([y:int A] D21 y x) (D' x))
                 <- adm D1 D' D.
```

☑

A questo punto, il teorema di eliminazione del taglio per la logica delle formule di Harrop ereditarie lineari si ottiene con una semplice induzione.

Teorema 6.4.3 (*Eliminazione del taglio*)

Per ogni derivazione $\mathcal{D}$, eventualmente contenente tagli, di $\Gamma; \Delta \longrightarrow A$, esiste una derivazione priva di tagli $\mathcal{D}'$ di $\Gamma; \Delta \longrightarrow A$. □

Viene rappresentato mediante la famiglia di tipi:

```
ce:  pr+ A -> pr A -> type.
```

6.5 Mahjongg

Come nostro ultimo esempio, facciamo vedere la codifica in *LLF* di un gioco, *Mahjongg*, popolare fra gli utenti di workstation. Il gioco consiste di 144 *tasselli* suddivisi nei 36 *tipi* mostrati in Figura 6.8: vi sono quattro tasselli per ognuno dei tipi mostrati, con l'eccezione delle stagioni e dei vegetali, che costituiscono due tipi, ma con diverse istanze.

Il gioco inizia in una configurazione iniziale quale quella mostrata in Figura 6.9. Lo scopo del gioco è quello di accoppiare tasselli dello stesso tipo, togliendoli dalla tavola di gioco, fintantoché quest'ultima risulta vuota. Chiaramente, vi sono dei vincoli: non si può usare un tassello

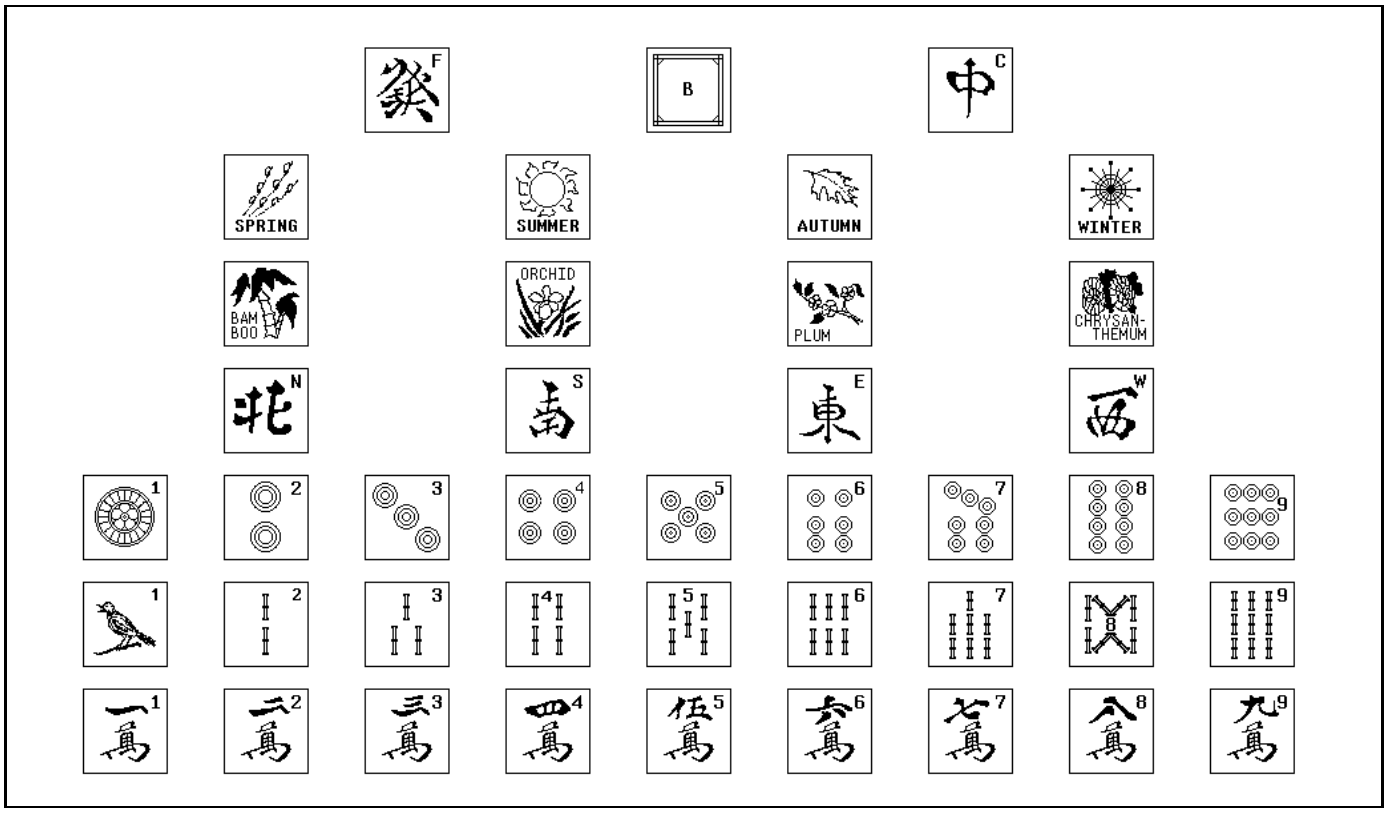


Figura 6.8: Tasselli del Gioco Originale Mahjongg

sormontato da un altro tassello e inoltre un tassello è utilizzabile solamente se è libero a destra oppure a sinistra.

Il trattamento di quest'esempio in *LLF* mostra alcune nuove possibilità offerte dal nostro linguaggio. Partiamo con la parte piú semplice, ossia la codifica dei tasselli. Ci basiamo su tre famiglie di tipi, le quali rappresentano i tipi di tasselli, le istanze all'interno di ciascun tipo, e i tasselli in sè. Abbiamo le seguenti dichiarazioni:

```
tp :  type.
inst :  type.
tle :  type.
```

Formiamo i tasselli individuali accoppiando un tipo e un'istanza per mezzo del costruttore `tile`, definito dalla seguente dichiarazione:

```
tile :  tp -> inst -> tle.
```

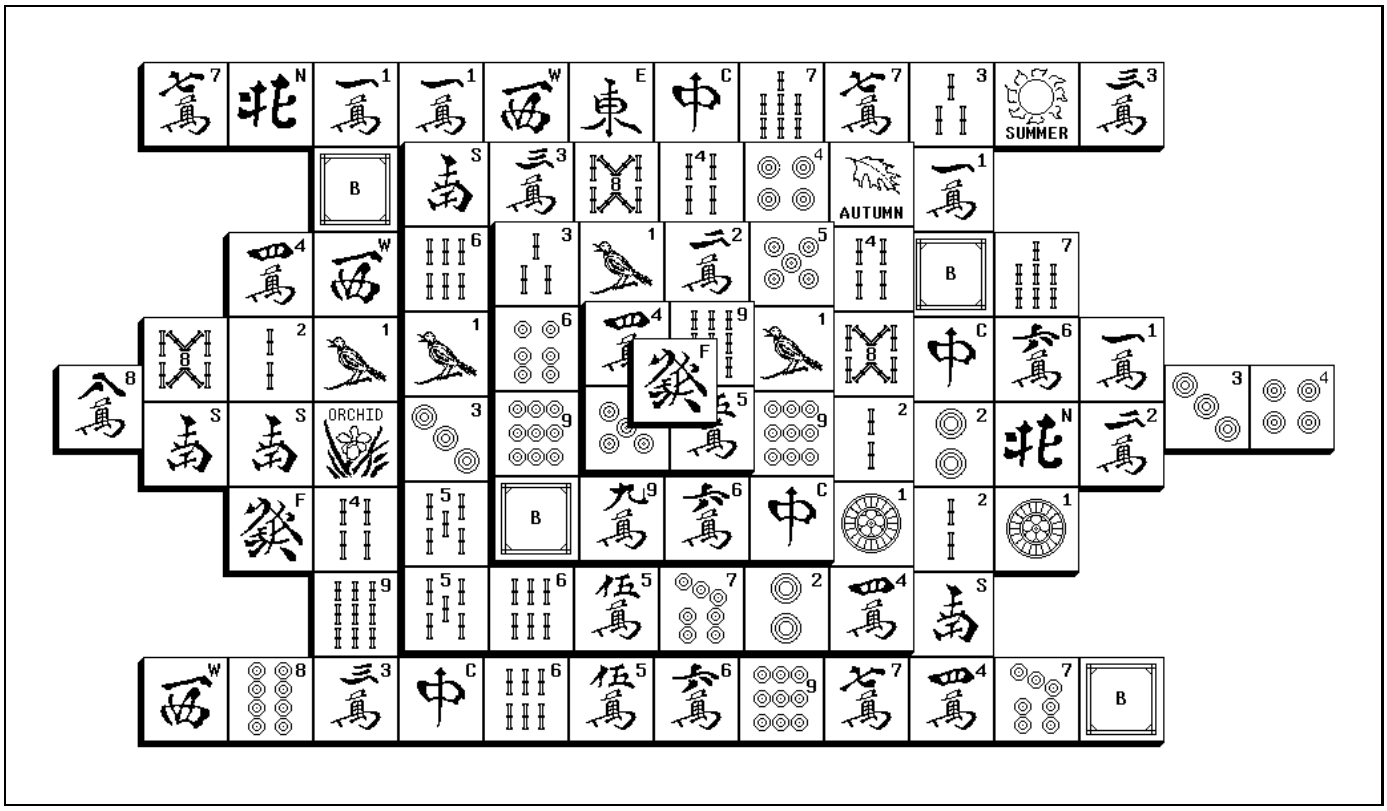


Figura 6.9: Una Configurazione Iniziale in Mahjongg

Passiamo ora alla rappresentazione del gioco vero e proprio. Assumiamo di avere la codifica dei tasselli di una partita. Rappresentiamo ciascun tassello, durante il gioco, tramite una dichiarazione nel contesto. Iniziamo con i tasselli che sono stati rimossi dalla tavola di gioco: li codifichiamo mediante assunzioni nella parte intuizionistica del contesto di *LLF* che marchiamo con la costante `out`. Abbiamo la seguente dichiarazione:

```
out :  tle -> type.
```

Chiaramente, queste assunzioni vengono fatte dinamicamente durante lo svolgersi di una partita a *Mahjongg*.

Non rappresentiamo i tasselli presenti sulla tavola di gioco in sè ma i *vincoli* a cui sono sottoposti. A questo scopo, ci serve la costante `in`, dichiarata come segue:

```
in :  tle -> type.
```

Per ogni tassello presente sul tavolo di gioco, abbiamo una dichiarazione nel contesto lineare che esprime i vincoli a cui è sottoposto. Pertanto, diversamente dagli esempi esaminati in precedenza,

il contesto lineare non contiene semplicemente dati, ma procedure. Piú precisamente, rappresentiamo un tassello t sormontato dai tasselli $s_1, \ldots, s_n$, e vincolato a destra e a sinistra dai tasselli $r_1, \ldots, r_m$ e $l_1, \ldots, l_p$ rispettivamente mediante l'assunzione lineare

```
<linear> t :    (in ⌜t⌝   <- out ⌜s1⌝ <- ... <- out ⌜sn⌝
                          <- out ⌜r1⌝ <- ... <- out ⌜rm⌝)
                & (in ⌜t⌝   <- out ⌜s1⌝ <- ... <- out ⌜sn⌝
                          <- out ⌜l1⌝ <- ... <- out ⌜lp⌝)
```

dove i singoli tasselli sono rappresentati come prima. Ognuna di queste clausole può venire utilizzata solamente se uno dei due congiunti è dimostrabile, ossia se tutti i tasselli sovrastanti t sono stati rimossi e i tutti i tasselli a destra oppure a sinistra sono stati tolti dalla scacchiera. Quando queste condizioni sono verificate, la clausola può venire utilizzata per il gioco. Essendo lineare, verrà eliminata in quel momento dal contesto. L'idea è allora di rimpiazzarla con un'assunzione del tipo `out` $\ulcorner t \urcorner$. Il gioco in sè viene allora rappresentato dalle seguenti due clausole:

```
match :  play
            o- in (tile T N1)
            o- in (tile T N2)
            o- (out (tile T N1)
                -> out (tile T N2)
                -> match).
done  :  play.
```

dove `play` è dichiarato come

```
play :  type.
```

Si noti che la seconda clausola può avere successo solamente quanto il contesto lineare è vuoto, e quindi rappresenta il termine del gioco.

Possiamo quindi presentare il teorema di adeguatezza per la rappresentazione proposta. Chiaramente, essendo l'intera discussione informale, anche l'enunciato di questo teorema di rifà a nozioni intuitive. Si noti che stiamo operando al livello semantico, ove la sintassi è consiste della forma dei tasselli ed è banale.

Teorema 6.5.1 (*Adeguatezza della rappresentazione di Mahjongg*)

Dati i tipi di tasselli $c_1, \ldots, c_p$, le istanze $i_1, \ldots, i_q$, una configurazione iniziale consistente dei tasselli $t_1, \ldots, t_n, \ldots, t_{n+m}$ dove ogni t_i è sottoposta ai vincoli $C(t_i)$, un configurazione corrente tale che i tasselli $t_1, \ldots, t_n$ sono già state rimosse dalla scacchiera mentre i tasselli $t_{n+1}, \ldots, t_{n+m}$ vi sono ancora presenti, esiste una biiezione composizionale tra proseguimenti vincenti della partita corrente e termini LLF *canonici M tali che il giudizio*

$$\begin{bmatrix} c_1\texttt{:tp}, & \ldots, & c_p\texttt{:tp}, \\ i_1\texttt{:inst}, & \ldots, & i_q\texttt{:inst}, \\ x_1\texttt{:out}\ulcorner t_1 \urcorner, & \ldots, & x_n\texttt{:out}\ulcorner t_n \urcorner, \\ x_{n+1} \hat{:} \ulcorner C(t_{n+1}) \urcorner, & \ldots, & y_{n+m} \hat{:} \ulcorner C(t_{n+m}) \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \texttt{play}$$

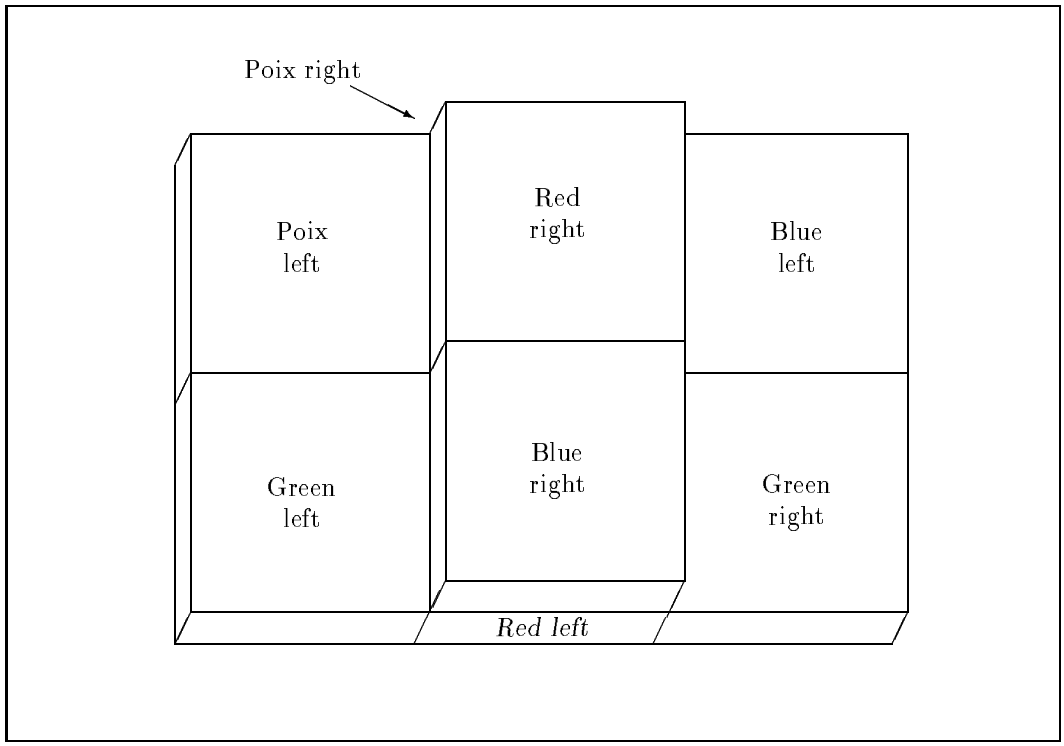


Figura 6.10: Una Configurazione Iniziale nel Problema dell'Accoppiamento dei Calzini

è derivabile. □

A titolo illustrativo, mostriamo l'esempio concreto data dalla configurazione iniziale in Figura 6.10. In questa figura si è voluto rappresentare il comune problema dell'appaiamento di quattro paia di calzini al momento in cui escono da una lavatrice. Abbiamo le seguenti dichiarazioni per i tipi di calzini (blu, rosso, verde e à poix), le loro istanze (destro o sinistro) e i vincoli a cui sono sottoposto come mostrato in Figura:

```
% Types of tiles.

blue  : tp.
red   : tp.
green : tp.
poix  : tp.
```

```
% Instances of each tile

left  : inst.
right : inst.


% Initial configuration

<linear> blue_r  :   in (tile blue right).
<linear> green_l :   in (tile green left)
                   & (in (tile green left)
                        <- out (tile red left)).
<linear> red_l   :   (in (tile red left)
                        <- out (tile blue right)
                        <- out (tile green left))
                   & (in (tile red left)
                        <- out (tile blue right)
                        <- out (tile green right)).
<linear> green_r :   (in (tile green right)
                        <- out (tile red left))
                   & in (tile green right).

<linear> red_r   :   in (tile red right).
<linear> poix_l  :   in (tile poix left)
                   & (in (tile poix left)
                        <- out (tile poix right)).
<linear> poix_r  :   (in (tile poix right)
                        <- out (tile red right)
                        <- out (tile poix left))
                   & (in (tile poix right)
                        <- out (tile red right)
                        <- out (tile blue left)).
<linear> blue_l  :   (in (tile blue left)
                        <- out (tile poix right))
                   & in (tile blue left).
```

Abbiamo la seguente soluzione esemplificativa

```
green_l <-> green_r
blue_l  <-> blue_r
red_l   <-> red_r
poix_r  <-> poix_l
```

per la quale viene calcolato il seguente proof-term:

```
                          match (<fst> green_l)       (<snd> green_r)
[gl:out (tile green left)]
[gr:out (tile green right)] match (<snd> blue_l)         blue_r
[bl:out (tile blue left)]
[br:out (tile blue right)]  match ((<fst> red_l) gl rb)  red_r
[rl:out (tile red left)]
[rr:out (tile red right)]   match ((<snd> poix_r) bl rr) (<fst> poix_l)
[pr:out (tile poix right)]
[pl:out (tile poix left)]   done
```

Parte III

Conclusioni

Capitolo 7

Conclusioni e Sviluppi Futuri

In questo capitolo conclusivo, riepiloghiamo il lavoro fatto e prospettiamo gli sviluppi che ne prenderanno origine nel futuro immediato e non.

7.1 Conclusions

In questa tesi, abbiamo presentato il logical framework lineare LLF. Dopo aver analizzato lo stato dell'arte, abbiamo messo in evidenza i punti deboli dei logical framework correnti: essendo basati su logiche e teorie dei tipi intuizionistiche, non permettono una rappresentazione efficiente di linguaggi basati su stati o risorse come i comuni linguaggi di programmazione imperativi. Questo li rende inutilizzabili in ambito industriale. Proponiamo di ovviare a questa defficienza basando il logical framework LLF su una teoria dei tipi lineare. Le caratteristiche intrinseche della logica lineare offre strumenti per modellare in maniera naturale ed effettiva la nozione di stato e quella di risorsa.

Come ogni logical framework, LLF consiste di due parti. Abbiamo innanzitutto un linguaggio formale, in questo caso una teoria dei tipi lineare, che permette di dare una rappresentazione ai giudizi e derivazioni di un linguaggio oggetto. In secondo luogo, abbiamo una metodologia di rappresentazione che ci guida nella formalizzazione delle entità del livello oggetto. Sia la teoria dei tipi che la metodologia di rappresentazione di LLF estendono in maniera conservativa quanto disponibile nel logical framework LF. Questo ha l'effetto che ogni specifica scritta fino ad oggi in LF è corretta ed ha lo stesso significato in LLF.

Abbiamo messo in evidenza le nuove possibilità di meta-rappresentazione di LLF mostrando come si possono codificare alcuni problemi che sfuggono allo spettro di applicabilità di LF. In particolare, abbiamo mostrato come rappresentare nel nostro formalismo comuni costrutti di programmazione disponibili nei linguaggi imperativi, logiche basate su risorse, e giochi caratterizzati da uno stato interno.

7.2 Sviluppi Futuri

Vorremmo continuare il lavoro iniziato con *LLF* in varie direzioni. In un immediato futuro, siamo interessati a migliorarne la meta-teoria, ad implementare il linguaggio e ad usarlo per ulteriori compiti di meta-rappresentazione. Abbiamo in mente anche alcuni sviluppi a lunga scadenza.

7.2.1 Aspetti Teorici

In questa tesi, abbiamo descritto la meta-teoria di *LLF* partendo da un sistema pre-canonico, caratterizzato dal fatto che ogni termine in esso derivabile è in forma η-espansa. Abbiamo motivato questo fatto dicendo che solamente termini del genere (anzi in forma canonica) vengono utilizzati per fare proof-search e nei teoremi di adeguatezza. Sebbene ciò sia vero, obbligare il programmatore a scrivere le sue segnature e query in forma canonica, o per lo meno η-lunga, è vincolante. Siamo convinti della possibilità pratica di rimuovere questo vincolo. Vorremmo formalizzarla definendo un sistema privo di vincoli e un algoritmo di conversion in forma η-espansa, che esiste già al livello proposizionale. Sfortunatamente, questa generalizzazione porta a notevoli complicazioni meta-teoriche.

Vorremo inoltre studiare la possibilità di mantenere tipi e termini impliciti nel nostro linguaggio e di ricostruirli automaticamente. Confidiamo sul fatto che risultati simili a quanto disponibili in *LF* valgano anche in *LLF*.

7.2.2 Implementazione

Uno dei nostro obiettivi prioritari è quello di costruire un'implementazione efficiente di *LLF*. In un primo momento, implementeremo un interprete che combina le tecniche impiegate nella corrente versione di *Elf* con le idee che abbiamo utilizzato nell'implementazione delle ultime versioni del linguaggio di programmazione logica *Lolli* [CHP96]. Piú stimolante ed impegnativo è il tentativo di definire un compilatore nello stile della *Warren Abstract Machine* (*WAM*). I lavori inizieranno nei prossimi giorni e ci prospettiamo di raggiungere il nostro scopo entro la fine dell'anno.

7.2.3 Applicazioni

Contemporaneamente all'implementazione di *LLF*, intendiamo sviluppare ulteriori applicazioni che mostrano le potenzialità del nostro linguaggio. Vorremo in particolare verificare se *LLF* è in grado di dare rappresentazioni effettive a sistemi intrinsecamente concorrenti o non-deterministici quali linguaggi di programmazione o specifica paralleli e algoritmi basati su riscrittura quale l'unificazione.

Vorremo inoltre collaudare *LLF* su problemi di semantica contestuale e sistemi logici complessi.

7.2.4 Oltre *LLF*

LLF è adeguato alla rappresentazione di formalismi che si basano pesantemente su una nozione di stato che viene modificato al procedere della computazione. La possibilità di vedere *LLF* come un linguaggio di programmazione logica permette di ottenere implementazioni compatte. Tuttavia, se l'obiettivo principale è l'efficienza, conviene rappresentare lo stato con opportune strutture di dati in un linguaggio di programmazione imperativo. Ovviamente, dovremmo implementare anche un motore inferenziale, con il possibile effetto di rendere il programma risultante molto complesso. Un promettente argomento di ricerca è la combinazione di questi due approcci: vorremo avere un compilatore che trasformi codice *LLF* di alto livello in un linguaggio intermedio che offre le strutture di dati solite di un linguaggio di programmazione imperativo, e le operazioni necessarie per accedervi. In particolare, molte applicazioni usano il contesto lineare come strutture di tipo vettore. Una volta riconosciuta una situazione di questo tipo, l'idea è di compilare parte del contesto come un vero vettore e le dichiarazioni che operano sulle singole assunzioni come operazioni di accesso al vettore. Una soluzione positiva a questo problema avrebbe l'effetto da un lato di migliorare l'efficienza di *LLF* e dall'altra di gettare nuova luce sulla connessione tra programmi imperativi e dichiarativi.

Molte classi di dimostrazioni costruttive si possono rappresentare facilmente in *LLF*. In particolare, dimostrazioni induttive basate su oggetti ricorsivamente definiti sono comuni quando si mettono in relazioni due formalismi (ad esempio tramite teoremi di correttezza e completezza). Grazie agli algoritmi di ricostruzione dei termini, è spesso piú veloce codificare queste dimostrazioni come programmi *Elf* o *LLF* che scriverle su carta. L'idea è allora di avere uno strumento, chiamato *schema checker*, che verifichi se l'insieme di clausole risultante è completo rispetto alle dichiarazioni del sistema formale rappresentato. Uno studio formale di queste problematiche nel caso di *Elf* è stato intrappreso e i primi risultati sono descritti in [Roh96, PR96]. Nessuno studio è ancora stato iniziato per il caso lineare sebbene darebbe origine a strumenti pratici altrettanto utili.

Una volta verificata da uno schema checker, sarebbe utile che una dimostrazione codificata come programma *Elf* o *LLF* potesse venire automaticamente trascritta nella notazione matematica consueta. Uno strumento del genere, chiamato *generatore di dimostrazione orientate all'uomo*, è desiderabile per due motivi: innanzitutto perché eviterebbe l'onere di trascrivere esplicitamente dimostrazioni, il ché è cruciale per lunghe dimostrazioni contenenti centinaia di casi, in cui è facilissimo fare errori di distrazione. In secondo luogo, il formato di output potrebbe essere progettato in modo tale da diventare uno standard. Questo sarebbe perticolarmente utile se *LLF* o una sua evoluzione prendesse piede nell'industria per applicazioni di larga scala, quale la specifica della correttezza di compilatori per linguaggi di programmazione imperativi. Si è tantato di includere strumenti ad hoc in *Elf* per ottenere traduzioni di clausole in regole di deduzione per mezzo di LaTeX, con un certo successo. Altri autori hanno prodotto dimostrazioni semi-formali in linguaggio naturale partendo da specifiche formali. I risultati ottenuti sono stati piú che soddisfacenti. L'applicazione di questi risultati al caso specifico di *LLF* non è ancora stato intrappreso, sebbene sarebbe altamente utile.

In un interprete per un comune linguaggio di programmazione logica, l'utente dà una query e si aspetta una risposta tramite sostituzioni a termine della computazione. Pertanto l'unità di interazione è la query. Le implementazioni concrete forniscono predicati di I/O che permettono all'utente di interagire con la computazione durante l'elaborazione di una query. Sfortunatamente, i predicati di I/O non hanno una semantica dichiarativa chiara. In questa tesi, abbiamo implicitamente adottato questo punto di vista parlando di *LLF*. Tuttavia, ciò non è pienamente soddisfacente: se raffiguriamo il contesto lineare come una rappresentazione dello stato del problema in esame, è sensato che l'utente abbi accesso alla computazione mentre lo stato viene trasformato. Si supponga che si voglia modellare un gioco a due giocatori, gli scacchi ad esempio rappresentando la scacchiera nel contesto lineare. Una volta calcolata un mossa, sarebbe opportuno che il sistema si fermasse, permettesse all'utente di accedere al contesto lineare (o almeno ad una sua parte) e aspettasse la mossa dell'utente. Questo non è correntemente possibile. Questo comportamento richiede un modello computazionale in cui il sistema e l'utente sono due *processi concorrenti* che comunicano a precisi *punti di sincronizzazione*. Questa visioni si potrebbe ampliare vedendo i singoli cammini in un albero di ricerca come processi interagenti. Sebbene tentante, questo modello di computazione apre numerosi interrogativi. Innanzitutto, impone pesanti restrizioni sulla nozione di backtracking e in generale sulla strategia di risoluzione del non-determinismo disgiuntivo: infatti, il backtracking non deve essere permesso dopo alcun punto di sincronizzazione in quanto darebbe origine al backtracking di processi indipendenti (si pensi ad un programma che fa una mossa a scacchi e poi cambia idea alcune giocate dopo). Pertanto, alcuni punti si scelta devono essere definitivi. Secondo, dovremmo dare uno statuto logico alla nozione di punto di sincronizzazione. Terzo, dobbiamo cambiare il nostro punto di vista sui proof-tree, attualmente statici, in quello di un oggetto che evolve interagendo con altri processi.

Siamo orientati a pensare che questi ultimi problemi siano risolubili in un sistema che differerice da *LLF* per la possibilità di avere piú di una formula o tipo nella conclusione di un giudizio. Nell'ambito logico, sistemi di questo tipo sono già stati studiati e hanno dato origine ai linguaggi di programmazione logica *Forum* [Mil94] e *Lygon* [HW95]. Questioni di questo tipo sono state studiate nella teoria dei tipi da Parigot. Investigazioni nella logica lineare sono state fatte da Girard in [Gir91, Gir93].

Appendice A

Type Preservation for *Mini-ML*

In this appendix, we have collected the *Elf* code and the adequacy theorems for the meta-representation example presented in Chapter 4.

A.1 *Elf* code for *Mini-ML*

In this section, we give the complete *Elf* code of the type preservation theorem for *Mini-ML*, presented in Chapter 4. More precisely, we show the representation of the syntax of this language in A.1.1, the code of its typing rules in A.1.2, the implementation of the continuation based evaluation procedure in A.1.3 and, at least, the meta-representation of the type preservation theorem for this language in Subsection A.1.4.

A.1.1 Syntax

```
exp    : type.  %name exp    E V
tp     : type.  %name tp     T
instr  : type.  %name instr  I
cont   : type.  %name cont   K


% Expressions

z       : exp.
s       : exp -> exp.
case    : exp -> exp -> (exp -> exp) -> exp.
pair    : exp -> exp -> exp.
fst     : exp -> exp.
snd     : exp -> exp.
lam     : (exp -> exp) -> exp.
app     : exp -> exp -> exp.
letval  : exp -> (exp -> exp) -> exp.
letname : exp -> (exp -> exp) -> exp.
fix     : (exp -> exp) -> exp.
```

```
% Types

nat   : tp.
cross : tp -> tp -> tp.                          %infix   right 90  cross
arrow : tp -> tp -> tp.                          %infix   right 100 arrow


% Instructions

eval : exp -> instr.
return : exp -> instr.
case*  : exp -> exp -> (exp -> exp) -> instr.
pair*  : exp -> exp -> instr.
fst*   : exp -> instr.
snd*   : exp -> instr.
app*   : exp -> exp -> instr.


% Continuations

init : cont.
-    : cont -> (exp -> instr) -> cont.           %infix left 80 -
```

A.1.2 Typing

```
tpe : exp -> tp -> type.          %name tpe Te
tpi : instr -> tp -> type.        %name tpi Ti
tpK : cont -> tp -> tp -> type.   %name tpK TK

%mode -tpe +E *Te
%lex E

%mode -tpi +I *Ti
%lex I

%mode -tpK +K *T1 *T2
%lex K


% Expressions

tpe_z       : tpe z nat.
tpe_s       : tpe (s E) nat
                <- tpe E nat.
tpe_case    : tpe (case E E1 E2) T
```

```
              <- tpe E nat
              <- tpe E1 T
              <- ({x:exp} tpe x nat -> tpe (E2 x) T).
tpe_pair    : tpe (pair E1 E2) (T1 cross T2)
              <- tpe E1 T1
              <- tpe E2 T2.
tpe_fst     : tpe (fst E) T1
              <- tpe E (T1 cross T2).
tpe_snd     : tpe (snd E) T2
              <- tpe E (T1 cross T2).
tpe_lam     : tpe (lam E) (T1 arrow T2)
              <- ({x:exp} tpe x T1 -> tpe (E x) T2).
tpe_app     : tpe (app E1 E2) T1
              <- tpe E1 (T2 arrow T1)
              <- tpe E2 T2.
tpe_letval  : tpe (letval E1 E2) T2
              <- tpe E1 T1
              <- ({x:exp} tpe x T1 -> tpe (E2 x) T2).
tpe_letname : tpe (letname E1 E2) T
              <- tpe (E2 E1) T.
tpe_fix     : tpe (fix E) T
              <- ({x:exp} tpe x T -> tpe (E x) T).


% Instructions

tpi_eval    : tpi (eval E) T
              <- tpe E T.
tpi_return  : tpi (return V) T
              <- tpe V T.
tpi_case*   : tpi (case* V E1 E2) T
              <- tpe V nat
              <- tpe E1 T
              <- ({x:exp} tpe x nat -> tpe (E2 x) T).
tpi_pair*   : tpi (pair* V E) (T1 cross T2)
              <- tpe V T1
              <- tpe E T2.
tpi_fst*    : tpi (fst* V) T1
              <- tpe V (T1 cross T2).
tpi_snd*    : tpi (snd* V) T2
              <- tpe V (T1 cross T2).
tpi_app*    : tpi (app* V E) T1
              <- tpe V (T2 arrow T1)
              <- tpe E T2.


% Continuations
```

```
tpK_init : tpK init T T.
tpK_lam  : tpK (K - I) T1 T2
            <- ({x:exp} tpe x T1 -> tpi (I x) T)
            <- tpK K T T2.
```

A.1.3 Evaluation

```
ev : cont -> instr -> exp -> type.   %name ev E

%mode -ev +C +I -A
%lex C


% Expressions

ev_z        : ev K (eval z) A
               <- ev K (return z) A.
ev_s        : ev K (eval (s E)) A
               <- ev (K - [x:exp] return (s x)) (eval E) A.
ev_case     : ev K (eval (case E E1 E2)) A
               <- ev (K - [x:exp] case* x E1 E2) (eval E) A.
ev_pair     : ev K (eval (pair E1 E2)) A
               <- ev (K - [x:exp] pair* x E2) (eval E1) A.
ev_fst      : ev K (eval (fst E)) A
               <- ev (K - [x:exp] fst* x) (eval E) A.
ev_snd      : ev K (eval (snd E)) A
               <- ev (K - [x:exp] snd* x) (eval E) A.
ev_lam      : ev K (eval (lam E)) A
               <- ev K (return (lam E)) A.
ev_app      : ev K (eval (app E1 E2)) A
               <- ev (K - [x:exp] app* x E2) (eval E1) A.
ev_letval   : ev K (eval (letval E1 E2)) A
               <- ev (K - [x:exp] eval (E2 x)) (eval E1) A.
ev_letname  : ev K (eval (letname E1 E2)) A
               <- ev K (eval (E2 E1)) A.
ev_fix      : ev K (eval (fix E)) A
               <- ev K (eval (E (fix E))) A.


% Values

ev_init     : ev init (return V) V.
ev_cont     : ev (K - I) (return V) A
               <- ev K (I V) A.
```

```
% Auxiliary instructions

ev_case*1   : ev K (case* z E1 E2) A
               <- ev K (eval E1) A.
ev_case*2   : ev K (case* (s V) E1 E2) A
               <- ev K (eval (E2 V)) A.
ev_pair*    : ev K (pair* V E) A
               <- ev (K - [x:exp] return (pair V x)) (eval E) A.
ev_fst*     : ev K (fst* (pair V1 V2)) A
               <- ev K (return V1) A.
ev_snd*     : ev K (snd* (pair V1 V2)) A
               <- ev K (return V2) A.
ev_app*     : ev K (app* (lam E1) E2) A
               <- ev (K - [x:exp] eval (E1 x)) (eval E2) A.
```

A.1.4 Type Preservation

```
pr : ev K I A -> tpi I T -> tpK K T T' -> tpe A T' -> type.

%mode -pr +E +Ti +TK -Ta
%lex E


% Expressions

pr-ev_z        : pr (ev_z E) (tpi_eval tpe_z) TK Ta
                  <- pr E (tpi_return tpe_z) TK Ta.
pr-ev_s        : pr (ev_s E) (tpi_eval (tpe_s Te)) TK Ta
                  <- pr E
                        (tpi_eval Te)
                        (tpK_lam TK [x][t:tpe x nat] tpi_return (tpe_s t))
                        Ta.
pr-ev_case     : pr (ev_case E) (tpi_eval (tpe_case Te'' Te' Te)) TK Ta
                  <- pr E
                        (tpi_eval Te)
                        (tpK_lam TK [x][t:tpe x nat] tpi_case* Te'' Te' t)
                        Ta.
pr-ev_pair     : pr (ev_pair E) (tpi_eval (tpe_pair Te'' Te')) TK Ta
                  <- pr E
                        (tpi_eval Te')
                        (tpK_lam TK [x][t:tpe x T'] tpi_pair* Te'' t)
                        Ta.
pr-ev_fst      : pr (ev_fst E) (tpi_eval (tpe_fst Te)) TK Ta
                  <- pr E
                        (tpi_eval Te)
                        (tpK_lam TK [x][t:tpe x (T' cross T'')] tpi_fst* t)
                        Ta.
```

```
pr-ev_snd      : pr (ev_snd E) (tpi_eval (tpe_snd Te)) TK Ta
                  <- pr E
                        (tpi_eval Te)
                        (tpK_lam TK [x][t:tpe x (T' cross T'')] tpi_snd* t)
                        Ta.
pr-ev_lam      : pr (ev_lam E) (tpi_eval (tpe_lam Te)) TK Ta
                  <- pr E (tpi_return (tpe_lam Te)) TK Ta.
pr-ev_app      : pr (ev_app E) (tpi_eval (tpe_app Te'' Te')) TK Ta
                  <- pr E
                        (tpi_eval Te')
                        (tpK_lam TK [x][t:tpe x (T' arrow T'')]tpi_app* Te'' t)
                        Ta.
pr-ev_letval   : pr (ev_letval E) (tpi_eval (tpe_letval Te'' Te')) TK Ta
                  <- pr E
                        (tpi_eval Te')
                        (tpK_lam TK [x][t:tpe x T'] tpi_eval (Te'' x t))
                        Ta.
pr-ev_letname  : pr (ev_letname E) (tpi_eval (tpe_letname Te)) TK Ta
                  <- pr E (tpi_eval Te) TK Ta.
pr-ev_fix      : {Te:{x} tpe x T -> tpe (X x) T}
                 pr (ev_fix E) (tpi_eval (tpe_fix Te)) TK Ta
                  <- pr E (tpi_eval (Te (fix X) (tpe_fix Te))) TK Ta.


% Values

pr-ev_init     : pr ev_init (tpi_return Tv) tpK_init Tv.
pr-ev_cont     : {Tv: tpe V T}
                 pr (ev_cont E) (tpi_return Tv) (tpK_lam TK Ti) Ta
                  <- pr E (Ti V Tv) TK Ta.


% Auxiliary instructions

pr-ev_case*1   : pr (ev_case*1 E) (tpi_case* Te'' Te' tpe_z) TK Ta
                  <- pr E (tpi_eval Te') TK Ta.
pr-ev_case*2   : {Tv: tpe V nat}
                 pr (ev_case*2 E) (tpi_case* Te'' Te' (tpe_s Tv)) TK Ta
                  <- pr E (tpi_eval (Te'' V Tv)) TK Ta.
pr-ev_pair*    : pr (ev_pair* E) (tpi_pair* Te Tv) TK Ta
                  <- pr E
                        (tpi_eval Te)
                        (tpK_lam TK [x][t:tpe x T'']tpi_return (tpe_pair t Tv))
                        Ta.
pr-ev_fst*     : pr (ev_fst* E) (tpi_fst* (tpe_pair Tv'' Tv')) TK Ta
                  <- pr E (tpi_return Tv') TK Ta.
pr-ev_snd*     : pr (ev_snd* E) (tpi_snd* (tpe_pair Tv'' Tv')) TK Ta
```

```
                  <- pr E (tpi_return Tv'') TK Ta.
pr-ev_app*     : pr (ev_app* E) (tpi_app* Te'' (tpe_lam Te')) TK Ta
                  <- pr E
                        (tpi_eval Te'')
                        (tpK_lam TK [x][t:tpe x T''] tpi_eval (Te' x t))
                        Ta.
```

A.2 Adequacy Theorems

In this section, we have collected all the adequacy theorems for the meta-representation of *Mini-ML*. We do not show the proofs. Here, Σ is the signature displayed in Section A.1.

A.2.1 Syntax

Adequacy theorem A.1 (*Mini-ML expressions*)

There is a compositional bijection $\ulcorner \_ \urcorner$ *between Mini-ML expression with free variables among* $x_1, \ldots, x_n$ *and canonical LF objects* M *such that the judgment*

$$x_1 : \mathtt{exp}, \ldots, x_n : \mathtt{exp} \vdash^{\mathrm{LF}}_{\Sigma} M \Uparrow \mathtt{exp}$$

is derivable. □

Adequacy theorem A.2 (*Mini-ML types*)

There is a compositional bijection $\ulcorner \_ \urcorner$ *between Mini-ML types and canonical LF objects* M *such that*

$$\cdot \vdash^{\mathrm{LF}}_{\Sigma} M \Uparrow \mathtt{tp}$$

is derivable □

Adequacy theorem A.3 (*Mini-ML instructions*)

There is a compositional bijection $\ulcorner \_ \urcorner$ *between Mini-ML instructions with free variables among* $x_1, \ldots, x_n$ *and canonical LF objects* M *such that the judgment*

$$x_1 : \mathtt{exp}, \ldots, x_n : \mathtt{exp} \vdash^{\mathrm{LF}}_{\Sigma} M \Uparrow \mathtt{instr}$$

is derivable. □

Adequacy theorem A.4 (*Mini-ML continuations*)

There is a compositional bijection $\ulcorner \_ \urcorner$ *between Mini-ML continuations with free variables among* $x_1, \ldots, x_n$ *and canonical LF objects* M *such that the judgment*

$$x_1 : \mathtt{exp}, \ldots, x_n : \mathtt{exp} \vdash^{\mathrm{LF}}_{\Sigma} M \Uparrow \mathtt{cont}$$

is derivable. □

A.2.2 Typing

Adequacy theorem A.5 (*Mini-ML expression typing*)

Given a Mini-ML expression e and a type τ, there is a compositional bijection $\ulcorner \_ \urcorner$ between derivations of

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash^e e : \tau$$

and LF objects M such that

$$\begin{bmatrix} x_1 : \mathtt{exp}, & \ldots, & x_n : \mathtt{exp}, \\ t_1 : \mathtt{tpe}\ x_1\ \ulcorner \tau_1 \urcorner, & \ldots, & t_n : \mathtt{tpe}\ x_n\ \ulcorner \tau_n \urcorner \end{bmatrix} \vdash^{\mathrm{LF}}_{\Sigma} M \Uparrow \mathtt{tpe}\ \ulcorner e \urcorner\ \ulcorner \tau \urcorner$$

is derivable. □

Adequacy theorem A.6 (*Mini-ML instruction typing*)

Given a Mini-ML instruction i and a type τ, there is a compositional bijection $\ulcorner \_ \urcorner$ between derivations of

$$x_1 : \tau_1, \ldots, x_n : \tau_n \vdash^i i : \tau$$

and LF objects M such that

$$\begin{bmatrix} x_1 : \mathtt{exp}, & \ldots, & x_n : \mathtt{exp}, \\ t_1 : \mathtt{tpe}\ x_1\ \ulcorner \tau_1 \urcorner, & \ldots, & t_n : \mathtt{tpe}\ x_n\ \ulcorner \tau_n \urcorner \end{bmatrix} \vdash^{\mathrm{LF}}_{\Sigma} M \Uparrow \mathtt{tpi}\ \ulcorner i \urcorner\ \ulcorner \tau \urcorner$$

is derivable. □

Adequacy theorem A.7 (*Mini-ML continuation typing*)

Given a closed Mini-ML continuation K and types τ and τ', there is a compositional bijection $\ulcorner \_ \urcorner$ between derivations of

$$\vdash^K K : \tau \Rightarrow \tau'$$

and LF objects M such that

$$\cdot \vdash^{\mathrm{LF}}_{\Sigma} M \Uparrow \mathtt{tpK}\ \ulcorner K \urcorner\ \ulcorner \tau \urcorner\ \ulcorner \tau' \urcorner$$

is derivable. □

A.2.3 Evaluation

Adequacy theorem A.8 (*Mini-ML evaluation*)

Given a closed Mini-ML continuation K, a closed instruction i, a closed expression v, there is a compositional bijection $\ulcorner \_ \urcorner$ between derivations of

$$K \vdash i \hookrightarrow v$$

and LF objects M such that

$$\cdot \vdash^{\mathrm{LF}}_{\Sigma} M \Uparrow \mathtt{ev}\ \ulcorner K \urcorner\ \ulcorner i \urcorner\ \ulcorner v \urcorner$$

is derivable. □

A.2.4 Type Preservation

Adequacy theorem A.9 (*Mini-ML type preservation*)

Given a closed Mini-ML continuation K, a closed instruction i, a closed expression v, types τ and τ', LF objects E, T_i and T_K such that the judgments

$$\cdot \vdash^{\mathrm{LF}}_{\Sigma} E \Uparrow \mathtt{ev}\ \ulcorner K \urcorner\ \ulcorner i \urcorner\ \ulcorner v \urcorner$$
$$\cdot \vdash^{\mathrm{LF}}_{\Sigma} T_i \Uparrow \mathtt{tpi}\ \ulcorner i \urcorner\ \ulcorner \tau \urcorner$$
$$\cdot \vdash^{\mathrm{LF}}_{\Sigma} T_K \Uparrow \mathtt{tpK}\ \ulcorner K \urcorner\ \ulcorner \tau \urcorner\ \ulcorner \tau' \urcorner$$

are derivable, there exist LF objects T and M such that

$$\cdot \vdash^{\mathrm{LF}}_{\Sigma} T \Uparrow \mathtt{tpe}\ \ulcorner v \urcorner\ \ulcorner \tau' \urcorner$$

and

$$\cdot \vdash^{\mathrm{LF}}_{\Sigma} M \Uparrow \mathtt{pr}\ \ulcorner K \urcorner\ \ulcorner i \urcorner\ \ulcorner \tau \urcorner\ \ulcorner \tau' \urcorner\ \ulcorner v \urcorner\ E\ T_i\ T_K\ T$$

is derivable. □

Appendice B

Proofs from Chapter 5

In this appendix, we collect all the properties needed to support the statements in Chapter 5. We describe how to prove these properties, but we do not go in very deep details in most cases. The main reason for this choice is that the deductive systems in Chapter 5, which judgments these properties relate, are defined by so many rules of inference that a detailed treatment would add several hundred pages to this dissertation.

The numbering of sections and of statements corresponds to the sectioning of Chapter 5, and to the numbering of lemmas and theorems there.

B.1 Language

There are no properties to prove from section 5.1.

B.2 Basic Operations

In this section, we describe the properties that relate the entities presented in Section 5.2, namely sequence concatenation, free variables, substitutions, context splitting and the functions $\overline{\cdots}$ and $\widehat{\cdots}$ that permit to access the intuitionistic and linear part of the context respectively.

We first describe some properties of the functions $\overline{\cdots}$ and $\widehat{\cdots}$.

Lemma 5.2.1 (*Properties of the intuitionistic and the linear part*)

$\overline{\overline{\Psi}} = \overline{\Psi}$, $\widehat{\widehat{\Psi}} = \widehat{\Psi}$ *and* $\overline{\widehat{\Psi}} = \widehat{\overline{\Psi}} = \cdot$.

Proof.

By induction on the structure of Ψ and inversion on the definition of $\overline{\cdots}$ and $\widehat{\cdots}$. ☑

Lemma 5.2.2 (*Intuitionistic/linear part distributes over context concatenation*)

$\overline{\Psi, \Psi'} = \overline{\Psi}, \overline{\Psi'}$ *and* $\widehat{\Psi, \Psi'} = \widehat{\Psi}, \widehat{\Psi'}$.

Proof.

By induction on the structure of Ψ' and inversion on definition of $\overline{\cdots}$ and $\widehat{\cdots}$, respectively. ☑

The following lemmas concern the properties of substitution in relation to the operations of context concatenation, the notion of free variable, and the functions $\overline{\cdots}$ and $\widehat{\cdots}$.

Lemma 5.2.3 (*Substitution respects syntactic categories*)

If M, P, A, K and Ψ are an object, a type family, a type, a kind and a context, respectively, then $[N/x]M$, $[N/x]P$, $[N/x]A$, $[N/x]K$ and $[N/x]\Psi$ are, respectively, an object, a type family, a type, a kind and a context.

Proof.

By induction on the structure of M, P, A, K and Ψ, keeping in mind that N is always an object level term. ☑

Lemma 5.2.4 (*Substitution distributes over context concatenation*)

$[N/x](\Psi, \Psi') = [N/x]\Psi, [N/x]\Psi'$.

Proof.

By induction on the structure of Ψ' and inversion on the definition of substitution for contexts. ☑

Lemma 5.2.5 (*Substitution distributes over the intuitionistic/linear part*)

$\overline{[N/x]\Psi} = [N/x]\overline{\Psi}$ *and* $\widehat{[N/x]\Psi} = [N/x]\widehat{\Psi}$.

Proof.

By induction on the structure of Ψ. ☑

Lemma 5.2.6 (*Substitution for variables non appearing in a term*)

If $x \notin \mathrm{FV}(U)$, then $[N/x]U = U$.

Proof.

By induction on the structure of U. ☑

Lemma 5.2.7 (*Nested substitutions*)

If $y \notin \mathrm{FV}(N)$, then $[N/x]([M/y]U) = [([N/x]M)/y]([N/x]U)$.

Proof.

We proceed by induction on the structure of U. All cases are trivial except the base case in which U is the variable x. We have indeed:

$$\begin{array}{rcll} [N/x]([M/y]x) & = & [N/x]x & \text{by definition of substitution} \\ & = & N & \text{by definition of substitution} \\ & = & [([N/x]M)/y]N & \text{by the previous lemma since } y \notin \mathrm{FV}(N) \\ & = & [([N/x]M)/y]([N/x]x) & \text{by definition of substitution} \end{array}$$

☑

Lemma 5.2.8 (*Substitutions and free variables*)

$\mathrm{FV}([N/x]U) \subseteq \mathrm{FV}(N) \cup (\mathrm{FV}(U) \setminus \{x\})$.

Proof.

By induction on the structure of U. ☑

Finally, we state the main properties of context splitting. The first two lemmas describe how concatenation interacts with splitting. The corollary shows how individual assumptions in the context are manipulated by this judgment, depending on whether they are intuitionistic or linear.

Lemma 5.2.9 (*Concatenation and splitting*)

If $\mathcal{C} :: \Psi', \Psi'' = \Psi_1 \bowtie \Psi_2$, *then* $\Psi_1 = \Psi_1', \Psi_1''$ *and* $\Psi_2 = \Psi_2', \Psi_2''$ *with* $\mathcal{C}' :: \Psi' = \Psi_1' \bowtie \Psi_2'$ *and* $\mathcal{C}'' :: \Psi'' = \Psi_1'' \bowtie \Psi_2''$.

Proof.

By induction on the structure of Ψ'' and inversion on the derivation $\mathcal{C}$. ☑

Corollary 5.2.10 (*Concatenation and splitting*)

i. *If* $\mathcal{C} :: \Psi', x : A, \Psi'' = \Psi_1 \bowtie \Psi_2$, *then* $\Psi_1 = \Psi_1', x : A, \Psi_1''$ *and* $\Psi_2 = \Psi_2', x : A, \Psi_2''$ *with* $\mathcal{C}' :: \Psi' = \Psi_1' \bowtie \Psi_2'$ *and* $\mathcal{C}'' :: \Psi'' = \Psi_1'' \bowtie \Psi_2''$.

ii. *If* $\mathcal{C} :: \Psi', x \hat{:} A, \Psi'' = \Psi_1 \bowtie \Psi_2$, *then either* $\Psi_1 = \Psi_1', x \hat{:} A, \Psi_1''$ *and* $\Psi_2 = \Psi_2', \Psi_2''$ *or* $\Psi_1 = \Psi_1', \Psi_1''$ *and* $\Psi_2 = \Psi_2', x \hat{:} A, \Psi_2''$, *with* $\mathcal{C}' :: \Psi' = \Psi_1' \bowtie \Psi_2'$ *and* $\mathcal{C}'' :: \Psi'' = \Psi_1'' \bowtie \Psi_2''$.

Proof.

Apply the previous lemma on $\Psi', x \hat{:} A$ and Ψ'' and then proceed by inversion on the resulting derivation concerning $\Psi', x \hat{:} A$. ☑

Lemma 5.2.11 (*Concatenation of split contexts*)

If $\mathcal{C}' :: \Psi' = \Psi_1' \bowtie \Psi_2'$ *and* $\mathcal{C}'' :: \Psi'' = \Psi_1'' \bowtie \Psi_2''$, *then* $\mathcal{C} :: \Psi', \Psi'' = \Psi_1', \Psi_1'' \bowtie \Psi_2', \Psi_2''$.

Proof.

By induction on $\mathcal{C}''$. ☑

Next, two lemmas that connect context splitting with substitutions and the concept of free variable.

Lemma 5.2.12 (*Splitting and substitution*)

If $\mathcal{C} :: \Psi = \Psi' \bowtie \Psi''$, *then* $\mathcal{C}^N :: [N/x]\Psi = [N/x]\Psi' \bowtie [N/x]\Psi''$.

Proof.

By induction on $\mathcal{C}$. ☑

Lemma 5.2.13 (*Splitting and free variables*)

If $\mathcal{C} :: \Psi = \Psi' \bowtie \Psi''$, *then* $\mathrm{FV}(\Psi) = \mathrm{FV}(\Psi') \cup \mathrm{FV}(\Psi'')$ *and* $\mathrm{dom}\,\Psi = \mathrm{dom}\,\Psi' \cup \mathrm{dom}\,\Psi''$.

Proof.

By induction on $\mathcal{C}$. ☑

Finally, we present a lemma that describe how assumptions are distributed by the context splitting judgment.

Lemma 5.2.14 (*Splitting and the intuitionistic/linear part*)

If $\mathcal{C} :: \Psi = \Psi' \bowtie \Psi''$, *then*

i. $\operatorname{dom} \overline{\Psi} = \operatorname{dom} \overline{\Psi'} = \operatorname{dom} \overline{\Psi''}$;

ii. $\operatorname{dom} \widehat{\Psi'} \cap \operatorname{dom} \widehat{\Psi''} = \varnothing$.

Proof.

By induction on $\mathcal{C}$. ☑

Corollary 5.2.15 (*Splitting and the intuitionistic/linear part*)

i. If $\mathcal{C} :: \overline{\Psi} = \Psi' \bowtie \Psi''$, *then* $\Psi' = \Psi'' = \overline{\Psi}$;

ii. If $\mathcal{C} :: \widehat{\Psi} = \Psi' \bowtie \Psi''$, *then* $\Psi' \cap \Psi'' = \cdot$.

Proof.

i. Since $\overline{\Psi}$ is intuitionistic, so are Ψ' and Ψ''. Therefore, by the previous lemma $\operatorname{dom} \overline{\Psi} = \operatorname{dom} \overline{\Psi} = \operatorname{dom} \overline{\Psi}$. Consequently, $\overline{\Psi} = \Psi' = \Psi''$.

ii. Similarly, since $\widehat{\Psi}$ does not contain intuitionistic assumptions, the same property holds for Ψ' and Ψ''. Therefore, by the previous lemma, $\operatorname{dom} \widehat{\Psi'} \cap \operatorname{dom} \widehat{\Psi''} = \varnothing$. Consequently, $\Psi' \cap \Psi'' = \cdot$. ☑

B.3 Pre-canonical Forms

This section contains the proofs of the statements presented in Section 5.3, as well as properties of minor importance, but nonetheless needed in those statements.

B.3.1 Presentation

There is nothing to prove from Section 5.3.1.

B.3.2 Equational Theory

In this section, we present the proofs of the statements concerning the definitional equality of LLF presented in Subsection 5.3.2.

We start with a series of minor lemmas about the equational theory. We will use them in proofs in the following sections. First, all relations involved in the definitional equality are reflexive.

Lemma 5.3.1 (*Reflexivity*)

For every term U *the following judgments are derivable:*

i. *If* $U \longrightarrow U$,

ii. *If* $U \longrightarrow^* U$,

iii. *If* $U \equiv U$.

Proof.

We prove (i) by induction on the structure of U. The remaining statements are then trivial. ☑

Next, a technical lemma stating that new free variables are not produced during reduction.

Lemma 5.3.2 (*Free variables and reduction*)

If either $\mathcal{R} :: U \longrightarrow V$ *or* $\mathcal{R} :: U \longrightarrow^* V$, *then* $\mathrm{FV}(U) \subseteq \mathrm{FV}(V)$.

Proof.

We proceed by induction on the structure of $\mathcal{R}$. ☑

The next lemmas serve the purpose of extending the inversion principle that applies to $\longrightarrow$ to its transitive closure $\longrightarrow^*$ and to the equivalence relation $\equiv$.

Lemma 5.3.3 (*Definitional equality is a congruence*)

Let $\sim$ *be one of* $\equiv$ *and* $\longrightarrow^*$. *If* $\mathcal{R} :: M \sim M'$, $\mathcal{R} :: A \sim A'$ *or* $\mathcal{R} :: K \sim K'$, *then it is possible to construct derivations* $\mathcal{R}'$ *for*

i. $\langle M, N\rangle \sim \langle M', N\rangle$, $\textsc{fst}\ M \sim \textsc{fst}\ M'$, $\textsc{snd}\ M \sim \textsc{snd}\ M'$,
$\langle N, M\rangle \sim \langle N, M'\rangle$,
$\hat{\lambda} x\!:\!A.\, M \sim \hat{\lambda} x\!:\!A'.\, M$, $M \mathbin{\hat{}} N \sim M' \mathbin{\hat{}} N$, $\lambda x\!:\!A.\, M \sim \hat{\lambda} x\!:\!A'.\, M$, $M\ N \sim M'\ N$,
$\hat{\lambda} x\!:\!A.\, M \sim \hat{\lambda} x\!:\!A.\, M'$, $N \mathbin{\hat{}} M \sim N \mathbin{\hat{}} M'$, $\lambda x\!:\!A.\, M \sim \hat{\lambda} x\!:\!A.\, M'$, $N\ M \sim N\ M'$,

ii. $A\ M \sim A'\ M$, $A \,\&\, B \sim A' \,\&\, B$, $A \multimap B \sim A' \multimap B$, $\Pi x\!:\!A.\, B \sim \Pi x\!:\!A'.\, B$,
$A\ M \sim A\ M'$, $B \,\&\, A \sim B \,\&\, A'$, $B \multimap A \sim B \multimap A'$, $\Pi x\!:\!B.\, A \sim \Pi x\!:\!B.\, A'$,

iii. $\Pi x\!:\!A.\, K \sim \Pi x\!:\!A'.\, K$,
$\Pi x\!:\!A.\, K \sim \Pi x\!:\!A.\, K'$.

Proof.

We first prove the lemma for $\longrightarrow^*$. We proceed by induction on the structure of $\mathcal{R}$. When analyzing the base rules for transitivity, **or*_r**, **fr*_r** and **fr*_r**, we use reflexivity and apply the appropriate congruence rule from Figure 5.4 to the premiss, obtain the desired conclusion, and then apply again the suitable base rule.

The treatment of $\equiv$ is similar, but this time we connect with the transitive closure relation by means of the property we have just proved for it. ☑

Corollary 5.3.4 (*Definitional equality is a congruence*)

Let $\sim$ *be one of* $\equiv$ *and* $\longrightarrow^*$. *If* $\mathcal{R}_M :: M \sim M'$, $\mathcal{R}_N :: N \sim N'$, $\mathcal{R}_A :: A \sim A'$, $\mathcal{R}_B :: B \sim B'$ *and* $\mathcal{R}_K :: K \sim K'$, *then it is possible to construct derivations* $\mathcal{R}$ *for*

i. $\langle M, N\rangle \sim \langle M', N'\rangle$, $\textsc{fst}\ M \sim \textsc{fst}\ M'$, $\textsc{snd}\ M \sim \textsc{snd}\ M'$,
$\hat{\lambda} x\!:\!A.\, M \sim \hat{\lambda} x\!:\!A'.\, M'$, $M \mathbin{\hat{}} N \sim M' \mathbin{\hat{}} N'$, $\lambda x\!:\!A.\, M \sim \hat{\lambda} x\!:\!A'.\, M'$, $M\ N \sim M'\ N'$,

ii. $A\ N \sim A'\ N'$, $A \,\&\, B \sim A' \,\&\, B'$, $A \multimap B \sim A' \multimap B'$, $\Pi x\!:\!A.\ B \sim \Pi x\!:\!A'.\ B'$,

iii. $\Pi x\!:\!A.\ K \sim \Pi x\!:\!A'.\ K'$.

Proof.

We use the lemma above to reduce one subterm of a binary construct at a time and then chain the resulting derivations by means of transitivity. ☑

Lemma 5.3.5 (*Definitional equality is invertible*)

Let $\sim$ *be one of* $\equiv$ *and* $\longrightarrow^*$. *If any of the following judgments is derivable*

i. $\langle M, N\rangle \sim \langle M', N'\rangle$, $\hat{\lambda} x\!:\!A.\ M \sim \hat{\lambda} x\!:\!A'.\ M'$, $\lambda x\!:\!A.\ M \sim \hat{\lambda} x\!:\!A'.\ M'$,

ii. $A\ N \sim A'\ N'$, $A \,\&\, B \sim A' \,\&\, B'$, $A \multimap B \sim A' \multimap B'$, $\Pi x\!:\!A.\ B \sim \Pi x\!:\!A'.\ B'$,

iii. $\Pi x\!:\!A.\ K \sim \Pi x\!:\!A'.\ K'$

then it is possible to construct derivations for $M \sim M'$, $N \sim N'$, $A \sim A'$, $B \sim B'$ *and* $K \sim K'$.

Proof.

We proceed as in the proof of lemma 5.3.3, but this time we apply inversion when we reach the base case for the transitivity rule. ☑

Notice that in the above lemma, the hypothesis concerning the level of objects only mentions constructors. Indeed, the rules for β-conversion prevent this statement from holding in the case of destructors.

Lemma 5.3.6 (*Shape of equivalent types*)

Let $\sim$ *be one of* $\equiv$ *and* $\longrightarrow^*$.

i. If $a \sim U$ *or* $U \sim a$, *then* $U = a$;

ii. If $\top \sim U$ *or* $U \sim \top$, *then* $U = \top$;

iii. If $A \,\&\, B \sim U$ *or* $U \sim A \,\&\, B$, *then* $U = A' \,\&\, B'$;

iv. If $A \multimap B \sim U$ *or* $U \sim A \multimap B$, *then* $U = A' \multimap B'$;

v. If $\Pi x\!:\!A.\ B \sim U$ *or* $U \sim \Pi x\!:\!A.\ B$, *then* $U = \Pi x\!:\!A'.\ B'$;

vi. If $\Pi x\!:\!A.\ K \sim U$ *or* $U \sim \Pi x\!:\!A.\ K$, *then* $U = \Pi x\!:\!A'.\ K'$.

Proof.

We proceed exactly as in the previous lemma: by induction on the structure of the derivation in the hypothesis, and inversion in the base case. ☑

We now turn our attention to the Church-Rosser theorem. Its proof requires a number of lemmas. We parallel the detailed proof given in [Pfe99]. However, the richer syntax of *LLF* leads to an explosion in the number of cases to analyze.

Lemma 5.3.7 (*Substitution and reduction*)

Assume that there exist a derivation for $N \longrightarrow N'$. *Then,*

i. if $\mathcal{R} :: U \longrightarrow U'$, *then* $\mathcal{R}' :: [N/x]U \longrightarrow [N'/x]U'$;

ii. if $\mathcal{R} :: U \longrightarrow^* U'$, *then* $\mathcal{R}' :: [N/x]U \longrightarrow^* [N'/x]U'$;

iii. if $\mathcal{R} :: U \equiv U'$, *then* $\mathcal{R}' :: [N/x]U \equiv [N'/x]U'$.

Proof.

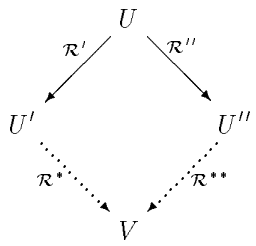
We proceed by induction on $\mathcal{R}$. The only non-trivial cases concern rules **or_beta_lin** and **or_beta_int**. We will expand the treatment of the first.

$$\boxed{\textbf{or\_beta\_lin}} \qquad \mathcal{R} = \dfrac{\begin{array}{cc}\mathcal{R}_1 & \mathcal{R}_2 \\ M_1 \longrightarrow M_1' & M_2 \longrightarrow M_2'\end{array}}{(\hat{\lambda}y{:}A.\,M_1)\,\hat{}\,M_2 \longrightarrow [M_2'/y]M_1'}\,\textbf{or\_beta\_lin}$$

with $U = (\hat{\lambda}y{:}A.\,M_1)\,\hat{}\,M_2$ and $U' = [M_2'/y]M_1'$.

$\mathcal{R}_1' :: [N/x]M_1 \longrightarrow [N'/x]M_1'$	by induction hypothesis on $\mathcal{R}_1$,
$\mathcal{R}_2' :: [N/x]M_2 \longrightarrow [N'/x]M_2'$	by induction hypothesis on $\mathcal{R}_2$,
$\mathcal{R}' :: [N/x]((\hat{\lambda}y{:}A.\,M_1)\,\hat{}\,M_2) \longrightarrow [([N'/x]M_2')/y]([N'/x]M_1')$	by rule **or_beta_lin** on $\mathcal{R}_1'$ and $\mathcal{R}_2'$, and definition of substitution,
$[([N'/x]M_2')/y]([N'/x]M_1') = [N'/x]([M_2'/y]M_1')$	by lemma 5.2.7 since y is local to M_1 (and to M_1') and therefore $y \notin \mathrm{FV}(N)$. ☑

Lemma 5.3.8 (*Diamond property*)



If $\mathcal{R}' :: U \longrightarrow U'$ *and* $\mathcal{R}'' :: U \longrightarrow U''$, *then there is a term* V *such that* $\mathcal{R}^* :: U' \longrightarrow V$ *and* $\mathcal{R}^{**} :: U'' \longrightarrow V$.

Proof.

We proceed by induction on the structure of U and by simultaneous inversion on $\mathcal{R}'$ and $\mathcal{R}''$. The cases involving kinds, types, type families, object constructors and object constants or variables are trivial since they determine the last rule applied in $\mathcal{R}'$ and $\mathcal{R}''$.

The cases in which the topmost operator in U is a destructor are slightly more complex since two rules are applicable, leading to a total of four cases to analyze. We consider the situation in which U is a term of the form $M\,\hat{}\,N$. The cases of the other destructors are treated similarly.

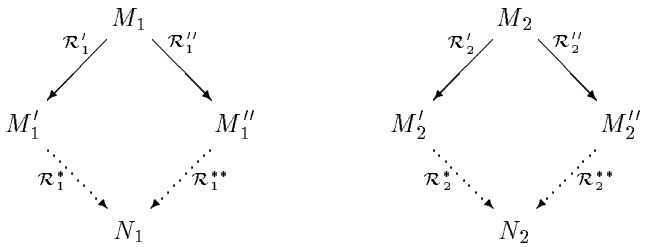
Let $U = M_1\,\hat{}\,M_2$. By inversion on the rules in Figure 5.4, a derivation of a judgment having this term as its left-hand side can only result from the application of rules **or_lapp** or **or_beta_lin**. Therefore, we have four combined case for $\mathcal{R}'$ and $\mathcal{R}''$.

or_lapp,**or_lapp**

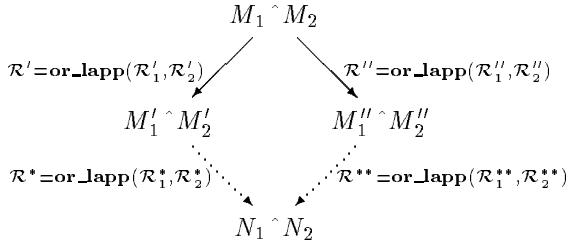
$$\mathcal{R}' = \dfrac{\begin{array}{cc}\mathcal{R}'_1 & \mathcal{R}'_2\\ M_1 \longrightarrow M'_1 & M_2 \longrightarrow M'_2\end{array}}{M_1 \hat{} M_2 \longrightarrow M'_1 \hat{} M'_2}\,\mathbf{or\_lapp} \quad \text{and} \quad \mathcal{R}'' = \dfrac{\begin{array}{cc}\mathcal{R}''_1 & \mathcal{R}''_2\\ M_1 \longrightarrow M''_1 & M_2 \longrightarrow M''_2\end{array}}{M_1 \hat{} M_2 \longrightarrow M''_1 \hat{} M''_2}\,\mathbf{or\_lapp}$$

with $U' = M'_1 \hat{} M'_2$ and $U'' = M''_1 \hat{} M''_2$.

We can apply the induction hypothesis to the premisses of these rules, completing the following diagrams:



Now, we apply rule **or_lapp** again to the derivations $\mathcal{R}^*_1$ and $\mathcal{R}^*_2$ of $M'_1 \longrightarrow N_1$ and $M'_2 \longrightarrow N_2$ respectively to obtain a derivation $\mathcal{R}^*$ of $M'_1 \hat{} M'_2 \longrightarrow N_1 \hat{} N_2$. Similarly we get a derivation $\mathcal{R}^{**}$ of $M''_1 \hat{} M''_2 \longrightarrow N_1 \hat{} N_2$. In this way, we have completed the following diagram, where the notation for derivations should be obvious.

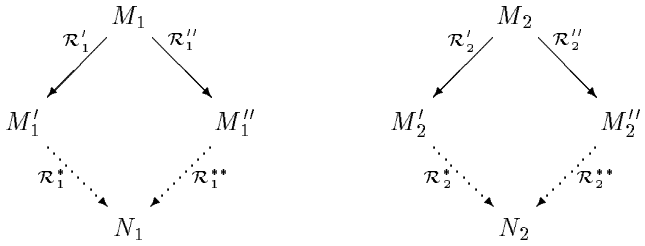


or_lapp,or_beta_lin

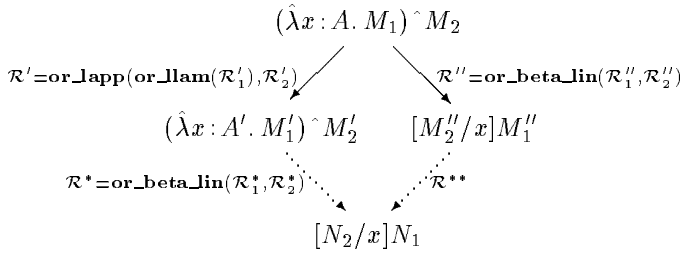
$$\mathcal{R}' = \dfrac{\begin{array}{cc}\mathcal{R}_1 & \mathcal{R}'_2\\ \hat{\lambda}x\!:\!A.\, M_1 \longrightarrow M' & M_2 \longrightarrow M'_2\end{array}}{(\hat{\lambda}x\!:\!A.\, M_1) \hat{} M_2 \longrightarrow M' \hat{} M'_2} \quad \text{and} \quad \mathcal{R}'' = \dfrac{\begin{array}{cc}\mathcal{R}''_1 & \mathcal{R}''_2\\ M_1 \longrightarrow M''_1 & M_2 \longrightarrow M''_2\end{array}}{(\hat{\lambda}x\!:\!A.\, M_1) \hat{} M_2 \longrightarrow [M''_2/x]M''_1}$$

with $U = (\hat{\lambda}x\!:\!A.\, M_1) \hat{} M_2$, $U' = M' \hat{} M'_2$ and $U'' = [M''_2/x]M''_1$.

By inversion on rule **or_llam** for $\mathcal{R}_1$, we have that $M' = \hat{\lambda}x : A'.\, M'_1$. Moreover, there exist a derivation $\mathcal{R}'_1$ of $M_1 \longrightarrow M'_1$. Therefore, we can apply the induction hypothesis and complete the following diagrams:



The substitution lemma 5.3.7 entails that there exist a derivation $\mathcal{R}^{**}$ for the judgment $[M_2''/x]M_1'' \longrightarrow [N_2/x]N_1$. In order to derive this term from the left side of the diamond, simply apply rule **or_llam** to make M_1' and N_1 functional, and then use the β-reduction rule to obtain the desired judgment. We have the following diagram:



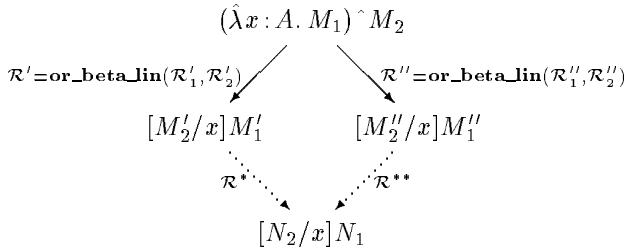
or_beta_lin,or_lapp Similar

or_beta_lin,or_beta_lin

$$\mathcal{R}' = \frac{\begin{array}{cc}\mathcal{R}_1 & \mathcal{R}_2'\\ M_1 \longrightarrow M_1' & M_2 \longrightarrow M_2'\end{array}}{(\hat{\lambda}x:A.\,M_1)\hat{}\,M_2 \longrightarrow [M_2'/x]M_1'} \quad \text{and} \quad \mathcal{R}'' = \frac{\begin{array}{cc}\mathcal{R}_1'' & \mathcal{R}_2''\\ M_1 \longrightarrow M_1'' & M_2 \longrightarrow M_2''\end{array}}{(\hat{\lambda}x:A.\,M_1)\hat{}\,M_2 \longrightarrow [M_2''/x]M_1''}$$

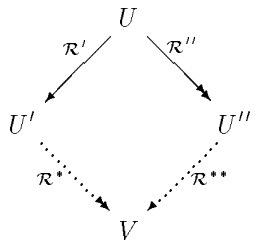
with $U = (\hat{\lambda}x:A.\,M_1)\hat{}\,M_2$, $U' = [M_2'/x]M_1'$ and $U'' = [M_2''/x]M_1''$.

By induction hypothesis, we obtain the same diagrams as in the previous cases. We refer to the names displayed there. By the substitution lemma 5.3.7, we obtain two derivations $\mathcal{R}^*$ and $\mathcal{R}^{**}$ for the judgments $[M_2'/x]M_1' \longrightarrow [N_2/x]N_1$ and $[M_2''/x]M_1'' \longrightarrow [N_2/x]N_1$ respectively. This achieves our purposes, as shown by the following diagram



☑

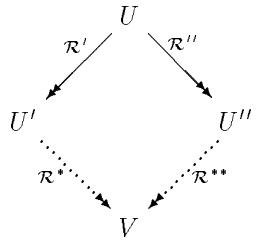
Lemma 5.3.9 (*Strip lemma*)



If $\mathcal{R}' :: U \longrightarrow U'$ *and* $\mathcal{R}'' :: U \longrightarrow^* U''$, *then there is a term* V *such that* $\mathcal{R}^* :: U' \longrightarrow^* V$ *and* $\mathcal{R}^{**} :: U'' \longrightarrow V$.

Proof.
By induction on the structure of $\mathcal{R}''$. In the base case, we apply the diamond property. ☑

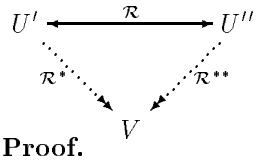
Lemma 5.3.10 (*Confluence*)



If $\mathcal{R}' :: U \longrightarrow^* U'$ *and* $\mathcal{R}'' :: U \longrightarrow^* U''$, *then there is a term* V *such that* $\mathcal{R}^* :: U' \longrightarrow^* V$ *and* $\mathcal{R}^{**} :: U'' \longrightarrow^* V$.

Proof.
By induction on the structure of $\mathcal{R}'$. In the base case, we apply the strip lemma. The other cases are trivial. ☑

Theorem 5.3.11 (*Church-Rosser*)



If $\mathcal{R} :: U' \equiv U''$, *then there is a term* V *such that* $\mathcal{R}^* :: U' \longrightarrow^* V$ *and* $\mathcal{R}^{**} :: U'' \longrightarrow^* V$.

Proof.

By induction on the structure of $\mathcal{R}$. In the two base cases, we apply the reflexivity lemma and confluence. ☑

B.3.3 Fundamental Properties

In this subsection, we give the proofs of the results presented in Section 5.3.3. We first provide a sketch of the proof of the free variables lemma.

Lemma 5.3.12 (*Free variables*)

i. If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\downarrow V$, *then* $\mathrm{FV}(U) \subseteq \mathrm{dom}\,\Psi$ *and* $\mathrm{FV}(V) \subseteq \mathrm{dom}\,\overline{\Psi}$;

ii. If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\downarrow V$ *or* $\mathcal{P} :: \vdash_\Sigma \Psi \Uparrow Ctx$, *then* $\mathrm{FV}(\Psi) \subseteq \mathrm{dom}\,\overline{\Psi}$;

iii. If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\downarrow V$ *or* $\mathcal{P} :: \vdash_\Sigma \Psi \Uparrow Ctx$ *or* $\mathcal{P} :: \vdash \Sigma \Uparrow Sig$, *then* $\mathrm{FV}(\Sigma) = \varnothing$;

iv. If $\mathcal{P} :: \Psi, x :\!\hat{:} A, \Psi' \vdash_\Sigma U \Uparrow\!\downarrow V$ *or* $\mathcal{P} :: \vdash_\Sigma \Psi, x :\!\hat{:} A, \Psi' \Uparrow Ctx$, *then* $\mathrm{FV}(A) \cup \mathrm{FV}(\Psi) \subseteq \mathrm{dom}\,\overline{\Psi}$.

Proof.

We proceed by induction on the structure of $\mathcal{P}$. The four parts of this statement all depend on each other, leading to a total number of 112 cases to analyze, or just 31 if we combine the conclusions of all the individual parts.

Many of these cases are trivial and rely on simple properties of the set-theoretic operations used in the definition of the concept of free variable and domain. Other cases require properties that we proved in Section B.2. As an example, we treat the case of rule **opa_lapp**; all other cases are simpler.

If rule **opa_lapp** has been applied, then $\mathcal{P}$ has the following shape, where the context in the conclusion has been written in the format needed for item (*iv*):

$$\mathcal{P} = \dfrac{\begin{array}{ccc}\mathcal{P}_1 & \mathcal{P}_2 & \mathcal{C} \\ \Psi_1 \vdash_\Sigma M \downarrow B_1 \multimap B_2 & \Psi_2 \vdash_\Sigma N \Uparrow B_1 & \Psi = \Psi_1 \bowtie \Psi_2\end{array}}{\underbrace{\Psi', x :\!\hat{:} A, \Psi''}_{\Psi} \vdash_\Sigma M \hat{} N \downarrow B_2}\ \textbf{opa\_lapp}$$

i. By induction hypothesis on $\mathcal{P}_1$, $\mathrm{FV}(B_1 \multimap B_2) \subseteq \mathrm{dom}\,\overline{\Psi_1}$. By definition, $\mathrm{FV}(B_2) \subseteq \mathrm{FV}(B_1 \multimap B_2)$. By lemma 5.2.14, $\mathrm{dom}\,\overline{\Psi_1} = \mathrm{dom}\,\overline{\Psi}$. Therefore, $\mathrm{FV}(B_2) \subseteq \mathrm{dom}\,\overline{\Psi}$.

By induction hypothesis on $\mathcal{P}_1$ we have also that $\mathrm{FV}(M) \subseteq \mathrm{dom}\,\Psi_1$. Moreover, by induction hypothesis on $\mathcal{P}_2$, $\mathrm{FV}(N) \subseteq \mathrm{dom}\,\Psi_2$. By definition, $\mathrm{FV}(M \hat{} N) = \mathrm{FV}(M) \cup \mathrm{FV}(N)$. Finally, by lemma 5.2.13, $\mathrm{dom}\,\Psi = \mathrm{dom}\,\Psi_1 \cup \mathrm{dom}\,\Psi_2$. Therefore, $\mathrm{FV}(M \hat{} N) \subseteq \mathrm{dom}\,\Psi$.

ii. By induction hypothesis, $\mathrm{FV}(\Psi_1) \subseteq \mathrm{dom}\,\overline{\Psi_1}$ and $\mathrm{FV}(\Psi_2) \subseteq \mathrm{dom}\,\overline{\Psi_2}$. By lemma 5.2.14, $\mathrm{dom}\,\overline{\Psi} = \mathrm{dom}\,\overline{\Psi_1} = \mathrm{dom}\,\overline{\Psi_2}$. By lemma 5.2.13, $\mathrm{FV}(\Psi) = \mathrm{FV}(\Psi_1) \cup \mathrm{FV}(\Psi_2)$. Therefore $\mathrm{FV}(\Psi) \subseteq \mathrm{dom}\,\overline{\Psi}$.

iii. By induction hypothesis on either $\mathcal{P}_1$ or $\mathcal{P}_2$, we have that $\mathrm{FV}(\Sigma) = \varnothing$.

iv. We have two case to analyze depending on whether x is linear or intuitionistic.

(a) If x is intuitionistic, then by corollary 5.2.10 we have that $\Psi_1 = \Psi_1', x:A, \Psi_1''$ and $\Psi_2 = \Psi_2', x:A, \Psi_2''$ with $\Psi' = \Psi_1' \bowtie \Psi_2'$ and $\Psi'' = \Psi_1'' \bowtie \Psi_2''$.
By induction hypothesis, $\mathrm{FV}(A) \subseteq \mathrm{dom}\,\overline{\Psi_1'}$ and $\mathrm{FV}(A) \subseteq \mathrm{dom}\,\overline{\Psi_2'}$. However, by lemma 5.2.14, $\mathrm{dom}\,\overline{\Psi'} = \mathrm{dom}\,\overline{\Psi_1'} = \mathrm{dom}\,\overline{\Psi_2'}$. Therefore $\mathrm{FV}(A) \subseteq \mathrm{dom}\,\overline{\Psi'}$.
Moreover, $\mathrm{FV}(\Psi_1') \subseteq \mathrm{dom}\,\overline{\Psi_1'}$ $(= \mathrm{dom}\,\overline{\Psi'})$ and $\mathrm{FV}(\Psi_2') \subseteq \mathrm{dom}\,\overline{\Psi_2'}$ $(= \mathrm{dom}\,\overline{\Psi'})$. By lemma 5.2.13, $\mathrm{FV}(\Psi') = \mathrm{FV}(\Psi_1') \cup \mathrm{FV}(\Psi_2')$. Therefore, $\mathrm{FV}(\Psi') \subseteq \mathrm{dom}\,\overline{\Psi'}$.

(b) If x is linear, corollary 5.2.10 leads us to distinguishing two further subcases, that are handled similarly to the intuitionistic case just described. ☑

The following two lemmas state properties of pre-canonical contexts. They are needed in the development of some of the proofs below.

Lemma 5.3.13 (*Intuitionistic context*)

i. If $\mathcal{P} :: \vdash_\Sigma \Psi \Uparrow \mathit{Ctx}$, *then* $\mathcal{P}' :: \vdash_\Sigma \overline{\Psi} \Uparrow \mathit{Ctx}$;

ii. If $\mathcal{P} :: \vdash_\Sigma \Psi \Uparrow \mathit{Ctx}$ *and* $\mathcal{S} :: \Psi = \Psi' \bowtie \Psi''$, *then* $\mathcal{P}' :: \vdash_\Sigma \Psi' \Uparrow \mathit{Ctx}$ *and* $\mathcal{P}'' :: \vdash_\Sigma \Psi'' \Uparrow \mathit{Ctx}$.

Proof.
By induction on $\mathcal{P}$. In (ii), we rely on the fact that $\overline{\Psi} = \overline{\Psi'} = \overline{\Psi''}$, by corollary 5.2.15. ☑

Lemma 5.3.14 (*Types in the context*)

If $\mathcal{P} :: \vdash_\Sigma \Psi, x \hat{:} A, \Psi' \Uparrow \mathit{Ctx}$, *then* $\mathcal{P}' :: \overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}$.

Proof.
By induction on the structure of Ψ'. ☑

We now consider the structural properties of *LLF*, namely permutation and weakening.

Lemma 5.3.15 (*Structural properties*)

Permutation:

i. If $\mathcal{P}' :: \Psi, \Psi', x \hat{:} A, \Psi'' \vdash_\Sigma U \Uparrow\!\downarrow V$ *and* $\mathcal{P}'' :: \overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}$, *then* $\mathcal{P} :: \Psi, x \hat{:} A, \Psi', \Psi'' \vdash_\Sigma U \Uparrow\!\downarrow V$;

ii. If $\mathcal{P}' :: \vdash_\Sigma \Psi, \Psi', x \hat{:} A, \Psi'' \Uparrow \mathit{Ctx}$ *and* $\mathcal{P}'' :: \overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}$, *then* $\mathcal{P} :: \vdash_\Sigma \Psi, x \hat{:} A, \Psi', \Psi'' \Uparrow \mathit{Ctx}$.

Weakening:

If $\mathcal{P}' :: \Psi \vdash_\Sigma U \Uparrow\!\!\downarrow V$ *and* $\mathcal{P}'' ::\vdash_\Sigma \Psi, \overline{\Psi'} \Uparrow Ctx$, *then* $\mathcal{P} :: \Psi, \overline{\Psi'} \vdash_\Sigma U \Uparrow\!\!\downarrow V$.

Proof.

We proceed by simultaneous induction on the structure of $\mathcal{P}'$ and, in the case of permutation, on the length of Ψ''. We also consider $\mathcal{P}''$ smaller than any subderivation in $\mathcal{P}'$ at the moment of applying the induction hypothesis.

Permutation

The proof is complicated by the rules that require the context to be intuitionistic and by the need to deal with context splitting. These cases are handled by the lemmas in Section B.2. The base case for (ii) is complicated by the fact that it relies on weakening. We first show one relatively simple case.

$$\boxed{\textbf{opc\_eq}} \quad \mathcal{P} = \dfrac{\begin{array}{ccc}\mathcal{P}_1' & \mathcal{R} & \mathcal{P}_2' \\ \Psi, \Psi', x :\!\!\hat{:} A, \Psi'' \vdash_\Sigma M \Uparrow B_2 & B_1 \equiv B_2 & \overline{\Psi, \Psi', x :\!\!\hat{:} A, \Psi''} \vdash_\Sigma B_1 \Uparrow \text{TYPE}\end{array}}{\Psi, \Psi', x :\!\!\hat{:} A, \Psi'' \vdash_\Sigma M \Uparrow B_1}\ \textbf{opc\_eq}$$

with $U = M$ and $V = B_1$.

$\mathcal{P}_1 :: \Psi, x :\!\!\hat{:} A, \Psi', \Psi'' \vdash_\Sigma M \Uparrow B_2$		by induction hypothesis on $\mathcal{P}_1'$,
$\overline{\Psi, \Psi', x :\!\!\hat{:} A, \Psi''} = \overline{\Psi}, \overline{\Psi'}, \overline{x :\!\!\hat{:} A}, \overline{\Psi''}$		by lemma 5.2.2,
$\mathcal{P}_2 :: \overline{\Psi}, \overline{x :\!\!\hat{:} A}, \overline{\Psi'}, \overline{\Psi''} \vdash_\Sigma B_1 \Uparrow \text{TYPE}$		by induction hypothesis on $\mathcal{P}_2'$,
$\overline{\Psi}, \overline{x :\!\!\hat{:} A}, \overline{\Psi'}, \overline{\Psi''} = \overline{\Psi, x :\!\!\hat{:} A, \Psi', \Psi''}$		by lemma 5.2.2,
$\mathcal{P} :: \Psi, x :\!\!\hat{:} A, \Psi', \Psi'' \vdash_\Sigma M \Uparrow B_1$		by rule **opc_eq** on $\mathcal{P}_1$, $\mathcal{R}$ and $\mathcal{P}_2$.

$\boxed{\Psi'' = \cdot}$ With an auxiliary induction on the length of Ψ', we prove that if $\mathcal{P}' :: \vdash_\Sigma \Psi, \Psi', x :\!\!\hat{:} A \Uparrow Ctx$, then $\mathcal{P} :: \vdash_\Sigma \Psi, x :\!\!\hat{:} A, \Psi' \Uparrow Ctx$. There are two cases to analyze.

- $\Psi' = \cdot$. Then $\mathcal{P} = \mathcal{P}'$.
- $\Psi' = \Psi^*, y :\!\!\hat{:} B$. By inversion, we have the following derivation for $\mathcal{P}'$, where the rules applied can be either **cp_int** or **cp_lin**, depending on whether x and y are intuitionistic or linear.

$$\mathcal{P}' = \dfrac{\dfrac{\begin{array}{cc}\mathcal{P}'^* & \mathcal{P}_B' \\ \vdash_\Sigma \Psi, \Psi^* \Uparrow Ctx & \overline{\Psi, \Psi^*} \vdash_\Sigma B \Uparrow \text{TYPE}\end{array}}{\vdash_\Sigma \Psi, \Psi^*, y :\!\!\hat{:} B \Uparrow Ctx} \quad \begin{array}{c}\mathcal{P}_A' \\ \overline{\Psi, \Psi^*, y :\!\!\hat{:} B} \vdash_\Sigma A \Uparrow \text{TYPE}\end{array}}{\vdash_\Sigma \Psi, \Psi^*, y :\!\!\hat{:} B, x :\!\!\hat{:} A \Uparrow Ctx}$$

An application of lemma 5.3.13 to $\mathcal{P}'^*$ enables us to obtain a derivation of $\vdash_\Sigma \overline{\Psi, \Psi^*} \Uparrow Ctx$ that we use to derive $\overline{\Psi, \Psi^*} \vdash_\Sigma A \Uparrow \text{TYPE}$ by weakening on hypothesis $\mathcal{P}''$. We can now apply the appropriate context formation rule to this derivation and $\mathcal{P}'^*$ in order to produce a derivation $\mathcal{P}^{**}$ of $\vdash_\Sigma \Psi, \Psi^*, x :\!\!\hat{:} A \Uparrow Ctx$.

We now apply the induction hypothesis on $\mathcal{P}^{**}$ and get a derivation $\mathcal{P}^*$ of $\vdash_\Sigma \Psi, x :\!\!\hat{:} A, \Psi^* \Uparrow Ctx$.

Now we need a derivation $\mathcal{P}_B$ of $\overline{\Psi, x :\hat{:} A, \Psi^*} \vdash_\Sigma B \Uparrow \text{TYPE}$. In order to obtain it, we apply weakening to $\mathcal{P}'_B$ and $\mathcal{P}^{**}$ and get a derivation of $\overline{\Psi, \Psi^*, x :\hat{:} A} \vdash_\Sigma B \Uparrow \text{TYPE}$. Now, we can apply the induction hypothesis on this derivation in order to obtain $\mathcal{P}_B$.

By applying the appropriate context rule on $\mathcal{P}^*$ and $\mathcal{P}_B$, we obtain the desired derivation $\mathcal{P}$ of $\vdash_\Sigma \Psi, x :\hat{:} A, \Psi^*, y :\hat{:} B \Uparrow Ctx$.

Weakening

The only non-trivial cases concern rules **opc_llam** and **opc_ilam**, for which we need permutation in order to move the assumption made by these rules back to the end of the context after weakening. In those cases, we need to run an auxiliary induction in order to have permutation go in the desired direction. We also rely on lemmas 5.2.1 and 5.2.2 in the rules characterized by a strictly intuitionistic context. ☑

Corollary 5.3.16 (*Reverse permutation*)

i. If $\mathcal{P}' :: \Psi, x :\hat{:} A, \Psi', \Psi'' \vdash_\Sigma U \Uparrow\!\!\downarrow V$ *and* $\mathcal{P}'' :: \overline{\Psi, \Psi'} \vdash_\Sigma A \Uparrow \text{TYPE}$, *then* $\mathcal{P} :: \Psi, \Psi', x :\hat{:} A, \Psi'' \vdash_\Sigma U \Uparrow\!\!\downarrow V$;

ii. If $\mathcal{P}' :: \vdash_\Sigma \Psi, x :\hat{:} A, \Psi', \Psi'' \Uparrow Ctx$ *and* $\mathcal{P}'' :: \overline{\Psi, \Psi'} \vdash_\Sigma A \Uparrow \text{TYPE}$, *then* $\mathcal{P} :: \vdash_\Sigma \Psi, \Psi', x :\hat{:} A, \Psi'' \Uparrow Ctx$.

Proof.

By induction on the structure of Ψ'. ☑

The flattening lemma below is not properly needed for the proofs in this appendix, but is important for adapting the adequacy theorem of an object language so that the encoding in *LLF* results correct even for complex linearity patterns.

Lemma 5.3.17 (*Flattening*)

If $\mathcal{P} :: \Psi, x :\hat{:} A, \Psi' \vdash_\Sigma M \Uparrow\!\!\downarrow B$, *then* $\mathcal{P}' :: \Psi, x : A, \Psi' \vdash_\Sigma M \Uparrow\!\!\downarrow B$.

Proof.

By induction on the structure of $\mathcal{P}$. ☑

Next, we consider the transitivity lemmas and consistency. As we said in Chapter 5, the order in which these results are presented is critical since linear transitivity depends on consistency, which depends on intuitionistic transitivity.

Lemma 5.3.18 (*Intuitionistic transitivity*)

i. If $\mathcal{P}_N :: \overline{\Psi} \vdash_\Sigma N \Uparrow A$ *and* $\mathcal{P} :: \Psi, x : A, \Psi' \vdash_\Sigma U \Uparrow\!\!\downarrow V$, *then* $\mathcal{P}' :: \Psi, [N/x]\Psi' \vdash_\Sigma [N/x]U \Uparrow\!\!\downarrow [N/x]V$;

ii. If $\mathcal{P}_N :: \overline{\Psi} \vdash_\Sigma N \Uparrow A$ *and* $\mathcal{P} :: \vdash_\Sigma \Psi, x : A, \Psi' \Uparrow Ctx$, *then* $\mathcal{P}' :: \vdash_\Sigma \Psi, [N/x]\Psi' \Uparrow Ctx$.

Proof.

We proceed by induction on the structure of $\mathcal{P}$. We limit the presentation to a number of interesting cases.

opc_a
$$\mathcal{P} = \dfrac{\begin{array}{c}\mathcal{P}_1\\ \Psi, x:A, \Psi' \vdash_\Sigma\ M \downarrow P\end{array}}{\Psi, x:A, \Psi' \vdash_\Sigma\ M \Uparrow P}\ \textbf{opc\_a}$$

with $U = M$ and $V = P$.

$\mathcal{P}_1'$ :: $\Psi, [N/x]\Psi' \vdash_\Sigma\ [N/x]M \downarrow [N/x]P$ by induction hypothesis on $\mathcal{P}_1$,
$[N/x]P$ is a type family by lemma 5.2.3,
$\mathcal{P}'$ :: $\Psi, [N/x]\Psi' \vdash_\Sigma\ [N/x]M \Uparrow [N/x]P$ by rule **opc_a** on $\mathcal{P}_1'$.

opc_eq
$$\mathcal{P} = \dfrac{\begin{array}{ccc}\mathcal{P}_1 & \mathcal{R} & \mathcal{P}_2\\ \Psi, x:A, \Psi' \vdash_\Sigma\ M \Uparrow B_1 & B_1 \equiv B_2 & \overline{\Psi, x:A, \Psi'} \vdash_\Sigma\ B_2 \Uparrow \text{TYPE}\end{array}}{\Psi, x:A, \Psi' \vdash_\Sigma\ M \Uparrow B_2}\ \textbf{opc\_eq}$$

with $U = M$ and $V = B_2$.

$\mathcal{P}_1'$ :: $\Psi, [N/x]\Psi' \vdash_\Sigma\ [N/x]M \Uparrow [N/x]B_1$ by induction hypothesis on $\mathcal{P}_1$,
$\mathcal{R}'$ :: $[N/x]B_1 \equiv [N/x]B_2$ by the substitution lemma and reflexivity,
$\mathcal{P}_2'$ :: $\overline{\Psi, [N/x]\Psi'} \vdash_\Sigma\ [N/x]B_2 \Uparrow \text{TYPE}$ by induction hypothesis on $\mathcal{P}_2$ and lemma 5.2.2,
$\mathcal{P}'$ :: $\Psi, [N/x]\Psi' \vdash_\Sigma\ [N/x]M \Uparrow [N/x]B_2$ by rule **opc_eq** on $\mathcal{P}_1'$, $\mathcal{R}'$ and $\mathcal{P}_2'$.

opa_ivar We must distinguish three cases depending on whether the variable y mentioned by this rule is declared in Ψ, is x itself, or is declared in Ψ'.

- $\boxed{y = x}$ Then
$$\mathcal{P} = \dfrac{\begin{array}{c}\mathcal{P}_1\\ \vdash_\Sigma\ \overline{\Psi, x:A, \Psi'} \Uparrow Ctx\end{array}}{\overline{\Psi, x:A, \Psi'} \vdash_\Sigma\ x \downarrow A}\ \textbf{opa\_ivar}$$

with $U = x$, $V = A$ and the context being intuitionistic.
By induction hypothesis on $\mathcal{P}_1$, there exist a derivation of $\vdash_\Sigma\ \overline{\Psi, [N/x]\Psi'} \Uparrow Ctx$. We can then apply weakening on $\mathcal{P}_N$ and obtain a derivation $\mathcal{P}_1'$ of $\overline{\Psi, [N/x]\Psi'} \vdash_\Sigma\ N \Uparrow A$. By rule **opa_c**, we get a derivation $\mathcal{P}'$ of $\overline{\Psi, [N/x]\Psi'} \vdash_\Sigma\ N \downarrow A$. By the free variables lemma, $\text{FV}(A) \subseteq \text{dom}\ \Psi$. Thus, $x \notin \text{FV}(A)$ and, by lemma 5.2.6, $[N/x]A = A$.

- $\boxed{y \in \text{dom}\ \Psi}$ Then
$$\mathcal{P} = \dfrac{\begin{array}{c}\mathcal{P}_1\\ \vdash_\Sigma\ \overline{\Psi^*, y:B, \Psi^{**}, x:A, \Psi'} \Uparrow Ctx\end{array}}{\overline{\Psi^*, y:B, \Psi^{**}, x:A, \Psi'} \vdash_\Sigma\ y \downarrow B}\ \textbf{opa\_ivar}$$

with $\Psi = \Psi^*, y:B, \Psi^{**}$, $U = y$, $V = B$ and the context being intuitionistic.
By induction hypothesis on $\mathcal{P}_1$, there exist a derivation $\mathcal{P}_1'$ of $\vdash_\Sigma\ \overline{\Psi^*, y:B, \Psi^{**}, [N/x]\Psi'} \Uparrow Ctx$. Then, by rule **opa_ivar**, we get the desired derivation $\mathcal{P}'$ of $\overline{\Psi^*, y:B, \Psi^{**}, [N/x]\Psi'} \vdash_\Sigma\ y \downarrow B$. Now, simply observe that, by definition of substitution, $[N/x]y = y$ and, similarly to the previous case $[N/x]B = B$.

- $\boxed{y \in \mathrm{dom}\ \Psi'}$ Then $\mathcal{P} = \dfrac{\mathcal{P}_1}{\dfrac{\vdash_\Sigma\ \overline{\Psi, x:A, \Psi^*, y:B, \Psi^{**}} \Uparrow Ctx}{\overline{\Psi, x:A, \Psi^*, y:B, \Psi^{**}} \vdash_\Sigma\ y \downarrow B}\ \mathbf{opa\_ivar}}$

  with $\Psi' = \Psi^*, y:B, \Psi^{**}$, $U = y$, $V = B$ and the context being intuitionistic.
  By applying the induction hypothesis on $\mathcal{P}_1$, there exist a derivation of the judgment $\mathcal{P}_1'$ of $\vdash_\Sigma\ \overline{\Psi, [N/x]\Psi^*, y:[N/x]B, [N/x]\Psi^{**}} \Uparrow Ctx$. By definition of substitution, $[N/x]y = y$. By rule **opa_ivar**, we obtain the desired derivation $\mathcal{P}'$ of $\overline{\Psi, [N/x]\Psi^*, y:[N/x]B, [N/x]\Psi^{**}} \vdash_\Sigma y \downarrow [N/x]B$.

$\boxed{\mathbf{opa\_iapp}}$ $\mathcal{P} = \dfrac{\mathcal{P}_1 \qquad\qquad \mathcal{P}_2}{\dfrac{\Psi, x:A, \Psi' \vdash_\Sigma\ M_2 \downarrow \Pi y:B_1.\ B_2 \quad \overline{\Psi, x:A, \Psi'} \vdash_\Sigma\ M_1 \Uparrow B_1}{\Psi, x:A, \Psi' \vdash_\Sigma\ M_2\ M_1 \downarrow [M_1/y]B_2}\ \mathbf{opa\_iapp}}$

with $U = M_2\ M_1$ and $V = [M_1/y]B_2$.

$\mathcal{P}_1' :: \Psi, [N/x]\Psi' \vdash_\Sigma [N/x]M_2 \downarrow \Pi y:[N/x]B_1.\ [N/x]B_2$	by induction hypothesis on $\mathcal{P}_1$ and definition of substitution,
$\mathcal{P}_2' :: \overline{\Psi, [N/x]\Psi'} \vdash_\Sigma [N/x]M_1 \Uparrow [N/x]B_1$	by induction hypothesis on $\mathcal{P}_2$,
$[([N/x]M_1)/y]([N/x]B_2) = [N/x]([M_1/y]B_2)$	by lemma 5.2.7 since $y \notin \mathrm{FV}(N)$ by construction,
$\mathcal{P}' :: \Psi, [N/x]\Psi' \vdash_\Sigma [N/x](M_2\ M_1) \downarrow [N/x]([M_1/y]B_2)$	by rule **op_iapp** on $\mathcal{P}_1'$and $\mathcal{P}_2'$. ☑

Lemma 5.3.19 (*Consistency*)

i. If $\mathcal{P} :: \Psi \vdash_\Sigma M \Uparrow\!\downarrow A$, *then* $\mathcal{P}' :: \overline{\Psi} \vdash_\Sigma A \Uparrow \mathrm{TYPE}$;

ii. If $\mathcal{P} :: \overline{\Psi} \vdash_\Sigma A \Uparrow\!\downarrow K$, *then* $\mathcal{P}' :: \overline{\Psi} \vdash_\Sigma K \Uparrow Kind$;

iii. If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\downarrow V$, *then* $\mathcal{P}' :: \vdash_\Sigma \Psi \Uparrow Ctx$;

iv. If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\downarrow V$ *or* $\mathcal{P} :: \vdash_\Sigma \Psi \Uparrow Ctx$, *then* $\mathcal{P}' :: \vdash \Sigma \Uparrow Sig$.

Proof.

We proceed by induction on the structure of $\mathcal{P}$. We show two cases concerning (i). All other cases in this and the remaining parts are similar or simpler.

$\boxed{\mathbf{opc\_unit}}$ $\mathcal{P} = \dfrac{\mathcal{P}_1}{\dfrac{\vdash_\Sigma\ \Psi \Uparrow Ctx}{\Psi \vdash_\Sigma\ \langle\rangle \Uparrow \top}\ \mathbf{opc\_unit}}$

with $M = \langle\rangle$ and $A = \top$.

$\mathcal{P}_1' :: \vdash_\Sigma \overline{\Psi} \Uparrow Ctx$	by lemma 5.3.13 on $\mathcal{P}_1$,
$\mathcal{P}' :: \overline{\Psi} \vdash_\Sigma \top \Uparrow \mathrm{TYPE}$	by rule **fpc_top** on $\mathcal{P}_1'$.

opa_iapp
$$\mathcal{P} = \dfrac{\begin{array}{cc}\mathcal{P}_1 & \mathcal{P}_2\\ \Psi \vdash_\Sigma M_1 \downarrow \Pi x{:}A_2.\, A_1 & \overline{\Psi} \vdash_\Sigma M_2 \Uparrow A_2\end{array}}{\Psi \vdash_\Sigma M_1\ M_2 \downarrow [M_2/x]A_1}\ \mathbf{opa\_iapp}$$

with $M = M_1\ M_2$ and $A = [M_2/x]A_1$.

$\mathcal{P}_1' :: \overline{\Psi} \vdash_\Sigma \Pi x{:}A_2.\, A_1 \Uparrow \mathrm{TYPE}$	by induction hypothesis on $\mathcal{P}_1$,
$\mathcal{P}_1'' :: \overline{\Psi}, x{:}A_2 \vdash_\Sigma A_1 \Uparrow \mathrm{TYPE}$	by inversion on rule **fpc_dep** for $\mathcal{P}_1'$,
$\mathcal{P}' :: \overline{\Psi} \vdash_\Sigma [M_2/x]A_1 \Uparrow \mathrm{TYPE}$	by intuitionistic transitivity on $\mathcal{P}_1'$ with $\mathcal{P}_2$. ☑

Lemma 5.3.20 (*Linear transitivity*)

i. If $\mathcal{P}_N :: \Psi \vdash_\Sigma N \Uparrow A$ *and* $\mathcal{P} :: \overline{\Psi}, x \hat{:} A, \Psi' \vdash_\Sigma M \Uparrow\!\downarrow B$, *then* $\mathcal{P}' :: \Psi, \Psi' \vdash_\Sigma [N/x]M \Uparrow\!\downarrow B$;

ii. If $\mathcal{P}_N :: \Psi \vdash_\Sigma N \Uparrow A$ *and* $\mathcal{P} :: \vdash_\Sigma \overline{\Psi}, x \hat{:} A, \Psi' \Uparrow Ctx$, *then* $\mathcal{P}' :: \vdash_\Sigma \Psi, \Psi' \Uparrow Ctx$.

Proof.

We proceed by induction on the structure of $\mathcal{P}$ in the first part, and by induction on the length of Ψ' in the second. We present two cases. The only other complex case arises from the application of rule **opa_lapp** in $\mathcal{P}$, and requires corollary 5.2.10.

opa_lvar
$$\mathcal{P} = \dfrac{\begin{array}{c}\mathcal{P}_1\\ \vdash_\Sigma \overline{\Psi}, x \hat{:} A, \overline{\Psi'} \Uparrow Ctx\end{array}}{\overline{\Psi}, x \hat{:} A, \overline{\Psi'} \vdash_\Sigma x \downarrow A}\ \mathbf{opa\_lvar}$$

with $U = x$, $V = A$ and the contexts being intuitionistic.

$\mathcal{P}^* :: \vdash_\Sigma \overline{\Psi}, \overline{\Psi'} \Uparrow Ctx$	by lemma 5.3.13, since x is linear,
$\mathcal{P}_1' :: \overline{\Psi}, \overline{\Psi'} \vdash_\Sigma N \Uparrow A$	by weakening on $\mathcal{P}_N$ thanks to $\mathcal{P}^*$,
$\mathcal{P}' :: \overline{\Psi}, \overline{\Psi'} \vdash_\Sigma N \downarrow A$	by rule **opa_c** on $\mathcal{P}_1'$.

$\boxed{\Psi' = \cdot}$ By the consistency lemma applied on $\mathcal{P}_N$, there is a derivation $\mathcal{P}'$ of $\vdash_\Sigma \Psi' \Uparrow Ctx$. Then simply notice that $[N/x]\cdot = \cdot$. ☑

Next, we prove the unicity of types and kinds and the inversion lemmas, which permit us to consider unique inversion patterns, even when the rules in Figure 5.2–5.2 offer multiple inversion choices.

Lemma 5.3.21 (*Unicity of types and kinds*)

i. If $\mathcal{P}' :: \Psi' \vdash_\Sigma M \Uparrow\!\downarrow A'$ *and* $\mathcal{P}'' :: \Psi'' \vdash_\Sigma M \Uparrow\!\downarrow A''$ *with* $\Psi'_{|\mathrm{FV}(M)} = \Psi''_{|\mathrm{FV}(M)}$ *and where the arrows do not need to match, then* $A' \equiv A''$;

ii. If $\mathcal{P}' :: \overline{\Psi} \vdash_\Sigma A \Uparrow\!\downarrow K'$ *and* $\mathcal{P}'' :: \overline{\Psi} \vdash_\Sigma A \Uparrow\!\downarrow K''$ *where the arrows do not need to match, then* $K' \equiv K''$.

Proof.

We proceed by induction on the structure of $\mathcal{P}'$ and $\mathcal{P}''$. The idea is to unfold these derivations until an introduction or elimination rule is unearthed, then we apply the induction hypothesis on the subterms. Rule **opa_lapp** is problematic due to context splitting and is the reason for the third premiss in the first part. We show two cases:

opc_eq, any rule

$$\mathcal{P}' = \dfrac{\mathcal{P}'_1 \quad \mathcal{R}' \quad \mathcal{P}'_2}{\Psi' \vdash_\Sigma M \Uparrow B' \quad B' \equiv A' \quad \overline{\Psi'} \vdash_\Sigma A' \Uparrow \text{TYPE}}\ \textbf{opc\_eq}$$

$$\Psi' \vdash_\Sigma M \Uparrow A'$$

$\mathcal{R} :: B' \equiv A''$ by induction hypothesis on $\mathcal{P}'_1$,

$A' \equiv A''$ by symmetry and transitivity on $\mathcal{R}'$ and $\mathcal{R}$.

opa_lapp,**opa_lapp**

$$\mathcal{P}' = \dfrac{\mathcal{P}'_1 \quad \mathcal{P}'_2 \quad \mathcal{C}'}{\Psi'_1 \vdash_\Sigma M_1 \downarrow B' \multimap A' \quad \Psi'_2 \vdash_\Sigma M_2 \Uparrow B' \quad \Psi' = \Psi'_1 \bowtie \Psi'_2}\ \textbf{opa\_lapp}$$

$$\Psi' \vdash_\Sigma M_1 \hat{} M_2 \downarrow A'$$

and

$$\mathcal{P}'' = \dfrac{\mathcal{P}''_1 \quad \mathcal{P}''_2 \quad \mathcal{C}''}{\Psi''_1 \vdash_\Sigma M_1 \downarrow B'' \multimap A'' \quad \Psi''_2 \vdash_\Sigma M_2 \Uparrow B'' \quad \Psi'' = \Psi''_1 \bowtie \Psi''_2}\ \textbf{opa\_lapp}$$

$$\Psi'' \vdash_\Sigma M_1 \hat{} M_2 \downarrow A''$$

with $M = M_1 \hat{} M_2$.

We know by hypothesis that $\Psi'_{|\mathrm{FV}(M_1 \hat{} M_2)} = \Psi''_{|\mathrm{FV}(M_1 \hat{} M_2)}$. Since $\mathrm{FV}(M_1) \subseteq \mathrm{FV}(M_1 \hat{} M_2)$, we have also that $\Psi'_{|\mathrm{FV}(M_1)} = \Psi''_{|\mathrm{FV}(M_1)}$. By lemma 5.2.13, we have that $\mathrm{dom}\,\Psi'_1 \subseteq \mathrm{dom}\,\Psi'$ and $\mathrm{dom}\,\Psi''_1 \subseteq \mathrm{dom}\,\Psi''$. By the free variables lemma 5.3.12 applied to $\mathcal{P}'_1$ and $\mathcal{P}''_1$, $\mathrm{FV}(M_1) \subseteq \mathrm{dom}\,\Psi'_1$ and $\mathrm{FV}(M_1) \subseteq \mathrm{dom}\,\Psi''_1$. This permits us to conclude that $\Psi'_{1|\mathrm{FV}(M_1)} = \Psi''_{1|\mathrm{FV}(M_1)}$.

Therefore, we can apply the induction hypothesis on ($\mathcal{P}'_1$ and $\mathcal{P}''_1$), obtaining that $B' \multimap A' \equiv B'' \multimap A''$. By the invertibility of definitional equality (lemma 5.3.5), we have that $A' \equiv A''$. ☑

Lemma 5.3.22 (*Inversion*)

i. *If* $\mathcal{P} :: \Psi \vdash_\Sigma \langle M, N\rangle \Uparrow\!\downarrow A \,\&\, B$, *then* $\mathcal{P}' :: \Psi \vdash_\Sigma M \Uparrow\!\downarrow A$ *and* $\mathcal{P}'' :: \Psi \vdash_\Sigma N \Uparrow\!\downarrow B$;

ii. *If* $\mathcal{P} :: \Psi \vdash_\Sigma \hat{\lambda} x{:}A.\, M \Uparrow\!\downarrow A' \multimap B$, *then* $\mathcal{P}' :: \Psi, x \hat{:} A \vdash_\Sigma M \Uparrow\!\downarrow B$ *and* $\mathcal{R} :: A \equiv A'$;

iii. *If* $\mathcal{P} :: \Psi \vdash_\Sigma \lambda x{:}A.\, M \Uparrow\!\downarrow \Pi x{:}A'.\, B$, *then* $\mathcal{P}' :: \Psi, x{:}A \vdash_\Sigma M \Uparrow\!\downarrow B$; *and* $\mathcal{R} :: A \equiv A'$.

Proof.

By induction on the structure of $\mathcal{P}$. All these results apply the same technique: the derivation is unfolded until an introduction or an elimination appears as the last inference rule applied in $\mathcal{P}$. We show one case from (*ii*).

$$\boxed{\textbf{opc\_eq}} \quad \mathcal{P} = \dfrac{\begin{array}{ccc} \mathcal{P}_1 & \mathcal{R} & \mathcal{P}_2 \\ \Psi \vdash_\Sigma \hat{\lambda} x\!:\!A.\, M \Uparrow B' & B' \equiv A' \multimap B & \overline{\Psi} \vdash_\Sigma A' \multimap B \Uparrow \textsc{type} \end{array}}{\Psi \vdash_\Sigma \hat{\lambda} x\!:\!A.\, M \Uparrow A' \multimap B}\ \textbf{opc\_eq}$$

By lemma 5.3.6, we have that $B' = A'' \multimap B''$. Then, by induction hypothesis on $\mathcal{P}_1$, we obtain $\mathcal{P}_1' :: \Psi, x \hat{:} A \vdash_\Sigma M \Uparrow B''$ and $\mathcal{R}_1' :: A \equiv A''$. By lemma 5.3.5, we have $\mathcal{R}_1'' :: A'' \equiv A'$ and $\mathcal{R}' :: B'' \equiv B$. By rule **feq_trans** on $\mathcal{R}_1$ and $\mathcal{R}'$, we get $\mathcal{R} :: A \equiv A'$.

Now, by consistency on $\mathcal{P}$, we have $\mathcal{P}_2 :: \overline{\Psi} \vdash_\Sigma A' \multimap B \Uparrow \textsc{type}$. By inversion on rule **fpc_limp**, we get $\mathcal{P}_2' :: \overline{\Psi} \vdash_\Sigma B \Uparrow \textsc{type}$. Then by rule **opc_eq** on $\mathcal{P}_1'$, $\mathcal{R}'$ and $\mathcal{P}_2'$, we finally obtain the desired derivation $\mathcal{P}'$ of $\Psi, x \hat{:} A \vdash_\Sigma M \Uparrow B$. ☑

Lemma 5.3.23 (*Inversion on object constructors*)

i. *If* $\mathcal{P} :: \Psi \vdash_\Sigma \langle\rangle \Uparrow\!\downarrow A$, *then* $A = \top$;

ii. *If* $\mathcal{P} :: \Psi \vdash_\Sigma \langle M_1, M_2\rangle \Uparrow\!\downarrow A$, *then* $A = A_1 \,\&\, A_2$;

iii. *If* $\mathcal{P} :: \Psi \vdash_\Sigma \hat{\lambda} x\!:\!A_1.\, M \Uparrow\!\downarrow A$, *then* $A = A_1' \multimap A_2$ *with* $A_1 \equiv A_1'$;

iv. *If* $\mathcal{P} :: \Psi \vdash_\Sigma \lambda x\!:\!A_1.\, M \Uparrow\!\downarrow A$, *then* $A = \Pi x\!:\!A_1'.\, A_2$ *with* $A_1 \equiv A_1'$.

Proof.

We proceed by induction on the structure of $\mathcal{P}$. All cases are trivial except showing that rule **opc_a** does not apply.

Assume ab absurdum that there exist a derivation $\mathcal{P}$ of $\Psi \vdash_\Sigma \langle\rangle \Uparrow\!\downarrow P$. Then by inversion, $\mathcal{P}$ can result only from the application of rules **opc_a**, **opc_eq**, **opa_c** or **opa_eq**. Let h be a measure of the height of $\mathcal{P}$ calculated by counting the number of consecutive occurrence of these rules in the leftmost path of this derivation. For $h > 0$, it is easy to see that there exist a derivation $\mathcal{P}'$ of measure $h - 1$ of $\Psi \vdash_\Sigma \langle\rangle \Uparrow\!\downarrow P'$ with $P \equiv P'$. However, no rule is applicable in the case in which $h = 0$. Thus, $\mathcal{P}$ cannot have a finite measure, and therefore, $\mathcal{P}$ itself must contain an infinite number of rules, which is against our definition of derivation.

We proceed similarly for the other constructors. ☑

Finally, we prove the extensionality lemma, which relates the shape of pre-canonical object level terms to their types.

Lemma 5.3.24 (*Extensionality*)

i. *If* $\mathcal{P} :: \Psi \vdash_\Sigma M \Uparrow \top$, *then* $M = \langle\rangle$;

ii. *If* $\mathcal{P} :: \Psi \vdash_\Sigma M \Uparrow A \,\&\, B$, *then* $M = \langle M_1, M_2\rangle$;

iii. *If* $\mathcal{P} :: \Psi \vdash_\Sigma M \Uparrow A \multimap B$, *then* $M = \hat{\lambda} x\!:\!A'.\, M_1$ *with* $A \equiv A'$;

iv. *If* $\mathcal{P} :: \Psi \vdash_\Sigma M \Uparrow \Pi x\!:\!A.\, B$, *then* $M = \lambda x\!:\!A'.\, M_1$ *with* $A \equiv A'$.

Proof.

By induction on the structure of $\mathcal{P}$. Notice that rule **opc_a** does not apply and therefore we do not need to access the pre-atomicity judgments. The base cases correspond to the constructors mentioned in the lemma. The only other case concerns the equivalence rule, which is handled by induction. ☑

B.3.4 Strong Normalization

This section contains the proofs of the results presented in Subsection 5.3.4. We also state and prove auxiliary properties that are either needed in the proofs of these results, or that are interesting by themselves.

First, we prove the equivalence of one-step and parallel nested reduction strategies. Further properties, namely confluence and the Church-Rosser theorem for the one-step reduction strategy, are given.

Lemma 5.3.25 (*Soundness and completeness of* $\equiv_1$ *w.r.t.* $\equiv$)

i. *If* $U \longrightarrow U'$, *then either* $U = U'$ *or* $U \longrightarrow_1^+ U'$;

ii. *If* $U \longrightarrow^* U'$, *then either* $U = U'$ *or* $U \longrightarrow_1^+ U'$;

iii. *If* $U \longrightarrow_1 U'$, *then* $U \longrightarrow U'$;

iv. *If* $U \longrightarrow_1^+ U'$, *then* $U \longrightarrow^* U'$.

v. $U \equiv_1 U'$ *iff* $U \equiv U'$.

Proof.

By induction on a derivation for the hypothesis. ☑

Corollary 5.3.26 (*Confluence of* $\longrightarrow_1^+$)

If $U \longrightarrow_1^+ U'$ *and* $U \longrightarrow_1^+ U''$, *then one of the following cases applies*

i. $U' = U''$;

ii. $U' \longrightarrow_1^+ U''$;

iii. $U'' \longrightarrow_1^+ U'$;

iv. *there is a term* V *such that* $U' \longrightarrow_1^+ V$ *and* $U'' \longrightarrow_1^+ V$.

Proof.

We use the above equivalence and confluence for the parallel nested reduction strategy. The four cases in the conclusion arise from the validity of reflexivity in $\longrightarrow^*$, but not in $\longrightarrow_1^+$. ☑

Corollary 5.3.27 (*Church-Rosser property for* $\equiv_1$)

If $U' \equiv_1 U''$, *then one of the following cases applies*

i. $U' = U''$;

ii. $U' \longrightarrow_1^+ U''$;

iii. $U'' \longrightarrow_1^+ U'$;

iv. *there is a term* V *such that* $U' \longrightarrow_1^+ V$ *and* $U'' \longrightarrow_1^+ V$.

Proof.

Similar to confluence, but this time we use the Church-Rosser property of $\equiv$. ☑

At this point, we are ready to prove the subject reduction lemma for the one-step reduction strategy. This result is then transfered to the parallel nested reduction strategy.

Lemma 5.3.28 (*Subject reduction*)

If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\downarrow V$ *and* $\mathcal{R} :: U \longrightarrow_1 U'$, *then* $\mathcal{P}' :: \Psi \vdash_\Sigma U' \Uparrow\!\downarrow V$.

Proof.

We proceed by induction on the structure of $\mathcal{P}$ and inversion on $\mathcal{R}$. We show one case of the proof.

opa_iapp $\quad \mathcal{P} = \dfrac{\begin{array}{cc}\mathcal{P}_1 & \mathcal{P}_2\end{array}}{\Psi \vdash_\Sigma M\ N \downarrow [N/x]B}\ \textbf{opa\_iapp}$ $\quad \dfrac{\Psi \vdash_\Sigma M \downarrow \Pi x{:}A.\,B \quad \overline{\Psi} \vdash_\Sigma N \Uparrow A}{}$

with $U = M\ N$ and $V = [N/x]B$.

By inversion on $\mathcal{R}$, we have three cases to analyze:

- $\mathcal{R} = \dfrac{\mathcal{R}_1}{M\ N \longrightarrow_1 M'\ N}\ \textbf{or1\_iapp1}$ with $U' = M'\ N$.
  By induction hypothesis on $\mathcal{P}_1$ and $\mathcal{R}_1$, there exist a derivation $\mathcal{P}_1'$ of $\Psi \vdash_\Sigma M' \downarrow \Pi x : A.\,B$. Now, by rule **opa_iapp** on $\mathcal{P}_1'$ and $\mathcal{P}_2$, we obtain the desired derivation $\mathcal{P}'$ of $\Psi \vdash_\Sigma M'\ N \downarrow [N/x]B$.

- We proceed similarly if $\mathcal{R}$ ends with an application of rule **or1_iapp2**.

- $\mathcal{R} = \dfrac{}{\lambda x{:}A'.\,M'\ N \longrightarrow_1 [N/x]M'}\ \textbf{or1\_beta\_int}$ with $M = \lambda x{:}A'.\,M'$ and $U' = [N/x]M'$.
  By the inversion lemma 5.3.22 on $\mathcal{P}_1$, we obtain a derivation $\mathcal{R}^*$ of $A \equiv A'$ and a derivation $\mathcal{P}_1^*$ of $\Psi, x : A' \vdash_\Sigma M' \downarrow B$. By consistency, we get a derivation of $\vdash_\Sigma \Psi, x : A' \Uparrow Ctx$ and therefore, by inversion on rule **cp_int**, there is a derivation $\mathcal{P}^*$ of $\overline{\Psi} \vdash_\Sigma A' \Uparrow \textsc{type}$. At this point, we can apply rule **opc_eq** on $\mathcal{P}_2$, $\mathcal{R}^*$ and $\mathcal{P}^*$ and get a derivation $\mathcal{P}_2^*$ of $\overline{\Psi} \vdash_\Sigma N \Uparrow A'$. We can now apply intuitionistic transitivity (lemma 5.3.18) to $\mathcal{P}_1^*$ and $\mathcal{P}_2^*$, obtaining the desired derivation $\mathcal{P}'$ of $\Psi \vdash_\Sigma [N/x]M' \downarrow [N/x]B$. ☑

Corollary 5.3.29 (*Subject reduction for* $\longrightarrow^*$)

If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\downarrow V$ *and* $\mathcal{R} :: U \longrightarrow^* U'$, *then* $\mathcal{P}' :: \Psi \vdash_\Sigma U' \Uparrow\!\downarrow V$.

Proof.

If $U = U'$, the validity of this statement is trivial. Otherwise, we use the soundness and completeness lemma above to reduce to the transitive closure of the one-step case, changing the second premiss to $\mathcal{R}_1 :: U \longrightarrow_1^+ U'$. Then we proceed by induction on structure of $\mathcal{R}_1$. ☑

We now turn to proving a number of lemmas concerning the encoding of LLF terms into $\lambda^{\times\rightarrow}$. These results will be needed in the proof of the adequacy of the translation.

Lemma 5.3.30 (*Type-encoding under substitution*)

i. $\tau([N/x]A) = \tau(A)$;

ii. $\tau([N/x]K) = \tau(K)$.

Proof.

By induction on the structure of A and K. ☑

Lemma 5.3.31 (*Type-encoding of equivalent terms*)

Let U and U' be types or kinds. Then,

i. If $\mathcal{R} :: U \longrightarrow U'$, *then* $\tau(U) = \tau(U')$;

ii. If $\mathcal{R} :: U \longrightarrow^* U'$, *then* $\tau(U) = \tau(U')$;

iii. If $\mathcal{R} :: U \equiv U'$, *then* $\tau(U) = \tau(U')$;

Proof.

We proceed by induction on the structure of $\mathcal{R}$. The proof of (ii) relies on (i), and the proof of (iii) requires (ii). ☑

Lemma 5.3.32 (*Compositionality of the object-encoding*)

$|[N/x]U| = [|N|/x]|U|$.

Proof.

We proceed by induction on the structure of U. We show one case.

$\boxed{U = \hat{\lambda}y{:}A.\,M}$

$$
\begin{array}{lll}
 & |[N/x](\hat{\lambda}y{:}A.\,M)| & \\
= & |\hat{\lambda}y{:}[N/x]A.\,[N/x]M| & \text{by definition of substitution in } LLF, \\
= & (\lambda^s \bar{y}{:}\omega.\,\lambda^s y{:}\tau([N/x]A).\,|[N/x]M|)\,|[N/x]A| & \text{by the definition of the encoding,} \\
= & (\lambda^s \bar{y}{:}\omega.\,\lambda^s y{:}\tau(A).\,[|N|/x]|M|)\,[|N|/x]|A| & \text{by induction hypothesis and lemma 5.3.30,} \\
= & [|N|/x]((\lambda^s \bar{y}{:}\omega.\,\lambda^s y{:}\tau(A).\,|M|)\,|A|) & \text{by definition of substitution in } \lambda^{\times\rightarrow}, \\
= & [|N|/x]|\hat{\lambda}y{:}A.\,M| & \text{by the definition of the encoding.}
\end{array}
$$

☑

We are now ready to prove that the encoding proposed in Section 5.3.4 preserves LLF derivability. This will be an essential component of the proof of strong normalization to follow.

Lemma 5.3.33 (*Adequacy of the encoding*)

i. If $\mathcal{P} :: \Psi \vdash_\Sigma M \Uparrow\!\downarrow A$, *then* $\mathcal{S} :: \tau(\Psi) \vdash_{\tau(\Sigma)} |M| : \tau(A)$;

ii. If $\mathcal{P} :: \overline{\Psi} \vdash_\Sigma A \Uparrow\!\downarrow K$, *then* $\mathcal{S} :: \tau(\overline{\Psi}) \vdash_{\tau(\Sigma)} |A| : \tau(K)$;

iii. If $\mathcal{P} :: \overline{\Psi} \vdash_\Sigma K \Uparrow \mathit{Kind}$, *then* $\mathcal{S} :: \tau(\overline{\Psi}) \vdash_{\tau(\Sigma)} |K| : \omega$.

Proof.

We proceed by induction on the structure of $\mathcal{P}$. We present four among the most difficult cases.

opc_eq $\quad \mathcal{P} = \dfrac{\mathcal{P}_1 \quad \mathcal{R} \quad \mathcal{P}_2}{\Psi \vdash_\Sigma M \Uparrow B \quad B \equiv A \quad \overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE}}\ \textbf{opc\_eq}$

$\mathcal{S} :: \tau(\Psi) \vdash_{\tau(\Sigma)} |M| : \tau(B)$ — by induction hypothesis on $\mathcal{P}_1$,
$\tau(A) = \tau(B)$ — by lemma 5.3.31 on $\mathcal{R}$.

opc_llam $\quad \mathcal{P} = \dfrac{\mathcal{P}_1}{\Psi, x \hat{:} A_1 \vdash_\Sigma M_1 \Uparrow A_2}\ \textbf{opc\_llam}$

with $M = \hat{\lambda} x : A_1 . M_1$ and $A = A_1 \multimap A_2$.

$\mathcal{S}_1 :: \tau(\Psi), x : \tau(A_1) \vdash_{\tau(\Sigma)} |M_1| : \tau(A_2)$ — by induction hypothesis and definition of $\tau(\_)$,
$\mathcal{S}' :: \tau(\Psi) \vdash_{\tau(\Sigma)} \lambda^s x : \tau(A_1). |M_1| : \tau(A_1) \to^s \tau(A_2)$ — by rule **st_lam** on $\mathcal{S}_1$,
$\mathcal{S}'' :: \tau(\Psi), y : \omega \vdash_{\tau(\Sigma)} \lambda^s x : \tau(A_1). |M_1| : \tau(A_1 \multimap A_2)$ — by weakening in $\lambda^{\times\to}$ and definition of the encoding,
$\mathcal{S}''' :: \tau(\Psi) \vdash_{\tau(\Sigma)} \lambda^s y : \omega . \lambda^s x : \tau(A_1). |M_1| : \omega \to^s \tau(A_1 \multimap A_2)$ — by rule **st_lam** on $\mathcal{S}''$,
$\mathcal{P}' :: \vdash_\Sigma \Psi, x \hat{:} A_1 \Uparrow \mathit{Ctx}$ — by consistency on $\mathcal{P}_1$,
$\mathcal{P}'' :: \overline{\Psi} \vdash_\Sigma A_1 \Uparrow \text{TYPE}$ — by inversion on rule **cp_lin** for $\mathcal{P}'$,
$\mathcal{S}^* :: \tau(\overline{\Psi}) \vdash_{\tau(\Sigma)} |A_1| : \tau(\text{TYPE})$ — by induction hypothesis on $\mathcal{P}''$,
$\mathcal{S}^{**} :: \tau(\Psi) \vdash_{\tau(\Sigma)} |A_1| : \omega$ — by weakening in $\lambda^{\times\to}$ and definition of $\tau(\_)$,
$\mathcal{S} :: \tau(\Psi) \vdash_{\tau(\Sigma)} (\lambda^s y : \omega . \lambda^s x : \tau(A_1). |M_1|)\, |A_1| : \tau(A_1 \multimap A_2)$ — by rule **st_app** on $\mathcal{S}'''$ and $\mathcal{S}^{**}$.

opa_iapp $\quad \mathcal{P} = \dfrac{\mathcal{P}_1 \quad \mathcal{P}_2}{\Psi \vdash_\Sigma M_1 \downarrow \Pi x : A_2 . A_1 \quad \overline{\Psi} \vdash_\Sigma M_2 \Uparrow A_2}\ \textbf{opa\_iapp}$

with $M = M_1\ M_2$ and $A = [M_2/x]A_1$.

$\mathcal{S}_1 :: \tau(\Psi) \vdash_{\tau(\Sigma)} |M_1| : \tau(\Pi x : A_2 . A_1)$ — by induction hypothesis on $\mathcal{P}_1$,
$\mathcal{S}_1 :: \tau(\Psi) \vdash_{\tau(\Sigma)} |M_1| : \tau(A_2) \to^s \tau(A_1)$ — by definition of $\tau(\_)$,

$\mathcal{S}_2 :: \tau(\overline{\Psi}) \vdash_{\tau(\Sigma)} |M_2| : \tau(A_2)$ by induction hypothesis on $\mathcal{P}_2$ and weakening,
$\mathcal{S} :: \tau(\Psi) \vdash_{\tau(\Sigma)} |M_1|\,|M_2| : \tau([M_2/x]A_1)$ by rule **st_app** on $\mathcal{S}_1$ and $\mathcal{S}_2$, and lemma 5.3.30.

fpc_dep

$$\mathcal{P} = \frac{\mathcal{P}_1}{\dfrac{\overline{\Psi}, x:A_1 \vdash_\Sigma A_2 \Uparrow \text{TYPE}}{\overline{\Psi} \vdash_\Sigma \Pi x:A_1.\, A_2 \Uparrow \text{TYPE}}\,\textbf{fpc\_dep}}$$

with $A = \Pi x:A_1.\, A_2$ and $K = \text{TYPE}$.

$\mathcal{S}' :: \tau(\overline{\Psi}), x:\tau(A_1) \vdash_{\tau(\Sigma)} |A_2| : \omega$ by induction hypothesis and definition of $\tau(\_)$,
$\mathcal{S}_3 :: \tau(\overline{\Psi}) \vdash_{\tau(\Sigma)} \lambda^s x:\tau(A_1).\,|A_2| : \tau(A_1) \to^s \omega$ by rule **st_lam** on $\mathcal{S}_1$,
$\mathcal{P}' :: \vdash_\Sigma \overline{\Psi}, x:A_1 \Uparrow Ctx$ by consistency on $\mathcal{P}_1$,
$\mathcal{P}'' :: \overline{\Psi} \vdash_\Sigma A_1 \Uparrow \text{TYPE}$ by inversion on rule **cp_int** for $\mathcal{P}'$,
$\mathcal{S}_2 :: \tau(\overline{\Psi}) \vdash_{\tau(\Sigma)} |A_1| : \tau(\text{TYPE})$ by induction hypothesis on $\mathcal{P}''$,
$\mathcal{S}_1 :: \overline{\Psi} \vdash_{\tau(\Sigma)} \pi_{\tau(A_1)} : \omega \to^s (\tau(A_1) \to^s \omega) \to^s \omega$ by definition of $\tau(\Sigma)$ and rule **st_con**,
$\mathcal{S} :: \tau(\overline{\Psi}) \vdash_{\tau(\Sigma)} \pi_{\tau(A_1)}\,|A_1|\,(\lambda^s x:\tau(A_1).\,|A_2|) : \omega$ by two applications of rule **st_app** on $\mathcal{S}_1$, $\mathcal{S}_2$ and $\mathcal{S}_3$.
$\mathcal{S} :: \tau(\overline{\Psi}) \vdash_{\tau(\Sigma)} |\Pi x:A_1.\, A_2| : \tau(\text{TYPE})$ by definition of the encoding. ☑

Next, we show that the encoding preserves reductions, i.e. that every reduction in LLF corresponds to a sequence, in general, of reductions in $\lambda^{\times\to}$.

Lemma 5.3.34 (*Preservation of reduction sequences*)

If $\mathcal{R} :: U \longrightarrow_1 V$, *then* $\mathcal{R}^s :: |U| \xrightarrow{s}{}_1^+ |V|$.

Proof.

We proceed by induction on the structure of $\mathcal{R}$. We show two cases. All other cases are simpler.

or1_beta_lin

$$\mathcal{R} = \frac{}{(\hat{\lambda}x:A.\, M)\,\hat{}\,N \longrightarrow_1 [N/x]M}\,\textbf{or1\_beta\_lin}$$

with $U = (\hat{\lambda}x:A.\, M)\,\hat{}\,N$ and $V = [N/x]M$.

$\mathcal{R}'^s_1 :: (\lambda^s y:\omega.\,\lambda x:\tau(A).\,|M|)\,|A| \xrightarrow{s}{}_1 \lambda x:\tau(A).\,|M|$ by rule **str1_beta**,
$\mathcal{R}^s_1 :: (\lambda^s y:\omega.\,\lambda^s x:\tau(A).\,|M|)\,|A|\,|N| \xrightarrow{s}{}_1 (\lambda x:\tau(A).\,|M|)\,|N|$ by rule **str1_app1** on $\mathcal{R}'^s_1$,
$\mathcal{R}^s_2 :: (\lambda^s x:\tau(A).\,|M|)\,|N| \xrightarrow{s}{}_1 [|N|/x]|M|$ by rule **str1_beta**,
$\mathcal{R}^s :: (\lambda^s y:\omega.\,\lambda^s x:\tau(A).\,|M|)\,|A|\,|N| \xrightarrow{s}{}_1^+ [|N|/x]|M|$ by rules **st1*_r** and **st1*_trans** on $\mathcal{R}^s_1$ and $\mathcal{R}^s_2$,
$\mathcal{R}^s :: |(\hat{\lambda}x:A.\, M)\,\hat{}\,N| \xrightarrow{s}{}_1^+ |[N/x]M|$ by definition of the encoding and lemma 5.3.32.

fr1_with1 $$\mathcal{R} = \dfrac{\begin{array}{c}\mathcal{R}_1\\ A \longrightarrow_1 A'\end{array}}{A \,\&\, B \longrightarrow_1 A' \,\&\, B}\,\textbf{fr1\_with1}$$

with $U = A \,\&\, B$ and $V = A' \,\&\, B$.

$\mathcal{R}_1^s :: \lvert A\rvert \xrightarrow{s}{}_1^+ \lvert A'\rvert$	by induction hypothesis on $\mathcal{R}_1$,
$\mathcal{R}^s :: \pi_\& \lvert A\rvert \lvert B\rvert \xrightarrow{s}{}_1^+ \pi_\& \lvert A'\rvert \lvert B\rvert$	by two applications of the $\lambda^{\times\to}$ equivalent of lemma 5.3.3,
$\mathcal{R}^s :: \lvert A \,\&\, B\rvert \xrightarrow{s}{}_1^+ \lvert A' \,\&\, B\rvert$	by definition of the encoding. ☑

On the basis of these results, we can prove the the strong normalization theorem for *LLF*. For completeness, we report here the proof displayed in Section 5.3.4.

Theorem 5.3.35 (*Strong normalization*)

If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\downarrow V$, *then* U *is strongly normalizing.*

Proof.

By the adequacy of the translation (lemma 5.3.33), we have that $\tau(\Psi) \vdash_{\tau(\Sigma)} \lvert U\rvert : \tau(V)$ is derivable in $\lambda^{\times\to}$.

Assume ab absurdum that U is not strongly normalizing. Therefore, by the soundness of one-step reduction (lemma 5.3.25), there exists an infinite reduction sequence

$$U = U_0 \longrightarrow_1 U_1 \longrightarrow_1 \ldots$$

which, by reduction preservation (lemma 5.3.34), yields an infinite reduction sequence

$$\lvert U_0\rvert \xrightarrow{s}{}_1^+ \lvert U_1\rvert \xrightarrow{s}{}_1^+ \ldots$$

in $\lambda^{\times\to}$. However, this contradicts the strong normalization property for this language. ☑

We now prove a number of corollaries of this theorem. In particular, we prove that normal forms of derivable terms are unique and that their equivalence is decidable.

Corollary 5.3.36 (*Unicity of normal forms*)

If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\downarrow V$, $\mathcal{R}' :: U \longrightarrow^* U'$ *and* $\mathcal{R}'' :: U \longrightarrow^* U''$ *with both* U' *and* U'' *normal, then* $U' = U''$.

Proof.

By confluence, there exist a term V such that $U' \longrightarrow^* V$ and $U' \longrightarrow^* V$. However, since U' and U'' are normal, they do not contain β-redices, and therefore $U' = V = U''$. ☑

Corollary 5.3.37 (*Shape of normal forms*)

Let M, N, A, B *and* K *normalizable terms. We have that*

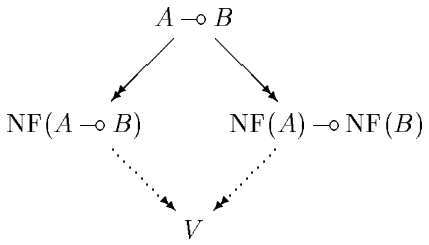
i. $\mathrm{NF}(\textsc{type}) = \textsc{type}$, $\mathrm{NF}(\Pi x\!:\!A.\,K) = \Pi x\!:\!\mathrm{NF}(A).\,\mathrm{NF}(B)$;

$$
\begin{array}{llcll}
\textit{ii}. & \mathrm{NF}(a) & = & a, & \mathrm{NF}(P\ M) = \mathrm{NF}(P)\,\mathrm{NF}(M), \\
 & \mathrm{NF}(\top) & = & \top, & \mathrm{NF}(A \,\&\, B) = \mathrm{NF}(A) \,\&\, \mathrm{NF}(B), \\
 & \mathrm{NF}(A \multimap B) & = & \mathrm{NF}(A) \multimap \mathrm{NF}(B), & \mathrm{NF}(\Pi x\!:\!A.\ B) = \Pi x\!:\!\mathrm{NF}(A).\ \mathrm{NF}(B); \\
\textit{iii}. & \mathrm{NF}(x) & = & x, & \mathrm{NF}(c) = c, \\
 & \mathrm{NF}(\langle\rangle) & = & \langle\rangle, & \mathrm{NF}(\langle M, N\rangle) = \langle \mathrm{NF}(M), \mathrm{NF}(N)\rangle, \\
 & \mathrm{NF}(\hat{\lambda} x\!:\!A.\ M) & = & \hat{\lambda} x\!:\!\mathrm{NF}(A).\ \mathrm{NF}(M), & \mathrm{NF}(\lambda x\!:\!A.\ M) = \lambda x\!:\!\mathrm{NF}(A).\ \mathrm{NF}(M), \\
 & \mathrm{NF}(\textsc{fst}\ M) & = & \textsc{fst}\ \mathrm{NF}(M) & \textit{if } M \neq \langle M', M''\rangle \\
 & \mathrm{NF}(\textsc{snd}\ M) & = & \textsc{snd}\ \mathrm{NF}(M) & \textit{if } M \neq \langle M', M''\rangle \\
 & \mathrm{NF}(M \hat{}\ N) & = & \mathrm{NF}(M) \hat{}\ \mathrm{NF}(N) & \textit{if } M \neq \hat{\lambda} x\!:\!A.\ M' \\
 & \mathrm{NF}(M\ N) & = & \mathrm{NF}(M)\ \mathrm{NF}(N) & \textit{if } M \neq \lambda x\!:\!A.\ M'
\end{array}
$$

Proof.

This corollary follows from the application of the confluence lemma 5.3.10 and the unicity of normal forms. We rely on the parallel nested reduction strategy in this proof, although the same result can be obtained with other strategies as well. We illustrate the technique by proving that $\mathrm{NF}(A \multimap B) = \mathrm{NF}(A) \multimap \mathrm{NF}(B)$. All other cases are similar.

An application of the confluence lemma permits the completion of the following diagram:

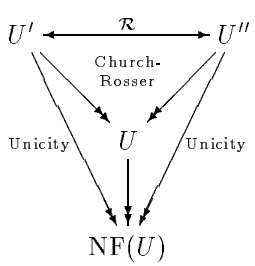


for some type V. However, $\mathrm{NF}(A \multimap B)$ is in normal form. Similarly, $\mathrm{NF}(A) \multimap \mathrm{NF}(B)$ cannot contain β-redices and therefore is in normal form also. By the unicity of normal forms, we have that $\mathrm{NF}(A \multimap B) = \mathrm{NF}(A) \multimap \mathrm{NF}(B)$ $(= V)$. ☑

Corollary 5.3.38 (*Decidability of the equational theory*)

If $\mathcal{P}' :: \Psi \vdash_\Sigma U' \Uparrow\!\downarrow V$ *and* $\mathcal{P}'' :: \Psi \vdash_\Sigma U'' \Uparrow\!\downarrow V$, *then it can be recursively decided whether* $U' \equiv U''$ *is derivable.*

Proof.



By the Church-Rosser property, U' and U'' have a common reduct U. By the subject reduction corollary 5.3.29, U is derivable. Therefore, by unicity (corollary 5.3.36), U', U and U'' share the same normal form $\mathrm{NF}(U)$. We have the diagram on the left.

By the strong normalization theorem, every sequence of reduction on U' and U'' eventually produces $\mathrm{NF}(U)$ after a finite number of steps. Therefore, a possible decision procedure for definitional equality is as follows: compute the normal forms of U' and U'' and then check whether they are equal. If they are, then U' is definitionally equal to U''. Otherwise, they are not equivalent. ☑

A further consequence of the strong normalization theorem is that we can normalize every entity in a derivable judgment, still preserving derivability.

Corollary 5.3.39 (*Normal forms*)

If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\downarrow V$, *then* $\mathcal{P}' :: \Psi \vdash_\Sigma \mathrm{NF}(U) \Uparrow\downarrow V$;

Proof.

By the strong normalization theorem, $U \longrightarrow^* \mathrm{NF}(U)$ is derivable. $\mathcal{P}'$ then results by subject reduction (corollary 5.3.29). ☑

Corollary 5.3.40 (*Normal forms*)

i. If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\downarrow V$, *then* $\mathcal{P}' :: \mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(U) \Uparrow\downarrow \mathrm{NF}(V)$;

ii. If $\mathcal{P} :: \vdash_\Sigma \Psi \Uparrow Ctx$, *then* $\mathcal{P}' :: \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(\Psi) \Uparrow Ctx$

iii. If $\mathcal{P} :: \vdash \Sigma \Uparrow Sig$, *then* $\mathcal{P}' :: \vdash \mathrm{NF}(\Sigma) \Uparrow Sig$.

Proof.

We first reduce U to normal form in (i) by means of the previous corollary, and then proceed by induction on the structure of $\mathcal{P}$. We show one case.

opa_iapp

$$\mathcal{P} = \dfrac{\begin{array}{cc}\mathcal{P}_1 & \mathcal{P}_2\\ \Psi \vdash_\Sigma M \downarrow \Pi x{:}A.\,B & \overline{\Psi} \vdash_\Sigma N \Uparrow A\end{array}}{\Psi \vdash_\Sigma M\ N \downarrow [N/x]B}\ \mathbf{opa\_iapp}$$

with $U = M\ N$, $V = [N/x]B$, and $(M\ N)$ in normal form.

$\mathcal{P}_1' :: \mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} M \downarrow \Pi x{:}\mathrm{NF}(A).\,\mathrm{NF}(B)$	by induction hypothesis on $\mathcal{P}_1$ and lemma 5.3.5,
$\mathcal{P}_2' :: \mathrm{NF}(\overline{\Psi}) \vdash_{\mathrm{NF}(\Sigma)} N \Uparrow \mathrm{NF}(A)$	by induction hypothesis on $\mathcal{P}_2$,
$\mathcal{P}_1^* :: \mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} M\ N \downarrow [N/x]\mathrm{NF}(B)$	by rule **opa_iapp** on $\mathcal{P}_1'$ and $\mathcal{P}_2'$,
$\mathcal{P}_1'' :: \overline{\Psi}, x{:}A \vdash_\Sigma B \Uparrow \mathrm{TYPE}$	by consistency on $\mathcal{P}_1$ and rule **fpc_dep**,
$\mathcal{R}' :: B \longrightarrow \mathrm{NF}(B)$	by strong normalization for $\mathcal{P}_1''$,
$\mathcal{R}'' :: [N/x]B \longrightarrow [N/x]\mathrm{NF}(B)$	by lemma 5.3.7 on $\mathcal{R}'$,
$\mathcal{P}^* :: \overline{\Psi} \vdash_\Sigma [N/x]B \Uparrow \mathrm{TYPE}$	by consistency on $\mathcal{P}$,
$\mathcal{R}''' :: [N/x]B \longrightarrow \mathrm{NF}([N/x]B)$	by strong normalization for $\mathcal{P}^*$,
$\mathcal{R}^* :: [N/x]\mathrm{NF}(B) \equiv \mathrm{NF}([N/x]B)$	by Church-Rosser on $\mathcal{R}''$ and $\mathcal{R}'''$, definition of normal form and rule **feq_red**,
$\mathcal{P}_2^* :: \mathrm{NF}(\overline{\Psi}) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}([N/x]B) \Uparrow \mathrm{TYPE}$	by corollary 5.3.39 on $\mathcal{P}^*$,
$\mathcal{P}' :: \mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} M\ N \downarrow \mathrm{NF}([N/x]B)$	by rule **opa_eq** on $\mathcal{P}_1^*$, $\mathcal{R}^*$ and $\mathcal{P}_2^*$. ☑

We conclude this section with the proof of the admissibility of rule **opa_c** for derivations of normal terms.

Lemma 5.3.41 (*Admissibility of* **opa_c**)

Let Σ^*, Ψ^*, M^*, A^* *and* K^* *be in normal form.*

i. *If* $\mathcal{P} :: \vdash \Sigma^* \Uparrow \mathit{Sig}$, *then* $\mathcal{P}' :: \vdash \Sigma^* \Uparrow \mathit{Sig}$

ii. *If* $\mathcal{P} :: \vdash_{\Sigma^*} \Psi^* \Uparrow \mathit{Ctx}$, *then* $\mathcal{P}' :: \vdash_{\Sigma^*} \Psi^* \Uparrow \mathit{Ctx}$

iii. *If* $\mathcal{P} :: \overline{\Psi^*} \vdash_{\Sigma^*} K^* \Uparrow \mathit{Kind}$, *then* $\mathcal{P}' :: \overline{\Psi^*} \vdash_{\Sigma^*} K^* \Uparrow \mathit{Kind}$

iv. *If* $\mathcal{P} :: \overline{\Psi^*} \vdash_{\Sigma^*} A^* \Uparrow\!\downarrow K$, *then* $\mathcal{P}' :: \overline{\Psi^*} \vdash_{\Sigma^*} A^* \Uparrow\!\downarrow K$

v. *If* $\mathcal{P} :: \Psi^* \vdash_{\Sigma^*} M^* \Uparrow\!\downarrow A$, *then*

- *if* A *is a type family or* $M^* = \top, \langle M, N\rangle, \hat{\lambda} x : A.\, M$ *or* $\lambda x : A.\, M$, *then* $\mathcal{P}' :: \Psi^* \vdash_{\Sigma^*} M^* \Uparrow A$
- *if* A *is a type family or* $M^* = c, x, \textsc{fst}\ M, \textsc{snd}\ M, M\hat{}N$ *or* $M\ N$, *then* $\mathcal{P}' :: \Psi^* \vdash_{\Sigma^*} M^* \downarrow A$

where $\mathcal{P}'$ *does not use the rule* ***opa_c****. Moreover, whenever* $\Psi' \vdash_{\Sigma^*} M' \downarrow A'$ *occurs in* $\mathcal{P}'$, *then* M' *is either a constant, a variable, a projection or one of the two applications.*

Proof.

We proceed by induction on the structure of $\mathcal{P}$. The difficult part of the proof concerns the judgments of the level of objects. We show two cases.

opc_pair
$$\mathcal{P} = \dfrac{\begin{array}{cc}\mathcal{P}_1 & \mathcal{P}_2\\ \Psi^* \vdash_{\Sigma^*} M_1^* \Uparrow A_1 & \Psi^* \vdash_{\Sigma^*} M_2^* \Uparrow A_2\end{array}}{\Psi^* \vdash_{\Sigma^*} \langle M_1^*, M_2^*\rangle \Uparrow A_1 \,\&\, A_2}\ \textbf{opc\_pair}$$

with $M^* = \langle M_1^*, M_2^*\rangle$ and $A = A_1 \,\&\, A_2$.

By the extensionality lemma 5.3.24, if A_1 is not a base type, then M_1^* must be a constructor. Therefore, by induction hypothesis on $\mathcal{P}_1$, there exists a derivation $\mathcal{P}_1'$ of $\Psi^* \vdash_{\Sigma^*} M_1^* \Uparrow A_1$ that does not use rule **opa_c** and such that whenever the judgment $\Psi' \vdash_{\Sigma^*} M' \downarrow A'$ occurs in $\mathcal{P}_1'$, then M' is either a constant, a variable, a projection or one of the two applications.

Similarly, we obtain a derivation $\mathcal{P}_2'$ of $\Psi^* \vdash_{\Sigma^*} M_2^* \Uparrow A_2$ with the same properties. Now, simply apply rule **opc_pair** to $\mathcal{P}_1'$ and $\mathcal{P}_2'$ in order to obtain the desired derivation $\mathcal{P}'$ of $\Psi^* \vdash_{\Sigma^*} \langle M_1^*, M_2^*\rangle \Uparrow A_1 \,\&\, A_2$ satisfying the statement of the lemma.

opa_fst
$$\mathcal{P} = \dfrac{\begin{array}{c}\mathcal{P}_1\\ \Psi^* \vdash_{\Sigma^*} M_1^* \downarrow A \,\&\, B\end{array}}{\Psi^* \vdash_{\Sigma^*} \textsc{fst}\ M_1^* \downarrow A}\ \textbf{opa\_fst}$$

with $M^* = \textsc{fst}\ M_1^*$.

By induction hypothesis on $\mathcal{P}_1$, we have two subcases to analyze:

- if $M_1^* = \top, \langle M, N\rangle, \hat{\lambda} x : A.\, M$ or $\lambda x : A.\, M$, then there exist a derivation $\mathcal{P}_1'$ of $\Psi^* \vdash_{\Sigma^*} M_1^* \Uparrow A \,\&\, B$ where $\mathcal{P}_1'$ does not use the rule **opa_c**.

  By the extensionality lemma 5.3.24, M_1^* can only have the form $\langle M, N\rangle$. However, if this happens, then M^* is not in normal form. Therefore, this case cannot apply.

- if $M_1^* = c, x, \text{FST}\ M, \text{SND}\ M, M\ \hat{}\ N$ or $M\ N$, then there exist a derivation $\mathcal{P}_1'$ of $\Psi^* \vdash_{\Sigma^*} M_1^* \downarrow A \,\&\, B$ such that $\mathcal{P}_1'$ does not use the rule **opa_c** and whenever $\Psi' \vdash_{\Sigma^*} M' \downarrow A'$ occurs in $\mathcal{P}_1'$, then M' is either a constant, a variable, a projection or one of the two applications. Then, $\mathcal{P}'$ is obtained by applying rule **opa_fst** to $\mathcal{P}_1'$. ☑

B.3.5 Algorithmic System

In this section, we give the proofs of the statements presented in Section 5.3.5. We have the following soundness and completeness results. Below, we also state the strengthening lemma for this system.

Theorem 5.3.42 (*Soundness of the algorithmic system*)

i. If $\mathcal{A} :: \Psi \vdash^a_\Sigma U \Uparrow\!\downarrow V$, *then* $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\downarrow V$ *and* V *is in normal form;*

ii. If $\mathcal{A} :: \vdash^a_\Sigma \Psi \Uparrow Ctx$, *then* $\mathcal{P} :: \vdash_\Sigma \Psi \Uparrow Ctx$;

iii. If $\mathcal{A} :: \vdash^a \Sigma \Uparrow Sig$, *then* $\mathcal{P} :: \vdash \Sigma \Uparrow Sig$.

Proof.

We proceed by induction on the structure of $\mathcal{A}$. We show one case. All other cases are easier.

$$\boxed{\textbf{oaa\_iapp}} \qquad \mathcal{A} = \dfrac{\begin{array}{cc}\mathcal{A}_1 & \mathcal{A}_2\end{array}}{\begin{array}{cc}\Psi \vdash^a_\Sigma M \downarrow \Pi x{:}A.\, B & \overline{\Psi} \vdash^a_\Sigma N \Uparrow A\end{array}}\ \textbf{oaa\_iapp}$$

$$\Psi \vdash^a_\Sigma M\ N \downarrow \text{NF}([N/x]B)$$

with $U = M\ N$ and $V = \text{NF}([N/x]B)$.

$\mathcal{P}_1' :: \Psi \vdash_\Sigma M \downarrow \Pi x{:}A.\, B$	by induction hypothesis on $\mathcal{A}_1$,
$\mathcal{P}_1'' :: \overline{\Psi} \vdash_\Sigma N \Uparrow A$	by induction hypothesis on $\mathcal{A}_2$,
$\mathcal{P}_1 :: \Psi \vdash_\Sigma M\ N \downarrow [N/x]B$	by rule **opa_iapp** on $\mathcal{P}_1'$ and $\mathcal{P}_1''$,
$\mathcal{P}_1^* :: \overline{\Psi} \vdash_\Sigma [N/x]B \Uparrow \text{TYPE}$	by consistency on $\mathcal{P}_1$,
$\mathcal{R} :: [N/x]B \equiv \text{NF}([N/x]B)$	by strong normalization and rule **feq_red**,
$\mathcal{P}_2 :: \overline{\Psi} \vdash_\Sigma \text{NF}([N/x]B) \Uparrow \text{TYPE}$	by the normal forms corollary 5.3.39 on $\mathcal{P}_1^1$,
$\mathcal{P} :: \Psi \vdash_\Sigma M\ N \downarrow \text{NF}([N/x]B)$	by rule **opa_eq** on $\mathcal{P}_1$, $\mathcal{R}$ and $\mathcal{P}_2$. ☑

Theorem 5.3.43 (*Completeness of the algorithmic system*)

i. If $\mathcal{P} :: \Psi \vdash_\Sigma U \Uparrow\!\downarrow V$, *then* $\mathcal{A} :: \Psi \vdash^a_\Sigma U \Uparrow\!\downarrow \text{NF}(V)$;

ii. If $\mathcal{P} :: \vdash_\Sigma \Psi \Uparrow Ctx$, *then* $\mathcal{A} :: \vdash^a_\Sigma \Psi \Uparrow Ctx$;

iii. If $\mathcal{P} :: \vdash \Sigma \Uparrow Sig$, *then* $\mathcal{A} :: \vdash^a \Sigma \Uparrow Sig$.

Proof.

By induction on the structure of $\mathcal{P}$ and the normal forms corollary 5.3.39. We show one case of the proof.

$$\boxed{\textbf{opc\_llam}} \quad \mathcal{P} = \dfrac{\mathcal{P}'}{\dfrac{\Psi, x \hat{:} A \vdash_\Sigma M \Uparrow B}{\Psi \vdash_\Sigma \hat{\lambda} x : A.\, M \Uparrow A \multimap B}\,\textbf{opc\_llam}}$$

with $U = \hat{\lambda} x : A.\, M$ and $V = A \multimap B$.

$\mathcal{A}' :: \Psi, x \hat{:} A \vdash^a_\Sigma M \Uparrow \mathrm{NF}(B)$	by induction hypothesis on $\mathcal{P}'$,
$\mathcal{A} :: \Psi \vdash^a_\Sigma \hat{\lambda} x : A.\, M \Uparrow \mathrm{NF}(A) \multimap \mathrm{NF}(B)$	by rule **oca_llam** on $\mathcal{A}'$,
$\mathcal{A} :: \Psi \vdash^a_\Sigma \hat{\lambda} x : A.\, M \Uparrow \mathrm{NF}(A \multimap B)$	by corollary 5.3.37 ☑

The strengthening lemma for the algorithmic system can then be proved by straightforward induction. On the basis of these result, we will then be able to prove this lemma for our original pre-canonical system.

Lemma 5.3.44 (*Strengthening*)

i. If $\mathcal{A} :: \Psi, x :\!\hat{:} A, \Psi' \vdash^a_\Sigma U \Uparrow\!\downarrow V$ *and* $x \notin \mathrm{FV}(\Psi') \cup \mathrm{FV}(U) \cup \mathrm{FV}(V)$, *then* $\mathcal{A}' :: \Psi, \Psi' \vdash^a_\Sigma U \Uparrow\!\downarrow V$;

ii. If $\mathcal{A} :: \vdash^a_\Sigma \Psi, x :\!\hat{:} A, \Psi' \Uparrow Ctx$ *and* $x \notin \mathrm{FV}(\Psi')$, *then* $\mathcal{A} :: \vdash^a_\Sigma \Psi, \Psi' \Uparrow Ctx$.

Proof.

By induction on the structure of $\mathcal{A}$. ☑

Lemma 5.3.45 (*Strengthening*)

i. If $\mathcal{P} :: \Psi, x :\!\hat{:} A, \Psi' \vdash_\Sigma U \Uparrow\!\downarrow V$ *and* $x \notin \mathrm{FV}(\Psi') \cup \mathrm{FV}(U) \cup \mathrm{FV}(V)$, *then* $\mathcal{P}' :: \Psi, \Psi' \vdash_\Sigma U \Uparrow\!\downarrow V$;

ii. If $\mathcal{P} :: \vdash_\Sigma \Psi, x :\!\hat{:} A, \Psi' \Uparrow Ctx$ *and* $x \notin \mathrm{FV}(\Psi')$, *then* $\mathcal{P}' :: \vdash_\Sigma \Psi, \Psi' \Uparrow Ctx$.

Proof.

By lemma 5.3.2 and the definition of normal form, we know that $\mathrm{FV}(\mathrm{NF}(V)) \subseteq \mathrm{FV}(V)$. Therefore, by the completeness theorem 5.3.43, we can transfer the premisses of this lemma in the algorithmic setting. We now apply the strengthening lemma for this system and get algorithmic versions of the conclusion of the two parts of this statement. Then, we take these results back in the pre-canonical system by means of the soundness theorem 5.3.42. ☑

The soundness and completeness results above allow us to import in the algorithmic system a number of properties proved in the previous sections in the pre-canonical case. In particular, the following properties are valid in the algorithmic presentation:

	Lemma 5.3.12	(*Free variables*)
	Lemma 5.3.13	(*Intuitionistic context*)
	Lemma 5.3.14	(*Types in the context*)
	Lemma 5.3.15	(*Structural properties*)
	Lemma 5.3.17	(*Flattening*)
	Lemma 5.3.18	(*Intuitionistic transitivity*)
	Lemma 5.3.19	(*Consistency*)
	Lemma 5.3.20	(*Linear transitivity*)
(∗)	Lemma 5.3.21	(*Unicity of types and kinds*)
(∗)	Lemma 5.3.22	(*Inversion*)
(∗)	Lemma 5.3.23	(*Inversion on object constructors*)
(∗)	Lemma 5.3.24	(*Extensionality*)

The results marked as (∗) rely on the definitional equality $\equiv$ on terms on the right-hand side of the main judgments. Their algorithmic version replace this relation with the syntactic identity $=$, since every derivable term in that position is in normal form.

A further property of the algorithmic system is the admissibility of rule **oaa_c**. We cannot however obtain it by soundness and completeness. We must instead redo, almost identically, the proof of that lemma. Nevertheless, we assume its validity also in the algorithmic setting and will, improperly, refer to it as lemma 5.3.41.

In the following, we will feel free to use these results when dealing with algorithmic judgments and derivations. We will refer to the numbering provided for the pre-canonical system.

B.3.6 Decidability

In this section, we report the proofs of the statements given in Subsection 5.3.6. The main property will be the decidability of type checking. We first prove the following lemma concerning the size of derivations.

Lemma 5.3.46 (*Upper bound on the size of a derivation*)

i. Let $h = \|\Psi \vdash^a_\Sigma U \Uparrow\Downarrow V\|$. *If* $\mathcal{A} :: \Psi \vdash^a_\Sigma U \Uparrow\Downarrow V$, *then* $\mathcal{A}' :: \Psi \vdash^a_\Sigma U \Uparrow\Downarrow V$,

ii. Let $h = \| \vdash^a_\Sigma \Psi \Uparrow Ctx\|$. *If* $\mathcal{A} :: \vdash^a_\Sigma \Psi \Uparrow Ctx$, *then* $\mathcal{A}' :: \vdash^a_\Sigma \Psi \Uparrow Ctx$,

iii. Let $h = \| \vdash^a \Sigma \Uparrow Sig\|$. *If* $\mathcal{A} :: \vdash^a \Sigma \Uparrow Sig$, *then* $\mathcal{A}' :: \vdash^a \Sigma \Uparrow Sig$,

where $\mathcal{A}'$ *has height less than* $2h$ *and contains at most* 3^{2h} *nodes.*

Proof.

$\mathcal{A}'$ is obtained from $\mathcal{A}$ by collapsing every chain of rules consisting uniquely in the alternation of **oca_a** and **oaa_c**. The bound on $\mathcal{A}'$ is then established by noticing that, when going from the conclusion to the premisses of introduction and elimination rules, the size of the judgments decreases. The factor 2 takes into account possible intervening applications of rules **fca_a**, **oca_a** and **oaa_c** for which the size of the premiss and the conclusion are the same. We present one case; the others are similar or easier.

oaa_c

$$\mathcal{A} = \dfrac{\begin{array}{c}\mathcal{A}_1 \\ \Psi \vdash^a_\Sigma M \Uparrow A\end{array}}{\Psi \vdash^a_\Sigma M \downarrow A}\,\textbf{oaa\_c}$$

with $U = M$ and $V = A$.

Before we start, it will be convenient to adopt some notation. By the definition of the size functions, $\|\Psi \vdash^a_\Sigma M \downarrow A\| = \|\Psi \vdash^a_\Sigma M \Uparrow A\|$. Let h be this value.

We proceed by inversion on the structure of $\mathcal{A}_1$. We show two of the five possible cases.

- $$\mathcal{A}_1 = \dfrac{\begin{array}{cc}\mathcal{A}_{11} & \mathcal{A}_{12} \\ \Psi \vdash^a_\Sigma M_1 \Uparrow A_1 & \Psi \vdash^a_\Sigma M_2 \Uparrow A_2\end{array}}{\Psi \vdash^a_\Sigma \langle M_1, M_2\rangle \Uparrow A_1 \,\&\, A_2}\,\mathbf{oca\_pair}$$

  with $M = \langle M_1, M_2\rangle$ and $A = A_1 \,\&\, A_2$. Let $h_1 = \|\Psi \vdash^a_\Sigma M_1 \Uparrow A_1\|$ and $h_2 = \|\Psi \vdash^a_\Sigma M_2 \Uparrow A_2\|$.

  By induction hypothesis, there exist derivations $\mathcal{A}'_{11}$ and $\mathcal{A}'_{12}$ of $\Psi \vdash^a_\Sigma M_1 \Uparrow A_1$ and $\Psi \vdash^a_\Sigma M_2 \Uparrow A_2$ respectively, with respective heights $h_{\mathcal{A}'_{11}} < 2h_1$ and $h_{\mathcal{A}'_{12}} < 2h_2$.

  Let $\mathcal{A}'_1$ the derivation of $\Psi \vdash^a_\Sigma \langle M_1, M_2\rangle \Uparrow A_1 \,\&\, A_2$ obtained by applying rule **oca_pair** to $\mathcal{A}'_{11}$ and $\mathcal{A}'_{12}$.

  We get the derivation $\mathcal{A}'$ of $\Psi \vdash^a_\Sigma \langle M_1, M_2\rangle \downarrow A_1 \,\&\, A_2$ by applying rule **oaa_c** to $\mathcal{A}'_1$. Its height $h_{\mathcal{A}'}$ is the largest between $h_{\mathcal{A}'_{11}} + 2$ and $h_{\mathcal{A}'_{12}} + 2$. We want to prove that $h_{\mathcal{A}'} < 2h$.

  By definition of the size functions, $\|\langle M_1, M_2\rangle\| = 1 + \|M_1\| + \|M_2\|$. Thus, we can infer that $h \geq h_1 + 1$ and $h \geq h_2 + 1$. Therefore $2h \geq 2h_1 + 2$ and $2h \geq 2h_2 + 2$. Consequently, $h_{\mathcal{A}'} < 2h$.

- $$\mathcal{A}_1 = \dfrac{\begin{array}{c}\mathcal{A}_{11} \\ \Psi \vdash^a_\Sigma M \downarrow P\end{array}}{\Psi \vdash^a_\Sigma M \Uparrow P}\,\mathbf{oca\_a} \qquad \text{with } A = P.$$

  By induction hypothesis, there is a derivation $\mathcal{A}'$ of $\Psi \vdash^a_\Sigma M \downarrow P$ with height lower than h.

In order to prove that the resulting derivation $\mathcal{A}'$ contains at most 3^{2h} nodes, simply notice that every rule in the algorithmic system has at most three premisses. ☑

We now present the the proofs of the decidability of type checking and type synthesis, first for the algorithmic system and then for the pre-canonical system. We first need to prove the decidability of context splitting.

Lemma 5.3.47 (*Decidability of context splitting*)

It can be recursively decided whether $\Psi = \Psi' \bowtie \Psi''$ is derivable. Moreover, given a context Ψ, there are exactly $2^{l(\Psi)}$ pairs (Ψ', Ψ'') such that $\Psi = \Psi' \bowtie \Psi''$, where $l(\Psi)$ is the number of linear assumptions in Ψ.

Proof.

We proceed by induction on the structure of Ψ.

- If $\Psi = \cdot$, then by inversion, it must be the case that $\Psi' = \cdot$ and $\Psi'' = \cdot$. Moreover, this is the only possible split.
- If $\Psi = \Psi_1, x : A$, then by inversion $\Psi' = \Psi'_1, x : A$ and $\Psi'' = \Psi''_1, x : A$. By induction hypothesis, it can be decided whether $\Psi_1 = \Psi'_1 \bowtie \Psi''_1$ and there are $2^{l(\Psi_1)}$ possible splits of Ψ_1. Since **s_int** is the only applicable rule and the only operation it involves is appending an item to a sequence, the validity of $\Psi = \Psi' \bowtie \Psi''$ is decidable. Moreover, the number of possible splits does not change, and $l(\Psi_1) = l(\Psi_1, x : A)$.

- If $\Psi = \Psi_1, x \hat{:} A$, then by inversion either $\Psi' = \Psi'_1, x \hat{:} A$ or $\Psi'' = \Psi''_1, x \hat{:} A$ depending on whether rule **s_lin1** or rule **s_lin2** has been applied. We consider the first case. By induction hypothesis, it can be decided whether $\Psi_1 = \Psi'_1 \bowtie \Psi''$ is derivable and the number of splits of Ψ_1 is $2^{l(\Psi_1)}$. The decidability of $\Psi = \Psi' \bowtie \Psi''$ then results from the application of rule **s_lin1**. We proceed similarly in the other case. Since there are two possible cases, each involving $2^{l(\Psi_1)}$ splits, the total number of possible splits is $2 \times 2^{l(\Psi_1)} = 2^{l(\Psi_1)+1} = 2^{l(\Psi_1, x \hat{:} A)}$. ☑

Lemma 5.3.48 (*Decidability of type checking in the algorithmic system*)

i. (Type checking)

It can be recursively decided whether $\Psi \vdash^a_\Sigma U \Uparrow\!\Downarrow V$, $\vdash^a_\Sigma \Psi \Uparrow Ctx$ *and* $\vdash^a \Sigma \Uparrow Sig$ *are derivable.*

ii. (Type synthesis)

Given a signature Σ, *a context* Ψ *and a term* U, *there is a recursive procedure that computes a term* V *such that the judgment* $\Psi \vdash^a_\Sigma U \Uparrow\!\Downarrow V$ *is derivable, or determines that no such* V *exists.*

Proof.

Let $H = 2||\Psi \vdash^a_\Sigma U \Uparrow\!\Downarrow V||$, $H = 2|| \vdash^a_\Sigma \Psi \Uparrow Ctx||$ or $H = 2|| \vdash^a \Sigma \Uparrow Sig||$, depending on the judgment we are operating on. Notice that H does not depend on the term V in the first case, therefore, this value is available also for the type synthesis problem. Let c be a counter, which initial value is 0 and that we will increment by one each time we succeed in matching the given judgment with one of the rules **fca_a**, **oca_a** and **oaa_c**. Instead, we reset it to zero when using another rule. If the premiss of **fca_a** matches some rule, then c will be reset immediately: no loops can occur at that level. At the level of objects, rules **oca_a** and **oaa_c** may alternate in sequence. We will give up the search for a derivation as soon as $c \geq H$, since, by lemma 5.3.46, if derivations for the judgments above exist at all, then one of them must have height less than H.

We proceed by induction, where the induction measure is defined as follows:

1. The premiss has the form $\Psi \vdash^a_\Sigma U \Uparrow\!\Downarrow V$.

   We order judgments so that $\Psi \vdash^a_\Sigma M \Uparrow\!\Downarrow A$ is larger than $\overline{\Psi} \vdash^a_\Sigma A \Uparrow \textsc{type}$, $\overline{\Psi} \vdash^a_\Sigma A \Uparrow\!\Downarrow K$ is larger than $\overline{\Psi} \vdash^a_\Sigma K \Uparrow Kind$, and in general $\Psi \vdash^a_\Sigma U \Uparrow\!\Downarrow V$ is larger than $\vdash^a_\Sigma \Psi \Uparrow Ctx$. If the size of the problem remains the same after these measures have been considered, we proceed by induction on the structure of M. If this measure does not change either, we rely on the value of $H - c$, as defined above.

2. The premiss has the form $\vdash^a_\Sigma \Psi \Uparrow Ctx$.

   This judgment is considered bigger than $\vdash^a \Sigma \Uparrow Sig$. It is bigger than $\Psi' \vdash^a_\Sigma U \Uparrow\!\Downarrow V$ if Ψ is bigger than Ψ'. Otherwise, we proceed by induction on the structure of Ψ.

3. The premiss has the form $\vdash^a \Sigma \Uparrow Sig$.

   This judgment is bigger than $\Psi \vdash^a_{\Sigma'} U \Uparrow\!\Downarrow V$ if Σ is bigger than Σ'. Otherwise, we proceed by induction on the structure of Σ.

We prove the decidability of both type checking and type synthesis at once. We show three representative cases.

$\boxed{U = \langle\rangle}$ First, we check whether $H - c = 0$. If so, we report a failure. In the following, we will assume that $H - c > 0$. We need to perform this check in all the cases involving objects. We will keep this issue implicit below.

If we are doing type checking, our second step is to check that the type V we are given is correct. The definition of the induction measure above ensures that the derivability of $\overline{\Psi} \vdash_\Sigma V \Uparrow \text{TYPE}$ can be decided. If the answer is positive, we can proceed, otherwise, we fail. Again, we assume this step in the remaining cases.

By inversion on the inference rules for the algorithmic system, a derivation of $\Psi \vdash^a_\Sigma \langle\rangle \downarrow V$, can be obtained only by means of rule **oaa_c** on the premiss $\Psi \vdash^a_\Sigma \langle\rangle \Uparrow V$. We can apply the induction hypothesis since H remains the same and c is incremented by one. If the answer of the decision procedure specifies that this judgment is not derivable, or returns a derivation of height greater than or equal to H, then we report a failure. Otherwise, by rule **oaa_c**, we have a derivation of the desired judgment. If we are interested in type synthesis, we simply return the type V' computed for it.

By inversion, the only inference rules that can yield a judgment of the form $\Psi \vdash^a_\Sigma \langle\rangle \Uparrow V$ are **oca_a** and **oca_unit**. In the first case, we proceed as above, with the additional constraint that V must be a base type.

In the second case, we must have $V = \top$. Nevertheless, we must apply the induction hypothesis (which is allowed by the measure defined above). If $\vdash^a_\Sigma \Psi \Uparrow Ctx$ is derivable, so is $\Psi \vdash^a_\Sigma \langle\rangle \Uparrow \top$. We can then return $V = \top$ in case of type synthesis, or check whether $V = \top$ in case of type checking.

If neither of these two cases applies, then there is no derivation for $\Psi \vdash^a_\Sigma \langle\rangle \Uparrow V$, whatever V is.

$\boxed{U = M \hat{} N}$ By inversion, $\Psi \vdash^a_\Sigma M \hat{} N \downarrow V$ can only be derived by means of rules **oaa_lapp** or **oaa_c**. In the latter case, we proceed as above. Otherwise, in order to apply the induction hypothesis, which is possible since the terms M and N in its premisses are smaller than $M \hat{} N$, we need to split the context. By the decidability of context splitting, there are only finitely many pairs (Ψ', Ψ'') such that $\Psi = \Psi' \bowtie \Psi''$. These pairs can be recursively generated by distributing the linear assumptions in Ψ in either Ψ' or Ψ'' while maintaining the order. It will be possible to type check the judgment above if there exists a split such that the types of the premisses can be inferred, and some other conditions hold (see below). Similarly for type synthesis. We will show that whenever more than one split yields valid derivations, they all compute the same types.

Let (Ψ', Ψ'') be a split of Ψ. We can apply the induction hypothesis (ii), synthesizing types $A' \multimap B$ and A'' for the judgments $\Psi' \vdash^a_\Sigma M \downarrow A' \multimap B$ and $\Psi'' \vdash^a_\Sigma N \Uparrow A''$, respectively. We first need to check that $A' = A''$. Then, in the case of type synthesis, we return B. In the case of type checking, we need to compare B with V. The consistency lemma ensures that B is derivable and we have checked beforehand that V is. Therefore, by the decidability of the definitional equality, we can compare B and V.

Assume that there are two or more splits (Ψ'_i, Ψ''_i) of Ψ such that $\Psi'_i \vdash^a_\Sigma M \downarrow A_i \multimap B_i$ and $\Psi''_i \vdash^a_\Sigma N \Uparrow A_i$ are derivable. By the free variables lemma 5.3.12, all Ψ'_i must declare the free variables in M. Therefore by unicity of types and kinds, lemma 5.3.21, all types $A_i \multimap B_i$ are equal, and consequently all B_i are the same type.

By inversion, $\Psi \vdash^a_\Sigma M \hat{} N \Uparrow V$ can only be derived by rule **oca_a**. Since the measure of the premiss is smaller, we can apply the induction hypothesis. If we are deciding type checking, we should fail if V is not a base type. If V is a base type, the induction hypothesis yields that $\Psi \vdash^a_\Sigma M \hat{} N \downarrow P$ is

derivable for some base type P. Then, we simply need to check whether $V = P$. If we are deciding type synthesis, the induction hypothesis yields a type V such that $\Psi \vdash^a_\Sigma M \hat{} N \downarrow V$ is derivable. We then need to check whether V is a base type in order to apply this rule. This check is clearly decidable.

$\boxed{U = M\ N}$ By inversion, $\Psi \vdash^a_\Sigma M\ N \downarrow V$ can only result from the application of rules **oaa_iapp** or **oaa_c**. In the latter case, we proceed as above. Similarly to the previous case, we infer types $\Pi x : A'.\, B$ and A'' for $\Psi \vdash^a_\Sigma M \downarrow \Pi x : A'.\, B$ and $\overline{\Psi} \vdash^a_\Sigma N \Uparrow A''$, respectively. After checking that $A' = A''$, we return $V = \mathrm{NF}([N/x]B)$ when deciding type synthesis. We instead compare $\mathrm{NF}([N/x]B)$ with the given type V in the case of type checking. In both cases, the normalization function is defined and computable.

We proceed as in the previous case in order to check whether $\Psi \vdash^a_\Sigma M\ N \Uparrow V$ is derivable or has a type. ☑

Theorem 5.3.49 (*Decidability of type checking*)

i. (Type checking)
It can be recursively decided whether $\Psi \vdash_\Sigma U \Uparrow\!\downarrow V$, $\vdash_\Sigma \Psi \Uparrow Ctx$ and $\vdash \Sigma \Uparrow Sig$ are derivable.

ii. (Type synthesis)
Given a signature Σ, a context Ψ and a term U, there is a recursive procedure that computes a term V such that the judgment $\Psi \vdash_\Sigma U \Uparrow\!\downarrow V$ is derivable, or determines that no such V exists.

Proof.

We want to reduce the proof of this statement to the similar problem we just solved in the algorithmic case. We must remember, however, that the term V in an algorithmic judgment $\Psi \vdash_\Sigma U \Uparrow\!\downarrow V$ is always canonical. This is perfectly acceptable if we are interested in synthesizing V: by consistency and strong normalization, if no canonical V permits deriving this judgment, then no V does. In the case of type checking, we need to check first whether V is normalizable, and if it is substitute it with its normal form. Indeed, if V is a type, by the previous lemma the problem of checking whether $\overline{\Psi} \vdash_\Sigma V \Uparrow \mathrm{TYPE}$ is derivable can be decided. If this judgment is indeed derivable, we know by the strong normalization theorem that $\mathrm{NF}(V)$ is defined and we can compute it. We proceed similarly if V is a kind.

By the previous lemma, we can recursively decide the problems of type checking and type synthesis for $\Psi \vdash^a_\Sigma U \Uparrow\!\downarrow V'$, $\vdash_\Sigma \Psi \Uparrow Ctx$ and $\vdash \Sigma \Uparrow Sig$ (where V' is a place holder in the case of type synthesis, and $\mathrm{NF}(V)$ for type checking). If the decision procedure for the algorithmic system gives a positive answer, then, we use the soundness theorem 5.3.42 to answer positively the question in the pre-canonical case. If the answer is instead negative, the completeness theorem 5.3.43 ensures that there are no derivations of these judgments in the pre-canonical system. ☑

B.4 Canonical Forms

In this section, we present the proofs of the results given in Section 5.4. Specifically, we prove the soundness and completeness of the canonical system for *LLF* with respect to the algorithmic system presented in Subsection 5.3.5. We will then prove that *LLF* is a conservative extension over *LF*.

First we relate the canonical and the algorithmic systems for *LLF*.

Theorem 5.4.1 (*Soundness of the canonical system*)

i. If $\mathcal{C}_\Sigma :: \vdash \Sigma \Uparrow Sig$, $\mathcal{C}_\Psi :: \vdash_\Sigma \Psi \Uparrow Ctx$ *and* $\mathcal{C} :: \Psi \vdash_\Sigma U \Uparrow\!\Downarrow V$, *then* $\mathcal{A} :: \Psi \vdash^a_\Sigma U \Uparrow\!\Downarrow V$;

ii. If $\mathcal{C}_\Sigma :: \vdash \Sigma \Uparrow Sig$ *and* $\mathcal{C} :: \vdash_\Sigma \Psi \Uparrow Ctx$, *then* $\mathcal{A} :: \vdash^a_\Sigma \Psi \Uparrow Ctx$;

iii. If $\mathcal{C} :: \vdash \Sigma \Uparrow Sig$, *then* $\mathcal{A} :: \vdash^a \Sigma \Uparrow Sig$.

Moreover, Σ, Ψ, U *and* V *are in normal form.*

Proof.

By induction on the structure of $\mathcal{C}$. A simple application of the inversion principle shows that in all the cases the judgment in the conclusion of the applied rules is normal assuming that the judgments in the premisses are. We show two cases.

kc_dep

$$\mathcal{C} = \dfrac{\mathcal{C}_1 \qquad \mathcal{C}_2}{\dfrac{\overline{\Psi} \vdash_\Sigma A \Uparrow \text{TYPE} \quad \overline{\Psi}, x:A \vdash_\Sigma K \Uparrow Kind}{\overline{\Psi} \vdash_\Sigma \Pi x:A.\, K \Uparrow Kind}\,\mathbf{kc\_dep}}$$

with $U = \Pi x:A.\, K$ and $V = Kind$ (improperly).

$\mathcal{C}'_\Psi :: \vdash_\Sigma \overline{\Psi}, x:A \Uparrow Ctx$	by rule **sc_obj** on $\mathcal{C}_\Psi$ and $\mathcal{C}_1$,
$\mathcal{A}_2 :: \overline{\Psi}, x:A \vdash^a_\Sigma K \Uparrow Kind$	and
$\overline{\Psi}$, A and K are in normal form	by induction hypothesis on $\mathcal{C}_2$ with $\mathcal{C}'_\Psi$ and $\mathcal{C}_\Sigma$,
$\mathcal{A} :: \overline{\Psi} \vdash^a_\Sigma \Pi x:A.\, K \Uparrow Kind$	by rule **kca_dep** on $\mathcal{A}_2$ since A is normal,
$\Pi x:A.\, K$ is in normal form	by corollary 5.3.37.

oa_lapp

$$\mathcal{C} = \dfrac{\mathcal{C}_1 \qquad \mathcal{C}_2 \qquad \mathcal{S}}{\dfrac{\Psi_1 \vdash_\Sigma M \downarrow A \multimap B \quad \Psi_2 \vdash_\Sigma N \Uparrow A \quad \Psi = \Psi_1 \bowtie \Psi_2}{\Psi \vdash_\Sigma M\,\hat{}\,N \downarrow B}\,\mathbf{oa\_lapp}}$$

with $U = M\,\hat{}\,N$ and $V = B$.

$\mathcal{C}'_\Psi :: \vdash_\Sigma \Psi_1 \Uparrow Ctx$ and $\mathcal{C}''_\Psi :: \vdash_\Sigma \Psi_2 \Uparrow Ctx$	by lemma 5.3.13 on $\mathcal{S}$ and $\mathcal{C}_\Psi$,
$\mathcal{A}_1 :: \Psi_1 \vdash^a_\Sigma M \downarrow A \multimap B$	and
Ψ_1, M, A and B are in normal form	by induction hypothesis on $\mathcal{C}_1$ with $\mathcal{C}'_\Psi$,
$\mathcal{A}_2 :: \Psi_2 \vdash^a_\Sigma N \Uparrow A$	and
Ψ_2 and N are in normal form	by induction hypothesis on $\mathcal{C}_2$ with $\mathcal{C}''_\Psi$,
$\mathcal{A} :: \Psi \vdash^a_\Sigma M\,\hat{}\,N \downarrow B$	by rule **oaa_lapp** on $\mathcal{A}_1$ and $\mathcal{A}_2$
Ψ, $(M\ N)$ and B are in normal form	by an easy induction and corollary 5.3.37. ☑

Theorem 5.4.2 (*Completeness of the canonical system*)

i. If $\mathcal{A} :: \Psi \vdash^a_\Sigma M \Uparrow\!\Downarrow A$, *then*

- *if* A *is a type family, then* $\mathcal{C} :: \text{NF}(\Psi) \vdash_{\text{NF}(\Sigma)} \text{NF}(M) \Uparrow\!\Downarrow A$,

- *if* $\mathrm{NF}(M) = \langle\rangle, \langle M', M''\rangle, \hat{\lambda}x{:}A'.\, M'$ *or* $\lambda x{:}A'.\, M'$, *then* $\mathcal{C} :: \mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(M) \Uparrow A$
- *if* $\mathrm{NF}(M) = x, c, \mathrm{FST}\ M', \mathrm{SND}\ M', M'\hat{}M''$ *or* $M'\ M''$, *then* $\mathcal{C} :: \mathrm{NF}(\Psi) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(M) \downarrow A$

ii. *If* $\mathcal{A} :: \overline{\Psi} \vdash^a_\Sigma A \Uparrow\!\!\downarrow K$, *then* $\mathcal{C} :: \mathrm{NF}(\overline{\Psi}) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(A) \Uparrow\!\!\downarrow K$;

iii. *If* $\mathcal{A} :: \overline{\Psi} \vdash^a_\Sigma K \Uparrow \mathit{Kind}$, *then* $\mathcal{C} :: \mathrm{NF}(\overline{\Psi}) \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(K) \Uparrow \mathit{Kind}$;

iv. *If* $\mathcal{A} :: \vdash^a_\Sigma \Psi \Uparrow \mathit{Ctx}$, *then* $\mathcal{C} :: \vdash_{\mathrm{NF}(\Sigma)} \mathrm{NF}(\Psi) \Uparrow \mathit{Ctx}$;

v. *If* $\mathcal{A} :: \vdash^a \Sigma \Uparrow \mathit{Sig}$, *then* $\mathcal{C} :: \vdash \mathrm{NF}(\Sigma) \Uparrow \mathit{Sig}$.

Proof.

By the normal form corollary 5.3.40, there exist derivations $\mathcal{A}'$ the judgments in the premisses that mention only terms in normal form (recall that the terms on the right-hand side of an algorithmic judgment are always normal). By the admissibility of **oaa_c** (lemma 5.3.41), we can further transform these derivations into derivations $\mathcal{A}''$ of the following form:

i. In case of object level terms, we have that

- if A is a type family or $\mathrm{NF}(M) = \langle\rangle, \langle M', M''\rangle, \hat{\lambda}x{:}A'.\, M'$ or $\lambda x{:}A'.\, M'$, then $\mathcal{A}'' :: \mathrm{NF}(\Psi) \vdash^a_{\mathrm{NF}(\Sigma)} \mathrm{NF}(M) \Uparrow A$
- if A is a type family or $\mathrm{NF}(M) = x, c, \mathrm{FST}\ M', \mathrm{SND}\ M', M'\hat{}M''$ or $M'\ M''$, then $\mathcal{A}'' :: \mathrm{NF}(\Psi) \vdash^a_{\mathrm{NF}(\Sigma)} \mathrm{NF}(M) \downarrow A$

ii. $\mathcal{A}'' :: \mathrm{NF}(\overline{\Psi}) \vdash^a_{\mathrm{NF}(\Sigma)} \mathrm{NF}(A) \Uparrow\!\!\downarrow K$;

iii. $\mathcal{A}'' :: \mathrm{NF}(\overline{\Psi}) \vdash^a_{\mathrm{NF}(\Sigma)} \mathrm{NF}(K) \Uparrow \mathit{Kind}$;

iv. $\mathcal{A}'' :: \vdash^a_{\mathrm{NF}(\Sigma)} \mathrm{NF}(\Psi) \Uparrow \mathit{Ctx}$;

v. $\mathcal{A}'' :: \vdash^a \mathrm{NF}(\Sigma) \Uparrow \mathit{Sig}$.

where $\mathcal{A}''$ does not contain any use of rule **oaa_c**, and whenever the judgment $\Psi' \vdash^a_\Sigma M' \downarrow A'$ occurs in $\mathcal{A}''$, then M' is either a constant, a variable, a projection or one of the two applications.

The completeness of the algorithmic system with respect to the pre-canonical system then results from an easy induction on the structure of $\mathcal{A}''$. ☑

As in Section B.3.5, the soundness and completeness results above give us for free a number of properties proved in the previous sections for the pre-canonical and the algorithmic case. In particular, the following properties are valid in the canonical system:

	Lemma 5.3.12	(*Free variables*)
	Lemma 5.3.13	(*Intuitionistic context*)
	Lemma 5.3.14	(*Types in the context*)
	Lemma 5.3.15	(*Structural properties*)
	Lemma 5.3.17	(*Flattening*)
	Lemma 5.3.18	(*Intuitionistic transitivity*)
	Lemma 5.3.19	(*Consistency*)
	Lemma 5.3.20	(*Linear transitivity*)
(∗)	Lemma 5.3.21	(*Unicity of types and kinds*)
(∗)	Lemma 5.3.22	(*Inversion*)
(∗)	Lemma 5.3.23	(*Inversion on object constructors*)
(∗)	Lemma 5.3.24	(*Extensionality*)
	Lemma 5.3.44	(*Strengthening*)

The results marked as (∗) rely on the definitional equality $\equiv$. Their canonical version replace this relation with the syntactic identity $=$, since every derivable term in this system is in normal form.

In the following, we will feel free to use these results when dealing with canonical judgments and derivations. We will refer to the numbering displayed above.

The next result in this section shows that LF is a sublanguage of LLF. We first prove that our language extends LF and then show that this conservation is conservative. In order to prove these results, we need a couple of simple lemmas stating that the LF fragment of the language of LLF is closed under some basic operations.

In the rest of this section, we will often use expressions such as "U is an LF term". By this, we mean that U belongs to the LF fragment of the language of LLF. We already noticed that our language extends the syntax of LF so that every syntactic entity in this language is valid in LLF.

Lemma 5.4.3 (*Closure of substitution for LF within LLF*)

If M and U are LF terms, then $[M/x]U$ is an LF term.

Proof.

By induction on the structure of U. ☑

Lemma 5.4.4 (*Closure of reduction for LF within LLF*)

If U is an LF term, and either $\mathcal{R} :: U \longrightarrow V$ or $\mathcal{R} :: U \longrightarrow^* V$, then V is an LF term.

Proof.

We proceed by induction on the structure of $\mathcal{D}$. ☑

We can now state and prove the results that show that LLF is a conservative extension over LF. First, we have that every canonical LF derivation is valid in the canonical system for LLF.

Theorem 5.4.5 (*Extension over LF*)

i. If $\mathcal{LF} :: \Gamma \vdash^{\mathrm{LF}}_{\Sigma} U \Uparrow\Downarrow V$, then $\mathcal{C} :: \Gamma \vdash_{\Sigma} U \Uparrow\Downarrow V$;

ii. If $\mathcal{LF} :: \vdash^{\mathrm{LF}}_{\Sigma} \Gamma \Uparrow Ctx$, then $\mathcal{C} :: \vdash_{\Sigma} \Gamma \Uparrow Ctx$;

iii. If $\mathcal{LF} :: \vdash^{\mathrm{LF}} \Sigma \Uparrow Sig$, then $\mathcal{C} :: \vdash \Sigma \Uparrow Sig$.

Proof.

We proceed by induction on the structure of $\mathcal{LF}$. All cases are immediate as soon as we notice that, for an LF context Γ, $\overline{\Gamma} = \Gamma$. ☑

Then, we have that whenever an LLF derivation mentions only entities in the LF fragment of this language, then it is isomorphic to an LF derivation for the corresponding judgment.

Theorem 5.4.6 (*Conservativity over LF*)

Let Σ, Γ, U and V be an LF signature, an LF context and two LF terms, respectively, then

i. If $\mathcal{C} :: \Gamma \vdash_\Sigma U \Uparrow\!\Downarrow V$, then $\mathcal{LF} :: \Gamma \vdash^{\mathrm{LF}}_\Sigma U \Uparrow\!\Downarrow V$;

ii. If $\mathcal{C} :: \vdash_\Sigma \Gamma \Uparrow Ctx$, then $\mathcal{LF} :: \vdash^{\mathrm{LF}}_\Sigma \Gamma \Uparrow Ctx$;

iii. If $\mathcal{C} :: \vdash \Sigma \Uparrow Sig$, then $\mathcal{LF} :: \vdash^{\mathrm{LF}} \Sigma \Uparrow Sig$.

Proof.

We proceed by induction on the structure of $\mathcal{C}$. In order to apply the induction hypothesis, we rely on the closure lemmas 5.4.3 and 5.4.4 to ensure that intermediate judgments appearing in this derivation still belong to the LF fragment of LLF. Again, we need to remember that for an LF context Γ, $\overline{\Gamma} = \Gamma$. ☑

B.5 Logic Programming in LLF

In this section, we have grouped the proofs of the statements presented in Section 5.5. We also present minor lemmas needed in these proofs.

B.5.1 Canonical Proofs

There are no properties to prove from Subsection 5.5.1.

B.5.2 Uniform Provability

In this section, we prove the soundness and completeness results that relate the canonical and the uniform provability system. We stated these theorems in Subsection 5.5.2.

Theorem 5.5.1 (*Soundness of uniform provability*)

i. If $\mathcal{U} :: \Psi \xrightarrow{u}_\Sigma M : A$, then $\mathcal{C} :: \Psi \vdash_\Sigma M \Uparrow A$;

ii. If $\mathcal{U} :: \Psi' \xrightarrow{u}_\Sigma M : A \gg N : P$ and $\mathcal{C}_M :: \Psi'' \vdash_\Sigma M \downarrow A$ with $\mathcal{S} :: \Psi = \Psi' \bowtie \Psi''$, then $\mathcal{C} :: \Psi \vdash_\Sigma N \downarrow P$.

Proof.

We proceed by induction on the structure of $\mathcal{U}$. We present one case.

$$\boxed{\textbf{i\_iapp\_dynamic}} \quad \mathcal{U} = \dfrac{\mathcal{U}_1 \qquad\qquad \mathcal{U}_2}{\Psi' \xrightarrow{u}_\Sigma M\ M' : \mathrm{NF}([M'/x]A_1) \gg N : P \quad \overline{\Psi'} \xrightarrow{u}_\Sigma M' : A_2}{\Psi' \xrightarrow{u}_\Sigma M : \Pi x : A_1.\, A_2 \gg N : P}$$

with $A = \Pi x : A_1.\, A_2$.

$\mathcal{C}' :: \overline{\Psi'} \vdash_\Sigma M' \Uparrow A_2$	by induction hypothesis on $\mathcal{U}_2$,
$\mathcal{C}'_M :: \Psi' \vdash_\Sigma M\ M' \downarrow \mathrm{NF}([M'/x]A_1)$	by rule **oa_iapp** on $\mathcal{C}_M$ and $\mathcal{C}'$,
$\mathcal{C} :: \Psi \vdash_\Sigma N \downarrow P$	by induction hypothesis on $\mathcal{U}_2$ and $\mathcal{C}'_M$. ☑

Theorem 5.5.2 (*Completeness of uniform provability*)

i. If $\mathcal{C} :: \Psi \vdash_\Sigma M \Uparrow A$, *then* $\mathcal{U} :: \Psi \xrightarrow{u}_\Sigma M : A$;

ii. If $\mathcal{C} :: \Psi' \vdash_\Sigma M \downarrow A$ *and* $\mathcal{U} :: \Psi'' \xrightarrow{u}_\Sigma M : A \gg N : P$ *with* $\mathcal{S} :: \Psi = \Psi' \bowtie \Psi''$, *then* $\mathcal{U}' :: \Psi \xrightarrow{u}_\Sigma N : P$.

Proof.

We proceed by induction on the structure of $\mathcal{C}$. ☑

B.5.3 Resolution

In this section, we report the proofs of the soundness and completeness of the resolution system presented in Subsection 5.5.3 with respect to the uniform provability system.

Lemma 5.5.3 (*Decomposition lemma*)

For any type A, base type P, terms M and N, there exist a G-term O and a G-formula G such that $M : A \gg N : P \setminus O : G$ is derivable.

Proof.

We proceed by induction on the structure of A. ☑

Theorem 5.5.4 (*Soundness of resolution*)

i. If $\mathcal{R} :: \Psi \xrightarrow{r}_\Sigma M : A$, *then* $\mathcal{U} :: \Psi \xrightarrow{u}_\Sigma M : A$;

ii. If $\mathcal{D} :: M : A \gg N : P \setminus O : G$ *and* $\mathcal{R} :: \Psi \xrightarrow{r}_\Sigma O : G$, *then* $\mathcal{U} :: \Psi \xrightarrow{u}_\Sigma M : A \gg N : P$.

Proof.

We proceed by mutual induction on the structure of $\mathcal{R}$ in the first part and $\mathcal{D}$ in the second. ☑

Theorem 5.5.5 (*Completeness of resolution*)

i. *If* $\mathcal{U} :: \Psi \xrightarrow{u}_{\Sigma} M : A$, *then* $\mathcal{R} :: \Psi \xrightarrow{r}_{\Sigma} M : A$;

ii. *If* $\mathcal{U} :: \Psi \xrightarrow{u}_{\Sigma} M : A \gg N : P$,
then $\mathcal{D} :: M : A \gg N : P \setminus O : G$ *and* $\mathcal{R} :: \Psi \xrightarrow{r}_{\Sigma} O : G$.

Proof.

By induction on the structure of $\mathcal{U}$. ☑

Appendice C

Type Preservation for *Mini-ML* with References

In this appendix, we have collected the *Elf* code and the adequacy theorems for the meta-representation of the type-preservation theorem for *Mini-ML* with references presented in Chapter 6.

C.1 *LLF* Source Code

In this section, we present the code for *MLR*, our extension of *Mini-ML* with memory cells, references to them, and the possibility of changing their values by means of assignment instructions. The program we present is an extension of the code displayed in Appendix A. The implementation language this time is *LLF*. Beyond the presence of new declarations dealing with the constructs of *MLR* not present in *Mini-ML*, the main difference is the use of linear linear rather than intuitionistic implication in the rules for evaluation and type preservation. We rely on the same names used in that appendix for declarations concerning the same entities.

The code is organized as follow: the syntax of *MLR* is presented in C.1.1, its typing rules in C.1.2, its evaluation rules in C.1.3, and finally the declaration for the type preservation theorem in C.1.4.

C.1.1 Syntax

```
exp    : type.  %name exp    E V
tp     : type.  %name tp     T
instr  : type.  %name instr  I
cont   : type.  %name cont   K
cell   : type.  %name cell   C
item   : type.  %name item   It
store  : type.  %name store  S
answer : type.  %name answer A


% Expressions
```

```
z       : exp.
s       : exp -> exp.
case    : exp -> exp -> (exp -> exp) -> exp.
pair    : exp -> exp -> exp.
fst     : exp -> exp.
snd     : exp -> exp.
lam     : (exp -> exp) -> exp.
app     : exp -> exp -> exp.
letval  : exp -> (exp -> exp) -> exp.
letname : exp -> (exp -> exp) -> exp.
fix     : (exp -> exp) -> exp.

rf      : cell -> exp.                          %prefix 80 rf
ref     : exp -> exp.
!       : exp -> exp.
noop    : exp.
;       : exp -> exp -> exp.                    %infix left  90  ;
is      : exp -> exp -> exp.                    %infix right 100 is


% Types

nat   : tp.
cross : tp -> tp -> tp.                         %infix   right 90  cross
arrow : tp -> tp -> tp.                         %infix   right 100 arrow

tref  : tp -> tp.                               %postfix       110 tref
cmd   : tp.


% Instructions

eval : exp -> instr.
return : exp -> instr.
case*  : exp -> exp -> (exp -> exp) -> instr.
pair*  : exp -> exp -> instr.
fst*   : exp -> instr.
snd*   : exp -> instr.
app*   : exp -> exp -> instr.

ref*   : exp -> instr.
deref* : exp -> instr.
is*1   : exp -> exp -> instr.                   %infix right 100 is*1
is*2   : exp -> exp -> instr.                   %infix right 100 is*2
```

```
% Continuations

init : cont.
-    : cont -> (exp -> instr) -> cont.          %infix left 80 -


% Store

estore : store.
and    : store -> item -> store.                %infix left 70 and
=      : cell -> exp -> item.                   %infix none 60 =


% Answers

close : store -> exp -> answer.
new   : (cell -> answer) -> answer.
```

C.1.2 Typing

```
tpc : cell -> tp -> type.        %name tpc Tc
tpe : exp -> tp -> type.         %name tpe Te
tpi : instr -> tp -> type.       %name tpi Ti
tpK : cont -> tp -> tp -> type.  %name tpK TK
tpS : store -> type.             %name tpS TS
tpa : answer -> tp -> type.      %name tpa Ta

%mode -tpc +C *Tc

%mode -tpe +E *Te
%lex E

%mode -tpi +I *Ti
%lex I

%mode -tpK +K *T1 *T2
%lex K

%mode -tpS +S
%lex S

%mode -tpa +A *Ta
%lex A


% Expressions
```

```
tpe_z       : tpe z nat.
tpe_s       : tpe (s E) nat
               <- tpe E nat.
tpe_case    : tpe (case E E1 E2) T
               <- tpe E nat
               <- tpe E1 T
               <- ({x:exp} tpe x nat -> tpe (E2 x) T).
tpe_pair    : tpe (pair E1 E2) (T1 cross T2)
               <- tpe E1 T1
               <- tpe E2 T2.
tpe_fst     : tpe (fst E) T1
               <- tpe E (T1 cross T2).
tpe_snd     : tpe (snd E) T2
               <- tpe E (T1 cross T2).
tpe_lam     : tpe (lam E) (T1 arrow T2)
               <- ({x:exp} tpe x T1 -> tpe (E x) T2).
tpe_app     : tpe (app E1 E2) T1
               <- tpe E1 (T2 arrow T1)
               <- tpe E2 T2.
tpe_letval  : tpe (letval E1 E2) T2
               <- tpe E1 T1
               <- ({x:exp} tpe x T1 -> tpe (E2 x) T2).
tpe_letname : tpe (letname E1 E2) T
               <- tpe (E2 E1) T.
tpe_fix     : tpe (fix E) T
               <- ({x:exp} tpe x T -> tpe (E x) T).

tpe_cell    : tpe (rf C) (T tref)
               <- tpc C T.
tpe_ref     : tpe (ref E) (T tref)
               <- tpe E T.
tpe_deref   : tpe (! E) T
               <- tpe E (T tref).
tpe_noop    : tpe noop cmd.
tpe_seq     : tpe (E1 ; E2) T2
               <- tpe E1 T1
               <- tpe E2 T2.
tpe_assign  : tpe (E1 is E2) cmd
               <- tpe E1 (T tref)
               <- tpe E2 T.


% Instructions

tpi_eval    : tpi (eval E) T
               <- tpe E T.
tpi_return  : tpi (return V) T
```

```
               <- tpe V T.
tpi_case*   : tpi (case* V E1 E2) T
               <- tpe V nat
               <- tpe E1 T
               <- ({x:exp} tpe x nat -> tpe (E2 x) T).
tpi_pair*   : tpi (pair* V E) (T1 cross T2)
               <- tpe V T1
               <- tpe E T2.
tpi_fst*    : tpi (fst* V) T1
               <- tpe V (T1 cross T2).
tpi_snd*    : tpi (snd* V) T2
               <- tpe V (T1 cross T2).
tpi_app*    : tpi (app* V E) T1
               <- tpe V (T2 arrow T1)
               <- tpe E T2.

tpi_ref*    : tpi (ref* V) (T tref)
               <- tpe V T.
tpi_deref*  : tpi (deref* V) T
               <- tpe V (T tref).
tpi_assign*1: tpi (V is*1 E) cmd
               <- tpe V (T tref)
               <- tpe E T.
tpi_assign*2: tpi (V1 is*2 V2) cmd
               <- tpe V1 (T tref)
               <- tpe V2 T.


% Continuations

tpK_init : tpK init T T.
tpK_lam  : tpK (K - I) T1 T2
            <- ({x:exp} tpe x T1 -> tpi (I x) T)
            <- tpK K T T2.


% Store

tpS_empty : tpS estore.
tpS_item  : tpS (S and (C = V))
             <- tpS S
             <- tpc C T
             <- tpe V T.


% Answers
```

```
tpa_close : tpa (close S V) T
             <- tpS S
             <- tpe V T.
tpa_new   : tpa (new A) T
             <- ({c: cell} tpc c T' -> tpa (A c) T).
```

C.1.3 Evaluation

```
%%% Contents of a cell

contains: cell -> exp -> type.   %name contains Cn

%mode -contains +C -V

% Only run-time assumptions



%%% Collection of all the cells in the store

collect: store -> type.          %name collect Col

col_empty : collect estore.
col_item  : collect (S and (C = V))
             o- contains C V
             o- collect S.

%mode -collect -S
%lex ASSUME TERMINATION



%%% Reading the value of a cell from the store

read: cell -> exp -> type.       %name read R

read_val  : read C V
             o- contains C V
             o- <T>.

%mode -read +C -V



%%% Evaluation

ev : cont -> instr -> answer -> type.   %name ev E
```

```
%mode -ev +C +I -A
%lex C


% Expressions

ev_z        : ev K (eval z) A
               o- ev K (return z) A.
ev_s        : ev K (eval (s E)) A
               o- ev (K - [x:exp] return (s x)) (eval E) A.
ev_case     : ev K (eval (case E E1 E2)) A
               o- ev (K - [x:exp] case* x E1 E2) (eval E) A.
ev_pair     : ev K (eval (pair E1 E2)) A
               o- ev (K - [x:exp] pair* x E2) (eval E1) A.
ev_fst      : ev K (eval (fst E)) A
               o- ev (K - [x:exp] fst* x) (eval E) A.
ev_snd      : ev K (eval (snd E)) A
               o- ev (K - [x:exp] snd* x) (eval E) A.
ev_lam      : ev K (eval (lam E)) A
               o- ev K (return (lam E)) A.
ev_app      : ev K (eval (app E1 E2)) A
               o- ev (K - [x:exp] app* x E2) (eval E1) A.
ev_letval   : ev K (eval (letval E1 E2)) A
               o- ev (K - [x:exp] eval (E2 x)) (eval E1) A.
ev_letname  : ev K (eval (letname E1 E2)) A
               o- ev K (eval (E2 E1)) A.
ev_fix      : ev K (eval (fix E)) A
               o- ev K (eval (E (fix E))) A.

ev_cell     : ev K (eval (rf C)) A
               o- ev K (return (rf C)) A.
ev_ref      : ev K (eval (ref E)) A
               o- ev (K - [x:exp] ref* x) (eval E) A.
ev_deref    : ev K (eval (! E)) A
               o- ev (K - [x:exp] deref* x) (eval E) A.
ev_noop     : ev K (eval noop) A
               o- ev K (return noop) A.
ev_seq      : ev K (eval (E1 ; E2)) A
               o- ev (K - [x:exp] eval E2) (eval E1) A.
ev_assign   : ev K (eval (E1 is E2)) A
               o- ev (K - [x:exp] x is*1 E2) (eval E1) A.


% Values

ev_init     : ev init (return V) (close S V)
```

```
              o- collect S.
ev_cont     : ev (K - I) (return V) A
              o- ev K (I V) A.


% Auxiliary instructions

ev_case*1   : ev K (case* z E1 E2) A
              o- ev K (eval E1) A.
ev_case*2   : ev K (case* (s V) E1 E2) A
              o- ev K (eval (E2 V)) A.
ev_pair*    : ev K (pair* V E) A
              o- ev (K - [x:exp] return (pair V x)) (eval E) A.
ev_fst*     : ev K (fst* (pair V1 V2)) A
              o- ev K (return V1) A.
ev_snd*     : ev K (snd* (pair V1 V2)) A
              o- ev K (return V2) A.
ev_app*     : ev K (app* (lam E1) E2) A
              o- ev (K - [x:exp] eval (E1 x)) (eval E2) A.

ev_ref*     : ev K (ref* V) (new A)
              o- ({c:cell} contains c V -o ev K (return (rf c)) (A c)).
ev_deref*   : ev K (deref* (rf C)) A
              o- read C V
              &  ev K (return V) A.
ev_assign*1 : ev K ((rf C) is*1 E) A
              o- ev (K - [x:exp] (rf C) is*2 x) (eval E) A.
ev_assign*2 : ev K ((rf C) is*2 V) A
              o- contains C V'
              o- (contains C V -o ev K (return noop) A).
```

C.1.4 Type Preservation

```
%%% Type preservation over cells

prCell : contains C V -> tpc C T -> tpe V T -> type.

%mode -prCell +Cn +Tc -Tv
%lex Cn

% Only run-time assumptions



%%% Type preservation over read values

prRead : read C V -> tpc C T -> tpe V T -> type.
```

```
%mode -prRead +R +Tc -Tv

prRead_val : prRead (read_val () Cn) Tc Tv
              o- prCell Cn Tc Tv
              &  <T>.



%%% Type preservation for collected values

prCollect : collect S -> tpS S -> type.

%mode -prCollect +Col -TS
%lex S

prCol_empty : prCollect col_empty tpS_empty.
prCol_item  : prCollect (col_item Col Cn) (tpS_item Tv Tc TS)
               o- prCell Cn Tc Tv
               o- prCollect Col TS.



%%% Type Preservation for evaluations

pr : ev K I A -> tpi I T -> tpK K T T' -> tpa A T' -> type.

%mode -pr +E +Ti +TK -Ta
%lex E


% Expressions

pr-ev_z        : pr (ev_z E) (tpi_eval tpe_z) TK Ta
                   o- pr E (tpi_return tpe_z) TK Ta.
pr-ev_s        : pr (ev_s E) (tpi_eval (tpe_s Te)) TK Ta
                   o- pr E
                         (tpi_eval Te)
                         (tpK_lam TK [x][t:tpe x nat] tpi_return (tpe_s t))
                         Ta.
pr-ev_case     : pr (ev_case E) (tpi_eval (tpe_case Te'' Te' Te)) TK Ta
                   o- pr E
                         (tpi_eval Te)
                         (tpK_lam TK [x][t:tpe x nat] tpi_case* Te'' Te' t)
                         Ta.
pr-ev_pair     : pr (ev_pair E) (tpi_eval (tpe_pair Te'' Te')) TK Ta
                   o- pr E
```

```
                    (tpi_eval Te')
                    (tpK_lam TK [x][t:tpe x T'] tpi_pair* Te'' t)
                    Ta.
pr-ev_fst      : pr (ev_fst E) (tpi_eval (tpe_fst Te)) TK Ta
                  o- pr E
                    (tpi_eval Te)
                    (tpK_lam TK [x][t:tpe x (T' cross T'')] tpi_fst* t)
                    Ta.
pr-ev_snd      : pr (ev_snd E) (tpi_eval (tpe_snd Te)) TK Ta
                  o- pr E
                    (tpi_eval Te)
                    (tpK_lam TK [x][t:tpe x (T' cross T'')] tpi_snd* t)
                    Ta.
pr-ev_lam      : pr (ev_lam E) (tpi_eval (tpe_lam Te)) TK Ta
                  o- pr E (tpi_return (tpe_lam Te)) TK Ta.
pr-ev_app      : pr (ev_app E) (tpi_eval (tpe_app Te'' Te')) TK Ta
                  o- pr E
                    (tpi_eval Te')
                    (tpK_lam TK [x][t:tpe x (T' arrow T'')]tpi_app* Te'' t)
                    Ta.
pr-ev_letval   : pr (ev_letval E) (tpi_eval (tpe_letval Te'' Te')) TK Ta
                  o- pr E
                    (tpi_eval Te')
                    (tpK_lam TK [x][t:tpe x T'] tpi_eval (Te'' x t))
                    Ta.
pr-ev_letname  : pr (ev_letname E) (tpi_eval (tpe_letname Te)) TK Ta
                  o- pr E (tpi_eval Te) TK Ta.
pr-ev_fix      : {Te:{x} tpe x T -> tpe (X x) T}
                 pr (ev_fix E) (tpi_eval (tpe_fix Te)) TK Ta
                  o- pr E (tpi_eval (Te (fix X) (tpe_fix Te))) TK Ta.

pr-ev_cell     : pr (ev_cell E) (tpi_eval (tpe_cell Tc)) TK Ta
                  o- pr E (tpi_return (tpe_cell Tc)) TK Ta.
pr-ev_ref      : pr (ev_ref E) (tpi_eval (tpe_ref Te)) TK Ta
                  o- pr E
                    (tpi_eval Te)
                    (tpK_lam TK [x][t:tpe x T] tpi_ref* t)
                    Ta.
pr-ev_deref    : pr (ev_deref E) (tpi_eval (tpe_deref Te)) TK Ta
                  o- pr E
                    (tpi_eval Te)
                    (tpK_lam TK [x][t:tpe x (T tref)] tpi_deref* t)
                    Ta.
pr-ev_noop     : pr (ev_noop E) (tpi_eval tpe_noop) TK Ta
                  o- pr E (tpi_return tpe_noop) TK Ta.
pr-ev_seq      : pr (ev_seq E) (tpi_eval (tpe_seq Te'' Te')) TK Ta
                  o- pr E
```

```
                      (tpi_eval Te'')
                      (tpK_lam TK [x][t:tpe x T'] tpi_eval Te')
                      Ta.
pr-ev_assign   : pr (ev_assign E) (tpi_eval (tpe_assign Te'' Te')) TK Ta
                  o- pr E
                      (tpi_eval Te')
                      (tpK_lam TK [x][t:tpe x (T tref)] tpi_assign*1 Te'' t)
                      Ta.


% Values

pr-ev_init     : pr (ev_init C) (tpi_return Tv) tpK_init (tpa_close Tv TS)
                  o- prCollect C TS.
pr-ev_cont     : {Tv: tpe V T}
                 pr (ev_cont E) (tpi_return Tv) (tpK_lam TK Ti) Ta
                  o- pr E (Ti V Tv) TK Ta.


% Auxiliary instructions

pr-ev_case*1   : pr (ev_case*1 E) (tpi_case* Te'' Te' tpe_z) TK Ta
                  o- pr E (tpi_eval Te') TK Ta.
pr-ev_case*2   : {Tv: tpe V nat}
                 pr (ev_case*2 E) (tpi_case* Te'' Te' (tpe_s Tv)) TK Ta
                  o- pr E (tpi_eval (Te'' V Tv)) TK Ta.
pr-ev_pair*    : pr (ev_pair* E) (tpi_pair* Te Tv) TK Ta
                  o- pr E
                      (tpi_eval Te)
                      (tpK_lam TK [x][t:tpe x T'']tpi_return (tpe_pair t Tv))
                      Ta.
pr-ev_fst*     : pr (ev_fst* E) (tpi_fst* (tpe_pair Tv'' Tv')) TK Ta
                  o- pr E (tpi_return Tv') TK Ta.
pr-ev_snd*     : pr (ev_snd* E) (tpi_snd* (tpe_pair Tv'' Tv')) TK Ta
                  o- pr E (tpi_return Tv'') TK Ta.
pr-ev_app*     : pr (ev_app* E) (tpi_app* Te'' (tpe_lam Te')) TK Ta
                  o- pr E
                      (tpi_eval Te'')
                      (tpK_lam TK [x][t:tpe x T''] tpi_eval (Te' x t))
                      Ta.

pr-ev_ref*     : pr (ev_ref* E) (tpi_ref* Tv) TK (tpa_new Ta)
                  o- ({c:cell}{Cn:contains c V}{Tc:tpc c T}
                     prCell Cn Tc Tv
                     -o pr (E c Cn) (tpi_return (tpe_cell Tc)) TK (Ta c Tc)).
pr-ev_deref*   : pr (ev_deref* (R , E)) (tpi_deref* (tpe_cell Tc)) Tk Ta
                  o- prRead R Tc Tv
```

```
                  &  pr E (tpi_return Tv) TK Ta.
pr-ev_assign*1 : pr (ev_assign*1 E) (tpi_assign*1 Te Tv) Tk Ta
                  o- pr E
                        (tpi_eval Te)
                        (tpK_lam TK [x][t:tpe x T] tpi_assign*2 t Tv)
                        Ta.
pr-ev_assign*2 : pr (ev_assign*2 E Cn') (tpi_assign*2 Tv (tpe_cell Tc)) Tk Ta
                  o- prCell Cn' Tc Tv'
                  o- (prCell Cn Tc Tv
                     -o pr (E Cn) (tpi_return (tpe_noop)) Tk Ta).
```

C.2 Adequacy Theorems

In this section, we have collected all the adequacy theorems for the meta-representation of *MLR*. We do not show the proofs. Here, Σ is the signature displayed in Section C.1.

C.2.1 Syntax

Adequacy theorem C.1 (*MLR expressions*)

There is a compositional bijection $\ulcorner\_\urcorner$ *between MLR expression with free variables among* $x_1, \ldots, x_n$ *and cells* $c_1, \ldots, c_m$*, and canonical LLF objects* M *such that the judgment*

$$x_1 : \mathtt{exp}, \ldots, x_n : \mathtt{exp}, c_1 : \mathtt{cell}, \ldots, c_m : \mathtt{cell} \vdash_\Sigma M \Uparrow \mathtt{exp}$$

is derivable. □

Adequacy theorem C.2 (*MLR types*)

There is a bijection $\ulcorner\_\urcorner$ *between MLR types and canonical LLF objects* M *such that*

$$\cdot \vdash_\Sigma M \Uparrow \mathtt{tp}$$

is derivable □

Adequacy theorem C.3 (*MLR instructions*)

There is a compositional bijection $\ulcorner\_\urcorner$ *between MLR instructions with free variables among* $x_1, \ldots, x_n$ *and cells* $c_1, \ldots, c_m$*, and canonical LLF objects* M *such that the judgment*

$$x_1 : \mathtt{exp}, \ldots, x_n : \mathtt{exp}, c_1 : \mathtt{cell}, \ldots, c_m : \mathtt{cell} \vdash_\Sigma M \Uparrow \mathtt{instr}$$

is derivable. □

Adequacy theorem C.4 (*MLR continuations*)

There is a compositional bijection $\ulcorner\_\urcorner$ *between MLR continuations with free variables among* $x_1, \ldots, x_n$ *and cells* $c_1, \ldots, c_m$*, and canonical LLF objects* M *such that the judgment*

$$x_1 : \mathtt{exp}, \ldots, x_n : \mathtt{exp}, c_1 : \mathtt{cell}, \ldots, c_m : \mathtt{cell} \vdash_\Sigma M \Uparrow \mathtt{cont}$$

is derivable. □

Adequacy theorem C.5 (*MLR stores*)

There is a compositional bijection $\ulcorner \_ \urcorner$ *between MLR closed stores mentioning cells* $c_1, \ldots, c_m$*, and canonical LLF objects* M *such that the judgment*

$$c_1 : \mathtt{cell}, \ldots, c_m : \mathtt{cell} \vdash_\Sigma M \Uparrow \mathtt{store}$$

is derivable. □

Adequacy theorem C.6 (*MLR Answers*)

There is a compositional bijection $\ulcorner \_ \urcorner$ *between MLR closed answers mentioning free cells* $c_1, \ldots, c_m$*, and canonical LLF objects* M *such that the judgment*

$$c_1 : \mathtt{cell}, \ldots, c_m : \mathtt{cell} \vdash_\Sigma M \Uparrow \mathtt{answer}$$

is derivable. □

C.2.2 Typing

Adequacy theorem C.7 (*MLR expression typing*)

Given an MLR expression e *and a type* τ*, there is a compositional bijection between derivations of*

$$\underbrace{c_1 : \tau_1, \ldots, c_p : \tau_p}_{\Omega};\ \underbrace{x_1 : \tau'_1, \ldots, x_n : \tau'_n}_{\Gamma} \vdash^e e : \tau$$

and LLF objects M *such that*

$$\begin{bmatrix} c_1 : \mathtt{cell}, \ \ldots, \ c_p : \mathtt{cell}, \ t_1 : \mathtt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, \ \ldots, \ t_p : \mathtt{tpc}\ c_p\ \ulcorner \tau_p \urcorner, \\ x_1 : \mathtt{exp}, \ \ldots, \ x_n : \mathtt{exp}, \ t'_1 : \mathtt{tpe}\ x_1\ \ulcorner \tau'_1 \urcorner, \ \ldots, \ t'_n : \mathtt{tpe}\ x_n\ \ulcorner \tau'_n \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \mathtt{tpe}\ \ulcorner e \urcorner\ \ulcorner \tau \urcorner$$

is derivable. □

Adequacy theorem C.8 (*MLR instruction typing*)

Given an MLR instruction i *and a type* τ*, there is a compositional bijection between derivations of*

$$\underbrace{c_1 : \tau_1, \ldots, c_p : \tau_p}_{\Omega};\ \underbrace{x_1 : \tau'_1, \ldots, x_n : \tau'_n}_{\Gamma} \vdash^i i : \tau$$

and LLF objects M *such that*

$$\begin{bmatrix} c_1 : \mathtt{cell}, \ \ldots, \ c_p : \mathtt{cell}, \ t_1 : \mathtt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, \ \ldots, \ t_p : \mathtt{tpc}\ c_p\ \ulcorner \tau_p \urcorner, \\ x_1 : \mathtt{exp}, \ \ldots, \ x_n : \mathtt{exp}, \ t'_1 : \mathtt{tpe}\ x_1\ \ulcorner \tau'_1 \urcorner, \ \ldots, \ t'_n : \mathtt{tpe}\ x_n\ \ulcorner \tau'_n \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \mathtt{tpi}\ \ulcorner i \urcorner\ \ulcorner \tau \urcorner$$

is derivable. □

Adequacy theorem C.9 (*MLR continuation typing*)

Given an MLR continuation K and types τ and τ', there is a compositional bijection between derivations of

$$\underbrace{c_1 : \tau_1, \ldots, c_p : \tau_p}_{\Omega};\ \underbrace{x_1 : \tau'_1, \ldots, x_n : \tau'_n}_{\Gamma} \vdash^K K : \tau \Rightarrow \tau'$$

and LLF objects M such that

$$\begin{bmatrix} c_1 : \mathtt{cell}, \ \ldots, \ c_p : \mathtt{cell}, \ t_1 : \mathtt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, \ \ldots, \ t_p : \mathtt{tpc}\ c_p\ \ulcorner \tau_p \urcorner, \\ x_1 : \mathtt{exp}, \ \ldots, \ x_n : \mathtt{exp}, \ t'_1 : \mathtt{tpe}\ x_1\ \ulcorner \tau'_1 \urcorner, \ \ldots, \ t'_n : \mathtt{tpe}\ x_n\ \ulcorner \tau'_n \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \mathtt{tpK}\ \ulcorner K \urcorner\ \ulcorner \tau \urcorner\ \ulcorner \tau' \urcorner$$

is derivable. □

Adequacy theorem C.10 (*MLR store typing*)

Given an MLR store S, there is a compositional bijection between typing derivations of

$$\underbrace{c_1 : \tau_1, \ldots, c_p : \tau_p}_{\Omega} \vdash^S \underbrace{c'_1 = v_1, \ldots, c'_n = v_n}_{S} : \underbrace{c'_1 : \tau'_1, \ldots, c'_n : \tau'_n}_{\Omega'}$$

and LLF objects M such that

$$\begin{bmatrix} c_1 : \mathtt{cell}, \ \ldots, \ c_p : \mathtt{cell}, \ t_1 : \mathtt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, \ \ldots, \ t_p : \mathtt{tpc}\ c_p\ \ulcorner \tau_p \urcorner, \\ c'_1 : \mathtt{cell}, \ \ldots, \ c'_n : \mathtt{cell}, \ t'_1 : \mathtt{tpc}\ c'_1\ \ulcorner \tau'_1 \urcorner, \ \ldots, \ t'_n : \mathtt{tpc}\ c'_n\ \ulcorner \tau'_n \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \mathtt{tpS}\ \ulcorner S \urcorner$$

is derivable. If some of the c_i's coincides with some of the c'_i's, we include only the declaration for one of these occurrences in the LLF signature above. □

Adequacy theorem C.11 (*MLR answer typing*)

Given an MLR answer A, there is a compositional bijection between derivations of

$$\underbrace{c_1 : \tau_1, \ldots, c_p : \tau_p}_{\Omega} \vdash^a a : \tau$$

and LLF objects M such that

$$\begin{bmatrix} c_1 : \mathtt{cell}, & \ldots, \ c_p : \mathtt{cell}, \\ t_1 : \mathtt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, \ \ldots, & t_p : \mathtt{tpc}\ c_p\ \ulcorner \tau_p \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \mathtt{tpA}\ \ulcorner a \urcorner\ \ulcorner \tau \urcorner$$

is derivable. □

C.2.3 Evaluation

Adequacy theorem C.12 (*MLR store items*)

Given a closed MLR store $S = (c_1 = v_1, \ldots, c_n = v_n)$, there is a compositional bijection between pairs $c_i = v_i$ in S and LLF objects M such that

$$\begin{bmatrix} c_1 : \mathtt{cell}, & \ldots, \ c_n : \mathtt{cell}, \\ h_1 \,\hat{:}\, \mathtt{contains}\ c_1\ \ulcorner v_1 \urcorner, \ \ldots, & h_n \,\hat{:}\, \mathtt{contains}\ c_n\ \ulcorner v_n \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \mathtt{read}\ \ulcorner c_i \urcorner\ \ulcorner v_i \urcorner$$

is derivable. □

Adequacy theorem C.13 (*MLR store contents*)

There is a compositional bijection between MLR stores $S = (c_1 = v_1, \ldots, c_n = v_n)$ *and LLF objects* M *such that*

$$\begin{bmatrix} c_1 : \texttt{cell}, & \ldots, & c_n : \texttt{cell}, \\ h_1 \hat{:} \texttt{contains}\ c_1\ \ulcorner v_1 \urcorner, & \ldots, & h_n \hat{:} \texttt{contains}\ c_n\ \ulcorner v_n \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \texttt{collect}\ \ulcorner S \urcorner$$

is derivable. □

Adequacy theorem C.14 (*MLR evaluation*)

Given a closed MLR continuation K*, a closed instruction* i*, a store* $S = (c_1 = v_1, \ldots, c_n = v_n)$ *and an answer* a*, there is a compositional bijection between derivations of*

$$K \vdash i \hookrightarrow_S a$$

and LLF objects M *such that*

$$\begin{bmatrix} c_1 : \texttt{cell}, & \ldots, & c_n : \texttt{cell}, \\ h_1 \hat{:} \texttt{contains}\ c_1\ \ulcorner v_1 \urcorner, & \ldots, & h_n \hat{:} \texttt{contains}\ c_n\ \ulcorner v_n \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \texttt{ev}\ \ulcorner K \urcorner\ \ulcorner i \urcorner\ \ulcorner a \urcorner$$

is derivable. □

C.2.4 Type Preservation

Adequacy theorem C.15 (*MLR type preservation for context items*)

Given a closed MLR store $S = (c_1 = v_1, \ldots, c_n = v_n)$*, a store context* $\Omega = (c_1 : \tau_1, \ldots, c_n : \tau_n)$ *and an* LLF *objects* R_i *and* T_i *such that the judgments*

$$\begin{bmatrix} c_1 : \texttt{cell}, & \ldots, & c_n : \texttt{cell}, \\ h_1 \hat{:} \texttt{contains}\ c_1\ \ulcorner v_1 \urcorner, & \ldots, & h_n \hat{:} \texttt{contains}\ c_n\ \ulcorner v_n \urcorner \end{bmatrix} \vdash_\Sigma R_i \Uparrow \texttt{read}\ \ulcorner c_i \urcorner\ \ulcorner v_i \urcorner$$

$$\begin{bmatrix} c_1 : \texttt{cell}, & \ldots, & c_n : \texttt{cell}, \\ t_1 : \texttt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, & \ldots, & t_n : \texttt{tpc}\ c_n\ \ulcorner \tau_n \urcorner \end{bmatrix} \vdash_\Sigma T_i \Uparrow \texttt{tpe}\ \ulcorner v_i \urcorner\ \ulcorner \tau_i \urcorner$$

are derivable, there exists an LLF objects M *such that*

$$\begin{bmatrix} c_1 : \texttt{cell}, & \ldots, & c_n : \texttt{cell}, \\ h_1 : \texttt{contains}\ c_1\ \ulcorner v_1 \urcorner, & \ldots, & h_n : \texttt{contains}\ c_n\ \ulcorner v_n \urcorner, \\ t_1 : \texttt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, & \ldots, & t_n : \texttt{tpc}\ c_n\ \ulcorner \tau_n \urcorner, \\ p_1 \hat{:} \texttt{prCell}\ h_1\ t_1, & \ldots, & p_n \hat{:} \texttt{prCell}\ h_n\ t_n \end{bmatrix}$$

$$\vdash_\Sigma M \Uparrow \texttt{prRead}\ c_i\ \ulcorner v_i \urcorner\ \ulcorner \tau_i \urcorner\ R_i\ t_i\ T_i$$

is derivable. □

Adequacy theorem C.16 (*MLR type preservation for contexts*)

Given a closed MLR store $S = (c_1 = v_1, \ldots, c_n = v_n)$*, a closed store context* $\Omega = (c_1 : \tau_1, \ldots, c_n : \tau_n)$*, and* LLF *objects* T *and* C *such that the judgments*

$$\begin{bmatrix} c_1 : \mathtt{cell}, & \ldots, & c_n : \mathtt{cell}, \\ h_1 \,\hat{:}\, \mathtt{contains}\ c_1\ \ulcorner v_1 \urcorner, & \ldots, & h_n \,\hat{:}\, \mathtt{contains}\ c_n\ \ulcorner v_n \urcorner \end{bmatrix} \vdash_\Sigma C \Uparrow \mathtt{collect}\ \ulcorner S \urcorner$$

$$\begin{bmatrix} c_1 : \mathtt{cell}, & \ldots, & c_p : \mathtt{cell}, \\ t_1 : \mathtt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, & \ldots, & t_p : \mathtt{tpc}\ c_p\ \ulcorner \tau_p \urcorner \end{bmatrix} \vdash_\Sigma T \Uparrow \mathtt{tpS}\ \ulcorner S \urcorner$$

are derivable, there exists an LLF objects M *such that*

$$\begin{bmatrix} c_1 : \mathtt{cell}, & \ldots, & c_n : \mathtt{cell}, \\ h_1 : \mathtt{contains}\ c_1\ \ulcorner v_1 \urcorner, & \ldots, & h_n : \mathtt{contains}\ c_n\ \ulcorner v_n \urcorner, \\ t_1 : \mathtt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, & \ldots, & t_n : \mathtt{tpc}\ c_n\ \ulcorner \tau_n \urcorner, \\ p_1 \,\hat{:}\, \mathtt{prCell}\ h_1\ t_1, & \ldots, & p_n \,\hat{:}\, \mathtt{prCell}\ h_n\ t_n \end{bmatrix} \vdash_\Sigma M \Uparrow \mathtt{prCollect}\ \ulcorner S \urcorner\ C\ T$$

is derivable. □

Adequacy theorem C.17 (*MLR type preservation*)

Given a closed MLR continuation K*, a closed instruction* i*, a closed answer* a*, a closed store* $S = (c_1 = v_1, \ldots, c_n = v_n)$*, a store context* $\Omega = (c_1 : \tau_1, \ldots, c_n : \tau_n)$*, types* τ *and* τ'*, and* LLF *objects* E*,* T_i*,* T_K *and* T_S *such that the judgments*

$$\begin{bmatrix} c_1 : \mathtt{cell}, & \ldots, & c_n : \mathtt{cell}, \\ h_1 \,\hat{:}\, \mathtt{contains}\ c_1\ \ulcorner v_1 \urcorner, & \ldots, & h_n \,\hat{:}\, \mathtt{contains}\ c_n\ \ulcorner v_n \urcorner \end{bmatrix} \vdash_\Sigma E \Uparrow \mathtt{ev}\ \ulcorner K \urcorner\ \ulcorner i \urcorner\ \ulcorner a \urcorner$$

$$\begin{bmatrix} c_1 : \mathtt{cell}, & \ldots, & c_p : \mathtt{cell}, \\ t_1 : \mathtt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, & \ldots, & t_p : \mathtt{tpc}\ c_p\ \ulcorner \tau_p \urcorner \end{bmatrix} \vdash_\Sigma T_i \Uparrow \mathtt{tpi}\ \ulcorner i \urcorner\ \ulcorner \tau \urcorner$$

$$\begin{bmatrix} c_1 : \mathtt{cell}, & \ldots, & c_p : \mathtt{cell}, \\ t_1 : \mathtt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, & \ldots, & t_p : \mathtt{tpc}\ c_p\ \ulcorner \tau_p \urcorner \end{bmatrix} \vdash_\Sigma T_K \Uparrow \mathtt{tpK}\ \ulcorner i \urcorner\ \ulcorner \tau \urcorner\ \ulcorner \tau' \urcorner$$

$$\begin{bmatrix} c_1 : \mathtt{cell}, & \ldots, & c_p : \mathtt{cell}, \\ t_1 : \mathtt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, & \ldots, & t_p : \mathtt{tpc}\ c_p\ \ulcorner \tau_p \urcorner \end{bmatrix} \vdash_\Sigma T_S \Uparrow \mathtt{tpS}\ \ulcorner S \urcorner$$

are derivable, there exists LLF objects T_a *and* M *such that*

$$\begin{bmatrix} c_1 : \mathtt{cell}, & \ldots, & c_p : \mathtt{cell}, \\ t_1 : \mathtt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, & \ldots, & t_p : \mathtt{tpc}\ c_p\ \ulcorner \tau_p \urcorner \end{bmatrix} \vdash_\Sigma T_a \Uparrow \mathtt{tpA}\ \ulcorner a \urcorner\ \ulcorner \tau' \urcorner$$

and

$$\begin{bmatrix} c_1 : \mathtt{cell}, & \ldots, & c_n : \mathtt{cell}, \\ h_1 : \mathtt{contains}\ c_1\ \ulcorner v_1 \urcorner, & \ldots, & h_n : \mathtt{contains}\ c_n\ \ulcorner v_n \urcorner, \\ t_1 : \mathtt{tpc}\ c_1\ \ulcorner \tau_1 \urcorner, & \ldots, & t_n : \mathtt{tpc}\ c_n\ \ulcorner \tau_n \urcorner, \\ p_1 \,\hat{:}\, \mathtt{prCell}\ h_1\ t_1, & \ldots, & p_n \,\hat{:}\, \mathtt{prCell}\ h_n\ t_n \end{bmatrix}$$

$$\vdash_\Sigma M \Uparrow \mathtt{pr}\ \ulcorner K \urcorner\ \ulcorner i \urcorner\ \ulcorner \tau \urcorner\ \ulcorner \tau' \urcorner\ \ulcorner a \urcorner\ E\ T_i\ T_K\ T_a$$

are derivable. □

Appendice D

Cut Elimination for Linear Hereditary Harrop Formulas

In this appendix, we have collected the *Elf* code and the adequacy theorems for the meta-representation of the cut-elimination theorem for the logic of linear hereditary Harrop formulas presented in Chapter 6.

D.1 *LLF* Source Code

In this Section, we provide the complete *LLF* code implementing the cut elimination theorem for the freely generated fragment of the language of linear hereditary Harrop formulas. We described the representation in Section 6.4. The language of formulas is given in Subsection D.1.1. The rules describing the provability relation for linear hereditary Harrop formulas is implemented in D.1.2. The admissibility lemma for the two forms of cut of this language is represented in D.1.3. Finally, Subsection D.1.4 contains the *LLF* code for the cut elimination procedure itself.

D.1.1 Formulas

```
o : type.     %name o A
i : type.     %name i T

top   : o.
with  : o -> o -> o.        %infix right 100 with
lolli : o -> o -> o.        %infix right 110 lolli
imp   : o -> o -> o.        %infix right 110 imp
all   : (i -> o) -> o.
```

D.1.2 Provability

```
% assumptions

int : o -> type.  %name int I
```

```
lin : o -> type.  %name lin L


% cut-free proofs

pr  : o -> type.  %name pr  D

id      : (lin A -o pr A).
clone   : (int A -> lin A -o pr C)              -o (int A -> pr C).

top_r   : <T>                                   -o pr top.
with_r  : pr A                 & pr B           -o pr (A with B).
lolli_r : (lin A -o pr B)                       -o pr (A lolli B).
imp_r   : (int A -> pr B)                       -o pr (A imp B).
all_r   : ({c:i} pr (A c))                      -o pr (all A).

with_l1 : (lin A -o pr C)                       -o (lin (A with B) -o pr C).
with_l2 : (lin B -o pr C)                       -o (lin (A with B) -o pr C).
lolli_l : pr A               -o (lin B -o pr C) -o (lin (A lolli B) -o pr C).
imp_l   : pr A               -> (lin B -o pr C) -o (lin (A imp B) -o pr C).
all_l   : {t:i}(lin (A t) -o pr C)              -o (lin (all A) -o pr C).


% proofs with cut

pr+ : o -> type.  %name pr+ C

id+      : (lin A -o pr+ A).
clone+   : (int A -> lin A -o pr+ C)            -o (int A -> pr+ C).

top_r+   : <T>                                  -o pr+ top.
with_r+  : pr+ A              & pr+ B           -o pr+ (A with B).
lolli_r+ : (lin A -o pr+ B)                     -o pr+ (A lolli B).
imp_r+   : (int A -> pr+ B)                     -o pr+ (A imp B).
all_r+   : ({c:i} pr+ (A c))                    -o pr+ (all A).

with_l1+ : (lin A -o pr+ C)                     -o (lin (A with B) -o pr+ C).
with_l2+ : (lin B -o pr+ C)                     -o (lin (A with B) -o pr+ C).
lolli_l+ : pr+ A             -o (lin B -o pr+ C) -o (lin (A lolli B) -o pr+ C).
imp_l+   : pr+ A             -> (lin B -o pr+ C) -o (lin (A imp B) -o pr+ C).
all_l+   : {t:i}(lin (A t) -o pr+ C)            -o (lin (all A) -o pr+ C).

cut      : pr+ A             -o (lin A -o pr+ C) -o pr+ C.
cut!     : pr+ A             -> (int A -> pr+ C) -o pr+ C.
```

D.1.3 Admissibility of `cut` and `cut!`

```
adm     : pr A -> (lin A -> pr C) -> pr C -> type.
adm!    : pr A -> (int A -> pr C) -> pr C -> type.

%mode +[A!] +[C!] +[D1!:pr A!] +[D2!:int A! -> pr C!] -[D!: pr C!] -[_:adm! D1! D2! D!]
%mode +[A]  +[C]  +[D1:pr A]   +[D2:lin A -> pr C]    -[D: pr C]   -[_:adm D1 D2 D]
%lex {A! A} {} {D1! D1} {D2! D2}



%%% Admissibility of cut

% Last rule of D1 operates on the left-hand side, so not on A

adm_id-*      : adm (id X) D2 (D2 X).
adm_clone-*   : adm (clone D11 Y) D2 (clone D' Y)
                  <- ({y1:int B}{y2:lin B} adm (D11 y1 y2) D2 (D' y1 y2)).
adm_with_l1-* : adm (with_l1 D11 Y) D2 (with_l1 D' Y)
                  <- ({y:lin B1} adm (D11 y) D2 (D' y)).
adm_with_l2-* : adm (with_l2 D11 Y) D2 (with_l2 D' Y)
                  <- ({y:lin B2} adm (D11 y) D2 (D' y)).
adm_lolli_l-* : adm (lolli_l D11 D12 Y) D2 (lolli_l D11 D' Y)
                  <- ({y:lin B2} adm (D12 y) D2 (D' y)).
adm_imp_l-*   : adm (imp_l D11 D12 Y) D2 (imp_l D11 D' Y)
                  <- ({y:lin B2} adm (D12 y) D2 (D' y)).
adm_all_l-*   : {Y: lin (all B)}
                  adm (all_l T D11 Y) D2 (all_l T D' Y)
                  <- ({y:lin (B T)} adm (D11 y) D2 (D' y)).


% Last rule of D2 operates on the right-hand side, so not on A

adm_*-id      : adm D1 id D1.
adm_*-top_r   : adm D1 ([x:lin A] top_r ()) (top_r ()).
adm_*-with_r  : adm D1 ([x:lin A] with_r (D21 x , D22 x)) (with_r (D' , D''))
                  <- adm D1 D21 D'
                  <- adm D1 D22 D''.
adm_*-lolli_r : adm D1 ([x:lin A] lolli_r (D21 x)) (lolli_r D')
                  <- ({y:lin C1} adm D1 ([x:lin A] D21 x y) (D' y)).
adm_*-imp_r   : adm D1 ([x:lin A] imp_r (D21 x)) (imp_r D')
                  <- ({y:int C1} adm D1 ([x:lin A] D21 x y) (D' y)).
adm_*-all_r   : adm D1 ([x:lin A] all_r (D21 x)) (all_r D')
                  <- ({c:i} adm D1 ([x:lin A] D21 x c) (D' c)).


% Last rule of D2 operates on the left-hand side, but not on A.
```

```
adm_*-clone   : adm D1 ([x:lin A] clone (D21 x) Y) (clone D' Y)
                 <- ({y1:int B} {y2:lin B}
                    adm D1 ([x:lin A] D21 x y1 y2) (D' y1 y2)).
adm_*_with_l1 : adm D1 ([x:lin A] with_l1 (D21 x) Y) (with_l1 D' Y)
                 <- ({y:lin B1} adm D1 ([x:lin A] D21 x y) (D' y)).
adm_*_with_l2 : adm D1 ([x:lin A] with_l2 (D21 x) Y) (with_l2 D' Y)
                 <- ({y:lin B1} adm D1 ([x:lin A] D21 x y) (D' y)).
adm_*-lolli_l1: adm D1 ([x:lin A] lolli_l (D21 x) D22 Y) (lolli_l D' D22 Y)
                 <- adm D1 D21 D'.
adm_*-lolli_l2: adm D1 ([x:lin A] lolli_l D21 (D22 x) Y) (lolli_l D21 D' Y)
                 <- ({y: lin B2} adm D1 ([x:lin A] D22 x y) (D' y)).
% no adm_*-imp_l1
adm_*-imp_l2  : adm D1 ([x:lin A] imp_l D21 (D22 x) Y) (imp_l D21 D' Y)
                 <- ({y: lin B2} adm D1 ([x:lin A] D22 x y) (D' y)).
adm_*-all_l   : {Y: lin (all B)}
                adm D1 ([x:lin A] all_l T (D21 x) Y) (all_l T D' Y)
                 <- ({y: lin (B T)} adm D1 ([x:lin A] D21 x y) (D' y)).


% Essential conversions: last rule of both D1 and D2 operates on A

% no rule for top
adm_with_r+with_l1  : adm (with_r (D11 , D12)) (with_l1 D21) D
                       <- adm D11 D21 D.
adm_with_r+with_l2  : adm (with_r (D11 , D12)) (with_l2 D21) D
                       <- adm D12 D21 D.
adm_lolli_r+lolli_l : adm (lolli_r D11) (lolli_l D21 D22) D
                       <- ({x:lin A1} adm (D11 x) D22 (D' x))
                       <- adm D21 D' D.
adm_imp_r+imp_l     : adm (imp_r D11) (imp_l D21 D22) D
                       <- ({x:int A1} adm (D11 x) D22 (D' x))
                       <- adm! D21 D' D.
adm_all_r+all_l     : adm (all_r D11) (all_l T D21) D
                       <- adm (D11 T) D21 D.



%%% Admissibility of cut!

% We never need to consider D1

% Last rule of D2 operates on the right-hand side, so not on A

adm!_*-top_r  : adm! D1 ([x:int A] top_r ()) (top_r ()).
adm!_*-with_r : adm! D1 ([x:int A] with_r (D21 x , D22 x)) (with_r (D' , D''))
                <- adm! D1 D21 D'
```

```
                <- adm! D1 D22 D''.
adm!_*-id     : adm! D1 ([x:int A] id Y) (id Y).
adm!_*-lolli_r: adm! D1 ([x:int A] lolli_r (D21 x)) (lolli_r D')
                <- ({y:lin C1} adm! D1 ([x:int A] D21 x y) (D' y)).
adm!_*-imp_r  : adm! D1 ([x:int A] imp_r (D21 x)) (imp_r D')
                <- ({y:int C1} adm! D1 ([x:int A] D21 x y) (D' y)).
adm!_*-all_r  : adm! D1 ([x:int A] all_r (D21 x)) (all_r D')
                <- ({c:i} adm! D1 ([x:int A] D21 x c) (D' c)).


% Last rule of D2 operates on the left-hand side, but not on A.

adm!_*-clone  : adm! D1 ([x:int A] clone (D21 x) Y) (clone D' Y)
                <- ({y1:int B} {y2:lin B}
                   adm! D1 ([x:int A] D21 x y1 y2) (D' y1 y2)).
adm!_*_with_l1: adm! D1 ([x:int A] with_l1 (D21 x) Y) (with_l1 D' Y)
                <- ({y:lin B1} adm! D1 ([x:int A] D21 x y) (D' y)).
adm!_*_with_l2: adm! D1 ([x:int A] with_l2 (D21 x) Y) (with_l2 D' Y)
                <- ({y:lin B1} adm! D1 ([x:int A] D21 x y) (D' y)).
adm!_*-lolli_l: adm! D1 ([x:int A] lolli_l (D21 x) (D22 x) Y) (lolli_l D' D'' Y)
                <- adm! D1 D21 D'
                <- ({y: lin B2} adm! D1 ([x:int A] D22 x y) (D'' y)).
adm!_*-imp_l  : adm! D1 ([x:int A] imp_l (D21 x) (D22 x) Y) (imp_l D' D'' Y)
                <- adm! D1 D21 D'
                <- ({y: lin B2} adm! D1 ([x:int A] D22 x y) (D'' y)).
adm!_*-all_l  : {Y: lin (all B)}
               adm! D1 ([x:int A] all_l T (D21 x) Y) (all_l T D' Y)
                <- ({y: lin (B T)} adm! D1 ([x:int A] D21 x y) (D' y)).


% Essential conversions: last rule of D2 operates on A

adm!_*+clone  : adm! D1 (clone D21) D
                 <- ({x:lin A} adm! D1 ([y:int A] D21 y x) (D' x))
                 <- adm D1 D' D.
```

D.1.4 Cut Elimination

```
ce: pr+ A -> pr A -> type.

%mode -ce +C -D
%lex C


ce_id      : ce (id+ X) (id X).
ce_clone   : ce (clone+ C1 X) (clone D1 X)
              <- ({x: int A}{y: lin A} ce (C1 x y) (D1 x y)).
```

```
ce_top_r   : ce (top_r+ ()) (top_r ()).
ce_with_r  : ce (with_r+ (C1 , C2)) (with_r (D1 , D2))
              <- ce C1 D1
              <- ce C2 D2.
ce_lolli_r : ce (lolli_r+ C1) (lolli_r D1)
              <- ({x:lin A} ce (C1 x) (D1 x)).
ce_imp_r   : ce (imp_r+ C1) (imp_r D1)
              <- ({x:int A} ce (C1 x) (D1 x)).
ce_all_r   : ce (all_r+ C1) (all_r D1)
              <- ({c:i} ce (C1 c) (D1 c)).

ce_with_l1 : ce (with_l1+ C1 X) (with_l1 D1 X)
              <- ({x:lin A} ce (C1 x) (D1 x)).
ce_with_l2 : ce (with_l2+ C1 X) (with_l2 D1 X)
              <- ({x:lin B} ce (C1 x) (D1 x)).
ce_lolli_l : ce (lolli_l+ C11 C12 X) (lolli_l D11 D12 X)
              <- ce C11 D11
              <- ({x:lin B} ce (C12 x) (D12 x)).
ce_imp_l   : ce (imp_l+ C11 C12 X) (imp_l D11 D12 X)
              <- ce C11 D11
              <- ({x:lin B} ce (C12 x) (D12 x)).
ce_all_l   : {X: lin (all A)}
             ce (all_l+ T C11 X) (all_l T D11 X)
              <- ({x:lin (A T)} ce (C11 x) (D11 x)).
ce_cut     : ce (cut C11 C12) D
              <- ce C11 D11
              <- ({x:lin A} ce (C12 x) (D12 x))
              <- adm D11 D12 D.
ce_cut!    : ce (cut! C11 C12) D
              <- ce C11 D11
              <- ({x:int A} ce (C12 x) (D12 x))
              <- adm! D11 D12 D.
```

D.2 Adequacy Theorems

In this section, we have collected all the adequacy theorems for the meta-representation of cut-elimination for the logic of linear hereditary Harrop formulas. We do not show the proofs. Here, Σ is the signature displayed in Section D.1

D.2.1 Formulas

Adequacy theorem D.1 (*Formulas*)

There is a compositional bijection $\ulcorner \_ \urcorner$ *between formulas containing individual constants* $c_1, \ldots, c_n$ *and canonical LLF objects* M *such that the judgment*

$$c_1 : \mathtt{i}, \ldots, c_n : \mathtt{i} \vdash_\Sigma M \Uparrow \mathtt{o}$$

is derivable. □

D.2.2 Provability

Adequacy theorem D.2 (*Cut-free derivations*)

Given formulas $A_1, \ldots, A_i$, $B_1, \ldots, B_l$ *and* C *mentioning individual constants* $c_1, \ldots, c_n$, *there is a compositional bijection* $\ulcorner \_ \urcorner$ *between cut-free derivations of*

$$A_1, \ldots, A_i;\ B_1, \ldots, B_l \longrightarrow C$$

and canonical LLF objects M *such that the judgment*

$$\begin{bmatrix} c_1 : \mathtt{i}, & \ldots, & c_n : \mathtt{i}, \\ x_1 : \mathtt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \mathtt{int}\ \ulcorner A_i \urcorner, \\ y_1 \hat{:} \mathtt{lin}\ \ulcorner B_1 \urcorner, & \ldots, & y_l \hat{:} \mathtt{lin}\ \ulcorner B_l \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \mathtt{pr}\ \ulcorner C \urcorner$$

is derivable. □

Adequacy theorem D.3 (*Derivations*)

Given formulas $A_1, \ldots, A_i$, $B_1, \ldots, B_l$ *and* C *mentioning individual constants* $c_1, \ldots, c_n$, *there is a compositional bijection* $\ulcorner \_ \urcorner$ *between derivations, possibly using the cut rules, of*

$$A_1, \ldots, A_i;\ B_1, \ldots, B_l \longrightarrow C$$

and canonical LLF objects M *such that the judgment*

$$\begin{bmatrix} c_1 : \mathtt{i}, & \ldots, & c_n : \mathtt{i}, \\ x_1 : \mathtt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \mathtt{int}\ \ulcorner A_i \urcorner, \\ y_1 \hat{:} \mathtt{lin}\ \ulcorner B_1 \urcorner, & \ldots, & y_l \hat{:} \mathtt{lin}\ \ulcorner B_l \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \mathtt{pr+}\ \ulcorner C \urcorner$$

is derivable. □

D.2.3 Admissibility of `cut` and `cut!`

Adequacy theorem D.4 (*Admissibility of* `cut` and `cut!`)

i. Let A and C be formulas, and $\Gamma = A_1, \ldots, A_i$, $\Delta' = B'_1, \ldots, B'_{l'}$ and $\Delta'' = B''_1, \ldots, B''_{l''}$ be contexts, mentioning individual constants $c_1, \ldots, c_n$. Given LLF terms D' and D'' such that the judgments

$$\begin{bmatrix} c_1 : \mathtt{i}, & \ldots, & c_n : \mathtt{i}, \\ x_1 : \mathtt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \mathtt{int}\ \ulcorner A_i \urcorner, \\ y'_1 \,\hat{:}\, \mathtt{lin}\ \ulcorner B'_1 \urcorner, & \ldots, & y'_{l'} \,\hat{:}\, \mathtt{lin}\ \ulcorner B'_{l'} \urcorner \end{bmatrix} \vdash_\Sigma D' \Uparrow \mathtt{pr}\ \ulcorner A \urcorner$$

$$\begin{bmatrix} c_1 : \mathtt{i}, & \ldots, & c_n : \mathtt{i}, \\ x_1 : \mathtt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \mathtt{int}\ \ulcorner A_i \urcorner, \\ y''_1 \,\hat{:}\, \mathtt{lin}\ \ulcorner B''_1 \urcorner, & \ldots, & y''_{l''} \,\hat{:}\, \mathtt{lin}\ \ulcorner B''_{l''} \urcorner, \\ z \,\hat{:}\, \mathtt{lin}\ \ulcorner A \urcorner & & \end{bmatrix} \vdash_\Sigma D'' \Uparrow \mathtt{pr}\ \ulcorner C \urcorner$$

are derivable, there exists LLF objects D' and M such that

$$\begin{bmatrix} c_1 : \mathtt{i}, & \ldots, & c_n : \mathtt{i}, \\ x_1 : \mathtt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \mathtt{int}\ \ulcorner A_i \urcorner, \\ y'_1 \,\hat{:}\, \mathtt{lin}\ \ulcorner B'_1 \urcorner, & \ldots, & y'_{l'} \,\hat{:}\, \mathtt{lin}\ \ulcorner B'_{l'} \urcorner, \\ y''_1 \,\hat{:}\, \mathtt{lin}\ \ulcorner B''_1 \urcorner, & \ldots, & y''_{l''} \,\hat{:}\, \mathtt{lin}\ \ulcorner B''_{l''} \urcorner \end{bmatrix} \vdash_\Sigma D \Uparrow \mathtt{pr}\ \ulcorner C \urcorner$$

and

$$\begin{bmatrix} c_1 : \mathtt{i}, & \ldots, & c_n : \mathtt{i}, \\ x_1 : \mathtt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \mathtt{int}\ \ulcorner A_i \urcorner, \\ y'_1 : \mathtt{lin}\ \ulcorner B'_1 \urcorner, & \ldots, & y'_{l'} : \mathtt{lin}\ \ulcorner B'_{l'} \urcorner, \\ y''_1 : \mathtt{lin}\ \ulcorner B''_1 \urcorner, & \ldots, & y''_{l''} : \mathtt{lin}\ \ulcorner B''_{l''} \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \mathtt{adm}\ \ulcorner A \urcorner\ \ulcorner C \urcorner\ D'\ [x{:}\mathtt{lin}\ \ulcorner A \urcorner] D''\ D$$

are derivable;

ii. Let A and C be formulas, and $\Gamma = A_1, \ldots, A_i$ and $\Delta = B_1, \ldots, B_l$ be contexts, mentioning individual constants $c_1, \ldots, c_n$. Given LLF terms D', D'' and D such that the judgments

$$\begin{bmatrix} c_1 : \mathtt{i}, & \ldots, & c_n : \mathtt{i}, \\ x_1 : \mathtt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \mathtt{int}\ \ulcorner A_i \urcorner \end{bmatrix} \vdash_\Sigma D' \Uparrow \mathtt{pr}\ \ulcorner A \urcorner$$

$$\begin{bmatrix} c_1 : \mathtt{i}, & \ldots, & c_n : \mathtt{i}, \\ x_1 : \mathtt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \mathtt{int}\ \ulcorner A_i \urcorner, \\ z : \mathtt{int}\ \ulcorner A \urcorner, & & \\ y_1 \,\hat{:}\, \mathtt{lin}\ \ulcorner B_1 \urcorner, & \ldots, & y_l \,\hat{:}\, \mathtt{lin}\ \ulcorner B_l \urcorner \end{bmatrix} \vdash_\Sigma D'' \Uparrow \mathtt{pr}\ \ulcorner C \urcorner$$

are derivable, there exists LLF objects D and M such that

$$\begin{bmatrix} c_1 : \mathtt{i}, & \ldots, & c_n : \mathtt{i}, \\ x_1 : \mathtt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \mathtt{int}\ \ulcorner A_i \urcorner, \\ y_1 \,\hat{:}\, \mathtt{lin}\ \ulcorner B_1 \urcorner, & \ldots, & y_l \,\hat{:}\, \mathtt{lin}\ \ulcorner B_l \urcorner \end{bmatrix} \vdash_\Sigma D \Uparrow \mathtt{pr}\ \ulcorner C \urcorner$$

and

$$\begin{bmatrix} c_1 : \mathtt{i}, & \ldots, & c_n : \mathtt{i}, \\ x_1 : \mathtt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \mathtt{int}\ \ulcorner A_i \urcorner, \\ y_1 : \mathtt{lin}\ \ulcorner B_1 \urcorner, & \ldots, & y_l : \mathtt{lin}\ \ulcorner B_l \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \mathtt{adm!}\ \ulcorner A \urcorner\ \ulcorner C \urcorner\ D'\ [x{:}\mathtt{int}\ \ulcorner A \urcorner] D''\ D$$

are derivable. □

D.2.4 Cut Elimination

Adequacy theorem D.5 (*Cut elimination*)

Let C be a formula, and $\Gamma = A_1, \ldots, A_i$, $\Delta = B_1, \ldots, B_l$, be contexts, mentioning individual constants $c_1, \ldots, c_n$. Then, given an LLF *term D such that the judgment*

$$\begin{bmatrix} c_1 : \mathtt{i}, & \ldots, & c_n : \mathtt{i}, \\ x_1 : \mathtt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \mathtt{int}\ \ulcorner A_i \urcorner, \\ y_1 \hat{:} \mathtt{lin}\ \ulcorner B_1 \urcorner, & \ldots, & y_l \hat{:} \mathtt{lin}\ \ulcorner B_l \urcorner \end{bmatrix} \vdash_\Sigma D \Uparrow \mathtt{pr}\ \ulcorner C \urcorner$$

is derivable, there exists LLF objects D' and M such that

$$\begin{bmatrix} c_1 : \mathtt{i}, & \ldots, & c_n : \mathtt{i}, \\ x_1 : \mathtt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \mathtt{int}\ \ulcorner A_i \urcorner, \\ y_1 \hat{:} \mathtt{lin}\ \ulcorner B_1 \urcorner, & \ldots, & y_l \hat{:} \mathtt{lin}\ \ulcorner B_l \urcorner \end{bmatrix} \vdash_\Sigma D' \Uparrow \mathtt{pr}{+}\ \ulcorner C \urcorner$$

and

$$\begin{bmatrix} c_1 : \mathtt{i}, & \ldots, & c_n : \mathtt{i}, \\ x_1 : \mathtt{int}\ \ulcorner A_1 \urcorner, & \ldots, & x_i : \mathtt{int}\ \ulcorner A_i \urcorner, \\ y_1 : \mathtt{lin}\ \ulcorner B_1 \urcorner, & \ldots, & y_l : \mathtt{lin}\ \ulcorner B_l \urcorner \end{bmatrix} \vdash_\Sigma M \Uparrow \mathtt{ce}\ \ulcorner C \urcorner\ \ D\ \ D'$$

are derivable. □

Bibliografia

[ABCJ94] David Albrecht, Frank A. Bäuerle, John N. Crossley, and John S. Jeavons. Curry-Howard terms for linear logic. In ??, editor, *Logic Colloquium '94*, pages ??–?? ??, 1994.

[Abr93] Samson Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

[AHM89] Arnon Avron, Furio Honsell, and Ian A. Mason. An overview of the Edinburgh logical framework. In G. Birtwistle and P. A. Subramanyan, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 323–240. Springer-Verlag, 1989.

[AHMP92] Arnon Avron, Furio A. Honsell, Ian A. Mason, and Robert Pollack. Using typed lambda calculus to implement formal systems on a machine. *Journal of Automated Reasoning*, 9(3):309–354, 1992. A preliminary version appeared as University of Edinburgh Report ECS-LFCS-87-31.

[AK91] H. Aït-Kaci. *Warren's Abstract Machine*. MIT Press, 1991.

[AP91] Jean-Marc Andreoli and Remo Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[Asp87] Andrea Asperti. A logic for concurrency. Technical Report ??, Department of Computer Science, University of Pisa, 1987.

[Bar92] Henk Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume II, pages 118–309. Oxford University Press, 1992.

[BBHdP93] Nick Benton, Gavin Bierman, J. Martin E. Hyland, and Valeria de Paiva. A term calculus for intuitionistic linear logic. In M. Bezem and J. F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 75–90. Springer-Verlag LNCS 664, 1993.

[BHN+94] Manfred Broy, Ursula Hinkel, Tobias Nipkow, Christian Prehofer, and Birgit Schieder. Interpreter verification for a functional language. In *Proceedings of the 14th Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer-Verlag LNCS, 1994. To appear.

[BM78] Robert S. Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1978.

[BM88] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.

[C+86] Robert L. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[CDDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux, and Gilles Kahn. A simple applicative language: Mini-ML. In *Proceedings of the 1986 Conference on LISP and Functional Programming*, pages 13–27. ACM Press, 1986.

[Cer92] Serenella Cerrito. A linear axiomatization of negation as failure. *Journal of Logic Programming*, 12:1–24, 1992.

[Cer96] Iliano Cervesato. A linear logical framework. Available on request, 1996.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*. North-Holland, Amsterdam, 1958.

[CGR92] Jawahar L. Chirimar, Carl A. Gunter, and Jon G. Riecke. Proving memory management invariants for a language based on linear logic. *Lisp and Functional Programming*, pages 139–150, 1992.

[CH88] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3), 1988.

[Chi95] Jawahar L. Chirimar. *Proof Theoretic Approach to Specification Languages*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.

[CHP96] Iliano Cervesato, Joshua S. Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In *Proceedings of the 1996 International Workshop on Extensions of Logic Programming*, Leipzig, Germany, 1996. To appear.

[Chu32a] Alonzo Church. A set of postulates for the foundation of logic I. *Annals of Mathematics*, 33:346–366, 1932.

[Chu32b] Alonzo Church. A set of postulates for the foundation of logic II. *Annals of Mathematics*, 34:839–864, 1932.

[Chu40] Alonzo Church. A formulation of a simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CKvC73] A. Colmerauer, H. Kanoui, and M. van Caneghem. Un systeme de communication homme-machine en francais. Research report, Groupe Intelligence Artificielle, Universite Aix-Marseille II, 1973.

[Coq91] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[dB72] N. G. de Bruijn. Lambda-calculus notation with nameless dummies: a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indag. Math.*, 34(5):381–392, 1972.

[dB80] N.G. de Bruijn. A survey of the project AUTOMATH. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980.

[DFH+93] Gilles Dowek, Amy Felty, Hugo Herbelin, Gérard Huet, Chet Murthy, Catherine Parent, Christine Paulin-Mohring, and Benjamin Werner. The Coq proof assistant user's guide. Rapport Techniques 154, INRIA, Rocquencourt, France, 1993. Version 5.8.

[dG95] Ph. de Groote, editor. *The Curry-Howard Isomorphism*, volume 8 of *Cahiers du Centre de Logique, Département de Philosophie, Université Catholique de Louvain*. Academia, 1995.

[DHM91] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ml. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, January 1991. To appear. Also available as CMU technical report CMU-CS-90-184.

[DS89] Peter Dybjer and Herbert Sander. A functional programming approach to the specification and verification of concurrent systems. *Formal Aspects of Computing*, 1:303–319, 1989.

[Dug93] Dominic Duggan. Unification with extended patterns. Technical Report CS-93-37, University of Waterloo, Waterloo, Ontario, Canada, July 1993. Revised March 1994 and September 1994.

[Ell90] Conal M. Elliott. *Extensions and Applications of Higher-Order Unification*. PhD thesis, School of Computer Science, Carnegie Mellon University, May 1990. Available as Technical Report CMU-CS-90-134.

[Gan80] R.O. Gandy. Proofs of strong normalization. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays in Combinatory Logic, Lambda Calculus and Formalism*, pages 457–477. Academic Press, 1980.

[GdQ92] Dov M. Gabbay and Ruy J. G. B. de Queiroz. Extending the Curry-Howard interpretation to linear, relevant and other resource logics. *Journal of Symbolic Logic*, 57(4):1319–1365, December 1992.

[Geu93] Herman Geuvers. *Logics and Type Systems*. PhD thesis, Katholieke Universiteit Nijmegen, 1993.

[GG89] V. Gehlot and Carl A. Gunter. Nets as tensor theories. In G. De Michelis, editor, *Proceedings of the tenth International Conference on Application and Theory of Petri Nets*, pages 174–191, Bonn, Germany, 1989. Extended and revised version available as Technical Report MS-CIS-89-68, University of Pennsylvania.

[Gir87] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[Gir91] Jean-Yves Girard. A new constructive logic: Classical logic. *Mathematical Structures in Computer Science*, 1:255–296, 1991.

[Gir93] Jean-Yves Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59:201–217, 1993.

[GLT88] Jean-Yves Girard, Yves Lafont, and P. Taylor. *Proofs and Types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, 1988.

[Göd31] Kurt Gödel. Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931.

[Göd90] Kurt Gödel. On an extension of finitary mathematics which has not yet been used. In S. Feferman, J. W. Dawson Jr., S. C. Kleene, G. H. Moore, R. Solovay, and J. van Heijenoort, editors, *Kurt Gödel, Collected Works*, volume II, pages 271–280. Oxford University Press, 1990.

[GP94] Didier Galmiche and Guy Perrier. On proof normalisation in linear logic. *Theoretical Computer Science*, 135(1):67–110, 1994. Also available as Technical Report CRIN 94-R-113 from the Centre di Recherche en Informatique de Nancy.

[GSS92] Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: A modular approach to polynomial time computability. *Theoretical Computer Science*, 97:1–66, 1992. Extended abstract in *Feasible Mathematics*, S. R. Buss and P. J. Scott editors, Proceedings of the MCI Workshop, Ithaca, NY, June 1989, Birkhauser, Boston, pp. 195–209.

[Gun92] Carl A. Gunter. *Semantics of Programming Languages: Structures and Techniques.* Foundations of computing. MIT Press, 1992.

[Har90] Robert Harper. Systems of polymorphic type assignment in LF. Technical Report CMU-CS-90-144, Carnegie Mellon University, Pittsburgh, Pennsylvania, June 1990.

[Her71] Jacques Herbrand. *Logical Writings.* Harvard University Press, Cambridge, Massachusetts, 1971. Edited by W. D. Goldfarb.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[HM90] John Hannan and Dale Miller. From operational semantics to abstract machines: Preliminary results. In M. Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 323–332, Nice, France, 1990.

[HM94] Joshua S. Hodas and Dale Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. Extended abstract in the Proceedings of the Sixth Annual Symposium on Logic in Computer Science, Amsterdam, July 15–18, 1991.

[Hod94] Joshua S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design and Implementation.* PhD thesis, University of Pennsylvania, Department of Computer and Information Science, 1994.

[HP92] John Hannan and Frank Pfenning. Compiler verification in LF. In Andre Scedrov, editor, *Seventh Annual IEEE Symposium on Logic in Computer Science*, pages 407–418, Santa Cruz, California, June 1992.

[HP95] Joshua S. Hodas and Jeff Polakow. Documentation of the preliminary distribution of the beta version of the prototype of HMC Forum 0.5b, October 1995.

[Hue76] Gérard Huet. *Résolution d'équations dans des langages d'ordre* $1, 2, \ldots, \omega$. PhD thesis, Université Paris VII, September 1976.

[Hue94] Gérard Huet. Residual theory in λ-calculus: A formal development. *Journal of Functional Programming*, 4(3):371–394, July 1994. Preliminary version available as INRIA Technical Report 2009, August 1993.

[HW95] James Harland and Michael Winikoff. Implementing the linear logic programming language Lygon. In J. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium*, pages 66–80, Portland, Oregon, 1995.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich*, pages 111–119. ACM Press, January 1987.

[Jut77] L.S. van Benthem Jutting. *Checking Landau's "Grundlagen" in the AUTOMATH System.* PhD thesis, Eindhoven University of Technology, 1977.

[Kle52] Stephen Cole Kleene. *Introduction to Metamathematics*. North-Holland, 1952.

[Kow74] Robert A. Kowalski. Predicate logic as a programming language. In *Proceedings of the IFIP Congress*, pages 569–574, Stockholm, Sweden, 1974. North-Holland.

[Kow79] Robert A. Kowalski. *Logic for Problem Solving*. Elsevier North-Holland, 1979.

[Kow88] Robert A. Kowalski. The early years of logic programming. *Communications of the ACM*, 31(1):38–44, 1988.

[KY93] Naoki Kobayashi and Akinori Yonezawa. ACL — a concurrent linear logic programming paradigm. In D. Miller, editor, *Proceedings of the 1993 International Logic Programming Symposium*, pages 279–294, Vancouver, Canada, October 1993. MIT Press.

[Laf88] Yves Lafont. The linear abstract machine. *Theoretical Computer Science*, 59:157–180, 1988. Some corrections in volume 62 (1988), pp. 327–328.

[Laf90] Yves Lafont. Interaction nets. In *Seventeenth Annual Symposium on Principles of Programming Languages*, pages 95–108, San Francisco, California, 1990. ACM Press.

[Lam95] Joachim Lambek. Bilinear logic in algebra and linguistics. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, pages 43–59. Cambridge University Press, 1995. Proceedings of the Workshop on Linear Logic, Ithaca, New York, June 1993.

[LM92] Patrick Lincoln and John Mitchell. Operational aspects of linear lambda calculus. In *Seventh Annual Symposium on Logic in Computer Science*, pages 235–246, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[LM94] D. Lester and S. Mintchev. Towards machine-checked compiler correctness for higher-order languages. In *Proceedings of the 8th Workshop on Computer Science Logic*. Springer LNCS, 1994. To appear.

[LP92] Zhaohui Luo and Robert Pollack. The LEGO proof development system: A user's manual. Technical Report ECS-LFCS-92-211, University of Edinburgh, May 1992.

[Luo89] Zhaohui Luo. ECC, an extended Calculus of Constructions. In Rohit Parikh, editor, *Fourth Annual Symposium on Logic in Computer Science*, pages 386–395, Pacific Grove, California, June 1989. IEEE Computer Society Press.

[Mac91] Ian Mackie. Lilac — a functional programming language based on linear logic. Master's thesis, Imperial College, London, 1991.

[Mil89] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. In Peter Schroeder-Heister, editor, *Proceedings of the International Workshop on Extensions of Logic Programming*, pages 253–281, Tübingen, Germany, 1989. Springer-Verlag LNAI 475.

[Mil92] Dale Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P. Mello, editors, *Proceedings of the Workshop on Extensions of Logic Programming*, pages 242–265. Springer-Verlag LNCS 660, 1992.

[Mil94] Dale Miller. A multiple-conclusion meta-logic. In S. Abramsky, editor, *Ninth Annual Symposium on Logic in Computer Science*, pages 272–281, Paris, France, July 1994. IEEE Computer Society Press.

[ML73] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118. North-Holland, 1973.

[ML80] Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science VI*, pages 153–175. North-Holland, 1980.

[ML85a] Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. Technical Report 2, Scuola di Specializzazione in Logica Matematica, Dipartimento di Matematica, Università di Siena, 1985.

[ML85b] Per Martin-Löf. Truth of a propositions, evidence of a judgement, validity of a proof. Notes to a talk given at the workshop *Theory of Meaning*, Centro Fiorentino di Storia e Filosofia della Scienza, June 1985.

[MM77] Alberto Martelli and Ugo Montanari. Theorem proving with structure sharing and efficient unification. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence*, page 543 ff, Boston, 1977. IJCAI.

[MN94] Lena Magnusson and Bengt Nordström. The ALF proof editor and its proof engine. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, pages 213–237. Springer-Verlag LNCS 806, 1994.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[MOM91] N. Martí-Oliet and J. Meseguer. From Petri nets to linear logic. *Mathematical Structures in Computer Science*, 1:66–101, 1991. Revised version of paper in LNCS 389.

[Moo94] J. Strother Moore. The mechanization of induction in the Boyer-Moore theorem prover. Lectures given at the Second International Summer School in Logic for Computer Science, Chambéry, France, July 1994.

[MP91] Spiro Michaylov and Frank Pfenning. Natural semantics and some of its meta-theory in Elf. In L.-H. Eriksson, L. Hallnäs, and P. Schroeder-Heister, editors, *Proceedings of the Second International Workshop on Extensions of Logic Programming*, pages 299–344, Stockholm, Sweden, January 1991. Springer-Verlag LNAI 596.

[MPP92] Dale Miller, Gordon Plotkin, and David Pym. A relevant analysis of natural deduction. Talk given at the Workshop on Logical Frameworks, Båstad, Sweden, May 1992.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[Nip95] Tobias Nipkow. More Church-Rosser proofs (in Isabelle). Unpublished manuscript, July 1995.

[NM88] Gopalan Nadathur and Dale Miller. An overview of λProlog. In Kenneth A. Bowen and Robert A. Kowalski, editors, *Fifth International Logic Programming Conference*, pages 810–827, Seattle, Washington, August 1988. MIT Press.

[Nor93] Bengt Nordström. The ALF proof editor. In *Proceedings of the Workshop on Types for Proofs and Programs*, pages 253–266, Nijmegen, 1993.

[Pau88] Lawrence C. Paulson. Isabelle: The next seven hundred theorem provers. In E. Lusk and R. Overbeek, editors, *Proceedings of the 9th International Conference on Automated Deduction*, pages 772–773, Argonne, Illinois, 1988. Springer Verlag LNCS 310. System abstract.

[Pau93] Lawrence C. Paulson. The Isabelle reference manual. Technical Report 283, University of Cambridge, Computer Laboratory, 1993.

[Pfe99] Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. *Journal of Automated Reasoning*, 199? To appear. A preliminary version is available as Carnegie Mellon Technical Report CMU-CS-92-186, September 1992.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe92] Frank Pfenning. Computation and deduction. Unpublished lecture notes, revised May 1994, May 1992.

[Pfe94a] Frank Pfenning. Elf: A meta-language for deductive systems. In A. Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, pages 811–815, Nancy, France, June 1994. Springer-Verlag LNAI 814. System abstract.

[Pfe94b] Frank Pfenning. Structural cut elimination in linear logic. Technical Report CMU-CS-94-222, Department of Computer Science, Carnegie Mellon University, December 1994.

[Pfe95a] Frank Pfenning. Sequent calculi. Course notes for the *Seminar on Linear Logic and Applications*, Carnegie Mellon University, Spring'95, 1995.

[Pfe95b] Frank Pfenning. Structural cut elimination. In D. Kozen, editor, *Proceedings of the Tenth Annual Symposium on Logic in Computer Science*, pages 156–166, San Diego, California, June 1995. IEEE Computer Society Press.

[PM89] Christine Paulin-Mohring. Extracting F_ω programs from proofs in the calculus of constructions. In *Sixteenth Annual Symposium on Principles of Programming Languages*, pages 89–104. ACM Press, January 1989.

[PM93] Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications, TLCA'93*, pages 328–345, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.

[Pol94] Robert Pollack. *The Theory of LEGO: A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[PR96] Frank Pfenning and Ekkehard Rohwedder. Mode and termination analysis for higher-order logic programs. In *Proceedings of the European Symposium on Programming*, Linköping, Sweden, April 1996.

[Pra91] Vaughan R. Pratt. Event spaces and their linear logic. In *Proceedings of the Workshop on Algebraic Methodology and Software Technology*, pages 1–23, Iowa City, Iowa, 1991. Springer-Verlag.

[PW90] David Pym and Lincoln Wallen. Investigations into proof-search in a system of first-order dependent function types. In G. Huet and G. Plotkin, editors, *Proceedings of the First Workshop on Logical Frameworks*, pages 435–452. Preliminary Version, May 1990.

[Pym90] David Pym. *Proofs, Search and Computation in General Logic*. PhD thesis, University of Edinburgh, 1990. Available as CST-69-90, also published as ECS-LFCS-90-125.

[Ras95] Ole Rasmussen. The Church-Rosser theorem in Isabelle: A proof porting experiment. Technical Report 364, University of Cambridge, Computer Laboratory, March 1995.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Roh96] Ekkehard Rohwedder. *Verifying the Meta-Theory of Deductive Systems*. PhD thesis, Carnegie Mellon University, 1996. Forthcoming.

[Roo91] Dirk Roorda. *Resource Logics: Proof-theoretical Investigations*. PhD thesis, University of Amsterdam, September 1991.

[Ros90] Lars Rossen. Formal Ruby. In J. Staunstrup, editor, *Formal Methods for VLSI Design*. Elsevier, 1990.

[RR94] Simona Ronchi della Rocca and Luca Roversi. Lambda calculus and intuitionistic linear logic. In *Logic Colloquium '94*, Clermont Ferrand, France, 1994.

[RW91] Colin Runciman and David Wakeling. Linearity and laziness. In J. Hughes, editor, *Proceedings of the Fifth ACM Conference on Functional Programming Languages and Computer Architecture*, pages 215–240, Cambridge, Massachusetts, 1991. Springer-Verlag LNCS 666.

[Sal90] Anne Salvesen. The Church-Rosser theorem for LF with $\beta\eta$-reduction. Unpublished notes to a talk given at the First Workshop on Logical Frameworks in Antibes, France, May 1990.

[Sch95] Carsten Schürmann. A computational meta logic for the Horn fragment of LF. Master's thesis, Carnegie Mellon University, 1995. Also Technical Report CMU-CS-95-218.

[Sha88] N. Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the Association for Computing Machinery*, 35(3):475–522, July 1988.

[Sha94] N. Shankar. *Metamathematics, Machines, and Gödel's Proof*, volume 38 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1994.

[Sny91] Wayne Snyder. *A Proof Theory for General Unification*. Birkhäuser, 1991.

[Sol89] U. Solitro. A typed calculus based on a fragment of linear logic. *Theoretical Computer Science*, 68:333–342, 1989.

[Sza70] M. E. Szabo, editor. *The Collected Papers of Gerhard Gentzen*, Studies in Logic. North-Holland, 1970.

[Tro86] Anne S. Troelstra. Strong normalization for typed terms with surjective pairing. *Notre Dame Journal of Formal Logic*, 27(4):547–550, October 1986.

[Tro93] Anne S. Troelstra. Natural deduction for intuitionistic linear logic. Prepublication Series for Mathematical Logic and Foundations ML-93-09, Institute for Language, Logic and Computation, University of Amsterdam, 1993.

[Wad90] Philip Wadler. Linear types can change the world. In M. Broy and C. B. Jones, editors, *IFIP TC 2 Working Conference on Programming Concepts and Methods*, pages 561–581, Sea of Gallilee, Israel, April 1990. North-Holland.

[Wad91] Philip Wadler. Is there a use for linear logic? In *Proceedings of the Symposium on Partial Evaluations and Semantics-Based Program Manipulation*, pages 255–273, New Haven, Connecticut, June 1991. Also in SIGPLAN Notices; vol.26, no.9; September 1991.

[War83] D. H. D. Warren. An abstract Prolog instruction set. Technical Note 306, SRI International, Menlo Park, California, 1983.