Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Formalizzazione di una rappresentazione canonica dello sharing per la valutazione call-by-value

Relatore:
Chiar.mo Prof.
Claudio Sacerdoti Coen

Presentata da:
Davide Davoli

Sessione 3
Anno Accademico 2019/2020

Ricorderai d'avermi atteso tanto,
e avrai negli occhi un rapido sospiro.

Giuseppe Ungaretti,
Il sentimento del tempo

Indice

1 Introduzione **7**

2 λ-calcolo **11**
- 2.1 Sintassi e riduzione 11
  - 2.1.1 Sintassi 11
  - 2.1.2 Riduzione 15
- 2.2 Calcolabilità 19
- 2.3 Due discipline di valutazione in compendio 23
  - 2.3.1 Plotkin's call-by-value 25
  - 2.3.2 Plotkin's call-by-name 27
- 2.4 A posteriori 29

3 Crumbling **31**
- 3.1 Complessità del λ-calcolo 31
- 3.2 Crumbling Abstract Machines 33

4 Tipi e funzioni di manipolazione **47**

5 Variabili **53**
- 5.1 Decisione di variabili e variabili fresche 54
  - 5.1.1 Occorrenze libere 57
- 5.2 Relazioni fra variabili 57

6 Crumbling e read-back **61**
- 6.1 read-back 61
- 6.2 underline 64
- 6.3 Lemmi su `underline` e `read_back` 68
  - 6.3.1 Lemma 4.2 68
  - 6.3.2 Remark 4.1 71
  - 6.3.3 Lemma 4.5 75
  - 6.3.4 Lemma 4.5.5 78
- 6.4 Size Lemma 81

7 alpha **83**
7.1 alpha 84
7.1.1 Definizioni informali 84
7.1.2 Implementazioni in Matita 86
7.1.3 `beta` e `gamma` 88
7.2 Lemmi riguardanti la funzione `alpha` 90
7.2.1 Size lemma - `alpha` 90
7.2.2 Lemma 5.5 90
7.2.3 Lemma 5.5.3 92
7.2.4 Lemma 5.5.4 94

8 Conclusioni **99**

Appendice A Matita: uso avanzato **101**
A.1 Introduzione ai proof assistants 101
A.2 p_Subst 102
A.3 Principi di induzione 116
A.4 Tecniche di riduzione in compendio 121

Capitolo 1

Introduzione

Il λ-calcolo è un formalismo di particolare interesse perché ha una definizione molto semplice, completamente ricorsiva e di alto livello che rende possibile costruire agevolmente dimostrazioni per induzione strutturale sui suoi termini, agevolando dunque l'analisi delle proprietà di tale formalismo. Per questa ragione la teoria del lambda calcolo è molto ricca di risultati interessanti, fra cui, per esempio, quella di Turing completezza o quella espressa dall'isomorfismo di Curry-Howard, il quale asserisce l'esistenza di una corrispondenza biunivoca fra termini ben formati del λ-calcolo tipato e derivazioni in deduzione naturale per la logica del prim'ordine. Inoltre, nel λ-calcolo ogni termine è una funzione anonima, una variabile o l'applicazione di due termini; ciò fa sì che, in tale formalismo, la nozione di calcolo coincida con quella di semplificazione di applicazioni di termini (funzioni) mediante una regola di riduzione β. Per questa ragione esso costituisce una solida base per lo sviluppo di linguaggi funzionali come LISP, Haskell, o i linguaggi della famiglia ML. Infine, in virtù dell'isomorfismo di Curry-Howard, il *lambda*-calcolo è anche una solida base per lo sviluppo di strumenti di dimostrazione assistita fra cui possiamo annoverare per esempio Coq, Agda, Isabelle o Matita.

Di contro, però, la definizione d'alto livello del lambda calcolo fa sì che questo non abbia una propria implementazione canonica su cui studiarne la complessità, cosicché la letteratura lo ha spesso considerato un formalismo poco adatto ad essere oggetto di tali studi; inoltre sono documentati fenomeni di esplosione della taglia di alcuni termini che, in caso di implementazione naïve del calcolo, fanno balzare la complessità di valutazione di questi ad ordini più che esponenziali. Solo di recente sono state proposte implementazioni del λ-calcolo che non soffrono di questa anomalia e che permettono la valutazione di λ-termini con un overhead polinomiale, bilineare o lineare nel numero di β-riduzioni necessarie alla riduzione del termine alla sua forma normale. Una di queste tecniche è quella del Crumbling, la quale prevede che i termini siano arricchiti da un costrutto di sharing esplicito, abbreviato ES ed espresso nella forma $t[x \leftarrow t']$, la cui semantica è simile a quella del costrutto `let` presente nei più comuni linguaggi funzionali: un ES lega all'interno del termine t la variabile x al termine

t'. Grazie allo sharing è possibile scomporre le β-riduzioni in operazioni più semplici da operare secondo discipline di valutazione finalizzate ad evitare fenomeni di esplosione del termine e a mantenere basso l'overhead dell'esecuzione. Lo sharing permette anche la rappresentazione dei termini nella loro Crumbled Form, che consiste nella decomposizione ricorsiva delle applicazioni mediante l'introduzione di ES.

Un lavoro preliminare per lo sviluppo di questa teoria è quello presentato in [5], in cui vengono descritte le procedure di traduzione dei termini in Crumbled Forms, il processo di queste da parte di una macchina astratta Crumble GLAM sia secondo una disciplina di valutazione closed CbV che secondo una politica open CbV e la riconversione delle Crumbled Forms in λ-termini; infine viene studiato l'overhead indotto dalla macchina astratta Crumble GLAM in entrambe le discipline di valutazione per evidenziare la ragionevolezza di questa implementazione. La correttezza di tutta la teoria proposta nell'articolo si regge sulla correttezza delle dimostrazioni dei numerosi lemmi che vi compaiono, per cui sulle 39 pagine in cui si dipana il lavoro, più della metà sono composte da dimostrazioni informali di tali lemmi. Tuttavia, nonostante il rigore con cui queste sono presentate, molti passaggi delle dimostrazioni sono omesse. Per questa ragione è stato utile intraprendere un'operazione di formalizzazione di questi lemmi e delle loro dimostrazioni al fine di evidenziare eventuali problemi nella teoria e di proporvi soluzioni.

Il mio lavoro ha riguardato prevalentemente la formalizzazione in Matita delle dimostrazioni circa la correttezza del procedimento di inizializzazione della macchina, quindi la dimostrazione di una serie di invarianti garantiti dall'operazione di conversione dei λ-termini in Crumbled Forms (detta crumbling o underline) e dalla funzione di conversione di Crumbled Forms in λ-termini (detta read-back). Due importanti invarianti sono stati il lemma che dimostra che la funzione di read-back è l'inversa sinistra di quella di crumbling ed il lemma che dimostra che il crumbling genera right-v evaluation contexts: un invariante della macchina Crumble GLAM necessario all'applicazione della disciplina di valutazione call-by-value. Una buona parte del lavoro ha riguardato la gestione dei nomi di variabile: l'articolo lavorava con l'assunzione implicita che, ove necessario, i nomi di variabili fresche generate dalle funzioni coincidessero. Ciò ha reso necessaria la scelta e l'implementazione ex-novo di politiche di gestione di nomi di variabili fresche in tutte le funzioni che ne facevano uso. Tali politiche, oltre ad essere semanticamente corrette, dovevano contribuire a non rilassare la nozione uguaglianza dalla sua formulazione forte a quella debole (intesa modulo α-conversione) per non indebolire le formulazioni dei lemmi ove non necessario. A causa di questa discrepanza alcuni lemmi si sono dimostrati essere addirittura falsi, per cui è stato necessario adottare diverse strategie di dimostrazione per evitarne l'applicazione o riformularne gli enunciati per poter ottenere i medesimi risultati da lemmi con enunciati più semplici ma dimostrabili. Siccome alcuni de lemmi che si sono verificati falsi erano utilizzati per dimostrare invarianti della macchina anche successive la fase di inizializzazione, la risoluzione di questi problemi ha toccato anche aspetti dinamici della macchina.

Per dare alcune misure dell'estensione di questo lavoro, la dimostrazione

formale di una decina di teoremi che in mettono in relazione un esiguo numero di funzioni e di tipi di dati ha richiesto la definizione di più di 400 lemmi e circa 100 funzioni e le circa 5 pagine di dimostrazione informale hanno richiesto la produzione di circa 12 KLOC di dimostrazioni formali.

In questo lavoro verrà presentato, in principio, un capitolo introduttivo nel quale verrà descritto il formalismo del λ-calcolo, successivamente verrà resposta la teoria delle crumbling abstract machines in riferimento a come viene presentata in [5]. Le parti successive del lavoro descriveranno nello specifico il lavoro di formalizzazione che ho operato in [14] sulla parte di [5] relativa all'inizializzazione della macchina Crumble GLAM, prestando particolare attenzione ai lemmi che hanno richiesto una totale o parziale riformulazione a causa della loro imprecisa formulazione.

A ciò è anche dovuto il lavoro di formalizzazione che ho raccolto in A, dove ho implementato e formalizzato alcuni aspetti del comportamento dinamico della macchina Crumble GLAM al fine di risolvere alcuni problemi nei risultati di [5] identificati nei capitoli precedenti.

Infine, ho deciso di includere un'appendice nella quale ho raccolto alcuni aspetti peculiari relativi all'uso del dimostratore interattivo Matita, che ho dovuto affrontare durante il presente lavoro di formalizzazione. In questo modo, spero che possano essere d'aiuto sia al lettore interessato ad approfondire l'uso di tali strumenti, che a quello interessato a comprendere le ragioni di alcune scelte di formalizzazione.

Capitolo 2

λ-calcolo

Il λ-calcolo è un formalismo di calcolo sviluppato da Alonzo Church e proposto fra il 1932 ed il 1933 come parte di sistema più ampio proposto come fondamento della matematica. Il sistema complessivo, tuttavia, fu dimostrato inconsistente da Kleene e Rosser nel 1935, ma l'insieme di assiomi concernenti la formalizzazione della nozione di calcolo e calcolabilità per mezzo di riduzione (riscrittura) di espressioni rimane di notevole interesse.

Il sotto-sistema in questione, il λ-calcolo, è tuttora oggetto di numerosi studi e ricerche poiché permette di esprimere e di calcolare funzioni per mezzo di una struttura sintattica molto semplice ed una sola regola di riduzione. A riguardo del potere espressivo di tale formalismo, possiamo dire che esso sia completo in senso di *Church*-Turing[1]. Tali ragioni, assieme alla natura strettamente induttiva della sua definizione, lo rendono uno strumento di alto livello molto potente di indagine per la teoria della calcolabilità.

Inoltre, in virtù dell'isomorfismo di Curry-Howard-Kolmogorov, è possibile identificare una corrispondenza biunivoca fra sistemi di tipo per il lambda calcolo e determinate logiche, per cui il λ-calcolo può essere un utile strumento per lo studio degli aspetti computazionali della logica e per lo sviluppo di dimostratori interattivi di teoremi.

2.1 Sintassi e riduzione

2.1.1 Sintassi

Dal momento che il λ-calcolo è un sistema finalizzato alla rappresentazione di funzioni, ci si aspetta che la sua sintassi preveda quantomeno un costrutto per definire funzioni ed un costrutto per applicare funzioni. Il primo di questi costrutti è il costrutto λ, da cui il nome del formalismo, che permette di fare astrazione di un termine, per mezzo di una variabile, all'interno di un altro.

[1] Come riportato da [7], fu Turing stesso, con una coppia di lavori pubblicati fra il 1936 ed il 1937, a dimostrare l'equivalenza fra il suo formalismo e quello di Church.

Questo tipo di costruttori è molto simile a quello di norma usato in matematica; si prenda, ad esempio, la definizione $f(x) \coloneqq x + 2$: il costruttore $\coloneqq$ lega al parametro x qualsiasi termine inserito all'interno delle parentesi, ed il nome di funzione f all'espressione di destra. In λ-calcolo le dichiarazioni di funzioni funzionano pressoché nello stesso modo, a differenza del fatto che le funzioni sono anonime. Ammettendo di avere la possibilità di esprimere l'operatore di somma infisso nel lambda calcolo, la f di sopra si scriverebbe come $\lambda x.x + 2$.

Più propriamente possiamo dire che il costruttore di funzione corrispondente matematico più simile a λ non sia $\coloneqq$, bensì $\mapsto$: esso infatti permette di definire funzioni che siano *anonime*, proprio come avviene nel λ-calcolo.

Il linguaggio formale del λ-termini, che chiameremo Λ, può essere espresso come il linguaggio $L(t)$ generato dalla (2.1), dove il simbolo x denota una possibile variabile da un insieme Var che ha un'infinità quantomeno numerabile di elementi.

$$t \coloneqq tt \mid \lambda x.t \mid x \tag{2.1}$$

Più propriamente, alcuni testi sono soliti la grammatica del λ-calcolo come in (2.2), in modo da poter formalmente catturare il vincolo sull'insieme Var precedentemente descritto.

$$\begin{aligned} t &\coloneqq tt \mid \lambda x.t \mid x \\ x &\coloneqq var|x' \end{aligned} \tag{2.2}$$

Entrambe le definizioni, ai fini della presente trattazione, sono da considerarsi equivalenti; qualora il lettore preferisca fare riferimento alla seconda allora è opportuno proporre anche la seguente definizione: $var \coloneqq L(x)$. Alcune osservazioni:

- La produzione $\lambda x.t$ viene comunemente detta "astrazione" poiché permette di costruire una funzione con corpo t in cui la variabile x è un'astrazione dell'argomento con cui verrà chiamata.

- La produzione tt prende solitamente il nome di "applicazione" e consiste, di fatto, nell'applicazione del termine di sinistra con il termine di destra come argomento. Questa notazione è poco comune sia in matematica che in informatica, laddove solitamente si preferisce indicare l'argomento di una funzione fra parentesi: per esempio, la funzione $f(3)$, in questa notazione, si riscriverebbe + 3 2.

- Di solito si assume implicitamente che le astrazioni abbiano la precedenza sulle applicazioni, dunque il termine $\lambda x.xy$ va interpretato come $\lambda x.(xy)$, piuttosto che come $(\lambda x.x)y$.

- Tradizionalmente le applicazioni associano a sinistra: ad esempio il termine xyz si costruisce come $(xy)z$.

- L'ipotesi di poter legare una sola variabile mediante il costrutto di astrazione non è affatto riduttiva: i costrutti di astrazione si possono annidare

senza perdere di generalità. Per convincersene basta pensare che anche aritmeticamente vale la relazione $\forall A, B, C.|(A \times B) \to C| = |A \to (B \to C)|$.

- Le applicazioni prendono anche il nome di termini e sono in genere espressi per mezzo delle lettere $t, u, s, \ldots$.
- Le astrazioni e le variabili prendono anche il nome di "valori". I valori in genere sono riassunti dalle lettere $v, w, \ldots$, mentre, nello specifico, le variabili sono di solito riassunte per mezzo delle lettere $x, y, z, \ldots$

In generale, la regola di riscrittura di un'applicazione agisce su termini della forma $(\lambda x.M)N$ e li riscrive come $M\{x \leftarrow N\}$. Dove, tale espressione denota il termine M all'interno del quale tutte le occorrenze libere di x sono state sostituite dal termine N. Per esempio, $(\lambda x.x)y$ (che è l'applicazione della funzione $(\lambda x.x)$ ad y) si riscrive come $x\{x \leftarrow y\} = y$.

Sui termini $t \in \Lambda$, anche al fine di poter definire formalmente l'operazione di sostituzione, è interessante definire l'insieme $FV(t)$ delle le variabili libere in t. Bisognerebbe, infatti, che l'operazione di sostituzione fosse in grado di evitare la cattura di variabili, in modo da non modificare la semantica del termine. La definizione di variabile libera per il λ-calcolo ricalca sostanzialmente quella della logica del prim'ordine ed è espressa come segue:

$$\begin{aligned} FV(x) &\coloneqq \{x\} \\ FV(t_1 t_2) &\coloneqq FV(t_1) \cup FV(t_2) \\ FV(\lambda x.t) &\coloneqq FV(t) \setminus \{x\} \end{aligned} \tag{2.3}$$

Per esempio, possiamo calcolare l'insieme delle variabili libere del termine $\lambda x.xy$ nel seguente modo: $FV(\lambda x.xy) = \{x, y\} \setminus \{y\}$. È interessante la definizione di termine chiuso, che ancora una volta ricalca la definizione di formula chiusa per la logica del prim'ordine.

Definizione 2.1.1 (Termine chiuso). Si dice che un termine $t \in \Lambda$ è *chiuso* se *e solo se* $FV(t) = \emptyset$.

Oltre alla definizione di variabile libera, è utile costruire l'insieme delle variabili che semplicemente appaiono in un termine come descritto dalla (5.3). Mediante tale definizione, infatti, è possibile costruire l'insieme delle variabili fresche per un determinato termine, i.e. che non vi compaiono. Tale costruzione sarà utile, lo vedremo, perché ci permetterà di avere un insieme dal quale prelevare (o equivalentemente: un modo per calcolare) variabili fresche qualora sia necessario inserirle nel termine.

$$\begin{aligned} IN(x) &\coloneqq \{x\} \\ IN(t_1 t_2) &\coloneqq IN(t_1) \cup IN(t_2) \\ IN(\lambda x.t) &\coloneqq IN(t) \cup \{x\} \end{aligned} \tag{2.4}$$

Definizione 2.1.2 (Variabile fresca). Una variabile k è fresca per il termine t se $k \in FRESH(t) \coloneqq Var \setminus IN(t)$.

La nozione di termine chiuso è interessante poiché, quando definiamo una funzione, ed in lambda calcolo tutti i termini sono funzioni, se tale funzione è chiusa, allora possibile interpretarla univocamente. Al contrario, se tale funzione è aperta, per poterla interpretare dobbiamo fare riferimento a definizioni esterne delle variabili in essa mancanti. Ad esempio, il termine $\lambda f.\lambda x.\lambda y.fxy$ è una funzione che si interpreta univocamente come segue: $f, x, y \mapsto f(x, y)$, che è una funzione chiusa, ben costruita, diremmo: senza riferimenti esterni. Al contrario, il termine $\lambda x.\lambda y.fxy$, si interpreta come $x, y \mapsto f(x, y)$, ma per poter attribuire a questa funzione una semantica è necessaria una definizione della funzione f.

Dualmente alla definizione di variabile libera, è interessante fornire la definizione di variabile legata:

Definizione 2.1.3. Diciamo che la variabile x, o meglio che *un'occorrenza* della variabile x è legata all'interno del termine t se nell'albero sintattico di t, x ha come antenato un'astrazione $\lambda x.$.

È più proprio parlare di *occorrenze* di variabili libere e legate, poiché una stessa variabile, all'interno del medesimo termine, può comparire contemporaneamente libera e legata. Per esempio, la variabile x nel termine $\lambda z.x\lambda x.xz$ appare libera a sinistra della seconda astrazione, e legata a destra di essa.

Durante la valutazione, le occorrenze di variabili legate possono diventare momentaneamente libere, ma questo passaggio avviene solo al fine di consentire la loro sostituzione con l'argomento con cui è stata invocata la funzione. Da questo punto di vista, si evince come le variabili legate siano una sorta di "segnaposto" per future sostituzioni, mentre le variabili libere rappresentino termini generici per i quali non è stata (ancora) fornita un'interpretazione.

A titolo illustrativo, proviamo a definire alcuni esempi informali di funzioni e di loro riscritture in λ-calcolo.

Esempio 2.1.1. Una delle funzione più semplici da esprimere in λ-calcolo è la funzione identità, che viene generalmente sintetizzata con il simbolo $I \coloneqq \lambda x.x$. Tale scrittura, infatti, descrive una funzione in cui il corpo x è legato dall'astrazione $\lambda x.$ al valore assunto dal termine con cui è applicata: applicandola ad un generico argomento z, si ottiene la riduzione che segue: $(\lambda x.x)z$ si riscrive come $x\{x \leftarrow z\}$ che equivale a z.

Esempio 2.1.2. Un altro termine interessante è il termine $\omega \coloneqq \lambda x.xx$. Esso si riscrive come l'auto-applicazione del termine su cui è chiamato, infatti: $(\lambda x.xx)z$ si riscrive come $xx\{x \leftarrow z\}$ che equivale a zz. Applicando il termine ω a se stesso otteniamo un effetto curioso, difatti: $\omega\omega = (\lambda x.xx)(\lambda x.xx)$ si riscrive come $xx\{x \leftarrow (\lambda x.xx)\}$ che equivale a $(\lambda x.xx)(\lambda x.xx) = \omega\omega$. Il termine $\Omega \coloneqq \omega\omega$ che abbiamo costruito si può sempre ridurre sempre in se stesso, questo fenomeno, da un punto di vista computazionale, è un esempio di divergenza.

In merito al ruolo delle variabili legate da λ, possiamo osservare che se in un termine qualsiasi, ad esempio I, sostituiamo all'interno del corpo dell'astrazione tutte le occorrenze della variabile x con una qualsiasi altra variabile che non compare nel termine, si prenda ad esempio y, ed anziché legare la x, leghiamo la y, la semantica della funzione non cambia: infatti $(\lambda x.x)z = (\lambda y.y)z = z$. Quest'operazione prende il nome di α-conversione.

Definizione 2.1.4 (α-convertibilità). Due termini t e t' si dicono α-convertibili, e si scrive $t \equiv_\alpha t'$ se t', è ottenibile da t per mezzo di una sequenza $\sigma_{t,t'}$ finita di riscritture delle λ-astrazioni $\lambda x.M$ di t con termini del tipo $\lambda y.(M\{x \leftarrow y\})$.

Si osserva che la relazione di α-convertibilità è una relazione di equivalenza, infatti, per ogni termine valgono:

1. $t \equiv_\alpha t' \rightarrow t' \equiv_\alpha t$
2. $t \equiv_\alpha t$ con $\sigma_{t,t} = nil$
3. $t \equiv_\alpha t' \wedge t' \equiv_\alpha t'' \rightarrow t \equiv_\alpha t''$ con $\sigma_{t,t''} = \sigma_{t,t'}\sigma_{t',t''}$

Per questa ragione, di norma, in letteratura la relazione di uguaglianza fra due termini t e t' è espressa modulo α-conversioni; tale operazione, inoltre, si rivelerà utile anche nella prossima sezione quando definiremo formalmente l'operazione di sostituzione.

2.1.2 Riduzione

Per dare una semantica all'operazione di β-riduzione è necessario innanzitutto formalizzare l'operazione di sostituzione su λ-termini. La definizione canonica tale funzione è simile alla seguente (2.5)[2].

$$\begin{aligned}
x\{x \leftarrow t\} &\coloneqq t \\
x\{y \leftarrow t\} &\coloneqq x \quad \text{se } x \neq y \\
(t_1 t_2)\{x \leftarrow t\} &\coloneqq (t_1\{x \leftarrow t\})(t_2\{x \leftarrow t\}) \\
(\lambda x.s)\{x \leftarrow t\} &\coloneqq (\lambda x.s) \\
(\lambda z.s)\{x \leftarrow t\} &\coloneqq \lambda z.(s\{x \leftarrow t\}) \quad \text{se } z \notin FV(t) \\
(\lambda z.s)\{x \leftarrow t\} &\coloneqq \lambda k.(s\{z \leftarrow k\}\{x \leftarrow t\}) \quad \text{se } z \in FV(t), \text{ con } k \in
\end{aligned} \tag{2.5}$$

Nell'ultima riga, la richiesta di k variabile fresca ha lo scopo di evitare che la riduzione di un termine possa cambiarne la semantica. L'operazione di α-conversione non dovrebbe modificare la semantica di nessun λ-termine: a meno di α-conversioni ci si aspetta che l'esito della valutazione di due termini α-equivalenti sia lo stesso, come del resto abbiamo verificato per il caso triviale di due diverse istanze di I in 2.1.1.

[2]Per una rapido excursus sulle possibili differenti definizioni di funzioni di sostituzione su λ-termini, il lettore può consultare A.2.

Purtroppo però, se non operiamo l'accortezza di α-convertire il termine su cui viene effettuata la sostituzione nel caso in cui la variabile legata sia libera nel sostituendo, causeremmo di certo fenomeni di cattura di variabili libere che modificherebbero la semantica del termine, come mostrato dal controesempio 2.1.3.

Esempio 2.1.3. Supponiamo di voler valutare $t \coloneqq (\lambda y.\lambda x.xyx)x$: se non operassimo la conversione prevista dalla 2.5, otterremmo il seguente termine: $\lambda x.xxx$, ma questo risultato è scorretto: ora il binder $\lambda x.$, lega tutte le variabili all'interno di xxx. Mentre il termine $t_\alpha \coloneqq (\lambda y.\lambda z.zyz)x$, che è α-equivalente a t, si ridurrebbe (correttamente) come $\lambda z.zxz$, dove il binder più interno continua a legare le sole due variabili z.

Come abbiamo anticipato, l'operazione di riduzione agisce su termini che hanno la forma $(\lambda x.M)N$, i quali sono detti redex o, in italiano, *redessi*, riscrivendoli come $M\{x \leftarrow N\}$, dove la funzione di sostituzione ha la specifica che ne abbiamo descritto in (2.5). Più formalmente, diremo che il termine $(\lambda x.M)N$ si riduce in un passo in $M\{x \leftarrow N\}$ e scriveremo: $(\lambda x.M)N \to_\beta M\{x \leftarrow N\}$.

Dal momento che i redessi possono trovarsi in una qualsiasi posizione di un termine t, è interessante generalizzare la nozione di riducibilità in un passo in modo da catturare quando interi termini (i.e. non semplici redessi), si riducono in un passo in altri. Per farlo definiamo tale relazione mediante le regole in (2.6) e la definizione 2.1.5.

$$
\frac{}{(\lambda x.M)N \to_\beta M\{x \leftarrow N\}}\ \lambda_a
$$
$$
\frac{M \to_\beta M'}{MN \to_\beta M'N}\ a_l \qquad \frac{N \to_\beta N'}{MN \to_\beta MN'}\ a_r \qquad \frac{M \to_\beta M'}{\lambda x.M \to_\beta \lambda x.M'}\ \lambda_b \tag{2.6}
$$

Definizione 2.1.5 (Raggiungibilità in un passo). Definiamo la relazione di raggiungibilità in un passo $\to_\beta$ come il più piccolo sottoinsieme di $\Lambda \times \Lambda$ chiuso rispetto le regole di 2.6.

Dalle regole espresse in 2.6 possiamo evincere che, da un punto di vista computazionale, il λ-calcolo sia molto flessibile: le riduzioni possono avvenire non solo nelle applicazioni di funzioni (λ_a), nelle espressioni che descrivono funzioni (a_l), negli argomenti di funzione (a_r), ma anche all'interno del corpo stesso delle funzioni (λ_b). Mentre le prime tre riduzioni sono comuni alla maggior parte dei linguaggi di programmazione (che sono a tutti gli effetti formalismi di calcolo) moderni, quest'ultima proprietà, se viste in termini moderni, esprime il fatto che il sistema di riduzione (i.e. di valutazione) dei λ-termini sia in grado di effettuare delle ottimizzazioni a tempo di esecuzione: infatti, le riduzioni del corpo delle funzione sono di fatto delle semplificazioni del codice da eseguire.

Sulla base della relazione di riducibilità in un passo possiamo anche definire la relazione di raggiungibilità in n passi come descritto in 2.1.7, infatti ci aspettiamo che di rado la valutazione di un termine si risolva in un solo passo di computazione.

Definizione 2.1.6 (Raggiungibilità in n passi). Diciamo che un termine t_n è raggiungibile in n passi a partire dal termine t_0, e scriviamo $t_0 \to_\beta^n t_n$, se può essere ottenuto mediante l'applicazione di n regole in (2.6). Formalmente, possiamo definire tale relazione come segue:

- $t \to_\beta^0 t$.
- $t \to_\beta^{n+1} t_{n+1}$ se $\exists t_n . t \to_\beta^n t_n \wedge t_n \to_\beta t_{n+1}$.

Definizione 2.1.7 (Raggiungibilità). Diciamo che un termine t^* è raggiungibile t, e scriviamo $t \to_\beta^* t^*$, esiste un numero n tale che $t \to_\beta^n t^*$ Formalmente, possiamo definire tale relazione come: $t \to_\beta^* t^* \leftrightarrow \exists n . t \to_\beta^n t^*$.

Un'ultima definizione che è utile dare è quella di termine normale: grazie ad essa è possibile avere un criterio univoco per discernere termini che sono ancora riducibili da termini che lo sono più, ossia per i quali la computazione ha raggiunto il termine.

Definizione 2.1.8 (Forma normale). Diciamo che un termine t è normale, e scriviamo $t \not\to_\beta$ se su di esso non è più possibile operare riduzioni. Formalmente scriviamo $t \not\to_\beta \leftrightarrow \nexists t' . t \to_\beta t'$.

La relazione di raggiungibilità così definita è non-deterministica: dato un termine t ed un termine t_n tale che $t \to_\beta^n t_n$, se in t sono presenti più redessi, possono esistere più riduzioni diverse che sono in grado di ridurre t in t_n. Un tale esempio, può essere quello espresso in 2.1.4.

Esempio 2.1.4. Il termine $t \coloneqq (II)(II)$ si riduce in I, che è un termine normale, ma le sequenze di riduzione che permettono di normalizzare t sono più di una, come mostrato dal seguente grafico:

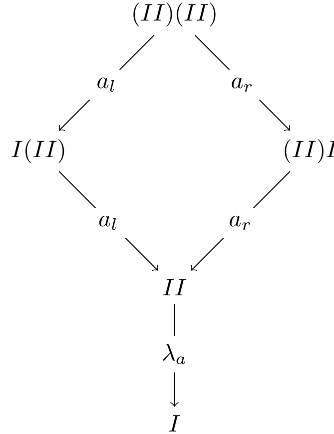

Figura 2.1: Diverse riduzioni del termine $(II)(II)$.

Definizione 2.1.9 (Formalismo di calcolo non-deterministico). Diciamo che un formalismo di calcolo è non-deterministico se e solo se $\exists t.t \to t' \wedge t \to t'' \wedge t' \neq t''$.

In generale, un formalismo di calcolo non-deterministico potrebbe essere problematico dal momento che, in linea di principio, nulla ci assicura che tutte le sequenze di riduzione siano normalizzanti, come si vede dall'esempio 2.1.5, e non è nemmeno certo che tutte le sequenze di riduzione di un dato termine t, generino termini t_* che siano a loro volta normalizzabili ed, infine, nulla assicura che le forme normali siano uniche.

Esempio 2.1.5. Il termine $(\lambda x.y)(\omega\omega)$ ha come forma normale y, ma le sequenze di riduzione che permettono di normalizzare t infinite, come mostrato dal grafico 2.2, in più: una di queste sequenze di riduzione è divergente e non porta mai alla normalizzazione del termine.

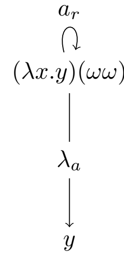

Figura 2.2: Diverse riduzioni del termine $(\lambda x.y)(\omega\omega)$.

Fortunatamente, però, il λ-calcolo gode di alcune proprietà le quali garantiscono che il non-determinismo della relazione di riducibilità non sia *patologico*: in particolare vedremo che se un termine ammette una forma normale, allora tale forma è unica e che la relazione di riducibilità in n passi sia *safe*.

Definizione 2.1.10 (Safety). Diciamo che un formalismo di calcolo, con relazione di transizione $\mapsto$ e chiusura transitiva di tale relazione $\mapsto^*$ è safe se e solo se $\forall t.\exists t_f.t \mapsto^* t_f \to \forall t'.t \mapsto^* t' \to t' \mapsto^* t_f$.

Teorema 2.1.1 (di Church-Rosser, di confluenza)
$\forall M, M', M'', N.(M \to_\beta^* M') \to (M \to_\beta^* M'') \to \exists N.M' \to_\beta^* N \wedge M'' \to_\beta^* N.$

Per la dimostrazione del teorema di Church-Rosser si faccia riferimento a [7]; al contrario, le proprietà di safety e di unicità della forma normale possono essere enunciate e dimostrate come semplici corollari di 2.1.1.

Corollario 2.1.1 (Unicità della forma normale). $\forall t, u, s.(t \to_\beta^* u \wedge u \not\to_\beta) \wedge (t \to_\beta^* s \wedge s \not\to_\beta) \to u = s.$

Dimostrazione. Per provare l'enunciato è sufficiente istanziare il teorema di Church-Rosser (2.1.1) su t, u, s. Otteniamo $\exists N.u \to_\beta^* N \wedge s \to_\beta^* N$, ma per ipotesi u ed s sono normali, per cui, per definizione di normalità, si riducono solo in se stessi, da cui $N = u$ e $N = s$.

Corollario 2.1.2 (Safety). $\forall t, u.(t \to_\beta^* u \wedge u \not\to_\beta) \to \forall s.t \to_\beta^* s \to s \to_\beta^* u$.

Dimostrazione. Per provare l'enunciato è sufficiente istanziare il teorema di Church-Rosser (2.1.1) su t, u, s. Otteniamo $\exists N.u \to_\beta^* N \wedge s \to_\beta^* N$, ma per ipotesi u è già forma normale, per cui $u = N$, da cui $s \to_\beta^* u$.

Questi risultati sono molto importanti per il λ-calcolo, poiché esprimono il fatto che sui termini di Λ possano essere sviluppate diverse discipline di valutazione deterministiche le quali, se normalizzano il termine di partenza t in una forma normale t_n, producono il medesimo risultato, altrimenti producono termini t_i che sono comunque riducibili ad un'unica forma normale t_n. Vedremo in 2.3 una rapida trattazione di alcune di queste discipline di valutazione per il λ-calcolo.

2.2 Calcolabilità

Come abbiamo già anticipato, il λ-calcolo è un formalismo Turing-completo, anzi: dal momento che la definizione del formalismo delle macchine di Turing è successiva a quella del λ-calcolo, sarebbe più opportuno dire che le MdT sono un formalismo λ-completo.

Ad ogni modo può sembrare contro-intuitivo che un formalismo così semplice possa essere sufficientemente espressivo per poter soddisfare la congettura di Church-Turing. Tuttavia, in questa sezione ne daremo una rapida prova mostrando una riduzione del formalismo della ricorsione generale al λ-calcolo, come proposto da Barendregt in [9].

Come prima cosa definiamo una codifica per i booleani e diamo un'implementazione di un primo costrutto di scelta:

Definizione 2.2.1 (`true` e `false`)
Definiamo i booleani `true` e `false` come segue:

- `true` $:= \lambda x.\lambda y.x$ $(\equiv K)$
- `false` $:= \lambda x.\lambda y.y$ $(\equiv K_*)$

Mediante tali costrutti è possibile implementare il costrutto di scelta `if-then-else` come:

Definizione 2.2.2 (`if-then-else`)
Definiamo il costrutto di `if b then m else n` come bmn.

Infatti si verifica che, dato un qualsiasi booleano b, l'espressione bmn si riduce in m se $b =$ `true` e in n se $b =$ `false`. Tramite i booleani è possibile costruire il tipo di dato `pair` con i rispettivi proiettori π_1 e π_2 come espresso dalla definizione 2.2.3.

Definizione 2.2.3 (pair)

- $\langle p, q \rangle \coloneqq \lambda x.xpq$
- $\pi_1(p) \coloneqq p\,\texttt{true}$
- $\pi_2(p) \coloneqq p\,\texttt{false}$

La correttezza di tale costruzione più essere verificata tramite l'esempio 2.7.

$$\pi_1(\langle p, q \rangle) = (\lambda x.xpq)\texttt{true} \to_\beta \texttt{true}pq \to_\beta p \tag{2.7}$$

Nel λ-calcolo è possibile anche codificare i numerali: specifiche funzioni che denotano i naturali. Mentre per i booleani e per le coppie la codifica è abbastanza uniforme in letteratura, per gli altri tipi di dati sono state sviluppate varie codifiche, due fra le più importanti sono quella di Scott e quella di Church[3]. Barendregt nella dimostrazione che riportiamo costruisce una diversa codifica, ma che poi dimostra equivalente a quella di Church. La costruzione procede come mostrato nella 2.2.5.

Definizione 2.2.4 (Numerali di Barendregt)

- $b_0 \coloneqq I$
- $b_{(S\ n)} \coloneqq \langle \texttt{false}, n \rangle$

Data questa codifica, è possibile costruire la alcune semplici funzione di manipolazione e di test come segue:

Definizione 2.2.5 (Successore, predecessore, zero)

- $\texttt{S}(x) \coloneqq \lambda x.\langle \texttt{false}, x \rangle$
- $\texttt{P}(x) \coloneqq \lambda x.x\texttt{false}$
- $\texttt{Zero}(x) \coloneqq \lambda x.x\texttt{true}$

Secondo la codifica data in 2.2.5, il numerale b_n è costituito da una tupla avente nelle prime n posizioni il termine `false` e nella $n+1$-esima posizione il termine I. Alla luce di questo fatto, la funzione successore $S(n)$ è corretta dal momento che costruisce una tupla avente un'occorrenza di `false` in testa ed n in coda. Allo stesso modo, la funzione predecessore è corretta per la riduzione mostrata in 2.8.

$$\begin{aligned} P(b_n) = P(\langle false, b_{n-1} \rangle) = (\lambda x.x\texttt{false})(\langle false, b_{n-1} \rangle) \to_\beta \\ \langle false, b_{n-1} \rangle\texttt{false} = \pi_2 \langle false, b_{n-1} \rangle \to_\beta^* b_{n-1} \end{aligned} \tag{2.8}$$

Nel caso specifico di $P(b_0)$, la codifica è in grado di catturare l'applicazione erronea ed il risultato è `false`, la verifica è la seguente:

$$P(b_0) = P(I) = (\lambda x.x\texttt{false})(I) \to_\beta I\texttt{false} \to_\beta^* false \tag{2.9}$$

[3] Per un'esposizione delle due codifiche dei dati e della loro equivalenza, il lettore può consultare [16].

Similmente alla 2.9, si può facilmente verificare la correttezza del predicato `Zero`. A questo punto, prima di dimostrare formalmente che tutte la classe delle funzioni ricorsive è implementabile in λ-calcolo, riportiamo la definizione formale di tale classe in 2.2.8. Per farlo, però è necessario formulare ancora qualche definizione:

Definizione 2.2.6 (Funzioni iniziali)

- $\forall i, n, 1 \leq i \leq n \rightarrow U_i^n(x_1, \ldots, x_n) \coloneqq x_i$
- $S(n) \coloneqq n + 1$
- $Z(n) \coloneqq 0$
- *Data un qualsiasi predicato P, $\mu m P(m)$ denota il più piccolo valore di m per cui tale funzione vale (se tale valore esiste), altrimenti è indefinito.*

Definizione 2.2.7
Una classe di funzioni $\mathcal{A}$ è chiusa per ricorsione per ricorsione primitiva, se per tutte le ϕ definite come:

$$\phi(0, \vec{n}) \coloneqq \chi(\vec{n})$$
$$\phi(k + 1, \vec{n}) \coloneqq \psi(\phi(k, \vec{n}), k, \vec{n})$$

$\chi, \psi \in \mathcal{A} \rightarrow \phi \in \mathcal{A}$

Definizione 2.2.8 (Funzioni ricorsive)
La classe $\mathcal{R}$ delle funzioni ricorsive è il più piccolo insieme per cui valgono le seguenti proprietà:

- *$\mathcal{R}$ è chiusa per composizione:* $\forall \chi, \phi_1, \ldots \phi_n . \chi \in \mathcal{R} \rightarrow \phi_1 \ldots \phi_n \in \mathcal{R} \rightarrow \chi(\phi_1(\vec{x}), \ldots, \phi_n(\vec{x})) \in \mathcal{R}$
- *$\mathcal{R}$ è chiusa per ricorsione primitiva* (2.2.7).
- *$\mathcal{R}$ è chiusa per minimizzazione:* $\forall \chi \in \mathcal{R}.(\forall \vec{n}. \exists m. \chi(\vec{n}, m) = 0 \rightarrow \mu m[\chi(\vec{n}, m) = 0] \in \mathcal{R})$.

Per dimostrare che la classe $\mathcal{R}$ è codificabile in λ-calcolo, definiamo le funzioni iniziali come in (2.10), poi verifichiamo che le le funzioni λ-definibili siano chiuse per composizione.

$$\begin{aligned} U_i^n &\coloneqq \lambda x_1, \ldots, \lambda x_n . x_i \\ S(x) &\coloneqq \lambda x. \langle \texttt{false}, x \rangle \\ Z(x) &\coloneqq \lambda x. b_0 \end{aligned} \tag{2.10}$$

Lemma 2.2.1 (Chiusura per composizione di Λ)
Λ è chiusa per composizione.

Dimostrazione. Siano $\chi, \psi_1, \dots, \psi_m$ funzioni λ-definite rispettivamente da $G, H_1, \dots, H_m$, allora

$$\phi(\vec{n}) \coloneqq \chi(\psi_1(\vec{n}), \dots, \psi_m(\vec{n}))$$

è λ-definita da F come

$$F \coloneqq \lambda\vec{x}.G(H_1\vec{x}) \dots (H_m\vec{x})$$

A questo punto rimangono da codificare in λ-calcolo la ricorsione ed un costrutto di minimizzazione. Per farlo, tuttavia, è opportuno introdurre in lambda calcolo un operatore di punto fisso, mediante il quale implementare la ricorsione.

Definizione 2.2.9 (Operatore di punto fisso)
Un operatore di punto fisso è una funzione Y che, data una funzione (un funzionale) $F : A \to A$, si verifica la seguente relazione: $YF = F(YF)$.

Una costruzione di un tale operatore è dovuta a Kleene ed è la seguente:

$$Y \coloneqq \lambda g.(\lambda f.g(ff))(\lambda f.g(ff)) \tag{2.11}$$

Per convincerci di come mediante l'operatore di punto fisso sia possibile codificare la ricorsione, facciamo riferimento al seguente esempio:

Esempio 2.2.1. Proviamo a costruire una funzione ricorsiva in λ-calcolo, mediante le costruzioni che abbiamo fornito sinora. Per esempio prendiamo la funzione di somma su naturali definita come segue:

$$Sum(x, y) \coloneqq \lambda x.\lambda y.\ \texttt{if}\ Z(x)\ \texttt{then}\ y\ \texttt{else}\ S(Sum(P(x), y))$$

Come abbiamo visto, tutti i termini presenti in tale definizione sono codificabili in λ-calcolo. Per codificare il comportamento ricorsivo di questa funzione, procediamo come segue:

$$SUM \coloneqq \lambda f.\lambda x.\lambda y.\ \texttt{if}\ Z(x)\ \texttt{then}\ y\ \texttt{else}\ S(f\ P(x)\ y)$$

Anche intuitivamente, se applichiamo l'operatore di punto fisso alla SUM appena definita, calcoliamo la f che è punto fisso della SUM, ossia la funzione che stiamo cercando di codificare ricorsivamente. Ma per convincercene ulteriormente verifichiamo cosa accade con la seguente computazione:

$$\begin{aligned}
(Y\ SUM)b_2\ b_3 \to^*_\beta & SUM\ (Y\ SUM)\ b_2\ b_3 \to^*_\beta \\
& \texttt{if}\ Zero(b_2)\ \texttt{then}\ b_3\ \texttt{else}\ S((Y\ SUM)\ P(b_2)\ b_3) \to^*_\beta \\
& \texttt{if false then}\ b_3\ \texttt{else}\ S((Y\ SUM)\ b_1\ b_3) \to^*_\beta \\
& S((Y\ SUM)\ P(b_1)\ b_3) \to^*_\beta \\
& S(\ \texttt{if}\ Zero(b_1)\ \texttt{then}\ b_3\ \texttt{else}\ S((Y\ SUM)\ P(b_1)\ b_3)) \to^*_\beta \\
& S(\ \texttt{if false then}\ b_3\ \texttt{else}\ S((Y\ SUM)\ b_0\ 3)) \to^*_\beta \\
& S(S((Y\ SUM)\ b_0\ b_3)) \to^*_\beta \\
& S(S\ \texttt{if}\ Zero(b_0)\ \texttt{then}\ b_3\ \texttt{else}\ S((Y\ SUM)\ P(b_0)\ b_3))) \to^*_\beta \\
& S(S\ \texttt{if true then}\ b_3\ \texttt{else}\ S((Y\ SUM)\ P(b_0)\ b_3))) \to^*_\beta \\
& S(S(b_3)) \to^*_\beta b_5
\end{aligned}$$

Similmente a quanto sopra, è possibile generalizzare questa costruzione per codificare la ricorsione primitiva: codifichiamo il *meta*-funzionale di ricorsione primitiva come in (2.12).

$$R \coloneqq \lambda r.\lambda F.\lambda G.\lambda H.\lambda x.\lambda \vec{y}.\ \texttt{if}\ Zero(x)\ \texttt{then}\ G\ \vec{y}\ \texttt{else}\ H\ (F\ (P\ x)\ \vec{y})\ (P\ x)\vec{y} \tag{2.12}$$

Grazie a questa costruzione, è possibile definire il funzionale vero e proprio di ricorsione come YR. Per codificare l'operatore di minimizzazione, possiamo osservare come sia semplice codificarlo ricorsivamente come:

$$M\ G\ \vec{x}\ y \coloneqq \lambda G.\lambda \vec{x}.\lambda y.\texttt{if}\ Zero\ (G\ \vec{x}\ y)\ \texttt{then}\ y\ \texttt{else}\ M\ G\ \vec{x}\ (S\ y)$$

Infine, dal momento che la ricorsione primitiva è codificabile in λ-calcolo mediante la costruzione descritta in (2.12), allora possiamo codificare anche il costruttore di minimizzazine appena descritto. Dunque, avendo dimostrato che la classe delle funzioni λ-definibili gode di tutte le proprietà descritte in 2.2.8, abbiamo dimostrato che le funzioni ricorsive sono λ-calcolabili.

2.3 Due discipline di valutazione in compendio

Dal momento che la nozione di riducibilità fra due termini non è deterministica (poiché non necessariamente, all'intreno di un termine, è presente un solo redesso), può essere utile definire delle discipline di valutazione deterministiche dei λ-termini in modo da costruire una semantica operazionale del λ-caclolo, così da avere una base solida (e deterministica) su cui modellare lo studio della complessità di tale formalismo e mediante la definizione di macchine astratte in grado di valutare i λ-termini.

In effetti, l'obiettivo delle macchine astratte per la valutazione di λ-termini è quello di fornire modelli di calcolo in grado di ridurre tali termini con overhead ragionevole nella lunghezza della riduzione in una specifica disciplina di valutazione. Chiaramente, se non si dispone di una disciplina di valutazione deterministica, non è possibile avere una misura univoca dell'overhead indotto dalla macchina astratta che la imlementa: per esempio, siccome abbiamo visto che alcune normalizzazioni possono avere lunghezza arbitrariamente lunga, senza il ricorso ad una determinata disciplina di valutazione non riusciamo a fissare un modello del costo di questa riduzione.

In letteratura sono state definite ed analizzate numerose discipline di valutazione: le più importanti sono senza dubbio la call-by-name (CbN) e la call-by-value (CbV), per le quali canonicamente si fa riferimento a [18]. In questa sezione ne daremo una sommaria presentazione al fine di agevolare il lettore nella trattazione che seguirà, dove la nozione di disciplina di valutazione sarà di notevole importanza. Per farlo, diamo alcune definizioni.

Definizione 2.3.1 (Disciplina di valutazione). Una disciplina di valutazione è una qualsiasi relazione $R : \Lambda \times \Lambda$ tale che $\forall t.\forall u.R(t, u) \to (t \to_\beta u)$.

Definizione 2.3.2 (Disciplina di valutazione deterministica). Una disciplina di valutazione R è deterministica se e solo se $\forall t, s, s'.R(t, s) \to R(t, s') \to s = s'$.

Definizione 2.3.3 (Disciplina di valutazione *left-to-right*). Una disciplina di valutazione è *left-to-right* se il sottotermine sinistro di un'applicazione viene semplificato prima del destro.

Definizione 2.3.4 (Disciplina di valutazione *right-to-left*). Una disciplina di valutazione è *right-to-left* se il sottotermine destro di un'applicazione viene semplificato prima del sinistro.

Definizione 2.3.5 (Disciplina di valutazione *closed*). Una disciplina di valutazione è *closed* se effettua riduzioni solo fra termini chiusi

Definizione 2.3.6 (Disciplina di valutazione *open*). Una disciplina di valutazione si dice *open* se effettua riduzioni anche qualora coinvolgano termini aperti.

Definizione 2.3.7 (Disciplina di valutazione *strong*). Una disciplina di valutazione si dice *strong* quando ammette la possibilità di effettuare riduzioni sotto λ-astrazioni (e dunque anche su termini aperti).

Definizione 2.3.8 (Disciplina di valutazione *weak*). Una disciplina di valutazione si dice *weak* quando non ammette la possibilità di effettuare riduzioni sotto λ-astrazioni.

Queste definizioni permettono di determinare una sorta di tassonomia delle discipline di valutazione per la quale possiamo intuire che le discipline di valutazione che per loro natura sono più semplici sono quelle weak e closed; nondimeno queste discipline di valutazione sono anche le più diffuse nei più comuni linguaggi di programmazione, dal momento che molto di rado a tempo di esecuzione avvengono semplificazioni all'interno del corpo di funzioni e che tutti gli encoding dei dati che abbiamo definito per il λ-calcolo sono termini chiusi, dunque, nel contesto della programmazione funzionale non dovrebbe essere necessario effettuare riduzioni fra termini aperti o sotto astrazioni.

Tuttavia, nel contesto dello sviluppo di dimostratori interattivi, è necessario implementare discipline di valutazione che siano open e strong, per cui è interessante anche studiare tali discipline di valutazione.

Talvolta, la definizione di discipline di valutazione può avvenire per mezzo di contesti di valutazione. Tale pratica è una buona norma poiché permette la rappresentazione più concisa delle discipline di valutazione. Nondimeno, nel contesto della progettazione di implementazioni del lambda calcolo, essa permette di definire una relazione di mapping fra contesti specifici all'interno di determinate strutture dati della macchina stratta (come ad esempio stack o dump) e contesti di una specifica disciplina di valutazione per dimostrare con più semplicità che la macchina astratta in questione realizza una specifica disciplina di valutazione.

Definizione 2.3.9 (Contesto di valutazione, evaluaton context). Un contesto di valutazione è un termine di Λ, all'interno del quale è stata inserita una speciale costante: $\langle\cdot\rangle$, detta "hole".

Di norma i contesti si costruiscono sulla base di specifiche grammatiche del lambda calcolo arricchendo tali grammatiche con una simbolo non terminale finalizzato a descrivere le possibili posizioni che la costante $\langle \cdot \rangle$ può assumere all'interno del termine.

Definizione 2.3.10 (Plugging). L'operazione di plugging di un termine t all'interno di un contesto C, di norma espressa con $C\langle t \rangle$, consiste con il rimpiazzamento non capture-avoiding della costante hole con t.

Come vedremo, non tutte le discipline di valutazione sono equivalenti in merito all'effettiva normalizzazione dei termini su cui operano, in particolare, non sempre sono in grado di identificare la forma normale di un termine, se esiste. Per catturare meglio questo concetto possono tornare utili le seguenti definizioni:

Definizione 2.3.11 (Termine fortemente normalizzante). Diciamo che un termine t è fortemente normalizzante se non esiste una successione di termini t_n con $n \in \mathbb{N}$ tale che $t_0 = t \wedge \forall i t_i \rightarrow_\beta t_{i+1}$.

Definizione 2.3.12 (Disciplina di valutazione completa). Diciamo che una disciplina di valutazione R è completa se esiste una successione t_n con $n \in \mathbb{N}$ tale che $\exists u.(t \rightarrow^*_\beta u) \rightarrow \forall i.R(t_i, t_{i+1}) \wedge \exists m.t_m = u$.

2.3.1 Plotkin's call-by-value

La disciplina di valutazione call-by-value definita da Plotkin prevede che la regola $\rightarrow_\beta$ sia applicata solo qualora l'argomento dell'applicazione sia un valore. Nel caso specifico di [18], dove la disciplina di valutazione è closed, i valori coincidono con le astrazini chiuse, mentre nel caso aperto, che non tratteremo, un valore può anche essere una generica variabile. Per rendere più semplice questa distinzione, quando si tratta di discipline di valutazione CbV, si è soliti arricchire la grammatica dei λ-termini con una sotto-produzione specifica distinguere le applicazioni dal resto dei termini. Il risultato è il seguente:

$$t \coloneqq t_1 t_2 \mid v$$
$$v \coloneqq x \mid \lambda x.t$$

Nella sua formulazione canonica, tale disciplina è definita solo fra i termini in $\Lambda^0 \coloneqq \{x \in \Lambda | FV(x) = \emptyset\}$, vedremo in seguito che se si prova ad estenderla a termini generici di Λ, tale disciplina di valutazione non sarà più normalizzante. Data la definizione (2.3.1), possiamo definire la disciplina di riduzione call-by-value $\rightarrow_p$ (dove la p a pedice fa riferimento a "Plotkin's call-by-value") come la più piccola relazione su Λ^0 per cui valgono regole simili alle seguenti[4]:

[4] L'aggettivo "simili" allude al fatto che la disciplina di valutazione call-by-value, possa essere definita sia right-to-left che left-to-right. Nello specifico, la definizione che ne daremo sarà right to left per analogia con quella adottata in [5].

$$\frac{}{(\lambda x.M)v \to M\{x \leftarrow v\}}\ \beta_v$$
$$\frac{M \to M'}{tM \to tM'}\ a_r \quad \frac{M \to M'}{Mv \to M'v}\ a_l \tag{2.13}$$

Una formulazione equivalente di questa relazione può essere espressa per mezzo della costruzione di contesti: definiamo i right-v evaluation contexts come segue:

$$R \coloneqq \langle\cdot\rangle \mid tR \mid Rv$$
$$\frac{}{(\lambda x.M)v \to M\{x \leftarrow v\}}\ \beta_v$$
$$\frac{M \to M'}{R[M] \to r[M']}$$

Chiaramente, data la grammatica dei right-v evaluation context, la seconda rappresentazione è equivalente alla prima. In merito al determinismo di questa disciplina di valutazione, possiamo dimostrare il seguente risultato:

Lemma 2.3.1 (Determinismo della disciplina $\to_p$)
Per ogni termine $t \in \Lambda^0$ è definita al più una riduzione secondo $\to_p$.

Dimostrazione. Per induzione sulla struttura del termine t:

- $t = v \in \Lambda^0$, le regole di 2.13 riducono solo applicazioni, dunque su tale termine non sono applicabili riduzioni.

- $t = t_1 t_2$, possiamo supporre senza mancare di generalità che il termine di destra non sia un valore (se lo fosse non sarebbe ulteriormente riducibile), l'unica regola che fa match con questa costruzione è a_r. Per ipotesi induttiva il sottotermine di destra riduce al più in un modo, dunque anche $t_1 t_2$ riduce solo in un modo.

- $t = t_1 v$, potrebbero fare match sia la regola a_l che a_r, ma a_r non sarebbe in grado di ridurre, mentre similmente a sopra a_l potrebbe generare al più una riduzione

- $t = v_1 v_2$, le regole a_l e a_r non sarebbero in grado di normalizzare i sottotermini, l'unica regola applicabile è β_v.

Per quanto riguarda la normalizzazione, purtroppo, osserviamo che questa disciplina di valutazione non è completa indipendentemente dal fatto che il termine di partenza sia o meno fortemente normalizzante. Un semplice motivazione di questo fatto può essere identificata nel fatto che tale disciplina è *weak*, dunque, non effettuando riduzioni sotto astrazioni, non è in grado di semplificare tutti i redessi che si trovano nei corpi delle funzioni. Tuttavia, anche qualora non siano presenti redessi sotto astrazioni, tale disciplina di valutazione può non essere in grado di normalizzare i termini, per convincercene forniamo gli esempi che seguono:

Esempio 2.3.1. Supponiamo di voler valutare il termine chiuso: $(\lambda x.I)(\omega\omega)$. L'applicazione della disciplina $\to_p$ darà il seguente risultato:

$$\dfrac{\dfrac{}{\omega\omega \to_p \omega\omega}\ \beta_v}{(\lambda x.I)(\omega\omega) \to_p (\lambda x.I)(\omega\omega)}\ a_l$$

Ciò che otteniamo è una sequenza indefinitamente lunga di riduzioni di $(\lambda x.I)(\omega\omega)$ in se stesso. Per questa ragione la disciplina $\to_p$ non è completa, dal momento che si può verificare che $(\lambda x.I)(\omega\omega) \to_\beta^* I$.

Esempio 2.3.2. In questo caso vogliamo dimostrare che la disciplina di valutazione $\to_p$ non scala naturalmente ai termini aperti. Per farlo, parafrasiamo un esempio tratto da [4].

$$\dfrac{\overline{xx \to_p}}{(\lambda y.Iy)(xx) \to_p}\ a_l$$

Come evidenziato dall'albero d'inferenza di cui sopra, la disciplina di valutazione $\to_p$ non è in grado di ridurre efficacemente il termine aperto $(\lambda y.Iy)(xx)$ nonostante questo abbia una forma normale xx.

Quest'anomalia è dovuta al fatto che l'ipotesi con cui Plotkin ha definito la disciplina di valutazione $\to_p$ identifichi una condizione di armonia fra sintassi e normalizzazione (riportata in 2.3.2) troppo stringente e che non scala a termini aperti.

Lemma 2.3.2 (Armonia di Plotkin)
Un termine è normale se e solo se è un valore chiuso

La nozione di armonia, che determina la correttezza della disciplina di valutazione $\to_p$, non scala al caso di termini aperti. Per ovviare a questa mancanza, gli autori di [4], hanno esteso la disciplina CbV di Plotkin, definendo una disciplina call-by-value adeguata anche per i termini aperti. L'interesse specifico per tale strategia di valutazione, detta del "Fireball calculus", è dovuto al fatto che una delle crumbling abstract machines che sono state oggetto della formalizzazione che presenteremo in seguito, l'Open Crumble GLAM, implementa tale disciplina di valutazione.

2.3.2 Plotkin's call-by-name

La definizione di Plotkin della disciplina call-by-name, se confrontata con la disciplina call-by-value, è più semplice: essa è definita su un numero minore di regole e non prevede che per lanciare una β-riduzione l'argomento assuma forme particolari. Come per il caso della disciplina CbV, possiamo definire la disciplina di valutazione call-by-name $\to_n$ come la più piccola relazione di Λ per cui valgono le seguenti regole:

$$\frac{}{(\lambda x.M)N \to M\{x \leftarrow N\}}\ \beta \qquad \frac{M \to M'}{MN \to M'N}\ a_l \tag{2.14}$$

Ancora una volta è possibile esprimere questa regola facendo ricorso a contesti di valutazione come segue:

$$C \coloneqq \ \langle \cdot \rangle \mid Ct$$

$$\frac{}{(\lambda x.M)N \to M\{x \leftarrow N\}}\ \beta \qquad \frac{M \to M'}{C\langle M \rangle \to C\langle M \rangle}$$

Per quanto riguarda il determinismo di questa disciplina di valutazione possiamo procedere come segue:

Lemma 2.3.3 (Determinismo della disciplina $\to_p$)
Per ogni termine $t \in \Lambda$ è definita al più una riduzione si $\to_n$.

Dimostrazione. Per induzione sulla struttura del termine t:

- $t = x$, le regole di 2.14 riducono solo applicazioni, dunque su tale termine non sono applicabili riduzioni.

- $t = \lambda x.t$, le regole di 2.14 riducono solo applicazioni, dunque su tale termine non sono applicabili riduzioni.

- $t = t_1 t_2$, in questo caso non possono essere applicate entrambe le regole a_l e β, dal momento che se il temine di sinistra è un'astrazione, allora non è ulteriormente riducibile. Al contrario, se il termine di sinisara è un'applicazione, la regola β non può essere applicata.

Ancora una volta, dal momento che questa disciplina di valutazione non semplifica sotto astrazione possiamo dire che non sia completa. Tuttavia, possiamo osservare che per termini in cui la disciplina di valutazione CbV fallisce, la disciplina CbN è in grado di trovare la forma normale; prendiamo i seguenti esempi:

Esempio 2.3.3. Supponiamo di voler valutare il termine: $(\lambda x.I)(\omega\omega)$. L'applicazione della disciplina $\to_v$ lo riscriverà direttamente in I, riuscendo a normalizzarlo in un solo passo.

La ragione per cui su certi termini in cui la disciplina CbV fallisce, la disciplina CbN converge, è dovuto al fatto che quest'ultima risuce gli argomenti di una funzione solo se questi compaiono nel sottotermine sinistro di un'applicazione, mentre la disciplina CbV prova a ridurli indipendentemente dal fatto che se ne faccia uso o meno all'interno del corpo della funzione.

Esempio 2.3.4. Anche nel caso della riduzione di termini aperti, il comportamento della disciplina CbN è migliore rispetto a quello della disciplina CbV: siccome non sono richieste ipotesi, la valutazione delle astrazioni non si blocca qualora il termine non sia un valore. Prendiamo il caso dell'esempio 2.3.2, la riduzione avviene come segue:

$$(\lambda y.Iy)(xx) \rightarrow_n I(xx) \rightarrow_n (xx)$$

Nonostante la disciplina CbN non sia completa, una sua estensione *strong*, quella del "Normal order", è stata dimostrata essere completa da Curry e Feys in [10]. Il fatto di introdurre la possibilità di ridurre sotto le astrazioni, rende necessaria l'introduzione di un ordine specifico con cui operare le riduzioni, nello specifico tale disciplina prevede che vengano valutati i termini da sinistra a destra dal più esterno al più interno, per questa ragione è anche detta left-outermost reduction.

2.4 A posteriori

A questo punto della trattazione possiamo ridefinire più formalmente alcune definizioni che avevamo fornito all'inizio della presente sezione.

Definizione 2.4.1 (Disciplina di valutazione *left-to-right*). Una disciplina di valutazione $\rightarrow$ è *left-to-right* se vale:

$$\frac{M \rightarrow M'}{MN \rightarrow M'N}$$

Ma non:

$$\frac{N \rightarrow N'}{MN \rightarrow MN'}$$

Definizione 2.4.2 (Disciplina di valutazione *right-to-left*). Una disciplina di valutazione $\rightarrow$ è *right-to-left* se vale:

$$\frac{N \rightarrow N'}{MN \rightarrow MN'}$$

Ma non:

$$\frac{M \rightarrow M'}{MN \rightarrow M'N}$$

Definizione 2.4.3 (Disciplina di valutazione *strong*). Una disciplina di valutazione si dice *strong* se vale:

$$\frac{M \rightarrow_\beta M'}{\lambda x.M \rightarrow_\beta \lambda x.M'}$$

Definizione 2.4.4 (Disciplina di valutazione *weak*). Una disciplina di valutazione si dice *weak* se non vale:

$$\frac{M \rightarrow_\beta M'}{\lambda x.M \rightarrow_\beta \lambda x.M'}$$

Infine, puntualizziamo solo un ultimo aspetto: dal momento che la disciplina call-by-name sembra essere *più completa* della disciplina call-by-value, potrebbe essere lecito chieresi per quale ragione, la disciplina di valutazione CbV sia in molti contesti preferita alla CbN: la ragione è che, mediamente, la disciplina CbV richiede meno riduzioni per valutare il termine, per esempio prendiamo il seguente caso:

$$(\lambda x.xxx)(II) \to_p (\lambda x.xxx)I \to_p III \to_p II \to_p I$$

$$(\lambda x.xxx)(II) \to_n ((II)(II))(II) \to_n (I(II))(II) \to_n (II)(II) \to_n I(II) \to_n II \to_n I$$

Anche da questo semplice esempio, è semplice intuire che il motivo di questo fenomeno è dovuto al fatto che la disciplina CbV riduce completamente gli argomenti prima di usarli all'interno del corpo delle funzioni. Mediamente dunque, qualora ci si aspetti che un medesimo argomento compaia meno di una volta all'interno del termine da ridurre, la disciplina CbN sarà preferibile, poiché non riduce inutilmente i termini, mentre qualora ci si aspetti che i termini compaiano più di una volta, la disciplina di valutazione call-by-value darà risultati migliori .

Capitolo 3

Crumbling

3.1 Complessità del λ-calcolo

Il λ-calcolo, come si è visto, è stato definito ad un livello molto elevato. Se da un lato ciò permette di analizzare agevolmente l'espressività di tale formalismo e, data la sua completezza, del calcolo in generale, dall'altro rende pressoché inoperabili studi di complessità. Questo fenomeno è dovuto al fatto che, mentre in formalismi di calcolo di più basso livello, come per esempio RAM o MdT la presenza di una macchina teorica di calcolo garantisce l'esistenza di un insieme di operazioni atomiche di complessità costante per via delle quali studiare la complessità di una computazione coincide con contare il numero di operazioni atomiche. Nel lambda calcolo l'operazione di riduzione difficilmente può essere considerata di complessità costante, per cui lo studio della complessità di una computazione deve avvenire diversamente. L'operazione di riduzione più importante del calcolo è, come abbiamo visto, quella di β-riduzione; in un contesto di trattazione teorica del calcolo essa viene intesa come la *riscrittura* di un termine al posto di tutte le occorrenze legate di una variabile all'interno di un termine[1]. Formalmente:

$$(\lambda x.t)u \coloneqq t\{x \leftarrow u\}$$

Come si può immaginare, la complessità di tale riscrittura difficilmente può essere considerata costante, o quantomeno limitata, come mostrato dall'esempio 3.1.1.

[1] Per una trattazione più dettagliata del λ-calcolo, fare riferimento a 2

Esempio 3.1.1. Si prendano come esempio le seguenti riduzioni:

$$(\lambda x.x)y = y \tag{3.1}$$
$$(\lambda x.x)t_1 = t_1 \tag{3.2}$$
$$(\lambda x.y)x = y \tag{3.3}$$
$$(\lambda x.y)t_1 = y \tag{3.4}$$
$$(\lambda x.xx)t_1 = t_1 t_1 \tag{3.5}$$

Già da questi pochi esempi è possibile notare come la complessità di una singola operazione di riduzione, se interpretata in maniera naïve come riscrittura, possa essere disomogenea in base alla forma dei termini da ridurre: ad esempio la 3.1 in linea di principio dovrebbe avere complessità minore o uguale della 3.2 dal momento che il termine t_1 non può essere più piccolo di una variabile.

L'analisi si complica ulteriormente se andiamo a considerare casi di riduzioni quali la 3.3 o la 3.4: in questi casi, dove la sostituzione di fatto non avviene, sarebbe opportuno poter disporre di modelli di calcolo in cui la complessità di entrambe le riduzioni sia costante. Infine, la presenza di riduzioni che fanno crescere la taglie del termine mette in dubbio anche l'ipotesi che la taglia iniziale di un termine possa limitare superiormente la complessità di una singola operazione di β-riduzione.

Ulteriormente alle ragioni esposte sopra, è anche stata verificata (si prenda come riferimento [1]) la presenza di fenomeni di esplosione della taglia che sono indipendenti dalla strategia di valutazione adottata e che dunque, in un'implementazione naïve del lambda calcolo sono ineluttabili. Per queste ragioni, la letteratura ha sempre considerato il λ-calcolo un formalismo poco adatto agli studi di complessità. Solo di recente sono stati prodotti risultati interessanti, che riguardano lo sviluppo di implementazioni del lambda calcolo la cui complessità è polinomiale, bilineare, oppure lineare nel numero di β-riduzioni necessarie alla normalizzazione del termine iniziale. Per una rapida rassegna di alcune di queste implementazioni si prenda sempre come riferimento [1], oppure [13] o [15].

Uno degli ultimi sviluppi in questo ambito di ricerca è quello delle crumbling abstract machines: implementazioni *ragionevoli*[2] del lambda calcolo che permettono di ridurre un termine con una complessità bilineare nel numero di β-riduzioni e nella taglia iniziale del termine, minimizzando il numero di strutture dati necessarie alla valutazione. Inoltre, tali macchine sono caratterizzate da un'elevata scalabilità: ad esempio la medesima architettura definita per la disciplina call-by-value può essere utilizzata senza necessità di cambi architetturali per la valutazione di termini aperti. Un'altra delle peculiarità di queste macchine è che, al contrario di altre implementazioni del λ-calcolo per mezzo di ES, la valutazione di crumbled forms non causa overhead asintotici dovuti alle ricerca di redessi (si confronti [17]).

[2]Nella presente trattazione, il termine "ragionevole" assume la stessa connotazione del termine "reasonable" definito in [1], ossia se la complessità di valutazione è polinomiale nel numero di β-riduzioni e nella taglia iniziale del termine.

3.2 Crumbling Abstract Machines

Il lavoro introduttivo per la definizione di Crumbling Abstract Machines è stato raccolto e formalizzato in [5]. In tale articolo viene presentata la rappresentazione di λ-termini per mezzo di crumbled forms nonché la definizione di una macchina astratta, detta Crumble GLAM, che implementa una disciplina di calcolo closed weak call-by-value studiandone la complessità, la correttezza e definendo estensioni che implementano discipline di valutazione weak open call-by-value e gettando le basi per lo studio di una crumbling abstract machine strong call-by-value e call-by-need.

Per questi motivi è interessante intraprendere una formalizzazione dei tipi, delle funzioni e dei teoremi definiti e dimostrati in [5] in modo da assicurare la correttezza dei risultati teorici su cui si basa lo studio di tutta questa famiglia di macchine astratte. Questo corpus di dimostrazioni formali è stato raccolto e pubblicato in [14]. Presentiamo ora un rapido sommario informale di come funziona il processo di crumbling in modo da contestualizzare quella che altrimenti sarebbe una trattazione estremamente tecnica e fine a se stessa.

Sharing

Il crumbling prevede che la sintassi del λ-calcolo si arricchita da un costrutto di explicit sharing (anche detto "explicit substitution" ed abbreviato con la sigla ES). Un tale costrutto è, ad esempio il `let` x `=` u `in` t comune nella maggior parte dei linguaggi funzionali, la cui semantica è quella di rappresentare con la variabile x il termine u all'interno di t. Per ragioni di brevità, esso può essere espresso nella forma $t[x \leftarrow u]$.

L'utilità pratica di un simile costrutto è dovuta al fatto che esso rende possibile decomporre la β-riduzione in due sotto-operazioni come mostrato dalla 3.6, svincolando l'operazione di β-riduzione da quella di sostituzione su termini.

$$(\lambda x.t)u \rightarrow_{\beta_{ES}} t[x \leftarrow u] \rightarrow_{ES} t\{x \leftarrow u\} \tag{3.6}$$

In questo modo è possibile ritardare l'operazione di sostituzione a piacere evitando fenomeni di esplosione della taglia del termine, così da rendere possibile la costruzione di implementazioni ragionevoli del λ-calcolo.

Crumbled forms

Una volta aggiunto il costrutto di sharing al lambda calcolo è possibile, dato un termine qualsiasi t, costruire la crumbled form $\underline{t}$: un termine ottenuto da t mediante la sostituzione di applicazioni annidate con costrutti di explicit sharing. Questa operazione prende il nome di crumbling e verrà analizzata nel dettaglio più sotto in 3.2 e più avanti in un'apposita sezione: 6.2; per la trattazione corrente ci limitiamo a fornire solo un esempio informale dell'esito del crumbling sul termine $((\lambda x.x(xx))y)((\lambda z.z)y)$ dimodoché il lettore possa formarsi un'idea di come procede tale operazione:

$$\underline{((\lambda x.x(xx))y)((\lambda z.z)y)} = w'w[w' \leftarrow (\lambda x.(xx')[x' \leftarrow xx])y][w \leftarrow (\lambda z.z)y]$$

Si noti che i costrutti di explicit sharing possono essere introdotti anche all'interno dei corpi di funzioni ($((\lambda x.x(xx))y)$ si riscrive in $(\lambda x.(xx')[x' \leftarrow xx])y)$ e che, a meno di astrazioni, gli ES sono appiattiti in una medesima lista. La forma di questi λ-termini è detta *crumbled form*, dove la nomenclatura "crumbled" vuole alludere al fatto che i termini risultino spezzati, "sbriciolati" per mezzo di costrutti di ES. La valutazione delle crumbled forms prevede che queste vengano trasformate per mezzo di specifiche regole di riduzione che fanno uso di funzioni di manipolazione fino al raggiungimento di una forma normale, la quale verrà ricomposta nel termine finale rimuovendo i costrutti di ES.

Nella sua forma non tipata, il crumbling coincide sostanzialmente con la rappresentazione dei λ-termini per mezzo di ANF descritta in [15]. La complessità della valutazione di termini espressi secondo tale rappresentazione è stata dimostrata essere soggetta ad anomalie da Kennedy in [17]. I casi patologici presi in esame sono dovuti alla *necessità* di definire regole di commutazione per la valutazione di costrutti espliciti di match o if-then-else: una di tali anomalie è il potenziale overhead quadratico indotto dalla necessità di concatenare liste di ES, mentre altre anomalie riguardano per esempio possibili problemi di tipaggio e la necessità di replicare costrutti di sharing. Fortunatamente però, come vedremo, la restrizione a forme crumbled non necessita di regole di commutazione e dunque non dà luogo a tali anomalie.

Crumbled environments

I costrutti di explicit-sharing sono raggruppati in liste dette Crumbling Environments che aggregano quanti più costrutti di ES possibile all'esterno della medesima astrazione. In questo modo i crumbled environments giocano il duplice ruolo di codificare l'ambiente di valutazione dei termini e, dato l'ordinamento loro indotto in fase di costruzione della crumbled form, di codificare il contesto di valutazione del termine: infatti molte implementazioni del lambda calcolo hanno bisogno di strutture dati ausiliarie, dette *stack* o *dump*, in cui vengono inseriti sequenzialmente quelli che, nel caso del crumbling, sono gli argomenti delle applicazioni annidate, prima di essere spostati nell'environment. In tal modo, l'ordine con cui tali argomenti vengono inseriti nello *stack* determina l'ordine in cui i termini devono essere valutati.

Come vedremo in seguito quando verrà presentata l'operazione di crumbling in 3.7, la natura sequenziale degli environment permette di codificare già a tempo di compilazione (i.e. *crumbling*) il contesto di valutazione dei termini, per cui sarà sufficiente percorrere sequenzialmente i crumbled environment per identificare i redessi. Per questo motivo, dunque, sarà necessario che i crumbled environment siano implementati in maniera efficiente, in modo che idealmente sia possibile accedere ad ogni ES sia sequenzialmente per identificare i redessi ed applicare la disciplina di valutazione, che in maniera random per risolvere nomi di variabili, informalmente: la crumbled form $(x, e[x \leftarrow t]e')$ si deve ridurre atomicamente a $(t, e[x \leftarrow t]e')$.

Per questo motivo, similmente a quanto avviene di norma in letteratura quando si ha a che fare con macchine che implementano il λ-calcolo per mez-

$$
\begin{aligned}
t &\coloneqq t_1 t_2 \mid v \mid \text{if } t \text{ then } u \text{ else } t \\
v &\coloneqq x \mid v_{\neg x} \\
v_{\neg x} &\coloneqq \lambda x.t \mid \text{true} \mid \text{false} \mid \text{err} \\
R &\coloneqq \langle\cdot\rangle \mid tR \mid Rv \mid \text{if } R \text{ then } u \text{ else } t
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x.t)v &\mapsto_{\beta_v} t\{x \leftarrow v\} \\
\text{if true then } t \text{ else } s &\mapsto_{ift} t \\
\text{if false then } t \text{ else } s &\mapsto_{iff} s \\
\text{if } t \text{ then } u \text{ else } s &\mapsto_{ife} \text{err} \quad \text{if } t = \lambda x.u \vee t = \text{err} \\
tu &\mapsto_{@e} \text{err} \quad \text{if } t \in true, false, err
\end{aligned}
$$

$$
\begin{aligned}
R\langle t\rangle \mapsto_a R\langle t'\rangle &\quad \text{if } t \mapsto_a t' \text{ for } a \in \{\beta_v, ift, iff, if, @e\} \\
\to_{pif} \coloneqq &\quad \mapsto_{\beta_v} \cup \mapsto_{ift} \cup \mapsto_{iff} \cup \mapsto_{ife} \cup \mapsto_{ife}
\end{aligned}
$$

Figura 3.1: Sintassi e regole del λ_{Pif}-calculus

zo di ambienti globali[3], i nomi di variabile in [5] sono rappresentati per mezzo di indirizzi di memoria così da permettere la risoluzione di tali nomi in tempo costante. Inoltre in [5] viene presentata un'implementazione in OCaml dei crumbled environmets per cui la concatenazione di due environment sia eseguibile in tempo costante, così da evitare l'overhead quadratico per le ANF teorizzato da Kennedy in [17] dovuto anche all'ipotesi che il costo della concatenazione fosse lineare.

Pif-calculus

Per smentire le altre anomalie supposte da Kennedy in [17] per quanto riguarda la valutazione di ANF con costrutti espliciti di matching e `if-then-else`, gli autori di [5] definiscono una nozione di calcolo detta Pif-calculus che consiste sostanzialmente in quella introdotta da Plotikin [18], ma arricchita dal costrutto esplicito di `if-then-else` e dal termine `err` come mostrato in 3.1. In questo modo, dimostrando che la complessità di valutazione di λ_{Pif}-termini è bilineare nonostante la presenza esplicita del costrutto di `if-then-else`, si dimostra che le anomalie ipotizzate da Kennedy in [17] non si verificano per le crumbling abstract machines. Dal momento che il costrutto di `if-then-else esplicito` (come quello di matching) non aumenta il potere espressivo del calcolo e che esso è introdotto della grammatica del λ_p-calcolo solo per ribadire il fatto che il crumbling non soffre dell'anomalia di Kennedy, il sottoinsieme del calcolo che abbiamo deciso di formalizzare e di analizzare in questa trattazione è solamente

[3] Un'alternativa agli ambienti globali sarebbe quella degli ambienti locali. Nelle macchine che implementano il λ-calcolo in questo modo, ad ogni termine è associato un ambiente e la risoluzione di un nome di variabile coinciderebbe con il look-up ricorsivo di tale nome negli ambienti annidati attorno ad essa. Si dimostra in [2] che, allo stato dell'arte attuale, dal punto di vista della complessità le implementazioni del calcolo con ambienti locali e globali sono asintoticamente equivalenti.

$$
\begin{aligned}
t &\coloneqq t_1 t_2 \mid v \mid \text{if } t \text{ then } u \text{ else } t \\
v &\coloneqq x \mid v_{\neg x} \\
v_{\neg x} &\coloneqq \lambda x.t \\
R &\coloneqq \langle \cdot \rangle \mid tR \mid Rv
\end{aligned}
$$

$$
(\lambda x.t)v \mapsto_{\beta_v} t\{x \leftarrow v\}
$$

$$
\begin{aligned}
R\langle t \rangle \mapsto_{\beta_v} R\langle t' \rangle \quad & \text{if } t \mapsto_{\beta_v} t' \\
\rightarrow_p \coloneqq \quad & \mapsto_{\beta_v}
\end{aligned}
$$

Figura 3.2: Sintassi e regole del λ_p-calculus

quella del tradizionale λ_{Plot}-calcolo (anche detto λ_p-calcolo) riassunto in 3.2. In 3.1 e 3.2, al fine di rappresentare più concisamente la disciplina di valutazione call-by-value, è stata definita la grammatica dei right-v-evaluation contexts per mezzo del non terminale R. Dalla grammatica con testa R, si osserva che in un termine il contesto ha sempre al più una costante $\langle t \rangle$, dunque la regola $R\langle t \rangle \mapsto_{\beta_v} R\langle t' \rangle$ identifica sempre una ed una sola riduzione all'interno del contesto. Tale regola, infine, descrive la disciplina di valutazione right-to-left CbV nel senso che, data la BNF dei right-v-evaluation contexts, in un'applicazione si riduce prima il termine di destra e si effettua una β-riduzione solo se l'argomento è un value.

Questa nozione è utile perché, come vedremo in 6.3.4, dimostrando che in ogni momento della valutazione le crumbled form della macchina astratta con disciplina CbV (denominata Crumble GLAM) si decodificano in right-v evaluation contexts, saremo in grado di dimostrare che le regole di valutazione di tale macchina implementano la disciplina $\rightarrow_p$. Allo stesso modo, quando verrà presentata la Open Crumble GLAM per la disciplina di valutazione open CbV, sarà possibile dimostrare che questa implementa correttamente la open CbV dimostrando che le crumbled form generate dalle regole di transizione della macchina si decodificano sempre in right-f contexts.

Crumbling e crumbles, informalmente

Come anticipato, il crumbling è l'elemento comune ad una famiglia di macchine astratte in grado di implementare diverse discipline di valutazione del λ-calcolo. Da un punto di vista tecnico, queste macchine differiscono le une dalle altre semplicemente per l'insieme di regole di valutazione adottate, mentre da un punto di vista teorico differiscono solamente per gli invarianti che ne determinano la correttezza e la complessità. Hanno dunque in comune le medesime strutture dati e l'impianto teorico che ne garantisce l'efficacia: il teorema d'implementazione. Per questo motivo, presenteremo ora la più semplice delle crumbling-abstract-machines: la macchina Crumble GLAM che implementa una disciplina di valutazione closed CbV di λ_p-termini ed istanzieremo in segui-

to su di essa il teorema di implementazione, lasciando [5] come riferimento al lettore interessato a questi aspetti di scalabilità della macchina Crumble GLAM a diverse discipline di valutazione.

Il processo di crumbling sul λ_p-calcolo mappa λ-termini in crumbled forms; prima di formalizzare il processo di crumbling, forniamo una grammatica di tali forme per mezzo della seguente BNF:

$$\begin{aligned} c &\coloneqq (b, e) \\ b &\coloneqq v_1 v_2 \mid v \\ v &\coloneqq x \mid \lambda x.c \\ e &\coloneqq \epsilon \mid es \\ s &\coloneqq [x \leftarrow b] \end{aligned}$$

Già a questo livello è possibile intravedere una certa somiglianza fra la grammatica dei λ_p-termini e quella delle crumbled form: infatti se si pensa di fare l'inlining della produzione $v_{\neg x}$ all'interno della v in 3.2 e se si astrae la BNF appena fornita dall'aggiunta dei crumbled environments, è facile desumere come i bites b non siano altro che i termini t del λ_p-calcolo e che le due v, che rappresentano i valori, siano in fin de conti equivalenti.

Detto ciò, il processo di crumbling procede in maniera simile a quanto avevamo già mostrato alcuni paragrafi più su: le applicazioni annidate vengono spezzate da costrutti di ES, separati solo qualora si incontri un'astrazione; una definizione un po' più formale di tale funzione, che riprenderemo più avanti, può essere data come segue[4]:

$$\begin{aligned} \overline{x} &\coloneqq x \\ \overline{\lambda x.t} &\coloneqq \lambda x.\underline{t} \\ \underline{v} &\coloneqq (\overline{v}, \epsilon) \\ \underline{vw} &\coloneqq (\overline{v}\overline{w}, \epsilon) \\ \underline{tv} &\coloneqq (x\overline{v},\ [x \leftarrow b]) \quad (*) \\ \underline{ut} &\coloneqq \underline{ux} \mathbin{@} ([x \leftarrow b]e) \quad (*) \end{aligned} \tag{3.7}$$

$(*)$ Dove t non è un valore e $\underline{t} = (b, e)$ e x è fresca

Dualmente al processo di crumbling è definito un processo di *de-crumbling*, in grado di mappare crumbled form a λ_p-termini. Come abbiamo anticipato, questa operazione in [5] viene definita "read-back" ed è espressa con il simbolo $\cdot_\downarrow$.

$$\begin{aligned} x_\downarrow &\coloneqq x \\ (b, \epsilon)_\downarrow &\coloneqq b_\downarrow \\ (b, e[x \leftarrow b'])_\downarrow &\coloneqq (b, e)_\downarrow\{x \leftarrow b_\downarrow\} \\ (\lambda x.c)_\downarrow &\coloneqq \lambda x.c_\downarrow \\ (vw)_\downarrow &\coloneqq (v_\downarrow w_\downarrow) \end{aligned} \tag{3.8}$$

[4]Potrebbe risultare poco chiaro: la notazione @ che di norma denota la concatenazione di due liste viene usata per indicare la trasformazione $(b, e)@e' \coloneqq (b, ee')$.

Solo un appunto per quanto riguarda la 3.8: la notazione $\{x \leftarrow t\}$ indica la sostituzione del termine t a tutte le occorrenze libere della variabile x nel termine a cui tale espressione è giustapposta.

Similmente a quanto accade per i λ_p-termini, anche per le crumbled forms sono definiti contesti di valutazione come segue:

$$\begin{aligned} C &\coloneqq \langle\cdot\rangle \mid (b, E) \\ E &\coloneqq e[x \leftarrow \langle\cdot\rangle] \end{aligned} \tag{3.9}$$

Sui crumbled context sono definite le medesime operazioni di plugging che sono definite per i contesti su termini come mostrato in 3.10, inoltre è estesa la definizione di read-back anche ai crumbled-context come mostrato in 3.11.

$$(b, e[x \leftarrow \langle\cdot\langle])\langle(b', e') \coloneqq (b, e[x \leftarrow b']e') \tag{3.10}$$

$$\begin{aligned} \langle\cdot\rangle_{\downarrow} &\coloneqq \langle\cdot\rangle \\ (b, e[x \leftarrow \langle\cdot\rangle)_{\downarrow} &\coloneqq (b, e)\{x \leftarrow \langle\cdot\rangle\} \end{aligned} \tag{3.11}$$

L'utilità teorica dei crumbled context è dettata dal fatto che per mezzo di essi sia possibile dimostrare che le regole di riduzione su crumbled forms adottate dalla macchina astratta implementano specifiche discipline di valutazione: infatti, un importante invariante della Crumble GLAM sarà, come vedremo, quella del contextual-decoding che asserisce che comunque si suddivida una crumbled form in un contesto C in cui venga fatto plugging di un crumble c, il crumbled-context si ritraduce sempre in un right-v evaluation context. Questa condizione durante la valutazione può venire talvolta violata, ma è possibile dimostrarne una formulazione più debole (detta *weak* contextual decoding) la quale sostanzialmente asserisce che modulo applicazione di regole *tecniche* di transizione, vale il contextual decoding per tutte le crumbled-form intermedie. Chiaramente, proprietà analoghe, ma su diversi contesti saranno definite e dimostrate per le macchine che adottano diverse discipline di valutazione.

Le operazioni di crumbling e di read-back rispettano alcuni invarianti che sono necessari per riuscire a dimostrare il corretto funzionamento della Crumble GLAM mediante l'implementation theorem. Per il momento ci limitiamo ad elencarle in3.2.1, 3.2.2 e 3.2.1, rimandandone un'analisi più accurata a 6.3.

Lemma 3.2.1 (4.2)
$\forall t.\underline{t}_{\downarrow} = t \land \forall v.\overline{v}_{\downarrow} = v$

Lemma 3.2.2 (Remark 4.1)

1. $(\forall t.FV(\underline{t}) = FV(t)) \land (\forall v.FV(\overline{v}) = FV(v))$

2. $(\forall b.FV(b_{\downarrow}) = FV(b)) \land (\forall c.FV(c_{\downarrow}) = FV(c))$

3. *"The crumbling translation commutes with the renaming of free variables.", che può essere enunciato come segue:*
   $\forall t, x, y.x \in FV(t) \rightarrow \underline{t\{x \leftarrow y\}} = \underline{t}\{x \leftarrow y\}$

4. *"The crumbling translation and the readback map values to values"*

Lemma 3.2.3 (4.5)

1. *Freshness:* $\forall t.well_named(\underline{t})$
2. *Closure:* $\forall t.FV(t) = \emptyset \rightarrow FV(\underline{t}) = \emptyset$
3. *Disjointedness:* $\forall t, C, b, e.(\underline{t}) = C\langle b, e\rangle \rightarrow DOM(C) \cap FV(B) = \emptyset$
4. *Bodies: tutti i corpi di funzione in* $\underline{t}$ *sono la traduzione di termini.*
5. *Contextual decoding:* $\underline{t} = C\langle c\rangle \rightarrow rv_context(C_{\downarrow})$

Riduzione

Una volta operata la traduzione del λ_p-termine nell'equivalente crumbled form, la macchina effettua delle operazioni di riduzione su di esso mediante un insieme di regole di transizione che hanno lo scopo di identificare i redessi e di valutarli. Le regole della disciplina di valutazione CbV sono definite come in 3.13[5]. Per poter definire tali regole, è necessario fornire alcune definizioni:

Definizione 3.2.1. Il dominio di una crumbled form, in seguito indicato con $DOM(c)$, è l'insieme delle variabili che si trovano nella parte sinistra di ES contenuti nell'ambiente immediatamente interno a c.

Definizione 3.2.2. Indichiamo con $e(x)$ il look-up della variabile x all'interno del crumbled environment e. Si assume che tale operazione sia deterministica dal momento che, per l'invariante di well-namedness definita in 6.2.1, tali variabili sono uniche.

Definizione 3.2.3. Indichiamo con la notazione c^{α} la crumbled form ottenuta rinominando il dominio di c mediante variabili che siano distinte e fresche nella crumbled form c.

È infine necessario dare due ulteriori definizioni: quella di v-environment e di v-crumble. Un v-environment è la coda già valutata dell'environment di una crumbled form, mentre un v-crumble è un crumble già completamente valutato. Queste definizioni sono estremamente importanti perché ci permettono di dimostrare quanto avevamo già sottolineato in precedenza: i crumbled environments codificano i contesti di valutazione dei termini, basta infatti percorrerli da destra verso sinistra per identificare i redessi. Una volta effettuata una transizione, la porzione valutata dell'environment eventualmente cresce di un'unità verso sinistra fino al raggiungimento di un v-crumble. Infine, per mezzo di un teorema di normalità, è possibile dimostrare che il read-back di una crumbled form c è λ_p-normale se e solo se c è un v-crumble, definendo così una condizione di terminazione per l'esecuzione ella macchina astratta.

$$\begin{aligned} e_v &\coloneqq \epsilon \mid e_v[x \leftarrow v_{\neg x}] \\ c_v &\coloneqq (v_{\neg x}, e_v) \end{aligned} \tag{3.12}$$

[5]Per quanto riguarda la presente trattazione, dal momento che non tratteremo il caso del λ_{pif}-calcolo, ci limitiamo a presentare quella del λ_p-calcolo, rimandando il lettore interessato ad un approfondimento a [5] per l'insieme completo di regole di valutazione.

$$\begin{aligned} ((\lambda x.c)v, e_v) &\mapsto_{\beta_v} (c@[x \leftarrow v])^{\alpha}@e_v \\ (x, e_v) &\mapsto_{sub_{var}} (e_v(x), e_v) \quad (*) \\ (xv, e_v) &\mapsto_{sub_l} (e_v(x)v, e_v) \quad (*) \\ &(*) \text{ if } x \in DOM(e_v) \end{aligned} \tag{3.13}$$

Le regole della 3.13 permettono di definire la relazione $\rightarrow_{Cr}$ di riduzione in un passo sulla macchina Crumble GLAM come segue:

$$\rightarrow_{Cr} \coloneqq \mapsto_{\beta_v} \cup \mapsto_{sub_{var}} \cup \mapsto_{sub_l}$$

In primo luogo osserviamo che la relazione $\rightarrow_{Cr}$ genera crumbled form che sono sempre la concatenazione di un environment e generico e di un v-environment; in questo modo è ancora una volta verificato il fatto che per trovare i redessi basterà passare in rassegna il crumbled environment da destra a sinistra. Così facendo, non solo è possibile eliminare la necessità strutture dati che codificano il contesto di valutazione, ma è anche possibile abbassare la complessità della ricerca dei redessi ad O(1).

Tuttavia, l'operazione $\cdot^{\alpha}$ causa la ridenomina di tutti i nomi del dominio del crumbled environment di c. Questa operazione, da un punto di vista strettamente implementativo, consiste nell'effettuare una copia in memoria del corpo di funzione c al momento della valutazione poiché, lo ribadiamo, le crumbling abstract machines sono macchine ad environment globale nelle quali i nomi di variabile vengono implementati per mezzo di indirizzi di aree di memoria che di fatto contengono i termini ad esse associati dall'ambiente. Da un punto di vista di più alto livello questa operazione garantisce per definizione la well-namedness delle crumbled forms ed evita conflitti di nomi di variabile, rendendo deterministica l'operazione di valutazione di una variabile x in un crumbled environment e $e(x)$.

A questo punto il lettore potrebbe considerare interessante l'analisi di un esempio di esecuzione della macchina Crumble GLAM su un caso concreto. Parafrasando un caso preso in esame in [5], si consideri il seguente esempio:

Esempio 3.2.1. Sia dato il termine $t \coloneqq (\lambda x.x(xx))(\lambda z.z)$, studiamone la valutazione dalla macchina Crumble GLAM. Per prima cosa esso viene convertito nella sua crumbled form $\underline{t}$ come specificato dalla 3.7.

$$\begin{aligned} \underline{t} &= \underline{(\lambda x.x(xx))(\lambda z.z)} \\ &= (\overline{(\lambda x.x(xx))}\,\overline{(\lambda z.z)}, \epsilon) \\ &= ((\lambda x.\underline{x(xx)}, \epsilon)(\lambda z.(z, \epsilon)), \epsilon) \\ &= ((\lambda x.(xy, [y \leftarrow xx]), \epsilon)(\lambda z.(z, \epsilon)), \epsilon) \end{aligned}$$

Preliminarmente osserviamo che sia $\underline{x(xx)} = (xy, [y \leftarrow xx])$ che $\underline{t}$ rispettano l'invariante del contextual decoding espresso in : infatti $\underline{t}$ non può essere espresso come il plugging di nessun c in un crumbled context della forma $C = (b, E[x \leftarrow \langle\cdot\rangle])$, dal momento che l'environment è ϵ, per cui $C = \langle\cdot\rangle$, la cui read-back $\langle\cdot\rangle_{\downarrow} = \langle\cdot\rangle$ è un right-v evaluation context per definizione. Al contrario $(xy, [y \leftarrow xx])$,

oltre che a essere espresso come il plugging di se stesso in un contesto vuoto, può essere visto come il plugging di (xx, ϵ) in $(xy, [y \leftarrow \langle\cdot\rangle])$. Ma è anche vero che $(xy, [y \leftarrow \langle\cdot\rangle])_\downarrow = x\langle\cdot\rangle$ è un right-v evaluation context della forma tR.

Proviamo ora a verificare la correttezza dello stato iniziale: per farlo calcoliamo la read-back di $((\lambda x.(xy, [y \leftarrow xx]), \epsilon)(\lambda z.(z, \epsilon)), \epsilon)$.

$$\begin{aligned} ((\lambda x.(xy, [y \leftarrow xx]), \epsilon)(\lambda z.(z, \epsilon)), \epsilon)_\downarrow &= (\lambda x.(xy, [y \leftarrow xx]), \epsilon)_\downarrow(\lambda z.(z, \epsilon))_\downarrow \\ &= \lambda x.(xy, [y \leftarrow xx])_\downarrow(\lambda z.(z, \epsilon)_\downarrow) \\ &= \lambda x.(xy\{y \leftarrow xx\})(\lambda z.z) \\ &= \lambda x.(x(xx))(\lambda z.z) = t \end{aligned}$$

La proprietà $\forall t.\underline{t}_\downarrow$ sarà sempre vera in senso stretto (i.e. non modulo ridenomine delle variabili legate). Tale proprietà è, come vedremo, una delle ipotesi dell'implementation theorem; per ora ci limitiamo a questa verifica informale della validità del lemma in questo caso di studio, rimandando a 6.3.1 un'analisi più accurata dell'enunciato e della sua dimostrazione formale.

Prima di ridurre il termine mediante le regole della macchina Crumble GLAM, proviamo a ridurlo secondo la disciplina di valutazione closed CbV, in modo da sapere quale dovrebbe essere l'esito della normalizzazione.

$$t = \lambda x.(x(xx))(\lambda z.z)$$

Il termine di destra è un valore, per cui possiamo applicare la regola $\rightarrow_{\beta_v}$ ed ottenere il seguente termine:

$$(\lambda z.z)((\lambda z.z)(\lambda z.z))$$

A questo punto abbiamo due possibili redessi che corrispondono con le due applicazioni, ma quella più esterna ha come argomento $((\lambda z.z)(\lambda z.z))$ che per la grammatica del λ_p-calcolo non è un valore v, bensì un termine, per cui, per applicare la disciplina di valutazione closed CbV occorre semplificare prima l'applicazione di destra; il termine risultante sarà:

$$(\lambda z.z)(\lambda z.z) \rightarrow_p \lambda z.z$$

La crumbled form da cui ha inizio la valutazione è:

$$((\lambda x.(xy, [y \leftarrow xx]), \epsilon)(\lambda z.(z, \epsilon)), \epsilon)$$

Il crumbled environment di destra è un v-environment, il bite del crumble è un'applicazione, per cui, l'unica regola di riduzione che è possibile applicare è la $\mapsto_{\beta_v}$, per cui otterremo la seguente crumbled form:

$$(xy, [y \leftarrow xx][x \leftarrow (\lambda z.(z, \epsilon))])^\alpha = (ww', [w' \leftarrow ww][w \leftarrow (\lambda z.(z, \epsilon))])$$

Verifichiamo ora che la proprietà del contextual decoding non vale: infatti la crumbled form $(wy, [y \leftarrow ww][w \leftarrow (\lambda z.(z, \epsilon))])$ si può decomporre nella forma $C\langle c\rangle$ dove $C = (wy, [y \leftarrow ww][w \leftarrow \langle\cdot\rangle])$ e $c = (\lambda z.(z, \epsilon), \epsilon)$ oppure

$C' = (wy, [y \leftarrow \langle\cdot\rangle])$ e $c' = (ww, [w \leftarrow (\lambda z.(z, \epsilon))])$, mentre $C'_\downarrow = w\langle\cdot\rangle$ che è un right-v evaluation context, $C_\downarrow = (\langle\cdot\rangle(\langle\cdot\rangle\langle\cdot\rangle))$ chiaramente non lo è. Come avevamo anticipato, infatti, non sempre le crumbled form generate in fase di valutazione si decodificano in contesti di valutazione coerenti con la disciplina di valutazione adottata, ma quantomeno lo fanno *subito prima* di applicare regole del calcolo che è implementato (i.e. regole del calcolo che non sono tecniche della macchina). Un'altra caratterizzazione di questa proprietà, che è quella richiesta da [5] è la seguente: $\forall t, C', C, c.\underline{t} \rightarrow^*_{Cr} C'\langle C\langle c\rangle\rangle$ è un right-v evaluation context. Possiamo verificare la proprietà di cui sopra (detta *weak* contextual decoding) scomponendo $(wy, [y \leftarrow ww][w \leftarrow (\lambda z.(z, \epsilon))])$ come $C'\langle C\langle c\rangle\rangle$ ove $C' = (wy, [y \leftarrow \langle\cdot\rangle])$, $C = (ww, [w \leftarrow \langle\cdot\rangle])$ e $c = (\lambda z.(z, \epsilon), \epsilon)$ e concludere, dal momento che abbiamo già verificato che $C'_\downarrow = w\langle\cdot\rangle$ è un right-v evaluation context.

In questo caso, l'applicazione della regola $\mapsto_{\beta_v}$ ha aumentato di un'unità la porzione dell'ambiente che è già stata valutata, da cui $e_v = [w \leftarrow (\lambda z.(z, \epsilon))]$. A questo punto la riduzione procede come segue:

$$
\begin{aligned}
&(wy, [y \leftarrow ww][w \leftarrow (\lambda z.(z, \epsilon))]) \mapsto_{sub_l}\\
&(wy, [y \leftarrow (\lambda z.(z, \epsilon))w][w \leftarrow (\lambda z.(z, \epsilon))]) \mapsto_{\beta_v}\\
&(wy, [y \leftarrow k][k \leftarrow w][w \leftarrow (\lambda z.(z, \epsilon))]) \mapsto sub_{var}\\
&(wy, [y \leftarrow k][k \leftarrow (\lambda z.(z, \epsilon))][w \leftarrow (\lambda z.(z, \epsilon))]) \mapsto sub_{var}\\
&(wy, [y \leftarrow (\lambda z.(z, \epsilon))][k \leftarrow (\lambda z.(z, \epsilon))][w \leftarrow (\lambda z.(z, \epsilon))]) \mapsto sub_l\\
&((\lambda z.(z, \epsilon))y, [y \leftarrow (\lambda z.(z, \epsilon))][k \leftarrow (\lambda z.(z, \epsilon))][w \leftarrow (\lambda z.(z, \epsilon))]) \mapsto \beta_v\\
&(r, [r \leftarrow y][y \leftarrow (\lambda z.(z, \epsilon))][k \leftarrow (\lambda z.(z, \epsilon))][w \leftarrow (\lambda z.(z, \epsilon))]) \mapsto sub_{var}\\
&(r, [r \leftarrow (\lambda z.(z, \epsilon))][y \leftarrow (\lambda z.(z, \epsilon))][k \leftarrow (\lambda z.(z, \epsilon))][w \leftarrow (\lambda z.(z, \epsilon))]) \mapsto sub_{var}\\
&((\lambda z.(z, \epsilon)), [r \leftarrow (\lambda z.(z, \epsilon))][y \leftarrow (\lambda z.(z, \epsilon))][k \leftarrow (\lambda z.(z, \epsilon))][w \leftarrow (\lambda z.(z, \epsilon))])
\end{aligned}
$$

La crumbled form che abbiamo ottenuto è un v-crumble, la macchina Crumble GLAM è ora in grado di farne read-back ed ottenere l'esito finale della computazione:

$$
\begin{aligned}
&((\lambda z.(z, \epsilon)), [r \leftarrow (\lambda z.(z, \epsilon))][y \leftarrow (\lambda z.(z, \epsilon))][k \leftarrow (\lambda z.(z, \epsilon))][w \leftarrow (\lambda z.(z, \epsilon))])_\downarrow =\\
&(\lambda z.z)\{r \leftarrow (\lambda z.(z, \epsilon))\}\{y \leftarrow (\lambda z.(z, \epsilon))\}\{k \leftarrow (\lambda z.(z, \epsilon))\}\{w \leftarrow (\lambda z.(z, \epsilon))\} = \lambda z.z
\end{aligned}
$$

In riferimento all'esempio appena mostrato, possiamo convincerci intuitivamente della correttezza del processo di valutazione di λ-termini per mezzo di crumbling abstract machines, ma per poterlo dimostrare formalmente è necessario riprendere un importante risultato teorico: l'implementation theorem. Tale risultato, che fa riferimento ad uno studio di Accattoli e Guerrieri riportato in [3], permette di dimostrare la correttezza di qualsiasi implementazione astratta del λ-calcolo. Tale teorema è formulato come segue:

Teorema 1 (Implementation theorem). Dati:

- Una macchina astratta M che è una relazione di transizioni $\rightarrowtail_M$ su di un insieme di stati divise in:

  - *transizioni principali* $\rightarrowtail_p$, che corrispondono ai passi di valutazione del calcolo
  - *transizioni di overhead* $\rightarrowtail_o$, che sono specifiche della macchina

- una strategia di valutazione $\rightsquigarrow$ del λ-calcolo
- una procedura di decoding $\cdot_\downarrow$

M implementa $\rightsquigarrow$ per mezzo di $\cdot_\downarrow$ ogniqualvolta le seguenti condizioni siano verificate:

- Inizializzazione: esiste una funzione di encoding $\underline{\cdot}$ tale che $\forall t.\underline{t}_\downarrow = t$;
- Proiezione principale: $\forall s, s'.s \rightarrowtail_p s' \to s_\downarrow \rightsquigarrow s'_\downarrow$;
- Trasparenza dell'overhead: $\forall s, s'.s \rightarrowtail_o s' \to s_\downarrow = s'_\downarrow$;
- Determinismo: $\rightarrowtail_M$ è deterministica.
- Halt: Gli stati finali di M si decodificano in termini normali,
- Terminatine dell'overhead: le sequenze di transizioni di $\rightarrowtail_o$ terminano.

Alla luce di questi fatti, il processo di valutazione di un λ-termine da parte della Crumble GLAM può essere riassunto per mezzo del grafico in figura 3.3. La dimostrazione formale del risultato di inizializzazione, come anticipato può essere trovata in 6.3.1. Il lettore interessato ad approfondire le dimostrazioni degli altri enunciati può fare riferimento all'appendice A di [5], dove ne vengono fornite dimostrazioni accurate, salvo alcune imprecisioni che ho evidenziato e risolto nel lavoro di formalizzazione e che verranno discusse in seguito.

Una volta dimostrata la correttezza dell'implementazione del λ_{pif} calcolo, è il momento di effettuarne un accurato studio di complessità. Per far ciò si può procedere analizzando singolarmente la complessità di ciascuna delle operazioni di riduzione della Crumble GLAM, identificando successivamente quali sono i bound che legano il numero di occorrenze di ciascuna transizione in una sequenza di passi di riduzione normalizzante alla lunghezza di tale riduzione, per calcolare l'overhead complessivo indotto dalla valutazione.

Per far ciò, in [5] si procede scomponendo ognuna delle transizioni $\mapsto_*$ come segue:

- Ricerca del redesso
- Unplugging del redesso: divisione della crumbled form complessiva in un crumbled context C ed un crumble c su cui operare la riscrittura dettata dalla regola
- Riscrittura della crumbled form c in c' su cui è stata applicata la regola $\mapsto_*$
- Plugging della crumbled form c all'interno del crumbled context C

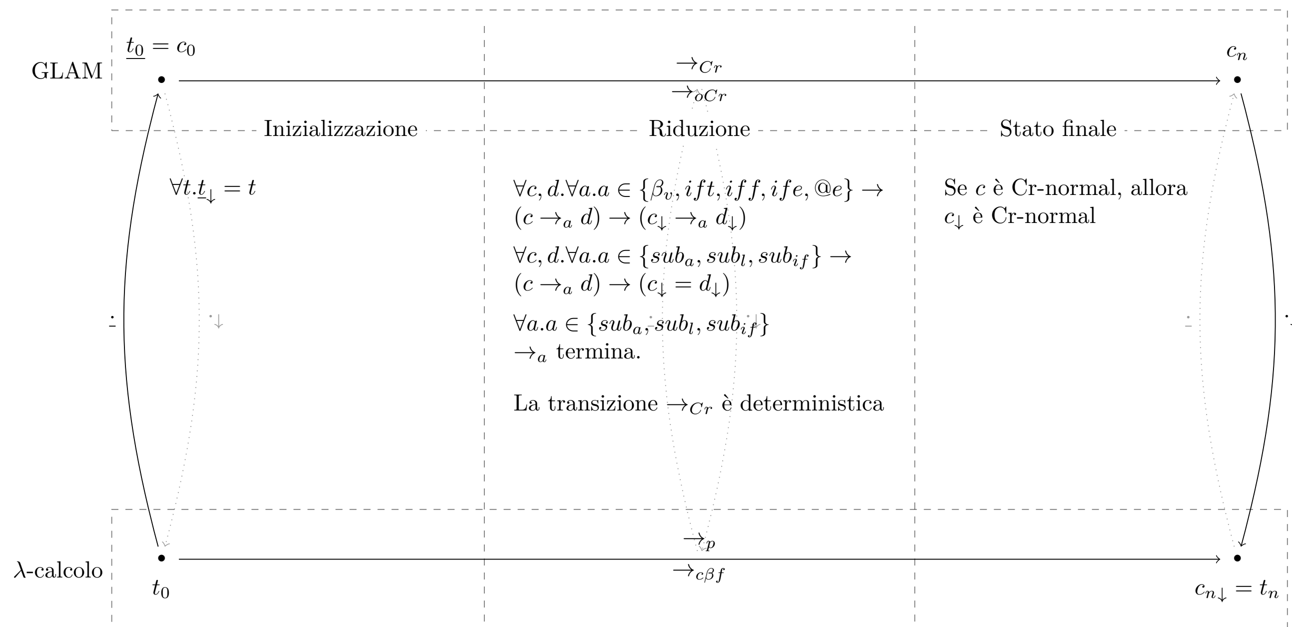

Figura 3.3: Schema di riduzione di un termine t_0 alla sua forma normale t_n per mezzo della macchina astratta Crumble GLAM e dei risultati teorici necessari per garantire la correttezza di ciascuna operazione.

La ricerca del redesso, come abbiamo visto, è implicita nella struttura sequenziale dei crumbled environments, mentre il costo della riscrittura delle crumbled form è determinato da uno degli invarianti della macchina Crumble GLAM, il quale asserisce che ogni corpo di funzione c' sottotermine di una crumbled form generata in fase di valutazione è, a meno di α-conversioni, un sottotermine del termine iniziale. Inoltre, è opportuno osservare che la dimensione delle crumbled forms prodotte dalla funzione $\underline{\cdot}$ è lineare nella dimensione del termine di partenza. In particolare, dimostreremo in 6.4, che $\forall t.|\underline{t}| \leq 5|\underline{t}|$. Per questi motivi, la complessità dell'operazione di riscrittura è limitata dalla taglia iniziale del termine. Come abbiamo anticipato, inoltre, la complessità della concatenazione dei crumbled environment può essere abbassata ad O(1): in [5] è presentata, in appendice, un'implementazione in OCaml di crumbled environments che godono di tale proprietà. Possiamo quindi concludere che la complessità di ciascuna operazione è dello stesso ordine della complessità di riscrittura.

Per il momento trascuriamo il costo di plugging e unplugging, che verrà analizzato in seguito e cerchiamo di limitare la lunghezza di una sequenza di riduzione della macchina Crumble GLAM: la proprietà di proiezione principale assicura collateralmente che il numero di riduzioni principali operate dalla Crumble GLAM sia esattamente uguale al numero di transizioni necessarie nel calcolo che essa implementa; per questo motivo il problema si riduce a quello di fornire un bound al rapporto fra le transizioni di overhead e quelle principali in modo da assicurarsi che il numero complessivo di transizioni della Crumble GLAM siano lineari nel numero di riduzioni del calcolo.

Per quanto riguarda il numero di transizioni di overhead si giunge al seguente risultato:

Lemma 3.2.4
Sia ρ una sequenza di riduzioni della Crumble GLAM, e $|\rho|_p$ il numero di transizioni principali in ρ e $|\rho|_x$ il numero di volte in cui la regola denotata a pedice con $\cdot_x$ compare in ρ, allora vale la seguente conclusione:

$$|\rho|_{sub_l} + |\rho|_{sub_{var}} + |\rho|_{sub_{if}} \leq 3|\rho|_p + 2$$

Occorrerebbe, a questo punto, determinare il costo delle operazioni di plugging ed unplugging necessarie per la valutazione. Per agevolare questa stima, in [5] viene presentata come implementazione alternativa della Crumble GLAM, la pointed Crumble GLAM, mediante la quale è possibile stimare con maggior accuratezza la complessità di tali operazione. Il risultato è che, anche in questo caso, la complessità è bilineare nel numero di β-riduzioni necessarie per la valutazione completa del termine. Per queste ragioni la complessità complessiva della Crumble GLAM risulta essere bilineare nel numero di β-riduzioni necessarie per la valutazione di un termine (e nella taglia iniziale di tale termine).

Capitolo 4

Tipi e funzioni di manipolazione

Una parte preliminare del lavoro di formalizzazione raccolto in [14] ha richiesto la definizione delle strutture dati su cui lavora la macchina Crumble GLAM. Da un punto di vista implementativo tale processo è stato molto semplice: è bastato trascrivere le BNF con cui tali strutture vengono definite in [5] con equivalenti dichiarazioni di tipi di dato astratto in Matita. Tuttavia, è stato scelto di semplificare i tipi presentati nell'articolo rimuovendo la produzione dell'`if-then-else` dalle grammatiche dei termini e delle crumbled forms. Tale costrutto, come visto in precedenza, non fa parte del λ-calcolo e non ne aumenta il potere espressivo, tuttavia molti linguaggi funzionali lo presentano in maniera esplicita per semplificare la scrittura di codice ed aumentare l'efficienza dell'esecuzione; per questa ragione in letteratura, è facile trovare grammatiche del λ-calcolo arricchite con strutture di scelta.

Come abbiamo anticipato, nel caso specifico di [5], gli autori hanno deciso di aggiungere l'`if-then-else` al linguaggio per evidenziare che è possibile definire macchine che implementano il λ-calcolo per mezzo di sharing e non soffrono delle anomalie ipotizzate da Kennedy. Alla luce di ciò, la correttezza del crumbling non dipende direttamente dalla presenza del costrutto di scelta nel linguaggio (che è ridondante), per cui è stato considerato ragionevole scegliere di snellire le dimostrazioni rimuovendo tale costrutto dalla grammatica dei termini e dei crumbles. Ne derivano le dichiarazioni che riportiamo in (4.1) e in (4.2) dimodoché possano essere un riferimento per le sezioni successive del lavoro.

$$\begin{aligned} t &\coloneqq t_1 t_2 \mid v \\ v &\coloneqq x \mid \lambda x.t \end{aligned} \tag{4.1}$$

$$\begin{aligned} c &\coloneqq (b, e) \\ b &\coloneqq v_1 v_2 \mid v \\ v &\coloneqq x \mid \lambda x.c \\ e &\coloneqq \epsilon \mid es \\ s &\coloneqq [x \leftarrow b] \end{aligned} \tag{4.2}$$

Il passaggio dalle BNF (4.1) (4.2) alle dichiarazioni dei rispettivi tipi di dati astratti in Matita è abbastanza semplice, come mostrato dalle dichiarazioni riportate in 4.1 e in 4.2:

```
1   inductive pTerm : Type[0] :=
2    | val_to_term: pValue → pTerm
3    | appl: pTerm → pTerm → pTerm
4
5   with pValue : Type[0] :=
6    | pvar: Variable → pValue
7    | abstr: Variable → pTerm → pValue
8    .
9
10  inductive pSubst : Type[0] :=
11   | psubst: Variable → pTerm → pSubst
12   .
```

Codice 4.1: Dichiarazione dei tipi mutui pTerm e pValue e del tipo semplice pSubst

```
1   inductive Crumble : Type[0] :=
2    | CCrumble: Bite → Environment → Crumble
3
4   with Bite : Type[0] :=
5    | CValue: Value → Bite
6    | AppValue: Value → Value → Bite
7
8   with Value : Type[0] :=
9    | var : Variable → Value
10   | lambda: Variable → Crumble → Value
11
12  with Environment : Type[0] :=
13   | Epsilon: Environment
14   | Snoc: Environment → Substitution → Environment
15
16  with Substitution: Type[0] :=
17   | subst: Variable → Bite → Substitution
18  .
```

Codice 4.2: Dichiarazione dei tipi mutui mutui Crumble, Bite, Environment, Value e Substitution

Sulle definizioni dei tipi pTerm e Crumble date in 4.1 e in 4.2 (e dei tipi ad essi mutualmente ricorsivi) è utile formulare le seguenti osservazioni:

- La definizione dei lambda termini data da Plotkin in [18] (e riportata in (4.1)) comporta la necessità di definire due tipi di dati mutualmente ricorsivi: quella dei pTerm e quella dei pValue. L'arricchimento del linguaggio

del lambda calcolo con i costrutti di ES e dei crumbled environments fa sì che l'effetto della mutualità fra pTerm e pValue induca una definizione mutua dei tipi che compongono i Crumble.

- Se si prova a mettere a confronto la grammatica dei pTerm con quella dei Crumble (e dei tipi a loro mutui) è semplice notare la corrispondenza che c'è fra i tipi pValue e Value e fra pTerm e Bite: i secondi sono rappresentazioni nel linguaggio della Crumble GLAM di ciò che i primi sono nella grammatica di Plotkin.
- Il tipo pSubst è contemporaneamente sia uno dei due argomenti della funzione informale di sostituzione $t\{x \leftarrow t'\}$, che il corrispondente, dalla parte dei pTerm, del tipo Substitution dalla parte dei Crumble.

Per quanto riguarda i Crumble osserviamo che, a causa della ricorsione mutua, non è stato possibile definire una grammatica più semplice di quella riportata in 4.2: per ragioni di armonia, Bite e Value devono essere mutui ed il fatto che all'interno delle Substitution compaiano Bite impone di inserire anche questi nella nella ricorsione mutua; infine, il fatto che l'argomento delle λ-astrazioni sia un Crumble impedisce di definire questo tipo indipendentemente dagli altri. Come vedremo in seguito, la mutualità dei tipi renderà molto onerose alcune dimostrazioni per le quali sarà necessario definire e adottare principi di induzione mutui e riformulare mutualmente gli enunciati dei teoremi.

Gli Environment sono definiti come liste di Substitution associative a sinistra. La ragione di questa scelta è dovuta al fatto che la macchina Crumble GLAM cerchi i redessi scorrendo sequenzialmente la lista da destra verso sinistra: con Environment di questo tipo l'implementazione della disciplina di valutazione call-by-value risulta più semplice perché la macchina può andare direttamente in ricorsione su di essi. Tuttavia, durante il processo di crumbling (come riportato in 6.3), è necessario codificare il contesto di valutazione dei termini inserendo le Substitution introdotte a sinistra dell'Environment nel quale risiedono i termini di partenza. Dunque, se da un lato la scelta di codificare gli Environment come liste associative a sinistra facilita la valutazione dei termini, dall'altro rende più complesso il processo di crumbling a causa del forzato utilizzo di una funzione di inserimento a sinistra, che diremo `push`. Contrariamente alla loro definizione, tuttavia, gli Environment utilizzati dall'articolo non sono né destri né sinistri: ciò è dettato sia dal fatto che le variabili debbano essere risolte in O(1), che dal fatto che la funzione `concat` (la quale giustappone sequenzialmente due ambienti) debba avere complessità costante: tale ottimizzazione non è possibile se le liste sono definite ricorsivamente. Anche per questo motivo, nelle dimostrazioni presenti nell'articolo, gli Environment sono spesso trattati in maniera molto informale. Questo fenomeno, in molti casi, ci indurrà a definire lemmi tecnici anche complessi per giustificare tali operazioni.

L'articolo definisce alcune generiche funzioni di manipolazione dei Crumble che saranno impiegate di frequente dalla Crumble GLAM. Nel frammento di codice 4.3 sono state riportare le definizioni date in Matita come riferimento per le parti successive del lavoro.

```
1  let rec push e s :=
2   match e with
3   [ Epsilon ⇒ Snoc Epsilon a
4   | Snoc e1 s1 ⇒ Snoc (push e1 s) (s1)
5   ].
6
7  let rec concat e f on f :=
8   match f with
9   [ Epsilon ⇒ e
10  | Snoc b' s ⇒ Snoc (concat e b') s]
11  .
12
13 definition at: Crumble → Environment → Crumble := λc,f.
14 match c with
15 [ CCrumble bite e  ⇒ CCrumble bite (concat e f)
16 ].
```

Codice 4.3: Funzioni di manipolazione di Environment e Crumble

Forniamo un rapido commento alla semantica delle funzioni definite in 4.3:

- la funzione `push` scorre ricorsivamente l'Environment `e` ed inserisce la sostituzione `s` alla sua estremità sinistra.
- la funzione `concat` concatena due ambienti `e` ed `f`.
- la funzione `at` accoda un Environment `f` ad un Crumble `c`.

Dopo aver definito i tipi di dati Crumble e pTerm sono stati introdotti i relativi contesti. Come abbiamo già anticipato in 3.2, un contesto è un termine che contiene una particolare costante, detta `hole` ed indicata con $\langle\cdot\rangle$. Nell'articolo vengono definite per i Crumbled contexts che riportiamo in (4.3):

$$\begin{aligned} C &\coloneqq \langle\cdot\rangle \,|\, (b, E) \\ E &\coloneqq e[x \leftarrow \langle\cdot\rangle] \end{aligned} \tag{4.3}$$

Per quanto riguarda le definizioni di contesti su termini del λ-calcolo, tuttavia, viene solo fornita la grammatica dei right-v evaluation contexts che riportiamo in (4.4).

$$R \coloneqq \langle\cdot\rangle \,|\, tR \,|\, Rv \tag{4.4}$$

Tali contesti sono particolarmente utili per la dimostrazione di un particolare invariante della Crumble GLAM, detto del contextual-decoding, m,a per ragioni che vedremo in seguito, non è affatto interessante implementarli come tipi di dato, per cui è stato scelto di definire al loro posto il tipo di dato TermContext ed un predicato `rv_context` che permette di determinare se un contesto generico sia o meno un right-v evaluation context.

$$T \coloneqq \langle\cdot\rangle \,|\, Tt \,|\, tT \tag{4.5}$$

```
1  inductive TermContext : Type[0] :=
2  | thole : TermContext
3  | term : pTerm → TermContext
4  | c_appl : TermContext → TermContext → TermContext
5  | c_abstr : Variable → TermContext → TermContext
6  .
```

Codice 4.4: Dichiarazione del tipo TermContext in Matita

Confrontando le due BNF in (4.4) e in (4.4) si può osservare che per poter implementare il predicato `rv_context` è necessario definire altri due sotto-predicati che, procedendo per ricorsione su di un contesto, riescano a determinare se questo è un termine oppure un valore. Detti questi predicati rispettivamente `tc_term` e `tc_value` possiamo definire `rv_context` come segue:

```
1  let rec rv_context T on T :=
2   match T with
3   [ thole ⇒ True
4   | term t ⇒ False
5   | c_appl t1 t2 ⇒ (tc_term (t1) ∧ rv_context (t2)) ∨ (rv_context (t1) ∧ tc_value
      (t2))
6   | c_abstr x TT ⇒ False
7   ]
8   .
```

Codice 4.5: Definizione ricorsiva del predicato `rv_context`

Come già noto, l'operazione più importante che si definisce sui contesti è quella di plugging, che consiste nel rimpiazzamento della costante $\langle\cdot\rangle$ nel termine t con un termine s e spesso indicata con $t\langle s\rangle$. Mentre per i TermContext, che sono tipi di dati ricorsivi, è necessario definire ricorsivamente la funzione di plugging, per i CrumbleContext si può procedere senza ricorsione sfruttando la keyword `definition`, come mostrato in 4.6:

```
1  definition plug_E := λE.λD.
2   match E with
3    [ envc e x ⇒ match D with
4      [ hole ⇒ E
5      | crc b ec ⇒ match ec with
6        [ envc f z ⇒ envc (concat (Snoc e [x ←b]) f) z]
7      ]
8    ]
9  .
10
11 definition plug_C := λC.λD.
12  match C with
13  [ hole ⇒ D
14  | crc b ec ⇒ crc b (plug_E ec D)
15  ]
16  .
17
18 let rec plug_t T t on T :=
19  match T with
20  [ thole ⇒ t
21  | term t' ⇒ t'
22  | c_appl u1 u2 ⇒ appl (plug_t u1 t) (plug_t u2 t)
23  | c_abstr x TT ⇒ val_to_term (abstr x (plug_t TT t))
24  ]
25  .
```

Codice 4.6: Definizioni delle operazioni di plugging

Capitolo 5

Variabili

Come anticipato, durante il lavoro di formalizzazione raccolto in [14] la gestione delle variabili ha richiesto particolare attenzione: l'articolo, infatti, trattava molto informalmente tale ambito. Per questo motivo è stato necessario un notevole sforzo di definizione di procedure e di teoremi che permettessero, da un lato, di far sì che le funzioni definite dall'articolo gestissero le variabili in modo da essere coerenti con la loro specifica e, dall'altro lato, che fossero in grado di implementare efficacemente meccanismi di gestione dei nomi di variabile.

Come vedremo in seguito, quello dei nomi di variabile è stato anche l'unico ambito in cui il lavoro di formalizzazione [14] ha potuto evidenziare criticità nella teoria della macchina Crumble GLAM. Per questo motivo riteniamo che sia interessante analizzare per sommi capi quali gli strumenti definiti per la gestione dei nomi di variabile. Infine, siccome la teoria delle variabili nel λ-calcolo è ben consolidata, la maggior parte dei risultati in questo capitolo saranno già noti al lettore, o comunque già trattati in 2. Dunque, non appesantendo troppo il lettore con l'onere di dover affrontare argomenti troppo specifici, questo capitolo vuole anche essere un'anticipazione di come opereranno i lavori di formalizzazione che saranno presentati più avanti.

Prima di procedere con lo studio di come sia stata operata la formalizzazione di questa parte del lavoro, è interessante spendere alcune osservazioni su quale sia il ruolo non banale dei nomi di variabile nelle implementazioni ad ambiente globale del λ-calcolo. Come abbiamo anticipato, una delle peculiarità delle implementazioni del λ-calcolo con ambiente globale è che questi spesso rendono possibile la valutazione di nomi di variabili nell'ambiente con complessità costante. Il modo adottato da [5] per ottenere questo risultato è quello di usare come nomi di variabile gli indirizzi di memoria in cui risiedono le rappresentazioni dei termini a loro associate mediante costrutti di ES. Per questa ragione, ognuna delle nozioni che definiremo sulle variabili, ha una controparte intuitiva in memoria. Ad esempio, la generazione di una variabile fresca per un determinato termine corrisponde alla ricerca di un indirizzo di memoria che non è ancora stato allocato in quel dato termine, similmente: la generazione di una variabile globalmente fresca coincide con la generazione di un indirizzo di memoria

libero globalmente. D'altro canto, una variabile libera è una variabile che punta ad un indirizzo che non è legato da alcun termine, per cui, trasponendo questa situazione a quella più comune di un linguaggio di programmazione, diremmo che quella variabile punta ad un indirizzo di memoria che è esterno allo spazio di nomi della funzione in cui si trova, il quale può essere, ad esempio, l'indirizzo di una procedura definita in un modulo esterno.

Risulta ora evidente che la gestione dei nomi di variabile sia una parte critica dell'implementazione di una crumbling abstract machine e che il fatto di aver identificato e risolto imprecisioni in questo ambito sia notevole importanza nel contesto del lavoro complessivo.

Per poter procedere allo studio delle variabili è stato necessario definire un tipo Variable. Gli elementi di questo tipo devono godere di due proprietà: essere almeno un'infinità numerabile e deve essere effettivamente decidibile l'uguaglianza di due variabili. In linea di principio, dunque, per il ruolo di variabili avremmo potuto utilizzare anche solo i numeri naturali, tuttavia l'interpretazione consueta di un naturale è completamente diversa da quella di una variabile, per cui tale operazione sarebbe stata impropria da un punto di vista semantico. Per tali ragioni si è scelto di definire il tipo Variable applicando un semplice costruttore `variable` (spesso abbreviato con la lettera ν) ad un $\mathbb{N}$. La definizione di tipo che ne è derivata è quella espressa in 5.1.

```
1 inductive Variable: Type[0] :=
2  | variable: nat → Variable
3 .
```

Codice 5.1: Dichiarazione del tipo Variable

5.1 Decisione di variabili e variabili fresche

Una volta che è stato definito con il nome Var l'insieme delle variabili, è possibile determinare dei sottoinsiemi di Var che godono di specifiche proprietà. La letteratura (si prenda per esempio [7]) solitamente definisce funzioni $L(t)$[1] $\longrightarrow (Var)$ che dato un termine qualsiasi determinano l'insieme delle sue variabili libere e delle sue variabili legate. Nel nostro caso, senza mancare di generalità, abbiamo preferito definire delle funzioni $L(t) \times Var \longrightarrow (B)$, questa scelta ci permette di lavorare agevolmente con funzioni caratteristiche di insiemi senza il bisogno di dover andare ad appesantire le dimostrazioni con operazioni su insiemi finiti.

Siccome la nozione di variabile libera per i λ-termini è già consolidata nella letteratura, non è molto informativo riportarla anche qui, mentre potrebbe essere più interessante vedere come questa è stata generalizzata al caso dei Crumble. Per farlo è stato necessario definire la nozione di dominio di un Crumble con il nome di DOM. Il dominio di un Crumble, come già specificato in 3.2.1, è la lista delle variabili legate dagli ES presenti nel suo Environment più esterno. Più formalmente:

[1] $L(t)$ rappresenta il linguaggio generato dalla grammatica T in (4.1).

$$\begin{aligned} DOM((b,e)) &:= DOM(e) \\ DOM(\epsilon) &:= \emptyset \\ DOM(e[x \leftarrow b]) &:= DOM(e) \cup \{x\} \end{aligned} \tag{5.1}$$

Che, una volta implementato in Matita, diventa:

```
1  let rec domb x c on c :=
2   match c with
3   [ CCrumble b e ⇒ domb_e x e ] .
4
5  let rec domb_e x e on e :=
6   match e with
7   [ Epsilon ⇒ false
8   | Snoc e s ⇒ match s with [ subst y b ⇒ (veqb x y) ∨ (domb_e x e)]
9   ]
10 .
```

Codice 5.2: Implementazione in Matita della funzione che calcola il dominio di un Crumble

A questo punto dovrebbe essere ancora più chiaro, anche da un punto di vista operativo, il ruolo degli Environment nel crumbling: essi legano le variabili del Bite (i.e. del termine) similmente alle λ-astrazioni, ma ad un livello più alto. In questo modo è possibile separare la logica computazionale, rappresentata dalla composizione annidata di astrazioni e applicazioni, dalla riscrittura, espressa dai costrutti di ES. Proprio in questo, come abbiamo anticipato, risiede la ragione per cui le implementazioni del λ-calcolo per mezzo di ES abbiano complessità ragionevoli.

Dal momento che i costrutti di ES contenuti negli Environment legano le variabili libere di un Crumble come fossero astrazioni, è necessario generalizzare la nozione di variabile libera per un Crumble in modo che tenga conto anche di questo aspetto. La soluzione adottata da [5] è la seguente:

$$\begin{aligned} FV((b,e)) &:= (FV(b) \setminus DOM(e)) \cup FV(e) \\ FV(v_1 v_2) &:= FV(v_1) \cup FV(v_2) \\ FV(\lambda x.c) &:= FV(c) \setminus \{x\} \\ FV(\epsilon) &:= \emptyset \\ FV(e[x \leftarrow b]) &:= (FV(e) \setminus \{x\}) \cup FV(b) \end{aligned} \tag{5.2}$$

```
1  let rec fvb x c on c : bool :=
2   match c with
3   [ CCrumble b e ⇒ ((fvb_b x b) ∧ ¬(domb_e x e)) ∨ fvb_e x e ]
4
5  and fvb_b x b on b :=
6   match b with
7   [ CValue v ⇒ fvb_v x v
8   | AppValue v w ⇒ (fvb_v x v) ∨ (fvb_v x w)
9   ]
10
11 and fvb_e x e on e :=
12  match e with
```

```
13   [ Epsilon ⇒ false
14   | Snoc e s ⇒ match s with [subst y b ⇒ ((fvb_e x e) ∧ (¬ veqb x y)) ∨ fvb_b x b]
15   ]
16
17  and fvb_v x v on v :=
18   match v with
19   [ var y ⇒ veqb x y
20   | lambda y c ⇒ (¬(veqb y x) ∧ fvb x c)
21   ]
22   .
```

Codice 5.3: Implementazione in matita della funzione `fvb`[2]

Similmente alla funzione `fvb`, ne è stata definita un analoga per i λ_p-termini detta `fvb_t` e allo stesso modo della famiglia di funzioni `fvb*` sono state famiglie di funzioni utili alla gestione dei nomi di variabile nel contesto dei Crumble. Esse sono `inb` e `fresh_var`. Per ragioni di economia di spazio, non forniamo il codice dell'implementazione di `inb` poiché, come si può dedurre dalla definizione data in (5.3), la sua implementazione è molto simile a quella di `fvb`. Tuttavia, osserviamo rapidamente che la funzione IN crea l'insieme delle variabili che appaiono nel termine su cui è lanciata, senza fare distinzioni fra occorrenze libere e legate.

$$
\begin{aligned}
IN((b,e)) &:= IN(b) \cup IN(e) \\
IN(v_1 v_2) &:= IN(v_1) \cup IN(v_2) \\
IN(\lambda x.c) &:= IN(c) \cup \{x\} \\
IN(\epsilon) &:= \emptyset \\
IN(e[x \leftarrow b]) &:= (IN(e) \cup \{x\}) \cup IN(b)
\end{aligned}
\tag{5.3}
$$

Invece il discorso si complica un poco per quanto riguarda la funzione `fresh_var`: al contrario delle funzioni di gestione delle variabili presentate finora, essa non è una funzione di decisione, ma di ricerca: la sua specifica è quella di cercare il più basso indice di variabile per cui vale il seguente invariante: $\forall$`x.inb` ν`x c = true` $\rightarrow$ `x` $\leq$ `fresh_var c`. Fatto ciò sarà banale costruire una variabile x fresca nel Crumble c come ν `fresh_var c`. In 5.4 riportiamo la definizione della funzione FRV che calcola la variabile fresca di un pTerm, tuttavia non riportiamo il codice della funzione realmente implementata, perché, facendo uso di Σ-tipi, tale implementazione introdurrebbe un livello di complessità non necessario ai fini della presente trattazione. Il lettore può benissimo immaginare che sia implementata similmente alle funzioni precedenza senza il rischio di mancare di dettaglio[3]. Allo stesso modo sono definite le funzioni mutualmente ricorsive sui tipi Crumble, Bite, Environment, Value e Substitution.

$$
\begin{aligned}
FRV(t_1 t_2) &:= max(FRV(t_1))(FRV(t_2)) \\
FRV(\nu x) &:= S\ x \\
FRV(\lambda x.t) &:= max(S\ x)(FRV(t))
\end{aligned}
\tag{5.4}
$$

[2]la funzione `veqb` è definita $Var \times Var \longrightarrow \mathbb{B}$ ed è la funzione di decisione della relazione di uguaglianza.

[3]Per una più adeguata trattazione dei Σ-tipi rimandiamo il lettore a A

5.1.1 Occorrenze libere

Un'altra funzione abbastanza importante che è stato necessario introdurre per portare a termine il lavoro di formalizzazione è la funzione che conta le occorrenze libere di una variabile in un pTerm. Come si potrà leggere in A.2, questa funzione è indispensabile per poter garantire la definibilità in Matita una funzione la funzione di sostituzione su pTerm. Uno degli aspetti interessanti di questa funzione è il fatto che in [5] non si faccia nemmeno riferimento alla sua necessità. Mentre la definizione della funzione `free_occ_t` è molto semplice, e dunque non la riportiamo, può essere interessante vedere come la definizione di tale funzione ci permetta di non dover definire per ricorsione la funzione `fvb_t`, poiché basterà implementarla nel seguente modo:

```
1  definition fvb_t := λx.λt. gtb (free_occ_t x t) 0.
2  definition fvb_tv := λx.λv. gtb (free_occ_v x v) 0.
```

Codice 5.4: Definizione delle funzioni `fvb_t` e `fvb_v`

Dando questa definizione, tutte le relazioni fra la `fvb_t` e la `free_occ_t` non richiedono di essere dimostrate perché sono implicite nella definizione di `fvb_t`. Inoltre, siccome la `free_occ_t` è implementata mediante una visita ricorsiva dell'albero di derivazione del termine, ma restituisce un risultato più espressivo di quello che la `fvb_t` avrebbe restituito con la medesima visita, definire la la `fvb_t` in funzione della `free_occ_t` non complica più del dovuto le dimostrazioni di proprietà riguardanti la più semplice `fvb_t`.

5.2 Relazioni fra variabili

Ora che abbiamo definito le nozioni di IN, FV e di FRV ci è possibile istanziare, mediante alcuni lemmi, le relazioni che valgono fra queste funzioni. La (5.5) è proprio la a caratterizzazione di `free_var` che abbiamo formulato più su, mentre la (5.6) è molto semplice: dice che tutte le variabili libere devono essere presenti nel termine.

$$\forall \mathtt{x}.\mathtt{inb}\ \nu\mathtt{x}\ \mathtt{c}\ =\ \mathtt{true} \rightarrow \mathtt{x} \leq \mathtt{fresh_var}\ \mathtt{c} \tag{5.5}$$

$$\forall \mathtt{x}.\mathtt{fvb}\ \mathtt{x}\ \mathtt{c}\ =\ \mathtt{true} \rightarrow \mathtt{inb}\ \mathtt{x}\ \mathtt{c}\ =\ \mathtt{true} \tag{5.6}$$

Si noti che la combinazione della (5.6) e della (5.5) fornisce un altro importante risultato:

$$\forall \mathtt{x}.\mathtt{inb}\ \nu\mathtt{x}\ \mathtt{c}\ =\ \mathtt{true} \rightarrow \mathtt{x} \leq \mathtt{fresh_var}\ \mathtt{c} \tag{5.7}$$

Informalmente, enunciati di questo tipo verrebbero liquidati scrivendo "triviale", tuttavia non c'è da stupirsi se abbiamo dovuto definire tali risultati: anche se i proof assistants forniscono comandi per la prova automatica di goal semplici, l'efficienza di questi costrutti, per ragioni legate alla dimensione dello spazio di ricerca dei risultati, è insufficiente per permettere di provare risultati di tipo in tempi praticabili.

Proponiamo, come esempio, la formalizzazione dell'enunciato (5.6) in 5.5, mentre né in questo caso né in futuro mostriamo esempi di dimostrazioni formali per le seguenti ragioni:

- Una dimostrazione formale è un oggetto (solitamente di vaste dimensioni) difficilmente interpretabile qualora non si disponga di un proof assistant in grado di mostrare, passo dopo passo, l'evoluzione del goal e delle ipotesi al seguito dell'applicazione di un tactical.
- Non è molto significativo, da un punto di vista pratico, conoscere nel dettaglio come è stata affrontata una dimostrazione formale qualora questa sia stata già verificata corretta dal proof assistant.

Per cui, qui come in seguito, dopo aver fornito l'enunciato formale di un lemma, ci limiteremo a fornire una descrizione informale della sua dimostrazione solo qualora questa non sia presente in [5] o qualora per alcune ragioni la si possa ritenere interessante.

```
lemma fv_to_in_crumble:
 (∀c.∀x. fvb x c = true → inb x c = true) ∧
  (∀b.∀x. fvb_b x b = true → inb_b x b = true) ∧
   (∀e.∀x. fvb_e x e = true → inb_e x e = true) ∧
    (∀v.∀x. fvb_v x v = true → inb_v x v = true) ∧
     (∀s.∀x. fvb_s x s = true → inb_s x s = true).
```

Codice 5.5: Enunciato formale del lemma che lega le nozioni di variabile libera e di variabile che occorre per il tipo Crumble

Dalla 5.5 possiamo osservare che è stato scelto di formulare congiuntamente l'enunciato del lemma principale con enunciati simili per i tipi mutualmente ricorsivi con il tipo Crumble. Questa scelta è stata indotta dal fatto che le funzioni `fvb` e `inb` effettuino ricorsivamente chiamate su tutta la catena di tipi mutui con Crumble. Per procedere in una maniera più semplice possibile con la dimostrazione formale è stato necessario definire una serie di predicati simili a (5.6), ma formulati sui tipi mutui a Crumble dimodoché, usando il principio di induzione mutua su Crumble `Crumble_mutual_ind` [4], il sistema d'inferenza di tipi di Matita potesse costruire automaticamente i tipi (i.e. gli enunciati) delle ipotesi induttive.

Ovviamente, per ragioni di simmetria, sono state enunciate e dimostrate proprietà simili anche per pTerm e pValue. Infine è stato utile osservare alcune semplici proprietà di decomposizione delle funzioni appena definite rispetto alle operazioni sugli ambienti mostrate in 4.3. Tuttavia, siccome sia `at` che `push` possono essere ricondotte al caso del `concat`[5], possiamo limitarci a mostrare tali lemmi enunciati per quest'ultima.

```
lemma domb_concat_distr:
 ∀x, f, e. domb_e x (concat e f) = (domb_e x e ∨ domb_e x f).
```

[4]Per una trattazione più accurata sui principi di induzione rimandiamo a A.3.

[5]Per push basta osservare che `push e s` = concat (Epsilon s) e.

```
lemma fv_concat:
 ∀f, e, x. fvb_e x (concat e f) = ((fvb_e x e ∧ ¬ domb_e x f) ∨
      fvb_e x f).

lemma fresh_var_concat:
 ∀f, e. fresh_var_e (concat e f) = max (fresh_var_e e )
     (fresh_var_e f).

lemma inb_concat: ∀f, e, x. inb_e x (concat e f) = (inb_e x e ∨
     inb_e x f).
```

Codice 5.6: Lemmi sulla proprietà distributiva delle funzioni di gestione dei nomi di variabile rispetto la funzione concat

La formulazione e la dimostrazione dei lemmi in mostrati 5.6 in linea di principio non era strettamente necessaria, ma è stata introdotta per le seguenti ragioni:

- A questo punto della formalizzazione non sono state introdotte altre funzioni di manipolazione di dati, per cui simili proprietà sarebbero dimostrate su costruttori di tipo i quali non catturano alcun comportamento funzionale. Ciò rende molto più semplice, in linea di principio, l'approccio ad una simile dimostrazione. Per esempio: provare un enunciato del tipo $\forall \mathtt{x}, \mathtt{v}, \mathtt{w}. \mathtt{inb_b\ x\ (AppValue\ v\ w)} = \mathtt{false} \rightarrow (\mathtt{inb_v\ x\ v} = \mathtt{false}) \land (\mathtt{inb_v\ x\ w} = \mathtt{false})$ non sarebbe molto utile dal momento che molto spesso la complessità della dimostrazione sarebbe paragonabile a quella di richiamare il risultato del lemma e di manipolarlo per inserirlo nel goal.

- Dal momento che gli Environment sono liste, provare degli enunciati che quantificano su di essi spesso richiede l'impiego di un principio di induzione anche qualora questi enunciati siano relativamente semplici; nulla vieterebbe di intraprendere la dimostrazione di una sotto-derivazione di un lemma applicando un principio di induzione, ma ciò renderebbe oltremodo più complessa la dimostrazione del lemma principale.

- Proprietà simili sono impiegate molto di frequente all'interno della dimostrazione perché, come abbiamo visto, la natura sequenziale degli Environment è, in un certo senso, la virtù della macchina Crumble GLAM per cui è lecito aspettarsi che questi vengano manipolati molto spesso.

Nondimeno, vedremo in seguito che sarà necessario provare la distributività rispetto a `concat` di altre funzioni più complesse di quelle presentate ora, mentre vedremo che per tali funzioni sarà necessario introdurre, per ragioni simili a quelle appena esposte, proprietà distributive anche per costruttori di tipo come per esempio `AppValue`.

Capitolo 6

Crumbling e read-back

Dal momento che il lavoro di formalizzazione raccolto in [14] non ha preso in esame il comportamento dinamico della macchina, le funzioni di crumbling e read-back sono le più importanti di questa trattazione. Esse svolgono il compito rispettivamente di convertire la rappresentazione tradizionali dei λ-termini in crumbled forms e viceversa: due processi che caratterizzano l'inizializzazione e la terminazione della valutazione. Tuttavia, tali funzioni sono importanti anche per lo studio dinamico del comportamento della Crumble GLAM perché compaiono in molti invarianti che è necessario dimostrare per garantire la correttezza della macchina: per convincersene basta pensare alla formulazione dell'implementation theorem che abbiamo dato in (1). In questa parte del lavoro, dopo avere ripreso in esame le definizioni informali di tali funzioni, ne mostreremo l'implementazione in Matita al fine di evidenziare le eventuali modifiche che abbiamo dovuto apportare alle definizioni per garantirne la terminazione.

Matita, infatti, richiede che le funzioni che vi vengono definite siano terminanti. Tuttavia, come è noto, il problema della terminazione di una procedura non è decidibile; per questo motivo, Matita richiede che le procedure dichiarate siano conformi ad un'approssimazione da sotto della proprietà di terminazione. Questo fatto talvolta impone, nell'ambito della formalizzazione, di definire le funzioni in maniera differente da come avverrebbe informalmente.

Sebbene da un punto di vista *ontologico* la funzione di crumbling sia antecedente alla funzione di read-back e nonostante [5] presenti prima la funzione di crumbling rispetto alla funzione di read-back, nella presente trattazione tratteremo prima la funzione di read-back, poiché, trattandosi di una semplificazione di una crumbled form, è quella che ha richiesto minor sforzo implementativo.

6.1 read-back

Come anticipato, la funzione di read-back, si occupa di ricostituire il pTerm codificato da un Crumble. Sostanzialmente, tale funzione scorre ricorsivamente tutti i crumbled environments presenti nella crumbled form chiamando ricor-

sivamente la funzione di sostituzione su pTerm `p_subst`[1] per reinserire tutti i termini condivisi nei crumbled environments all'interno del termine. La macchina Crumble GLAM, operativamente, si serve di questa funzione solo al termine della valutazione, ossia, quando non è più in grado di semplificare alcun redesso. Da un punto di vista teorico, invece, molti invarianti dell'implementation theorem (1) riguardano prorietà di λ_p-termini ottenuti dalla traduzione di crumbled forms per mezzo della funzione di read-back.

In [5], la funzione di read-back viene espressa mediante il simbolo $\cdot_\downarrow$, dove $\cdot$ rappresenta una crumbled form o un tipo mutuo. In questo contesto utilizzeremo il simbolo $\cdot_\downarrow$ per rappresentare la definizione informale della funzione di read-back, mentre utilizzeremo l'espressione `read_back*` per indicare una delle funzioni mutue che concorrono alla formalizzazione in Matita della $\cdot_\downarrow$. La definizione informale che se ne dà in [5] è la seguente:

$$\begin{aligned} x_\downarrow &\coloneqq x \\ (b,\epsilon)_\downarrow &\coloneqq b_\downarrow \\ (b, e[x \leftarrow b'])_\downarrow &\coloneqq (b,e)_\downarrow\{x \leftarrow b_\downarrow\} \\ (b, e[x \leftarrow b'])_\downarrow &\coloneqq (b,e)_\downarrow\{b \leftarrow_\downarrow\} \\ (\lambda x.c)_\downarrow &\coloneqq \lambda x.c_\downarrow \\ (vw)_\downarrow &\coloneqq (v_\downarrow w_\downarrow) \end{aligned} \tag{6.1}$$

Una traduzione letterale della funzione $\cdot_\downarrow$ in Matita potrebbe essere la seguente:

```
1  let rec read_back c on c :=
2   match c with
3   [ CCrumble b e ⇒ match e with
4                    [ Epsilon ⇒ read_back_b b
5                    | Snoc e1 s ⇒ match s with [ subst x' b1 ⇒ p_subst (read_back ⟨
                         b, e1⟩) (psubst x' (read_back_b b1))]
6                    ]
7   ]
8
9  and read_back_b b :=
10  match b with
11  [ CValue v ⇒ read_back_v v
12  | AppValue v w ⇒ appl (read_back_v v) (read_back_v w)
13  ]
14
15 and read_back_v v :=
16  match v with
17  [ var x ⇒ val_to_term (pvar x)
18  | lambda x c ⇒ val_to_term (abstr x (read_back c))
19  ]
20  .
```

Codice 6.1: Traduzione letterale della (6.1) in Matita

Tuttavia, se si prova a lanciare il comando di compilazione del codice, il risultato a console è il seguente messaggio di errore:

```
NTypeChecker failure: Recursive call (read_back ⟨b,e1⟩), ⟨
    b,e1⟩ is not smaller.
```

[1] Per una trattazione adeguata della funzione `p_subst` rimandiamo il lettore alla A.2.

Il motivo di questo errore è il fatto che, in Matita, le definizioni di funzioni con chiamate ricorsive su termini con un numero di costruttori di tipo annidati maggiori di quelli del parametro formale su cui è chiamata la ricorsione non sono ammesse. Il motivo di questa limitazione è, come anticipato, quello di garantire la terminazione delle funzioni definite. Per una trattazione più specifica del problema si rimanda il lettore alla trattazione fornita da A.2.

Per potere definire la funzione `read_back` si poteva agire in due diversi modi:

- Definire la funzione per induzione, anziché sul Crumble, sulla taglia del Crumble, in modo che la chiamata ricorsiva `(read_back_b b)` fosse ammessa grazie ad una definizione di taglia del Crumble in cui la taglia di un termine `b` fosse strettamente minore di quella di un qualsiasi termine ⟨`b`, `e`⟩. Questa alternativa avrebbe permesso una traduzione quasi letterale della funzione definita in [5] se non per il fatto che anziché essere definita per induzione sul tipo Crumble lo fosse sulla taglia dei termini.

- Modificare la struttura della `read_back` in modo che nel suo corpo non fosse necessario chiamare `read_back ⟨ b,e1⟩`. Del resto, questa strada sembra percorribile dal momento che la ricorsione preponderante nella 6.1 non è tanto quella sul termine `c`, quanto più quella sull'Environment `e`. Dunque, se fra la `read_back` e la `read_back_b` si inserisse una funzione intermedia che va in ricorsione solo sull'Environment `e`, allora sarebbe possibile definire la `read_back`.

Fra le due soluzioni, quella col trade-off migliore sembra la seconda, siccome, come analizzato in A.2, l'introduzione di funzione definite per ricorsione sulla taglia del termine molto spesso comporta un notevole sforzo implementativo. Per questo motivo, data l'intrinseca semplicità nella definizione della $\cdot_\downarrow$ si è preferito lasciar perdere l'analogia della `read_back` con la sua definizione in [5], ma guadagnare in semplicità. Quindi, la funzione adottata nel lavoro di formalizzazione è la seguente:

```
1  let rec aux_read_back rbb e on e :=
2   match e with
3   [ Epsilon ⇒ rbb
4   | Snoc e1 s ⇒ match s with [ subst x' b1 ⇒ p_subst (aux_read_back rbb e1)
        (psubst x' (read_back_b b1))]
5   ]
6
7  and read_back_b b :=
8   match b with
9   [ CValue v ⇒ read_back_v v
10  | AppValue v w ⇒ appl (read_back_v v) (read_back_v w)
11  ]
12
13 and read_back_v v :=
14  match v with
15  [ var x ⇒ val_to_term (pvar x)
16  | lambda x c ⇒ match c with
17                [ CCrumble b e ⇒ val_to_term (abstr x (aux_read_back (read_back_b
                      b) e))]
18  ]
19  .
20
21 let rec read_back c on c :=
```

```
22    match c with
23    [ CCrumble b e ⇒ aux_read_back (read_back_b b) e]
```

Codice 6.2: Implementazione della 6.1 in Matita

Come anticipato, l'introduzione della funzione `aux_read_back` permette di non dovere ricostruire nuovamente il Crumble ⟨`b,e1`⟩: il modo in cui ciò avviene è sostanzialmente quello di chiamare prima la ricorsione sul Bite `b` e solo in seguito quella sull'Environment `e`.

6.2 underline

Mentre il processo di read-back è sostanzialmente una decodifica, quello di underline è una codifica; in quanto tale, esso deve garantire che le crumbled forms così ottenute godano di determinati invarianti che garantiscono il corretto funzionamento della macchina. Per queste ragioni ci aspettiamo che la funzione di crumbling sia più complessa che quella di read-back.

Informalmente, la funzione di crumbling definita dall'articolo è composta da due funzioni mutualmente ricorsive: una per i pValue, indicata con il simbolo $\overline{\cdot}$ ed una per i pTerm, indicata con il simbolo $\underline{\cdot}$. La funzione $\underline{\cdot}$ passa in rassegna il proprio argomento: qualora incontri un pValue si limita a chiamare la funzione mutua $\overline{\cdot}$, mentre se incontra un'applicazione che coinvolge almeno un termine (dunque un'applicazione annidata), "stacca" i termini, effettua una chiamata ricorsiva su ciascuno di essi ed introduce nel crumbled environment un costrutto di ES fra i risultati delle chiamate ricorsive ed una variabile fresca introdotta al posto dei termini di partenza. La funzione $\overline{\cdot}$, passa in rassegna il proprio argomento e se sono variabili, le trascrive, mentre se sono λ-astrazioni trascrive l'argomento e chiama la funzione $\underline{\cdot}$ sul corpo dell'astrazione.

L'ordine in cui non vengono inseriti i costrutti di ES nel Crumble non è casuale, bensì è volto a garantire che in fase di valutazione, percorrendo l'Environment da destra a sinistra si applichi una disciplina di valutazione right-to-left closed CbV. Si prenda ad esempio l'ultima riga della 6.3: i costrutti di ES generati dal termine di destra verranno apposti a destra dell'Environment dimodoché percorrendo la lista da destra verso sinistra vengano valutati per primi.

Allo stesso modo della read-back, quando useremo le simbologie $\overline{\cdot}$ e $\underline{\cdot}$ faremo riferimento alle definizioni date in [5], mentre quando adotteremo le simbologie `overline` e `underline` faremo riferimento alle implementazioni che ne abbiamo dato in Matita. La definizione di $\overline{\cdot}$ e $\underline{\cdot}$ che compaiono [5] sono le seguenti:

$$\begin{aligned} \overline{x} &:= x \\ \overline{\lambda x.t} &:= \lambda x.\underline{t} \end{aligned} \tag{6.2}$$

$$\begin{aligned} \underline{v} &:= (\overline{v}, \epsilon) \\ \underline{vw} &:= (\overline{vw}, \epsilon) \\ \underline{tv} &:= (x\overline{v},\ [x \leftarrow b]) \quad (*) \\ \underline{ut} &:= \underline{ux} \ @ \ ([x \leftarrow b]e) \quad (*) \end{aligned} \tag{6.3}$$

$(*)$ Dove t non è un valore e $\underline{t} = (b, e)$ e x è fresca

Mentre per l'implementazione 6.2 non ci sono particolari problemi, l'implementazione in Matita della $\underline{\cdot}$ risulta un po' più problematica. In primo luogo, allo stesso modo della $\cdot_{\downarrow}$, anche questa definizione non è trascrivibile in Matita: infatti la definizione del caso $\underline{ut}$ prevede che la funzione vada in ricorsione su di un termine che non è strettamente più semplice del parametro formale. Inoltre, occorre implementare dei meccanismi per garantire che la funzione sia in grado di introdurre variabili fresche.

Per rendere la 6.3 definibile in Matita, si può pensare di scomporre il caso $\underline{ut}$ nei sotto-casi $\underline{vt}$ e $\underline{t't}$ ed osservare che $\underline{vt}$ si espanderebbe in $(\underline{vx}, \epsilon)$ @ $([x \leftarrow b]e)$ con le ipotesi $(*)$ su x, b ed e, mentre il caso $\underline{t't}$ si espanderebbe in $\underline{t'x}$ @ $([x \leftarrow b]e)$ sempre con le ipotesi di $(*)$, il quale a sua volta si ridurrebbe in $(x'\overline{x}, [x' \leftarrow b']e')$ @ $([x \leftarrow b]e) \equiv (x'x, [x' \leftarrow b']e'[x \leftarrow b]e)$ in cui ancora una volta valgono le ipotesi di $(*)$ su x', b' ed e'. Le espansioni appena presentate, a differenza della 6.3, non presentano chiamate ricorsive su termini non strettamente più piccoli di quelli del chiamante, per cui è possibile riscrivere la 6.3 come riportato nella 6.4.

$$
\begin{aligned}
\underline{v} &:= (\overline{v}, \epsilon) \\
\underline{vw} &:= (\overline{vw}, \epsilon) \\
\underline{tv} &:= (x\overline{v},\ [x \leftarrow b]) \quad (*) \\
\underline{vt} &:= (\overline{v}x,\ [x \leftarrow b]) \quad (*) \\
\underline{t't} &:= (x'x, [x' \leftarrow b']e'[x \leftarrow b]e) \quad (*)
\end{aligned}
\tag{6.4}
$$

$(*)$ Dove t non è un valore e $\underline{t} = (b, e)$ e x è fresca

A questo punto, per poter implementare la 6.4 in Matita, rimane solo da identificare un meccanismo per generare variabili che siano sempre fresche. Per farlo, facciamo ricorso alla definizione di FRV che abbiamo dato in 5.4: si osserva che dato un qualsiasi termine t, se denotiamo g una generica trasformazione di t tale che $g(t) = t' \wedge Var(t') = Var(t) \cup \{\nu FRV(t)\}$, allora $FRV(t') = S\ (FRV(t))$. Più informalmente: se modifichiamo un termine t in modo che all'insieme delle variabili che vi compaiono, sia aggiunta la variabile di indice $FRV(t)$, allora la FRV del termine così ottenuto sarà la FRV di t incrementata di uno.

Grazie a questa intuizione, dal momento che la `underline` si comporta come la g di cui sopra, si potrebbe quindi pensare di implementare la `underline` in modo che abbia la seguente signature: `pTerm` $\times$ `nat` $\rightarrow$ `Crumble` $\times$ `nat`, dove i secondi elementi delle coppie sono gli indici progressivi della FRV del Crumble che si sta costruendo. Così, ad ogni passaggio si potrà calcolare l'indice di una variabile fresca a partire dagli indici calcolati dalle chiamate ricorsive, inserirla all'interno del Crumble che si sta generando e determinare l'indice della nuova variabile fresca da restituire in output. Formalmente questa precondizione andrà espressa mediante un'ipotesi da vormulare ogniqualvolta, nell'enunciato di un teorema o all'interno di altre funzioni, si faccia uso della funzione `underline`.

Rimane solo un ultimo piccolo dettaglio da formalizzare: come decidere se l'argomento della `underline` è una t od una v. Una soluzione semplice potrebbe

essere quella che prevede, dato `r` un generico pTerm, di fare `match u` e, nel ramo `appl s s'`, allora comportarsi come nel caso t, mentre nel ramo `val_to_term w` comportarsi come nel caso v. Questo metodo, nonostante introduca dei sottotermini che non sono immediatamente necessari alla chiamata della funzione (quelli che nell'esempio più sopra sono `s`, `s'` e `w`), è implementabile in maniera semplice all'interno della `underline` e non richiede la definizione di nessuna funzione ausiliaria esterna.

Date queste considerazioni, la $\underline{\cdot}$ e la $\overline{\cdot}$ sono facilmente implementabili in Matita come segue:

```
1  let rec underline (t: pTerm) (s: nat): Crumble × nat :=
2   match t with
3   [ val_to_term v ⇒ match overline v s with
4     [ mk_Prod vv n ⇒  mk_Prod Crumble nat ⟨(CValue vv), Epsilon⟩ n]
5   | appl t1 t2 ⇒ match t2 with
6     [ val_to_term v2 ⇒ match t1 with
7       [ val_to_term v1 ⇒ match overline v1 s with
8         [ mk_Prod vv n ⇒ match overline v2 n with
9           [ mk_Prod ww m ⇒ mk_Prod Crumble nat ⟨AppValue (vv) (ww), Epsilon⟩ m ]
10        ]
11      | appl u1 u2 ⇒ match underline t1 s with
12        [ mk_Prod c n ⇒ match c with
13          [ CCrumble b e ⇒ match overline v2 n with
14            [ mk_Prod vv m ⇒ mk_Prod Crumble nat ⟨AppValue (var ν(m)) (vv), push e
                   [(ν(m)) ← b]⟩ (S m)]
15          ]
16        ]
17      ]
18    | appl u1 u2 ⇒ match underline t2 s with
19      [ mk_Prod c n ⇒ match c with
20        [ CCrumble b1 e1 ⇒ match t1 with
21          [ val_to_term v1 ⇒ match overline v1 n with
22            [ mk_Prod vv m ⇒  mk_Prod Crumble nat (at ⟨AppValue (vv) (var (νm)),
                   Epsilon⟩ (push e1 [νm←b1])) (S m)]
23          | appl u1 u2 ⇒ match underline t1 n with
24           [ mk_Prod c1 n1 ⇒ match c1 with
25             [ CCrumble b e ⇒ mk_Prod Crumble nat ⟨AppValue (var (ν(S(n1)))) (var
                    (νn1)), concat (push e [ν(S(n1)) ← b]) (push e1 [νn1 ← b1])⟩ (S
                    (S (n1)))]
26           ]
27          ]
28        ]
29      ]
30    ]
31  ]
32
33 and overline (x:pValue) (s: nat): Value × nat :=
34  match x with
35  [ pvar v ⇒ mk_Prod Value nat (var v) s
36  | abstr v t ⇒ match underline t s with
37    [ mk_Prod c n ⇒ mk_Prod Value nat (lambda (v) (c)) n  ]
38  ]
39 .
```

Codice 6.3: Formalizzazione delle funzioni $\underline{\cdot}$ e $\overline{\cdot}$ in Matita.

Dal momento che la funzione `underline` è una funzione di codifica, ci aspettiamo che il suo output debba godere di alcune proprietà di ben-formatezza essenziali per garantire il corretto funzionamento della macchina Crumble GLAM. Siccome l'aspetto peculiare del crumbling è l'inserimento di ES raggruppati in crumbled environments, è anche lecito aspettarsi che molte di queste proprietà

riguardino questi costrutti. In [5] una delle più importanti proprietà riguardanti la ben-formatezza delgi Environment è catturata dalla definizione informale di well-namedness.

Definizione 6.2.1 (well-namedness). Un Crumble c o un Environment e è well-named se tutte le variabili che occorrono a sinistra di un costrutto di ES al di fuori di un'astrazione sono a due a due distinte.

Se vogliamo dare un'interpretazione intuitiva alla condizione di well-namedness, alla luce del ruolo che hanno i nomi di variabile nel contesto delle crumbling-abstract-machines, possiamo asserire innanzitutto che la parte della definizione che richiede che i nomi siano distinti fra di loro è volta a garantire che l'interpretazione di una variabile in un ambiente non sia ambigua, ossia che ad una medesima locazione di memoria sia associato al più un termine ala volta. La seconda parte della definizione, quella che parla delle variabili sotto astrazione, garantisce che la disciplina di valutazione non effettui riduzioni sotto astrazioni, coerentemente con quanto richiesto dalla disciplina di valutazione dei λ_p-termini in 3.2.

Tuttavia questa formulazione della proprietà di well-namedness, non è abbastanza dettagliata per permetterci di concludere i risultati attesi: è stato dunque necessario aggiungere alcuni lemmi che descrivessero con maggior dettaglio la distribuzione dei nomi di variabile freschi introdotti dalla `underline`.

Alcuni utili proprietà sono per esempio quelle enunciate dai seguenti lemmi:

```
lemma line_monotone_names:
 (∀t.∀s.  snd … (underline t s)  s) ∧
  (∀v.∀s. snd … (overline v s)  s).

lemma line_names:
 (∀t.∀s. s  fresh_var_t t → snd … (underline t s)  fresh_var
     (fst … (underline t s))) ∧
  (∀v.∀s. s  fresh_var_tv v → snd … (overline v s)  fresh_var_v
      (fst … (overline v s))).
```

Codice 6.4: Caratterizzazioni del secondo valore di ritorno della `underline`

Questi due lemmi servono a garantire alcune proprietà sul secondo valore di ritorno della funzione: il primo dimostra che il secondo parametro della `underline` è crescente in senso lato, mentre il secondo che il secondo valore restituito dalla `underline` è una variabile fresca nel Crumble restituito come primo valore. Questi due lemmi sono un'importante verifica della correttezza dell'invariante sul secondo termine restituito dalla underline esposto precedentemente, dal momento che verificano formalmente che la funzione `underline` inserisce sempre variabili fresche nel termine.

Inoltre, tali risultati sono indispensabili per dimostrare la precondizione sul secondo argomento della funzione `underline` nelle ipotesi induttive quando si procede per induzione nella dimostrazione dei lemmi sulla `underline`: siccome la funzione `underline` richiede la precondizione `fresh_var t` $\leq$ `s`, quando

si formula un lemma sarà necessario elencarla fra le ipotesi, per questa ragione, quando si procederà per induzione, dimostrare tale precondizione sarà indispensabile per poter usare le ipotesi induttive.

Infine, per il secondo lemma, riusciamo ad intuire che tutti gli environment generati dalla `underline` hanno domini fra loro disgiunti. Un'ulteriore conferma di questo risultato (che però non verrà mai esplicitato e dimostrato formalmente) è data dai seguenti risultati, i quali asseriscono che, come aspettato, le variabili appartenente al dominio del crumbling di un termine sono comprese fra il paramento `s` fornito alla `underline` ed il secondo valore della coppia restituita dalla funzione.

```
definition interval_dom :=  λe, b. ∀x. domb_e (νx) e =true → b
    ≤ x.
definition bound_dom := λe, b. ∀x. domb_e  (νx) e = true → S x
    ≤ b.

lemma line_dom:
 (∀t. ∀s. (interval_dom match (fst … (underline t s)) with
     [CCrumble c e ⇒ e] s)).

lemma env_bound_lemma:
 (∀t. ∀s. (bound_dom match (fst … (underline t s)) with
     [CCrumble c e ⇒ e] (snd … (underline t s)))).
```

Codice 6.5: Bounds per i domini ottenuti mediante crumbling

Una volta dimostrati questi risultati, è stato finalmente possibile intraprendere la dimostrazione degli invarianti di inizializzazione formulati da [5] riguardanti la `read_back` e la `underline`.

6.3 Lemmi su `underline` e `read_back`

6.3.1 Lemma 4.2

La prima proposizione che [5] presenta su `underline` e `read_back` è, senza dubbio, uno dei risultati più importanti per l'inizializzazione della macchina virtuale Crumble GLAM: essa asserisce che la funzione $\cdot_\downarrow$ è l'inversa sinistra di $\underline{\cdot}$. In questo modo viene assicurato che lo stato iniziale goda di una delle proprietà fondamentali per la correttezza della macchina Crumble GLAM: ossia quella che il Crumble su cui la macchina opera, se ritradotto in un λ_p-termine sia esattamente il termine di partenza.

Lemma 6.3.1 (4.2)
$\forall t. \underline{t}_\downarrow = t \wedge \forall v. \overline{v}_\downarrow = v$

Per facilitare il confronto con [5], specifichiamo accanto all'intestazione del lemma il numero di riferimento in [5], in questo caso: 4.2.

La dimostrazione del lemma 6.3.1 in [5] ha richiesto la dimostrazione dei lemmi tecnici che seguono.

Definizione 6.3.1. $c\#e \coloneqq DOM(e) \cap FV(c) = \emptyset$

Lemma 6.3.2 (C.4)
$\forall b, c, e, x.x \notin DOM(e) \cup FV(e) \to (c@[x \leftarrow b]e)_\downarrow = (c@e)_\downarrow\{x \leftarrow (b, e)_\downarrow\}$

Lemma 6.3.3 (C.5)
$\forall c, e.c\#e \to (c@e)_\downarrow = c_\downarrow$

Lemma 6.3.4 (C.6)
$\forall c, e.c\#e \to (c@[x \leftarrow b]e)_\downarrow = c_\downarrow\{x \leftarrow (b, e)_\downarrow\}$

Il lemma C.6 (6.3.4) non è stato dimostrato perché la dimostrazione ne faceva uso solo nel caso del termine `if t then u else s`, rimosso dalla grammatica dei termini.

Il lemmi C.4 e C.5 non sono stati dimostrati alla lettera a causa delle differenze implementative fra le definizioni di $\cdot_\downarrow$ e $\underline{\cdot}$ con le rispettive formalizzazioni `read_back` e `underline`, ma sono stati dimostrati lemmi che avessero simile potere espressivo. I lemmi in 6.6 hanno un ruolo simile a quelli di C.4 e C.5; infatti, dal momento che la `read_back` è stata definita come wrapper della funzione `aux_read_back` (cfr. 6.2), non avrebbe avuto senso dimostrare il C.5 come formulato sopra siccome, nell'espansione della `read_back`, non compaiono chiamate ricorsive alla `read_back` stessa, bensì alla `aux_read_back`. Per queste ragioni, `stronger_aux_read_back3` può essere visto come l'analogo del lemma C.5 in cui al posto della `read_back` compare la `aux_read_back`. Similmente, il `push_lemma` in 6.6 è, nella formalizzazione, l'analogo del C.4 per le medesime ragioni di cui sopra.

```
lemma stronger_aux_read_back3: ∀e, t.
  (∀x. (domb_e x e = true → fvb_t x t = false)) →
     aux_read_back t e = t.

lemma push_lemma:
 ∀t, e, x, b. aux_read_back t (push e (subst x b)) =
     aux_read_back (p_subst t (psubst x (read_back_b b)) e.
```

Codice 6.6: Lemmi analoghi a C.4 e C.5

Provare il lemma `stronger_aux_read_back3` non è scontato: siccome la `read_back` agisce tramite chiamate alla `p_subst`, è necessario che la funzione di sostituzione su termini goda della proprietà 6.5 in senso stretto. Infatti, mentre nell'ambito teorico l'uguaglianza di termini è spesso intesa modulo `alpha`-conversioni, per ragioni di efficienza in [5] le uguaglianze dei termini debbono valere in senso stretto. Ma, come evidenziato in A.2, la 6.5 non vale in senso stretto con la definizione canonica di sostituzione (si prenda ad esempio A.2). Per questo motivo, per poter provare il lemma `stronger_aux_read_back3`, è stato necessario ridefinire la `p_subst` dimodoché non presentasse questa anomalia. Per dettagli specifici sull'implementazione della `p_subst`, rimandiamo il lettore a A.2.

$$\forall t, t', x.x \notin FV(t) \to t\{x \leftarrow t'\} = t \tag{6.5}$$

Come riportato in 6.2, la definizione della funzione `underline` differisce da quella della $\underline{\cdot}$ perché altrimenti non sarebbe stato possibile implementarla in Matita. In particolare, l'ultimo caso induttivo è stato riscritto espandendo la $\underline{\cdot}$ fino a rendere il termine della chiamata ricorsiva sufficientemente semplice per permetter alla definizione della funzione di essere accettata da Matita. Le modifiche hanno riguardato il caso $\underline{ut}$, scomponendolo nei sotto-casi $\underline{vt}$ e $\underline{t't}$. Queste modifiche hanno chiaramente indotto un'asimmetria fra le definizioni di `underline` e di $\underline{\cdot}$. Per questa ragione, molti risultati che nella definizione di $\underline{\cdot}$ sarebbero stati disponibili come ipotesi induttive su chiamate ricorsive sono stati dovuti ricostruire per mezzo di appositi lemmi tecnici. Nel caso specifico del lemma 4.2, mentre il caso $\underline{vt}$ è analogo a quello $\underline{tv}$, il caso $\underline{t't}$ ha richiesto la definizione di due lemmi tecnici per garantire che la concatenazione dei due environment generati dalle chiamate ricorsive non creasse interferisse durante la `read_back`, riportati in 6.7.

```
lemma ultra_concat_lemma:
  (∀f, x, e. domb_e x f = false →
    (\foralz. domb_e z f = true → fvb_e z e = false) →
     (aux_read_back (val_to_term (pvar x)) (concat e f)
  =aux_read_back (val_to_term (pvar x)) e)).

lemma iper_concat_lemma: ∀f, e, x.
  domb_e x e = false →
  (aux_read_back (val_to_term (pvar x)) (concat e f)
  =aux_read_back (val_to_term (pvar x)) f).
```

Codice 6.7: Lemmi aggiuntivi a C.4 e C.5 per il caso $\underline{t't}$

Per la dimostrazione formale del lemma 6.3.1, non è stato necessario introdurre altri significativi risultati (per comodità la dimostrazione formale fa uso del primo risultato del remark 6.3.6, ma la tesi sarebbe dimostrabile anche senza), al contrario è stato necessario introdurre e dimostrare un lemma riguardante la distributività dell'operazione di `read_back` senza il quale non è nemmeno possibile sfruttare le ipotesi immediate sui due sottotermini di un'applicazione. Tale lemma è così semplice che nelle dimostrazioni di [5] lo si dà per scontato. In effetti è intuitivamente vero che, all'interno di un medesimo ambiente, la `read_back` commuti con l'applicazione, ma giustificarlo con la definizione formale di `read_back` non è cosa semplice.

Lemma 6.3.5
distributività della `read_back`
∀ `t`, `u`. `read_back` (`appl t u`) `e` = `appl` (`read_back t e`) (`read_back u e`)

Dimostrazione. Per induzione su `e`

- Caso `Epsilon`, triviale
- Caso `e = Snoc e' [x←b]`

```
aux_read_back (appl t u) (Snoc e' [x←b])  =
p_subst (aux_read_back (appl t u) e') (pSubst x (read_back_b b)) =
```

```
p_subst (appl (aux_read_back t e')
              (aux_read_back u e')) (pSubst x (read_back_b b))
```

A questo punto ci sia aspetterebbe di poter concludere utilizzando proprietà distributiva della funzione `p_subst` per procedere come segue:

```
p_subst (aux_read_back (appl t u) e') (pSubst x (read_back_b b)) =
appl (p_subst (aux_read_back t e') (pSubst x (read_back_b b)))
     (p_subst (aux_read_back t e') (pSubst x (read_back_b b))) =
appl (aux_read_back t (Snoc e' [x←b]))
     (aux_read_back u (Snoc e' [x←b]))
```

Tuttavia la dimostrazione della proprietà distributiva della `p_subst` non è triviale, perché, per ragioni di definibilità, la `p_subst` è stata definita per induzione sulla taglia e mediante l'uso di Σ-tipi, per cui la dimostrazione di tale proprietà è appesantita da numerosi passaggi estremamente tecnici volti ad evitare problemi di tipi-dipendenti in fase di riduzione. Un'analisi più esauriente di questi fenomeni è fornita in A.2, dove sono anche presentati gli enunciati dei lemmi `p_subst_distro` e `p_subst_bound_irrelevance` (rispettivamente in A.11 e in A.12) necessari ricorsivamente per la dimostrazione del lemma 6.3.5.

Infine, da un punto di vista estremamente pratico, la dimostrazione del lemma 4.2 ha richiesto l'uso di tecniche di riduzione mirata: infatti, dal momento che la `read_back` effettua chiamate alla `p_subst` che a sua volta è zucchero sintattico per invocare la `p_subst_sig` con gli opportuni parametri, la taglia del goal, se normalizzato completamente, può crescere a tal punto da rendere impraticabile l'uso del proof-assistant. Un'accurata analisi della definizione della `p_subst_sig` e dei motivi per cui può causare fenomeni di esplosione della taglia dei goal è data in A.2, mentre per un rapido compendio delle tecniche di riduzione mirata e dei loro casi d'uso tipici, rimandiamo il lettore a A.4.

6.3.2 Remark 4.1

Per la dimostrazione dei lemmi 6.3.2 e 6.3.3 e di molti altri risultati che presenteremo in seguito, [5] fa uso dei risultati del Remark 4.1. Tali risultati riguardano soprattutto la gestione dei nomi di variabile; per i motivi precedentemente citati riguardanti il ruolo dei nomi di variabile nelle implementazioni del λ-calcolo con ambienti globali, in [5] tutte le uguaglianze fra termini, ove non specificato, valgono e vanno dimostrate in senso stretto. Inoltre, dal momento che la gestione dei nomi è del tutto trascurata in [5], la verifica della validità o meno di questi lemmi è stata molto importante per verificare l'efficacia dei meccanismi di gestione dei nomi di variabile che abbiamo definito. Dualmente, la gestione formale di nomi di variabili, lo vedremo, ci ha permesso di evidenziare che alcuni di questi risultati non sono validi.

Lemma 6.3.6 (Remark 4.1)

1. $(\forall t.FV(\underline{t}) = FV(t)) \wedge (\forall v.FV(\overline{v}) = FV(v))$
2. $(\forall b.FV(b_{\downarrow}) = FV(b)) \wedge (\forall c.FV(c_{\downarrow}) = FV(c))$

3. *"The crumbling translation commutes with the renaming of free variables.", che può essere enunciato come segue:*
   $\forall t, x, y. x \in FV(t) \to \underline{t\{x \leftarrow y\}} = \underline{t}\{x \leftarrow y\}$

4. *"The crumbling translation and the readback map values to values"*

Remark 4.1.1

Informalmente, questo risultato deriva dall'osservazione che, per come stata definita la funzione di `underline` in 6.3, ogniqualvolta venga aggiunta una variabile fresca `x`, questa viene immediatamente legata con l'aggiunta di una Substitution [`x` $\leftarrow$ `b`] nell'Environment immediatamente più esterno ad essa. Questa proprietà, dunque, garantisce che il processo di crumbling conservi i nomi delle variabili libere e legate e inoltre concorre a garantire che il processo di crumbling sia coerente: laddove viene introdotto un riferimento ad un termine condiviso, questo è effettivamente presente in memoria.

Per poter dimostrare tale risultato, tuttavia, è stato necessario introdurre la proposizione 6.6. Infatti, i vincoli espressi dai lemmi in 6.5 non forniscono particolari indicazioni sul fatto che le variabili nel dominio di un ambiente possano o meno essere libere.

$$\underline{t} = (b, e) \to (FV(e) \setminus DOM(e) = FV(e)) \tag{6.6}$$

Però, per provare la 6.6 è necessario avere informazioni riguardanti le variabili libere di un ambiente costruito dalla `underline`, per farlo è necessario conoscere l'insieme delle variabili libere su ciascun suo termine dopo la `underline`, ma questo risultato è il remark 4.1.1. Quindi, curiosamente, non c'è modo di provare la 6.6 se non congiuntamente ad un enunciato almeno tanto espressivo come il remark 4.1.1. Per questo motivo, il remark 4.1.1 è stato dato come banale corollario del lemma in 6.8.

```
1  lemma disjoint_dom:
2   (∀t, s, x. fresh_var_t t ≤ s →
3     (fvb_t x t = fvb x (fst ? ?(underline t s))) ∧
4     (fvb_e x match (fst … (underline t s)) with [CCrumble c e ⇒ e]=true →
5     domb_e x match (fst … (underline t s)) with [CCrumble c e ⇒ e]=false)) ∧
6    (∀v.∀s.∀x. (s  fresh_var_tv v) → fvb_tv x v = fvb_v x (fst ? ? (overline v s))).
```

Codice 6.8: Enunciato del lemma `disjoint_dom`

Remark 4.1.2

Il remark 4.1.2 è errato. L'errore deriva dal fatto che la definizione di variabile fresca utilizzata nell'articolo sia una definizione *sintattica* anziché *semantica*. Infatti, per la definizione di FV data in 5.2, l'insieme delle variabili libere di un Environment contiene anche le variabili che non trovano posto nel Crumble finale. In altre parole, essa tiene conto di tutte le variabili libere presenti nell'ambiente di memoria, senza curarsi del fatto che nel termine ci siano o meno puntatori alle locazioni di memoria in cui si trovano tali variabili.

Esempio 6.3.1.

$$FV((x, [y \leftarrow z])) = \{x, z\}$$
$$FV((x, [y \leftarrow z])_{\downarrow}) = FV((x, \epsilon)) = \{x\}$$

Per porre rimedio all'inconsistenza generata dalla falsità di questo enunciato, è bastato osservare che tutte le volte in cui in [5] veniva fatto uso di questo risultato, era per derivare la 6.7 e usare la contronominale di uno dei due congiunti; per questa ragione, non avrebbe avuto senso ridefinire la funzione FV perché aderisse alla nozione semantica di libertà, perciò è stato scelto scelto di limitarsi a dimostrare la 6.7.

$$(\forall b.FV(b_{\downarrow}) \subseteq FV(b)) \wedge (\forall c.FV(c_{\downarrow}) \subseteq FV(c)) \tag{6.7}$$

Remark 4.1.3

Il remark 4.1.3, se interpretato con l'uguaglianza intesa in senso stretto come richiesto da [5], è errato: per convincersene basta guardare l'esempio 6.3.2.

Esempio 6.3.2.

$$\underline{(\lambda x.y)((\lambda x.y)(\lambda x.y))} = ((\lambda x.(y, \epsilon))z, [z \leftarrow (\lambda x.(y, \epsilon))(\lambda x.(y, \epsilon))])$$
$$\underline{(\lambda x.y)((\lambda x.y)(\lambda x.y))\{y \leftarrow z\}} = ((\lambda x.(z, \epsilon))w, [w \leftarrow (\lambda x.(z, \epsilon))(\lambda x.(z, \epsilon))])$$

Come si vede dall'esempio, infatti, l'operazione di $\underline{\cdot}$, dovendo generare variabili che siano fresche, deve tenere conto del valore dei nomi delle variabili già presenti all'interno del termine per determinare il valore delle variabili introdotte nei costrutti ES.

In [5] il remark 4.1.3 veniva usato per dimostrare alcuni invarianti della riduzione: le proposizioni 5.5.3 e 5.5.4 per la versione open call-by-value della Crumble GLAM e le omologhe per la versione closed call-by-value[2]. Riportiamo le formulazioni che ne sono state date e le loro dimostrazioni informali per evidenziarne possibili soluzioni.

Lemma 6.3.7 (5.5.3)
Tutti i corpi di funzione in un Crumble c' raggiungibile durante la valutazione della macchina Crumble GLAM sono sottotermini del Crumble c iniziali a meno di ridenomina.

Dimostrazione. Le regole $\mapsto_{sub_{var}}$ e $\mapsto_{sub_l}$ possono copiare astrazioni, ma dal momento che le astrazioni erano già nell'ambiente, il claim segue direttamente dall'ipotesi induttiva. Per il caso della $\mapsto_{\beta_v}$, la regola può copiare un corpo di una funzione e ridenominarlo, ma dal momento che per il remark 4.1.3 la traduzione si commuta con la rinomina delle variabili fresche operata dalla funzione $\cdot^{\alpha}$, il risultato segue direttamente dall'ipotesi induttiva.

[2] Per l'enunciato completo si faccia riferimento a 7.0.1.

In questo caso il claim continua ad essere vero siccome la ridenomina di variabili libere si commuta con la traduzione di termini modulo α-conversioni. Purtroppo, un simile risultato è abbastanza difficile da formalizzare e dimostrare. Una formulazione da un potere espressivo più o meno equivalente, tuttavia è stata identificata e dimostrata in 7.2.3.

Lemma 6.3.8 (5.5.4, Weak contextual decoding)
Tutte le decomposizioni $C\langle(b, e_v)\rangle$ di Crumble raggiungibili dalla Crumble GLAM sono tali che tutti i prefissi C' di C siano right-v evaluation context.

Dimostrazione. ... Nel caso dell'applicazione della regola $\rightarrow_{\beta_v}$, il claim deriva dal fatto che, per il remark 4.1.3, ogni c^α sia un sottotermine del termine di partenza a meno di operazioni di ridenomina e dunque dal fatto che la ridenomina non vìoli i right-v evaluation contexts, dal momento che tutti i sotto-termini di partenza si decodificano in right-v evaluation context per il lemma del contextual decoding. ...

Anche in questo questo caso l'enunciato, continua ad essere vero: infatti, sia la ridenomina di variabili, che l'α-conversione di termini conservano i right-v evaluation context. Purtroppo, scegliere di ridimostrare questo risultato date le due precedenti osservazioni avrebbe richiesto l'introduzione di lemmi e di nozioni non banali come quella di α-convertibilità fra due termini ed il lemma di conservazione dei right-v evaluation context modulo ridenomine e conversioni. Per questo motivo, è stato scelto, in 7.2.4, di dimostrare l'enunciato direttamente.

Remark 4.1.4

Il remark 4.1.4 è triviale in quanto è una banale conseguenza del tipaggio. Sebbene questo risultato possa sembrare curioso, proviamo a darne una giustificazione teorica. Come asserito in [8] o in [12], un sistema di tipi può essere visto come un sistema di approssimazione di proprietà non decidibili. Per esempio, nei più comuni linguaggi di programmazione, come ad esempio C, i sistemi di tipi sono implementati al fine di evitare errori a tempo di esecuzione causati dall'errata interpretazione di dati (`int` con `char` e così via). Collateralmente, ogni sistema di tipo approssima esattamente la proprietà decidibile di appartenere ad un determinato tipo. Formalizzando le funzioni di [5], come visto in 4, siamo stati costretti a definire dei tipi di dati astratti e di conseguenza a tipare le funzioni di manipolazione. In questo modo, il sistema di inferenza di tipi del dimostratore interattivo Matita è stato in grado di garantire che le definizioni delle funzioni (nel caso specifico del remark 4.1.4, le funzioni $\cdot_\downarrow$ e $\bar{\cdot}$) non causassero errori a tempo di esecuzione e, collateralmente, che la funzione `read_back_v` avesse tipo pValue e che la funzione `overline` avesse tipo Value, dimostrando il remark.

Nonostante questa peculiarità, questo remark è di notevole importanza da un punto teorico: come abbiamo visto in 3.2, la disciplina di valutazione CbV richiede che gli argomenti delle applicazioni siano valori (i.e. astrazioni chiuse); se l'enunciato non fosse valido, la macchina Crumble GLAM non potrebbe implementare correttamente tale disciplina.

6.3.3 Lemma 4.5

Il lemma 4.5 ha lo scopo di garantire alcune proprietà fondamentali della funzione `underline`, mediante le quali si garantisce che lo stato iniziale della macchina Crumble GLAM goda degli invarianti necessari per il suo corretto funzionamento. La prima di queste proprietà è quella di well-namedness (espressa in 6.2.1) che garantisce che le variabili introdotte dal processo di crumbling siano distinte nel dominio del medesimo ambiente; in questo modo è garantito che ad uno stesso nome di variabile introdotto durante la fase di crumbling sia associato uno ed un solo ES.

Prima di presentare l'enunciato del lemma, mostriamo in 6.9 la formalizzazione che abbiamo dato del predicato di well-namedness e lo commentiamo.

```
1 lemma well_named_alpha:
2  ∀f, b, e. ∀n. fresh_var (at ⟨b, e⟩  f) ≤ n →
3   match (at ⟨b, e⟩ f) with [ CCrumble b e ⇒ ∀H. (w_well_named (pi1 … (alpha b e n
        H))=true) ∧ interval_dom match (pi1 … (alpha b e n H)) with [CCrumble b e ⇒
        e] n].
```

Codice 6.9: Definizione della funzione booleana `well_named`

La funzione implementa chiaramente la definizione di well-namedness, infatti, dato un Crumble ci si assicura che le variabili del dominio del suo Environment `e` siano distinte, poi, mediante le funzione `well_named_b` e `well_named_e`, si cercano altri Crumble sotto λ-astrazioni per reiterare l'applicazione del predicato di well-namedness.

Lemma 6.3.9 (4.5)

1. *Freshness:* $\forall t.well_named(\underline{t})$
2. *Closure:* $\forall t.FV(t) = \emptyset \rightarrow FV(\underline{t}) = \emptyset$
3. *Disjointedness:* $\forall t, C, b, e.(\underline{t}) = C\langle b, e\rangle \rightarrow DOM(C) \cap FV(B) = \emptyset$
4. *Bodies: tutti i corpi di funzione in* $\underline{t}$ *sono la traduzione di termini.*
5. *Contextual decoding:* $\underline{t} = C\langle c\rangle \rightarrow rv_context(C_{\downarrow})$

Lemma 4.5.1

La dimostrazione di questo enunciato non ha richiesto particolari accorgimenti, il risultato deriva dal fatto che la funzione `underline`, per come è stata costruita, inserisca sempre variabili fresh nell'Environment. Questo fatto è una diretta conseguenza del lemma `line_names` in 6.4. L'invariante di well-namedness garantisce che l'operazione di look-up di una variabile qualsiasi nell'Environment non sia ambigua e quindi contribuisce a dimostrare che, come richiesto dall'implementation theorem, la macchina Crumble GLAM sia deterministica.

Lemma 4.5.2

La proprietà di closure è un banale corollario del remark 4.1.1 (6.3.6), la necessità di introdurre questo lemma deriva dal fatto che, in [5], la prima crumbling abstract machine presentata, la Crumble GLAM, sia definita con un insieme di regole di transizione volte ad implementare una disciplina di valutazione weak closed CbV. Secondo tale strategia, un'applicazione può essere valutata solo se l'argomento è un'astrazione chiusa. Chiaramente, se questo lemma non fosse rispettato potrebbero verificarsi casi in cui nel Crumble su cui lavora la macchina sia possibile operare β-riduzioni che non sono possibili nel corrispondente pTerm o viceversa, rompendo uno degli invarianti dell'implementation thoerm (1).

Successivamente, in [5], vengono introdotte altre macchine (nel caso specifico la Open Crumble GLAM) che applicano diverse discipline di valutazione. Nel caso specifico della Open Crumble GLAM, la disciplina di valutazione adottata è quella weak open CbV per cui l'ipotesi di chiusura del termine iniziale, propria delle discipline di valutazione closed, non è più necessaria.

Lemma 4.5.3

Questo risultato è molto tecnico e difficilmente si riesce a fornirne un'interpretazione intuitiva soddisfacente. Nonostante ciò, proviamo comunque a formularne una: dalla definizione dell'operazione di read-back in 6.2 e dalle considerazioni date in 3.2 riguardo la valutazione dei termini, vediamo che entrambe le operazioni procedono percorrendo gli Environment da destra verso sinistra. Questo risultato, dunque, concorre con la sua formulazione dinamica a garantire che sia in fase di valutazione che di ritraduzione, i costrutti di ES che si trovano a sinistra di un Bite b condiviso siano ininfluenti nella sua valutazione e nella sua ritraduzione. Infatti nessuna variabile libera nel Bite b punta ad un termine condiviso che deve ancora essere preso in esame. Dunque questo risultato, ancora una volta, rafforza il fatto che la disciplina di valutazione della Crumble GLAM sia right-to-left.

Lemma 4.5.4

Questo lemma fa riferimento contemporaneamente a due accezioni: il primo è simile a quella del remark 4.1.4 e vuole garantire la corrispondenza fra le due grammatiche dei tipi Crumble (4.2) e pTerm (4.1) che avevamo informalmente enunciato quando le abbiamo introdotte. La seconda accezione di questo lemma vuole dimostrare che la funzione di traduzione sia in un certo senso fedele, ossia che non inserisca nel Crumble finali sottotermini che non fossero già presenti nel termine di partenza.

Nel contesto della formalizzazione, la prima accezione del lemma è triviale dal momento che la definizione formale della funzione `underline` ha fatto sì che tutti i dati venissero tipati, per questo motivo, dal momento che essa ha come tipo di ritorno Crumble e per la definizione del tipo stesso, tutti i corpi di funzione sono Crumble e per la definizione della `underline`, l'unico modo per generare un corpo di funzione è mediante la traduzione di un Term.

Per quanto riguarda la seconda accezione con cui il lemma viene utilizzato all'interno di [5], la prova di questo risultato si riduce con la verifica del fatto tutti i corpi di funzione generati dalla `underline` siano sottotermini del termine di partenza. Per fare ciò, è stato necessario definire due predicati mutui in grado di sintetizzare la relazione di essere sottotermine di un pTerm e di un pValue e cinque predicati per la relazione analoga relativa alla grammatica dei Crumble. Tale definizione è stata data come segue:

```
1  let rec subt t1 t2 on t2 :=
2   match t2 with
3   [ val_to_term v ⇒ subt_v t1 v
4   | appl u1 u2 ⇒ t1 = u1 ∨ t1 = u2 ∨ subt t1 u1 ∨ subt t1 u2
5   ]
6
7  and subt_v t v on v :=
8   match v with
9   [ pvar x ⇒ False
10  | abstr x t1 ⇒ t = t1 ∨ subt t t1
11  ]
12 .
13
14 definition tint := λt1.λt2. t1=t2 ∨ subt t1 t2.
15
16 let rec subc c d on d :=
17  match d with
18  [ CCrumble b e ⇒ subc_b c b ∨ subc_e c e]
19
20 and subc_b c b on b :=
21  match b with
22  [ CValue v ⇒ subc_v c v
23  | AppValue v w ⇒ subc_v c v ∨ subc_v c w
24  ]
25
26 and subc_v c v on v :=
27  match v with
28  [ var x ⇒ False
29  | lambda x d ⇒ c = d ∨ subc c d
30  ]
31
32 and subc_e c e on e :=
33  match e with
34  [ Epsilon ⇒ False
35  | Snoc e s ⇒ subc_s c s ∨ subc_e c e
36  ]
37
38 and subc_s c s on s :=
39  match s with
40  [ subst x b ⇒ subc_b c b ]
41  .
42
43 definition cinc := λc.λd. c=d ∨ subc c d.
```

Codice 6.10: Definizione formale dei predicati che esprimono la relazione di essere sottotermine di un termine `t1` per un dato termine `t2`

Date queste definizioni, l'enunciato del lemma 4.5.4 può essere formalizzato come segue:

```
(∀t, s, c. subc c (fst … (underline t s)) →
   ∃u, n. (c = fst … (underline u n)) ∧ (tint u t)) ∧
(∀v, s, c. subc_v c (fst … (overline v s)) →
```

```
  ∃u, n. (c = fst … (underline u n)) ∧ (subt_v u v)).
```

Codice 6.11: Formalizzazione del lemma 4.5.4

6.3.4 Lemma 4.5.5

Questa proprietà è di notevole importanza: le regole del calcolo della disciplina di valutazione call-by-value, richiedono che le β-riduzioni vengano applicate solo quando l'argomento dell'applicazione è un valore. Dunque questo lemma, unitamente al fatto che la macchina Crumble GLAM sia deterministica, concorre a garantire che la riduzione dei termini sia effettuata secondo la disciplina di valutazione call-by-value.

La dimostrazione di questo lemma ha richiesto la generalizzazione della `read_back` al caso di CrumbledContext; la definizione di tale operazione data in [5] è la 6.8, che a sua volta fa riferimento alla nozione di sostituzione della costante $\langle\cdot\rangle$, che tuttavia non è stata formalizzata.

$$
\begin{aligned}
\langle\cdot\rangle_{\downarrow} &\coloneqq \langle\cdot\rangle \\
(b, e[x \leftarrow \langle\cdot\rangle)_{\downarrow} &\coloneqq (b, e)\{x \leftarrow \langle\cdot\rangle\}
\end{aligned}
\tag{6.8}
$$

Per definire l'operazione di $\{x \leftarrow \langle\cdot\rangle\}$ nel caso generico di pTerm è stato necessario definire il tipo di dato astratto TermContext (4.4) e su di questo il predicato `rv_context` (6.8) in grado di determinare se un contesto generico sia o meno un right-v evaluation context. Infatti, non sarebbe stato possibile fornire un'implementazione degli rv-context utile sotto forma di tipo: come abbiamo dimostrato che l'operazione `underline` traduce pTerm in Crumble avremmo potuto definire l'operazione di read-back su CrumbledContexts in modo che il tipo di ritorno fosse già quello di rv-context. Per la definizione che ne abbiamo dato, avremmo avuto bisogno di una formalizzazione specifica della $\{x \leftarrow \langle\cdot\rangle\}$ con tipo di ritorno right-v evaluation context, ma per poter costruire tale funzione, avremmo dovuto giustificare formalmente che la variabile da sostituire con $\langle\cdot\rangle$ non si trovasse mai sotto o a sinistra di astrazioni, che equivale a dimostrare che i termini generati dalla `read_back_cc` siano right-v evaluation contexts.

Alla luce delle precedenti osservazioni, la formalizzazione dell'enunciato del lemma 4.5.5, può essere espressa come segue:

```
lemma four_dot_five_dot_five:
 (∀t, s, c, C.
  fresh_var_t t ≤ s →
   fst … (underline t s) = plug_c C c →
    rv_context (cc_read_back C)) ∧
 (∀v:pValue. True).
```

Codice 6.12: Formalizzazione del lemma 4.5.5

La dimostrazione di questo risultato ha richiesto un lavoro non banale di gestione degli ambienti: per esempio, nel caso induttivo dell'applicazione $t_1 t_2$, per poter usare l'ipotesi induttiva su uno dei due sottotermini di t, è necessario

poter esprimere $\underline{t}$ per come il plugging di un Crumble all'interno di un Clumble-Context. Per farlo è necessario saper identificare il punto preciso in cui dividere l'Environment per identificare quale sarebbe stata la porzione di environment a sinistra di $\langle\cdot\rangle$ (da inserire nel contesto richiesto dall'ipotesi induttiva) e quale a destra, come mostrato dall'esempio 6.3.3. Ma date le definizioni formali delle operazioni di `plug_c` e di `underline`, rispettivamente in 4.6 e 6.3, gli Environment generati in fase di crumbling possono assumere una delle forme espresse in (6.9), mentre, tralasciando il caso in cui $C = \langle\cdot\rangle$[3], gli Environment generati dal processo di plugging sono della forma espressa in 6.10. Per riuscire ad usare un'ipotesi di uguaglianza fra un Environment costruito come nella (6.9) ed uno costruito nella 6.10, e quindi a ricomporre il CrumbledContext necessario per l'ipotesi induttiva, è opportuno definire alcuni lemmi tecnici che permettno di dedurre delle uguaglianze di Environment sufficientemente piccoli da permettere di esprimere uno dei due composizione di pezzi dell'altro.

$$\begin{gathered}\texttt{Epsilon}\\ \texttt{push e s}\\ \texttt{concat (push e s) (push f t)}\end{gathered} \tag{6.9}$$

$$\texttt{concat (Snoc e s) f} \tag{6.10}$$

Esempio 6.3.3. Supponiamo ad esempio `t = appl t1 t2`, dove a loro volta `t1` e `t2` sono applicazioni. L'ipotesi `fst ... underline t s= plug_c C c`, nel caso in cui `C = crc b (envc f y)`, si forma come segue:

$$\begin{gathered}\langle \texttt{AppValue (var } \nu(\texttt{S(n1)})) \texttt{ (var } \nu(\texttt{n1})), \texttt{concat (push e } [\nu(\texttt{S(n1)}) \leftarrow \texttt{b}])\\ (\texttt{push e1 } [\nu\texttt{n1} \leftarrow \texttt{b1}])\rangle = \langle\ \texttt{r},\ \texttt{concat (Snoc f } [\texttt{y} \leftarrow \texttt{d}])\ \texttt{g}\rangle\end{gathered}$$

Dove $\texttt{c} = \langle \texttt{d}, \texttt{g}\rangle$. Per derivare `b` $=$ `AppValue` (`var` ν(`S(n1)`)) (`var` ν(`n1`)) non ci sono problemi, mentre per riuscire ad utilizzare il predicato di uguaglianza sugli Environment è necessario fare ancora del lavoro. In base alla lunghezza di `f`, possiamo avere:

1. `y` $=$ ν (`S n1`)

2. `dob_e y e = true`

3. `y` $=$ ν `n1`

4. `dob_e y e = true`

---

[3] in questo caso la conclusione è scontata per la definizione di `rv_context` data in 4.5.

Supponiamo di trovarci nel caso 3, allora avremmo la situazione che segue:

$$\begin{array}{cccccccccccc}
\mathtt{b} & \mathtt{b}_{\mathtt{e}}^{(\mathtt{1})} & \cdots & \cdots & \cdots & \mathtt{b}_{\mathtt{e}}^{(\mathtt{n})} & \mathtt{b1} & \mathtt{b}_{\mathtt{e1}}^{(\mathtt{1})} & \cdots & \cdots & \cdots & \mathtt{b}_{\mathtt{e1}}^{(\mathtt{m})} \\
\downarrow & \downarrow & \cdots & \cdots & \cdots & \downarrow & \downarrow & \downarrow & \cdots & \cdots & \cdots & \downarrow \\
\nu(\mathtt{S\ n1}) & \mathtt{x}_{\mathtt{e}}^{(\mathtt{1})} & \cdots & \cdots & \cdots & \mathtt{x}_{\mathtt{e}}^{(\mathtt{n})} & \nu\mathtt{n1} & \mathtt{x}_{\mathtt{e1}}^{(\mathtt{1})} & \cdots & \cdots & \cdots & \mathtt{x}_{\mathtt{e1}}^{(\mathtt{m})}
\end{array}$$

$$\begin{array}{cccccccccccc}
\mathtt{b}_{\mathtt{f}}^{(\mathtt{1})} & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \mathtt{b}_{\mathtt{f}}^{(\mathtt{l})} & \mathtt{d} & \mathtt{b}_{\mathtt{g}}^{(\mathtt{1})} & \mathtt{b}_{\mathtt{g}}^{(\mathtt{s})} \\
\downarrow & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \downarrow & \downarrow & \downarrow \\
\mathtt{x}_{\mathtt{f}}^{(\mathtt{1})} & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \mathtt{x}_{\mathtt{f}}^{(\mathtt{l})} & \mathtt{y} & \mathtt{x}_{\mathtt{g}}^{(\mathtt{1})} & \mathtt{x}_{\mathtt{g}}^{(\mathtt{s})}
\end{array}$$

In questo caso, per poter utilizzare l'uguaglianza dei due environment di cui sopra, occorre definire un lemma tecnico che permette di dimostrare l'esistenza di un Environment `j`, tale che:

- `concat (push e [ν(S n1) ← b]) (push j [ν n1 ← b1])= f`
- `e1 = concat (Snoc j [y ← d]) g`

Una volta ottenuto un risultato di questo tipo, è possibile sfruttare l'ipotesi induttiva sul termine di destra, perché è possibile esprimere la sua `underline` per mezzo di plugging: infatti, in riferimento al termine di sopra, `fst ... (underline t2 ?) = plug_c (crc b (envc j y)) (CCrumble d g))`.

Grazie a questa costruzione, si deriva che `(crc b (envc j y))` è un right-v evaluation context, ma a questo punto è banale derivare la conclusione dal momento che, per il lemma 6.3.3 (C.5)[4], l'Environment `j` e la variabile `y` sono ortogonali alla read-back di `t1`, per cui `read_back (crc r (envc f y)) = appl (read_back t1) (read_back_cc (crc b (envc j y)))` che è un right-v context dal momento che è l'applicazione di un termine ad un right-v context.

Per produrre questo tipo di risultati, è stato necessario dimostrare numerosi lemmi che in letteratura vengono detti di matching delle liste. L'idea che sta dietro a questa tipologia di risultati è quella di identificare tutti i pattern che possono essere generati dal confronto di due Environment composti, per poi procedere per casi su tutti i pattern e dimostrare uno specifico enunciato. Riportiamo in 6.13 uno dei risultati più importanti che sono stati dimostrati in questo ambito: quello del matching di un environment della forma *ese* con uno della forma *sese*; questo risultato è un caso generale di quello preso in esame da 6.3.3: si osserva, infatti, che il matching studiato in 6.3.3 è una banale riscrittura dell'ultimo disgiunto in 6.13.

```
lemma decomposition_lemma:
 ∀e, f, g, h, s, t, u. (concat (push e s) (push f t)=concat
     (Snoc g u) h) →
  (g=Epsilon ∧ s=u ∧ h = concat e (push f t)) ∨
```

[4]Come abbiamo già sottolineato, tale lemma non è stato dimostrato alla lettera, per cui si è fatto uso di una sua istanziazione più tecnica riguardante la read-back di un termine sotto la concatenazione di environment di cui il dominio uno dei due fosse disgiunto dalle variabili libere del termine.

```
(∃d, p. g = push d s ∧ e = concat (Snoc d u) p ∧ h =
    (concat (Snoc p t) f)) ∨
 (g = push e s ∧ t = u ∧ h = f) ∨
  (g = Snoc (push e s) t ∧ f = push h u) ∨
   (∃d. g = concat (Snoc (push e s) t) d ∧ f = concat d
       (push h u)).
```

Codice 6.13: Enunciato del lemma di matching di ambienti *ese* con ambienti *sese*

6.4 Size Lemma

L'ultimo risultato importante per lo studio dello stato di inizializzazione della macchina Crumble GLAM è quello del size lemma: in 3 abbiamo anticipato che questo risultato era fondamentale per la valutazione della complessità della macchina. Infatti, in [5] si dimostra che la complessità di valutazione di un temine qualsiasi `t` è $O(|t| * (|\rho| + 1))$ dove ρ è la lunghezza di una sequenza di riduzioni normalizzante secondo la disciplina CbV del λ_p-calcolo.

Per ottenere questo risultato è stato importante identificare la complessità di ciascuna operazione di riduzione della macchina Crumble GLAM, per poi studiare quali sono i rapporti fra il numero di volte in cui vengono applicate le regole di overhead ed il numero di volte in cui vengono applicate le regole proprie del calcolo implementato (i.e., nella Crumble GLAM, la disciplina weak CbV). In [5] si dimostra che il numero di volte in cui vengono applicate le transizioni di overhead è lineare nella lunghezza di ρ, dunque il numero complesivo di riduzioni opperata dalla Crumble continua ad essere lineare in ρ. Per studiare la complessità di ciascuna transizione, si parte dall'osservazione che ogni sottotermine delle crumbled form raggiungibili durante il calcolo è un sottotermine del Crumble iniziale, per cui, nel caso pessimo, la complessità di ciascuna operazione di riscrittura dei termini è lineare nella dimensione del sottotermine da riscrivere. Per questo motivo, è interessante studiare la relazione che sussiste fra la taglia del termine iniziale e quella della crumbled form ottenuta applicando su di esso la funzione `underline`.

A tal proposito è stato necessario definire una funzione per calcolare la taglia di un termine secondo la definizione data da [5], riproposta in 6.14 affinché il lettore si possa convincere che tale definizione è ragionevole. Infine è stato necessario formulare e dimostrare il lemma 6.15 che lega le taglie dei termini iniziali e dei loro tradotti. Nel caso specifico si dimostra, mediante una semplice visita per induzione strutturale del tipo pTerm, che $\forall t.|\underline{t}| \leq 5 \cdot |t| \wedge \forall v.|\underline{v}| \leq 5 \cdot |v|$.

```
1  let rec t_size t on t :=
2   match t with
3   [ val_to_term v ⇒ v_size v
4   | appl t1 t2 ⇒ S ((t_size t1) + (t_size t2))
5   ]
6
7  and v_size v on v :=
8   match v with
9   [ pvar v ⇒ 1
```

```
10   | abstr x t ⇒ S (t_size t)
11   ]
12  .
13
14  let rec c_size c on c :=
15   match c with
16   [ CCrumble b e ⇒ (c_size_b b + c_size_e e) ]
17
18  and c_size_b b on b :=
19   match b with
20   [ CValue v ⇒ c_size_v v
21   | AppValue v w ⇒ S (c_size_v v + c_size_v w)
22   ]
23
24  and c_size_e e on e :=
25   match e with
26   [ Epsilon ⇒ O
27   | Snoc e s ⇒(c_size_e e) + c_size_s s
28   ]
29
30  and c_size_v v on v :=
31   match v with
32   [ var x ⇒ S O
33   | lambda x c ⇒ S (c_size c)
34   ]
35
36  and c_size_s s on s :=
37   match s with
38   [ subst x b ⇒ S (c_size_b b)]
39   .
```

Codice 6.14: Funzioni che calcolano la taglia di pTerm, Crumble e tipi mutui

```
lemma size_lemma:
 (∀t.∀n. c_size (fst …  (underline t n)) ≤ 5 * (t_size t)) ∧
   (∀v.∀n. c_size_v (fst … (overline v n)) ≤ 5 * (v_size v)).
```

Codice 6.15: Enunciato formale del size lemma

Capitolo 7

alpha

Come abbiamo dimostrato in 6, il remark 4.1.3 (6.3.6) non è corretto. Questo fatto ha indotto la necessità di prendere in esame tutte le prove che ne fanno uso per indagare se tale risultato sia o meno condizione necessaria per concludere la tesi. Fortunatamente, il remark 4.1.3 è utilizzato per dimostrare solo due enunciati, per i quali il risultato in questione non è mai condizione necessaria: essi sono i punti 3 e la 4 delle invarianti dinamiche della macchina Crumble GLAM enunciate in 7.0.1.

Lemma 7.0.1 (5.5)
Ogni Crumble c_n raggiungibile in una computazione dalla macchina Crumble GLAM a partire dal Crumble iniziale c_0 gode delle seguenti invarianti:

1. *c_n è well-named*
2. *$FV(c_n) = \emptyset$*
3. *c_n è un sottotermine, a meno di ridenomine, di c_0*
4. *$\forall C, C', b, e_v. c_n = C\langle C'\langle (b, e_v)\rangle\rangle \rightarrow C_\downarrow$ è un right-v evaluation context.*

La necessità di risolvere alternativamente le dimostrazioni dei punti 3 e 4 di 7.0.1 ci ha indotti ad esplorare parte del comportamento dinamico della macchina Crumble GLAM il quale, di fatto, non avrebbe dovuto essere parte del presente lavoro di formalizzazione poiché sarà oggetto di successive ricerche. Di conseguenza, l'analisi dinamica della Crumble GLAM operata in questa sezione non sarà complessiva, ma sarà limitata a risolvere le sole falle indotte dal remark 4.1.3.

Il processo di normalizzazione operato dalla macchina Crumble GLAM è già stato esaustivamente esplicato in 3.2 per cui, in questa sezione, ci limiteremo ad avanzare considerazioni di carattere tecnico. Per prima cosa, osserviamo che le regole di transizione della Crumble GLAM (3.13) fanno uso delle funzioni di manipolazione dei Crumble già definite in 4 e di due ulteriori funzioni: $\cdot^\alpha$ e $e(\cdot)$, descritte in maniera informale rispettivamente in 7.1.1 e in 3.2.2.

In linea di principio, dunque, ci si potrebbe aspettare che sia necessario formalizzarle entrambe per potere concludere il nostro lavoro; tuttavia, il risultato del remark 4.1.3 è fortemente legato alla specifica della funzione $\cdot^{\alpha}$, dal momento questa opera mediante sostituzioni successive di variabili ed il remark 4.1.3 tratta nello specifico di rinomina di variabili libere. Infatti, nelle dimostrazioni informali che sono date in [5] dei punti 3 e 4 di 7.0.1 e che abbiamo riportato rispettivamente in 7.2.3 ed in 7.2.4, il remark 4.1.3 è utilizzato esclusivamente per concludere risultati riguardanti la funzione $\cdot^{\alpha}$.

Per queste ragioni non sarà necessario formalizzare la funzione $e(\cdot)$, ma ci si potrà limitare alla formalizzazione della funzione $\cdot^{\alpha}$.

7.1 alpha

7.1.1 Definizioni informali

La definizione informale data da [5] della funzione $\cdot^{\alpha}$ è quella riportata in 7.1.1. Il contesto in cui opera la funzione $\cdot^{\alpha}$ è quello dell'applicazione della regola di riduzione $\mapsto_{\beta_v}$(7.1), la quale ha lo scopo essenziale di emulare la riduzione dell'applicazione di due valori mediante l'inserimento di un costrutto di ES nell'Environment più esterno che lega la variabile astratta sul primo valore all'argomento, evitando così la necessità di effettuare sostituzioni che, lo ricordiamo, è il punto chiave dell'implementazione del λ-calcolo per mezzo di ambienti.

Definizione 7.1.1. Indichiamo con la notazione c^{α} la crumbled form ottenuta rinominando il dominio di c mediante variabili che siano distinte e fresche nella crumbled form c.

$$\begin{aligned} ((\lambda x.c)v, e_v) &\mapsto_{\beta_v} (c@[x \leftarrow v])^{\alpha}@e_v \\ (x, e_v) &\mapsto_{sub_{var}} (e_v(x), e_v) \quad (*) \\ (xv, e_v) &\mapsto_{sub_l} (e_v(x)v, e_v) \quad (*) \\ &(*) \text{ if } x \in DOM(e_v) \end{aligned} \tag{7.1}$$

Intuitivamente, la ragione per cui è necessario ridenominare tutto il dominio del crumble $(c@[x \leftarrow v])$ non è ben chiara: ci si aspetterebbe di potersi limitare alla sola ridenomina della x, dal momento che questa potrebbe essere già utilizzata in altre astrazioni e dunque una successiva applicazione del $\mapsto_{\beta_v}$ romperebbe il vincolo di well-namedness dell'Environment e, dunque, non sarebbe più possibile risolvere deterministicamente i nomi di variabile mediante la funzione $e(\cdot)$. Inoltre non ridenominare tutte le variabili dell'Environment, concorrerebbe ad abbassare (non asintoticamente), la complessità della valutazione. Purtroppo però, come evidenziato dall'esempio 7.1.1, la ridenomina di tutto l'Environment è inevitabile al fine di garantire l'invariante di well-namedness espresso dal lemma 5.5.1.

Esempio 7.1.1. Si prenda in esempio l'esempio mostrato in 7.2, in cui a e b sono valori generici, mentre la transizione $\mapsto^{\omega}_{\beta_v}$ è una variante della $\mapsto_{\beta_v}$, ma

che, anziché utilizzare la funzione $\cdot^\alpha$, fa uso di un'analoga funzione $\cdot^\omega$ che non rinomina tutto il dominio di $(c@[x \leftarrow v])$, ma si limita a rinominare la sola variabile x.

$$\begin{gathered}(ww', [w \leftarrow (ya)][w' \leftarrow yb][y \leftarrow \lambda x.(z, [z \leftarrow x])]) \mapsto_{sub_l} \\ (ww', [w \leftarrow (ya)][w' \leftarrow \lambda x.(z, [z \leftarrow x])b][y \leftarrow \lambda x.(z, [z \leftarrow x])]) \mapsto^\omega_{\beta_v} \\ (ww', [w \leftarrow (ya)][w' \leftarrow z][z \leftarrow x'][x' \leftarrow b][y \leftarrow \lambda x.(z, [z \leftarrow x])]) \mapsto_{sub_{var}} \\ (ww', [w \leftarrow (ya)][w' \leftarrow z][z \leftarrow b][x' \leftarrow b][y \leftarrow \lambda x.(z, [z \leftarrow x])]) \mapsto_{sub_{var}} \\ (ww', [w \leftarrow (ya)][w' \leftarrow b][z \leftarrow b][x' \leftarrow b][y \leftarrow \lambda x.(z, [z \leftarrow x])]) \mapsto^\omega_{\beta_v} \\ (ww', [w \leftarrow z][z \leftarrow x''][x'' \leftarrow a][w' \leftarrow b][z \leftarrow b][x' \leftarrow b][y \leftarrow \lambda x.(z, [z \leftarrow x])])\end{gathered} \tag{7.2}$$

È palese come la sequenza di riduzioni abbia generato due ES associati alla variabile z, rompendo l'invariante di well-namedness. Al contrario, se avessimo applicato la regola $\mapsto_{\beta_v}$, le due occorrenze di z sarebbero state ridenominate e dunque distinte fra loro.

Assodato che la definizione di una tale funzione è inevitabile, proviamo a formulare in 7.3 una definizione ricorsiva della funzione alpha che rispetti la specifica datane in 7.1.1.

$$\begin{aligned}(b, \epsilon)^\alpha &:= (b, \epsilon) \\ (b, e[x \leftarrow b'])^\alpha &:= ((b, e)^\alpha\{x \leftarrow z\})@[z \leftarrow b'](*)\end{aligned} \tag{7.3}$$

$(*)$ con z fresca in b, e, b'

Potrebbe sembrare fuorviante il fatto che la funzione $\cdot^\alpha$, per come è stata definita in (7.3), non effettui sostituzioni sul Bite b'. La motivazione di questa scelta è guidata dal fatto che tutti gli Environments generati durante la fase di valutazione di un Crumble sono formati di elementi $[x \leftarrow b]$ per cui $x \notin b$. Questo risultato può essere facilmente visto come conseguenza dell'invariante di disjointedness (espresso nel caso iniziale dal lemma 4.5.3 (6.3.9)) e per la sua istanza dinamica, il lemma D.6 che si trova in appendice di [5].

La definizione 7.3 mette in evidenza la necessità di definire una funzione in grado di effettuare sostituzioni di variabili su Crumble. Si potrebbe pensare che sia il caso di definire una funzione generica in grado di sostituire interi Crumble all'interno di Crumble, da utilizzare specificatamente per la sostituzione di variabili. In questo modo si produrrebbe una funzione (ed un insieme di lemmi utili su di essa) che potrebbw essere usata anche in seguito. Inoltre, sempre in questo modo, si definirebbe una funzione duale per il tipo Crumble alla `p_Subst` per i pTerm data in A.2, che potrebbe essere utile per dimostrare isomorfismi, ma a tal proposito è opportuno osservare quanto segue:

- In [5], la sostituzione di Crumble non viene mai nemmeno menzionata: la definizione stessa dei Crumble è finalizzata ad evitare la necessità di sostituire sottotermini.

- Oltre ai problemi di cattura dei nomi di variabili da parte delle λ-astrazioni già risolti dalla `p_Subst`, si aggiungerebbe il problema della cattura delle

variabili da parte delle ES negli Environment che è del tutto ortogonale al caso d'utilizzo che ne faremmo, siccome la funzione $\cdot^\alpha$ effettua sostituzioni con variabili fresche che, dunque, non possono essere catturate.

- L'implementazione di una tale funzione causerebbe l'adozione di misure atte ad evitare conflitti di nomi degli Environment.

- Dal momento che la grammatica dei Crumble è un'estensione della grammatica dei pTerm, ci si aspetta che la definizione delle sostituzione su Crubmle, e per le ragioni espresse nei punti precedenti, sia più complessa della definizione della `p_Subst` descritta in A.2.

Per le suddette ragioni, si è scelto di limitarsi a definire una famiglia di funzioni mutue, detta `ss*`, che ha il compito di sostituire variabili a variabili con l'ipotesi che la variabile da sostituire non compaia nel termine sostituendo. Questa ipotesi semplifica enormemente l'implementazione delle funzioni `ss*`, dal momento che, rendendo impossibili i fenomeni di cattura, evita le ridenomine; inoltre, quella di poter sostituire solo variabili *sufficientemente fresche* non è affatto un'ipotesi riduttiva, siccome la funzione $\cdot^\alpha$ è definita come come la sostituzione delle variabili del dominio di un dato Crumble con variabili fresche, che dunque non sono presenti nel Crumble di partenza.

La definizione delle funzioni di sostituzione semplice può avvenire come mostrato in 7.4.

$$\begin{aligned}
(b,e)\{x \leftarrow z\} &:= (b\{x \leftarrow z\}, e\{x \leftarrow z\}) \quad \text{if } x \notin DOM(e) \\
(b,e)\{x \leftarrow z\} &:= (b, e\{x \leftarrow z\}) \quad \text{if } x \in DOM(e) \\
(vw)\{x \leftarrow z\} &:= (v\{x \leftarrow z\})(w\{x \leftarrow z\}) \\
(\lambda y.c)\{x \leftarrow z\} &:= \lambda y.(c\{x \leftarrow z\}) \\
(\lambda x.c)\{x \leftarrow z\} &:= \lambda x.c \\
y\{x \leftarrow z\} &:= y \\
x\{x \leftarrow z\} &:= z \\
(e[y \leftarrow b])\{x \leftarrow z\} &:= (e\{x \leftarrow z\})[y \leftarrow b\{x \leftarrow z\}] \\
(e[x \leftarrow b])\{x \leftarrow z\} &:= e[x \leftarrow b\{x \leftarrow z\}]
\end{aligned} \tag{7.4}$$

7.1.2 Implementazioni in Matita

Per una migliore comprensione di questa parte, si consiglia di aver già letto l'appendice A.2 in merito all'implementazione della `p_subst`: in tale sezione viene esaminata dettagliatamente l'implementazione di una funzione per la sostituzione sui pTerm che, similmente alle funzioni `alpha` e `ssc`, necessita di prove di precondizioni per essere invocata. Tuttavia, mentre nel caso della `p_subst` le precondizioni non erano indirizzate a limitare la possibilità di fare uso della funzione a casi specifici, ma a garantire la definibilità della funzione stessa, in questo caso la richiesta di precondizioni è motivata dal fatto di permettere un'implementazione più leggera della funzione in virtù di determinate ipotesi.

L'implementazione delle `ssc` è molto semplice: l'unica accortezza adottata nell'implementazione riguarda la richiesta in input di una prova `(H: inb x c = false)` finalizzata a garantire che ogni uso di tale funzione sia legittimo in virtù delle precondizioni richieste dalla definizione 7.4. Come si evince dalla definizione riportata in 7.1, anche le funzioni mutue richiedono prove analoghe, che sono aperte dal termine `?` posto laddove dovrebbero apparire questi termini dimodoché si possano produrre specifiche dimostrazioni di tali enunciati *subito dopo* aver definito la funzione.

```
1   let rec ssc c y z on c: inb z c = false → Crumble :=
2    match c return λc. inb z c = false → Crumble with
3     [ CCrumble b e ⇒ λp. match domb_e y e with
4       [ true ⇒ ⟨b, sse e y z ?⟩
5       | false ⇒ ⟨ssb b y z ?, sse e y z ?⟩
6       ]
7     ]
8
9   and ssb b y z on b: inb_b z b = false → Bite :=
10   match b return λg. inb_b z g = false → Bite with
11   [ CValue v ⇒ λp. CValue (ssv v y z ?)
12   | AppValue v w ⇒ λp. AppValue (ssv v y z ?) (ssv w y z ?)
13   ]
14
15  and ssv v y z on v: inb_v z v = false → Value :=
16   match v return λv. inb_v z v = false → Value with
17   [ var x ⇒ λp. match veqb x y with [true ⇒ var z | false ⇒ var x]
18   | lambda x c ⇒ match veqb x y with [true ⇒ λp. lambda x c | false ⇒ λp. lambda x
        (ssc c y z ?)]
19   ]
20
21  and sse e y z on e: inb_e z e = false → Environment :=
22   match e return λe. inb_e z e = false → Environment with
23   [ Epsilon ⇒ λp. Epsilon
24   | Snoc e s ⇒ match s return λs. inb_e z (Snoc e s) = false → Environment with
25      [ subst w b ⇒ match veqb y w with
26        [ true ⇒ λp. Snoc e [w←ssb b y z ?]
27        |  false ⇒ λp. Snoc (sse e y z ?) [w←ssb b y z ?]
28        ]
29      ]
30   ]
31  .
```

Codice 7.1: Formalizzazione della funzione di ridenomina in Matita

L'implementazione `alpha` della funzione $\cdot^{\alpha}$, dovrà essere in grado di generare variabili fresche ogniqualvolta sia necessario. Per fare ciò, adottiamo un meccanismo simile a quello già descritto per la funzione `underline` in 6.2: imponiamo che, come parametro della funzione, sia richiesto un intero che sia maggiorante di tutti i valori delle variabili presenti nel Crumble da trasformare. In questo modo, ogni chiamata ricorsiva, incrementando tale valore, avrà sempre a disposizione un indice di variabile *sufficientemente fresca* da sostituire. Inoltre, dal momento che ogni chiamata ricorsiva, contrariamente a quanto avviene per la funzione `underline`, ridenomina esattamente una variabile, il calcolo del nuovo indice di variabile *sufficientemente fresca* sarà banale: basterà utilizzare il successore della variabile appena introdotta. Alla luce di queste osservazioni, la funzione `alpha` può essere implementata come mostrato in 7.2

```
1   let rec alpha (b: Bite) (e: Environment) (n: nat) on e:
```

```
2   fresh_var ⟨b, e⟩ ≤ n →
3    Σc. ∀m. fresh_var ⟨b, e⟩ ≤ m ∧ m < n → inb (νm) c = false :=
4   match e return λe. fresh_var ⟨b, e⟩ ≤ n → Σc. ∀m. fresh_var ⟨b, e⟩ ≤ m ∧ m < n →
        inb (νm) c = false  with
5   [ Epsilon ⇒ λp. mk_Sig … ⟨b, Epsilon⟩ (alpha_aux2 b n)
6   | Snoc e' s ⇒ match s return λs. fresh_var ⟨b, Snoc e' s⟩ ≤ n → Σc. ∀m.
        fresh_var ⟨b, Snoc e' s⟩ ≤ m ∧ m < n → inb (νm) c = false with
7     [subst y b' ⇒ λp. match alpha b e' (S n) (alpha_aux1 … (subst y b') … p) with
8       [ mk_Sig a h ⇒ mk_Sig …(at (ssc (a) y (νn) (alpha_aux3 b e' a n y b' h p))
            (Snoc Epsilon (subst (νn) b'))) (alpha_aux4 b e' a n y b' (alpha_aux3 b
            e' a n y b' h p) h p) ]
9     ]
10  ]
11  .
```

Codice 7.2: Formalizzazione della funzione $\cdot^{\alpha}$ in Matita

Per ragioni di definibilità legate al numero di costruttori di tipo annidati [1], la visita ricorsiva dell'albero sintattico del crumble è anticipata, per cui la chiamata della `ssc` ha come argomento `a`: il valore prodotto dalla chiamata ricorsiva di `alpha`. Sul Crumble `a`, per poter invocare la funzione `ssc`, è necessario provare la proposizione `(H: inb νn a = false)`. Questo fatto induce ancora una volta la necessità di utilizzare come tipo di ritorno della funzione `alpha` un Σ-tipo: possiamo così legare il termine `a` ad una prova `h` della proposizione $\forall \mathtt{m}.\ \mathtt{fresh_var}\ \langle \mathtt{b}, \mathtt{e} \rangle\ \leq\ \mathtt{n}\ \wedge\ \mathtt{m}\ <\ \mathtt{n} \rightarrow \mathtt{inb}\ (\nu \mathtt{m})\ \mathtt{a}\ =\ \mathtt{false}$. Da questa dimostrazione, che è a tutti gli effetti costruita dalla funzione `alpha` stessa, sarà possibile costruire la prova necessaria ad invocare la `ssc` utilizzandolo come parametro per l'invocazione del lemma `alpha_aux3` a riga 8, rendendo possibile la definizione stessa della funzione alpha, che altrimenti non potrebbe invocare la `ssc`.

7.1.3 `beta` e `gamma`

La funzione `alpha`, per come è definita, è poco elastica: essa richiede che come input si abbia sempre un Environment, ma soprattutto che il ritorno sia un Crumble. Questo aspetto può diventare scomodo utilizzarla qualora si vogliano commutare i costruttori di tipo al costruttore della funzione, o qualora si voglia usare tale funzione al fine di calcolare gli esiti di ridenomine su tipi diversi dal Crumble.

Per questi motivi è stata introdotta una funzione `alpha_e` che agisce come la funzione `alpha` ma solo sull'Environment; la definizione è molto simile a quella di `alpha`, ma con tipo $\lambda(\mathtt{e} : \mathtt{Environment}).\lambda(\mathtt{n} : \mathtt{nat}).\lambda(\mathtt{H} : \mathtt{fresh_vare} \leq \mathtt{nat}).$ `Environment`. Per poter effettuare rinomine simili, ma sugli altri tipi mutui con Crumble, è stato necessario definire ed implementare altre due funzioni: $\beta(\cdot)$ e $\gamma(\cdot)$ che permettono, inoltre, di disaccoppiare la fase di calcolo delle sostituzioni dalla fase di sostituzione vera e propria.

Per far ciò, abbiamo definito una coppia di funzioni, per prima la funzione `beta_e` di tipo `Environment` $\rightarrow$ `list` $(\mathtt{Variable} \times \mathtt{Variable})$ che dato un

[1] Per casi simili si faccia riferimento per esempio alle definizioni delle funzioni `underline`, `read_back` e `p_subst` rispettivamente in 6.2, 6.1 e A.2.

Environment e calcoli la lista di sostituzioni che verrebbero generate a partire da esso dalla funzione `alpha`. Mentre, le funzioni della famiglia `gamma*`, che ha come signature `list` (`Variable` $\times$ `Variable`) $\rightarrow *$ (dove $*$ è il tipo di ritorno della specifica funzione della famiglia), scorrono la lista della sostituzioni calcolate dalla `beta_e` e le riportano su di un dato qualsiasi. Le definizioni di tali funzioni sono riportate in (7.5) ed in (7.5).

$$\begin{aligned} \beta(\epsilon, n) &= nil \\ \beta(e[x \leftarrow y], n) &= \langle x, n \rangle :: \beta(e, S\ n) \end{aligned} \tag{7.5}$$

$$\begin{aligned} \gamma(t, nil) &= t \\ \gamma(t, \langle x, n \rangle :: l) &= \gamma(t, l)\{x \leftarrow n\} \end{aligned} \tag{7.6}$$

Uno degli aspetti fondamentali di queste due funzioni è che potrebbe suonare strano il motivo per cui sono in grado di emulare il comportamento della funzione $\cdot^{\alpha}$: non sembrerebbe possibile poter provare un enunciato del tipo $(b, e)^{\alpha}_{n} = (\gamma(b, \beta(e, n))), e^{\alpha})$ dove n denota l'indice di partenza per la ridenomina. Infatti le sostituzioni propagate dalla funzione alpha si arrestano qualora si incontri una ES nell'Environment che lega la variabile da sostituire, mentre le sostituzioni calcolate dalla funzione β ed applicate dalla funzione γ non si arrestano. Tuttavia, tale lemma è vero ed è stato anche dimostrato formalmente: la ragione per cui ciò è stato possibile deriva dal fatto che le sostituzioni che si trovano in coda nell'Environment saranno quelle applicate per prime: in questo modo, sostituzioni successive delle medesime variabili non avranno effetto poiché la variabile da sostituire sarà già stata sostituita da una variabile fresca e dunque non comparirà più nel termine.

Un caso d'uso specifico di tali funzioni, per esempio, può essere quello riguardante la commutazione di costruttori di tipo rispetto alle funzioni `alpha*`. Si prenda il seguente Environment: $[x \leftarrow b]e$, se si effettua su di esso una chiamata alla `alpha_e`, si riuscirebbe sì a calcolare l'esito della sostituzione sull'Environment, ma, dal momento che le funzioni `alpha*` sono opache rispetto alla struttura interna dei dati su cui operano, non sarebbe possibile accedere direttamente all'esito che la sostituzione ha su x e su b. Ciò a cui aspiriamo, invece, è riuscire a commutare la funzione `push`rispetto alla `alpha_e` sia sulla x che sulla b.

Grazie al potere espressivo di tali funzioni è stato possibile dimostrare con una certa semplicità dei lemmi come quello appena enunciato e riportato in 7.3, la cui dimostrazione altrimenti sarebbe stata molto complessa e senza il quale sarebbe stato impossibile dimostrare risultati come quello espresso dal lemma 5.5.3 ed ancor di più dal lemma 5.5.4.

```
1  lemma alpha_push: ∀ e, y, b, n, H.
2   pi1 … (alpha_e (push e [y←b]) n H) =push (pi1 … (alpha_e e n (alpha_push_aux1 …
       H)))
3    [ν(n+e_len e)←pi1 … (gamma_b b (beta_e e n) (alpha_push_aux2 … H))].
```

Codice 7.3: Enunciato del lemma che permette di commutare `push` ad `alpha_e`.

7.2 Lemmi riguardanti la funzione `alpha`

Per verificare la correttezza dell'implementazione della funzione `alpha` è stato interessante verificare la validità degli invarianti espressi dal lemma 5.5 (7.0.1) a seguito dell'applicazione della sola funzione $\cdot^{\alpha}$ a Crumble che già godessero di tali proprietà. In questo modo, è stato anche anticipato parzialmente il lavoro di formalizzazione degli aspetti dinamici del funzionamento della Crumble GLAM che sarà oggetto di future formalizzazioni.

7.2.1 Size lemma - `alpha`

Un primo risultato da verificare è stato quello riguardante la conservazione della taglia di un Crumble su cui viene applicata la `alpha`. Tale risultato, che intuitivamente è triviale, serve per garantire che l'applicazione della regola di transizione $\mapsto_{\beta_v}$ non causi un incremento della taglia del termine. Senza questa condizione, infatti, non sarebbe più possibile dimostrare che la Crumble GLAM effettua riduzioni con un overhead bilineare nel numero di β-riduzioni necessarie per la normalizzazione del termine.

L'enunciato di tale lemma fa uso delle medesime funzioni di calcolo della taglia già usate per la dimostrazione dell'invariante iniziale del Size Lemma, che abbiamo dato in 6.4. Intuitivmente questo ridultato è triviale dal momento che $\cdot^{\alpha}$ opera mediante la sostituzione di variabili con altre variabili. Il risultato è stato formalizzato come in 7.4 e dimostrato molto semplicemente per induzione sulla taglia. Il caso induttivo, che utilizza la funzione `ssc`, ha richiesto la dimostrazione di un lemma ausiliario sulla conservatività della taglia sulla famiglia di funzioni `ssc`, che è stato formulato come in 7.5 dimostrato anch'esso con una semplice visita induttiva del tipo Crumble e dei tipi mutui ad esso.

```
1  lemma size_alpha: ∀b,e.∀n.∀(H:fresh_var ⟨b, e⟩≤n). c_size (pi1 … (alpha b e n H))
       = c_size ⟨b, e⟩.
```

Codice 7.4: Enunciato del size lemma per la funzione `alpha`

```
1  lemma ssc_size:
2   (∀c, x, y. ∀(H: inb y c = false). c_size (ssc c x y H) = c_size c) ∧
3    (∀b.∀x, y. ∀(H: inb_b y b = false). c_size_b (ssb b x y H) = c_size_b b) ∧
4     (∀e.∀x, y. ∀(H: inb_e y e = false). c_size_e (sse e x y H) = c_size_e e) ∧
5      (∀v.∀x, y. ∀(H: inb_v y v = false). c_size_v (ssv v x y H) = c_size_v v) ∧
6       (∀s.∀x, y.  ∀(H: inb_s y s = false). c_size_s (sss s x y H) = c_size_s s).
```

Codice 7.5: Lemma di conservatività della taglia per le funzioni `ssc`

7.2.2 Lemma 5.5

Lemma 5.5.1

Il lemma 5.5.1 è la formulazione dinamica del lemma 4.5.1; come abbiamo già osservato, la proprietà di well-namedness è necessaria per garantire che l'operazione $e(x)$ sia deterministica. Per questo motivo, tale proprietà è un risulta-

to cruciale per il determinismo (e la correttezza) delle riduzioni operate dalla Crumble GLAM. Per quanto riguarda la funzione $\cdot^\alpha$, osserviamo che essa ha proprio come obiettivo quello di rinominare un Crumble dimodoché il suo Environment più esterno sia well-named, per questo motivo, la dimostrazione del lemma 5.5.1 non è propriamente la dimostrazione della conservatività della proprietà di well-namedness rispetto alla funzione $\cdot^\alpha$, ma la dimostrazione della proprietà di well-namedness a prescindere dal fatto che il Crumble di partenza lo fosse o meno.

In merito all'uso specifico della funzione $\cdot^\alpha$ nel contesto dell'applicazione della regola $\mapsto_{\beta_v}$, è necessario che la proprietà di well-namedness sia estesa anche alla porzione di Environment sul quale non viene applicata la funzione $\cdot^\alpha$. Per questa dimostrazione, dunque, ho operato in due fasi: dapprima ho dimostrato che la funzione generasse Crumble well-named, per poi dimostrare, per induzione sulla porzione di Environment concatenata a destra del Crumble su cui era applicata la $\cdot^\alpha$, che la proprietà di well-namedness continuasse a valere, dimostrando l'invariante vero e proprio.

Nello specifico, gli enunciati dimostrati sono quelli riportati in 7.6 ed in 7.7. Come si può notare dalla formulazione degli enunciati, in entrambi i casi è stato scelto di aiutarsi nella prova dimostrando, congiuntamente alla tesi, la proprietà `interval_dom` dell'Environment generato dalla funzione `alpha`. La proposizione `interval_dom e n` asserisce che il dominio dell'Environment `e` abbia valori maggiori od uguali a `n`. Assieme ad esso è talvolta utile fare uso del suo complementare `bound_dom`, che identifica un limite superiore al dominio di un Environment, per produrre risultati di well-namedness partizionando gli Environment in porzioni caratterizzati da domini appartenenti a diversi intervalli.

```
lemma w_well_named_alpha:
 ∀b, e. ∀n. ∀H: fresh_var ⟨b,e⟩ ≤ n.
  (w_well_named (pi1 … (alpha b e n H))=true) ∧ interval_dom
      match (pi1 … (alpha b e n H)) with [CCrumble b e ⇒ e] n.
```

Codice 7.6: Enunciato del lemma debole di well-namedness per la funzione `alpha`

```
lemma well_named_alpha:
 ∀f, b, e. ∀n. fresh_var (at ⟨b, e⟩  f) ≤ n →
  match (at ⟨b, e⟩ f) with [ CCrumble b e ⇒ ∀H. (w_well_named
      (pi1 … (alpha b e n H))=true) ∧ interval_dom match (pi1 …
      (alpha b e n H)) with [CCrumble b e ⇒ e] n].
```

Codice 7.7: Enunciato del lemma di well-namedness per la funzione `alpha`

Il lemma debole di well-namedness, per come è formulato, coincide con il goal del caso base della dimostrazione del secondo lemma se si procede per per induzione su `f`, mentre il caso induttivo, sostanzialmente, consiste con la dimostrazione del fatto che la concatenazione di due Environment well-named, laddove essi siano disgiunti, genera nuovamente un dominio well-named. Nello specifico, il predicato di interval-dom è sufficiente per garantire tale condizione e concludere la dimostrazione.

Lemma 5.5.2

Questo risultato, come abbiamo visto a proposito del lemma 4.1.2, è un invariante necessario per garantire la correttezza della disciplina di valutazione closed CbV. Per quanto concerne l'utilizzo della funzione $\cdot^\alpha$ nel contesto dell'applicazione della regola $\mapsto_{\beta_v}$, ci si può limitare ad osservare che l'implementazione che ne abbiamo dato conserva le variabili libere del Crumble su cui è applicata e, dunque, derivare il fatto che `alpha` conservi la chiusura dei termini come corollario della seguente proprietà: $\forall c.FV(c) = FV(c^\alpha)$. A sua volta, il fatto che la funzione `alpha` conservi le variabili fresche è garantito dal risultato espresso dal lemma 7.8, che ancora una volta deriva da un'osservazione fatta sulla funzione di sostituzione `ssc`: $\forall x, c.x \neq z \to x \neq y \to FV(c\{y \leftarrow z\}) = FV(c)$. Una volta dimostrata questa condizione (ancora una volta con una visita induttiva dei tipi mutui con Crumble) siamo in grado di concludere facilmente il risultato.

```
lemma alpha_fv_cons: ∀e, b, n, H. ∀x. fvb x (pi1 … (alpha b e n
    H)) = fvb x ⟨b, e⟩.
```

Codice 7.8: Enunciato del lemma di conservatività dell'insieme $FV(\cdot)$ per la funzione `alpha`

7.2.3 Lemma 5.5.3

Come anticipato mentre commentavamo i motivi per cui il lemma 4.1.3 fosse falso, questo è uno dei risultati in cui avremmo dovuto provvedere a formulare una dimostrazione alternativa poiché quella presentata in [5]. Riprendiamo la dimostrazione che ne è data nell'articolo per provare a trovare strade alternative.

Lemma 7.2.1
Tutti i corpi di funzione in un Crumble c' raggiungibile durante la valutazione della macchina Crumble GLAM sono sottotermini del crumble c iniziali a meno di ridenomina.

Dimostrazione. Le regole $\to_{sub_{var}}$ e $\to_{sub_l}$ possono copiare astrazioni, ma dal momento che le astrazioni erano già nell'ambiente, il claim segue direttamente dall'ipotesi induttiva. Per il caso della $\mapsto_{\beta_v}$, la regola può copiare un corpo di una funzione e ridenominarlo, ma dal momento che per il remark 4.1.3 la traduzione si commuta con la rinomina delle variabili fresche operata dalla funzione $\cdot^\alpha$, il risultato segue direttamente dall'ipotesi induttiva.

La dimostrazione sembra voler procedere per induzione sulla lunghezza della riduzione ed analizzare caso per caso tutte le possibili transizioni. Nello specifico ci troviamo a dover trattare il caso della regola di transizione $\mapsto_{\beta_v}$. Simbolicamente, la parte della dimostrazione in questione vuole procedere come in (7.7), dove $Sp_{c_0}(\cdot)$ è il predicato che denota il fatto che l'argomento sia un corpo del Crumble c_0 iniziale a meno di ridenomine.

$$\begin{gathered}((\lambda x.c_n)w_n, e_v) \mapsto_{\beta_v} (c_{n+1}@[x \leftarrow w_{n+1}])^\alpha e_v = \\ (c_0\{x_{a_1} \leftarrow z_{b_1}\}\dots\{x_{a_m} \leftarrow z_{b_m}\}@[x' \leftarrow w_0\{x_{a_1} \leftarrow z_{b_1}\}\dots\{x_{a_m} \leftarrow z_{b_m}\}])e_v \\ (\underline{t_0}\{x_{a_1} \leftarrow z_{b_1}\}\dots\{x_{a_m} \leftarrow z_{b_m}\}@[x' \leftarrow \underline{u_0}\{x_{a_1} \leftarrow z_{b_1}\}\dots\{x_{a_m} \leftarrow z_{b_m}])e_v\end{gathered} \tag{7.7}$$

Ma a questo punto, $Sp_{c_0}(e_v)$ per ipotesi induttiva dal momento che tutti i corpi di e_v erano corpi del termine prima che si applicasse la transizione. Per quanto riguarda i corpi di funzione in c_0, si vuole far vedere che:

$$(\underline{t_0}\{x_{a_1} \leftarrow z_{b_1}\}\dots\{x_{a_m} \leftarrow z_{b_m}\}@[x' \leftarrow \underline{u_0}\{x_{a_1} \leftarrow z_{b_1}\}\dots\{x_{a_m} \leftarrow z_{b_m}\}]) =$$
$$(\underline{t_0\{x_{a_1} \leftarrow z_{b_1}\}\dots\{x_{a_m} \leftarrow z_{b_m}\}}@[x' \leftarrow \underline{u_0\{x_{a_1} \leftarrow z_{b_1}\}\dots\{x_{a_m} \leftarrow z_{b_m}\}}])$$

Dove la riscrittura è giustificata per il remark (falso) 4.1.3 e la conclusione deriva dal fatto che t_0 sia sottotermine del termine iniziale per il remark 4.5.4. L'utilizzo del remark 4.1.3 è superfluo: la condizione di uguaglianza che esso esprime è troppo forte. La condizione che baseterebbe dimostrare non è un'uguagliaza stretta, ma è meno stringente: per la presenza della clausola "up to renaiming" ci potremmo limitare ad un'uguaglianza modulo rinomine. La stessa clausola non è ben formalizzata: la rinomina può agire anche sulle variabili legate da astrazioni, ma in realtà, dal momento che la $\cdot^\alpha$ sostituisce variabili fresche, le variabili legate da astrazioni non vengono di fatto mai rinominate, per cui si potrebbe riscrivere tale clausola con "up to $\cdot^\alpha$" ed il risultato continuerebbe ad essere vero. Per questa ragione, il lemma può essere riformulato con la condizione generica di ridenomina sostituita da quella di alpha $\cdot^\alpha$ e la prova si può riscrivere osservando che per la funzione $\cdot^\alpha$ valga una proprietà di idempotenza[2] per cui una serie annidata di applicazioni possa essere riassunta con l'applicazione più esterna.

La dimostrazione di questo risultato procede come mostrato da 7.9, dove $\cdot^\alpha_n$ denota l'applicazione della ridenomina con variabile fresca di indice n.

$$(b, e[x \leftarrow b'])^{\alpha\,\alpha}_{n\,m} = (b, e[x \leftarrow b'])^\alpha_m$$
$$((b,e)^\alpha_{n+1}\{x \leftarrow \nu n\}[\nu n \leftarrow b'])^\alpha_m = ((b,e))^\alpha_{m+1}\{x \leftarrow \nu m\}[\nu m + 1 \leftarrow b']$$
$$((b,e)^\alpha_{n+1}\{x \leftarrow \nu n\})^\alpha_m\{n \leftarrow \nu m\}[\nu m \leftarrow b'] = ((b,e))^\alpha_{m+1}\{x \leftarrow \nu m\}[\nu m + 1 \leftarrow b']$$
$$((b,e)^\alpha_{n+1})^\alpha_{m+1}\{x \leftarrow \nu n\}\{n \leftarrow \nu m\}[\nu m \leftarrow b'] = ((b,e))^\alpha_{m+1}\{x \leftarrow \nu m\}[\nu m + 1 \leftarrow b']$$
$$((b,e)^\alpha_{n+1})^\alpha_{m+1}\{x \leftarrow \nu m\}[\nu m \leftarrow b'] = ((b,e))^\alpha_{m+1}\{x \leftarrow \nu m\}[\nu m + 1 \leftarrow b']$$

A questo punto è possibile concludere usando l'ipotesi induttiva. Per poter giungere a questo risultato è stato necessario dimostrare altri due lemmi: il primo asserisce che in virtù delle ipotesi di freschezza delle variabili sostituite dalla funzione `alpha*`, sia possibile commutare l'applicazione della funzione `ss*` all'applicazione della funzione `alpha*`, mentre il secondo asserisce che due sostituzioni consecutive della forma $\{a \leftarrow b\}\{b \leftarrow c\}$ possano essere riassunte da una singola sostituzione $\{a \leftarrow c\}$. Dimostrati questi lemmi, l'enunciato per gli Environment viene formulato come mostrato da 7.9.

```
lemma alpha_e_idem: ∀e, n, m, H1, H2, H3. m n + e_len e →
 (pi1 … (alpha2_e (pi1 … (alpha2_e e n H1)) m H2)) = pi1 …
     (alpha2_e e m H3).
```

Codice 7.9: Enunciato del lemma d'idempotenza per la funzione `alpha_e`

[2]Per essere rigorosi, occorre sottolineare che la proprietà a cui facciamo riferimento non è d'idempotenza in senso propro: la funzione $\cdot^\alpha$ richiede due argomenti, dunque non ha senso parlare d'idempotenza. Più propriamente, dimostreremo che applicazioni successive della funzione $\cdot^\alpha$ annidate su di un termine c, saranno uguali all'esito della chiamata più esterna su c.

L'enunciato del lemma 7.3 richiede un'ipotesi esplicita di freschezza sulla variabile `m`, tale richiesta non è assolutamente limitante perché per la ridenomina possiamo scegliere indici arbitrariamente grandi, inoltre non è nemmeno strettamente necessaria per la dimostrazione del lemma, in quanto si può derivare dalle ipotesi `H1`, `H2` e `H3`. La ragione per cui è stata inserita nell'enunciato è solamente quella di snellirne la dimostrazione senza indebolire lo statement.

Aver dimostrato solo questo lemma e non lemmi analoghi per tipi come Bite o Crumble non è assolutamente limitante: la dimostrazione del caso della funzione `alpha` procede esattamente nello stesso modo, ma con qualche insidia in più dovuta al fatto che il tipo di ritorno della funzione `alpha` non compaia in input nella funzione stessa, quindi è necessario espandere il Crumble restituito dalla funzione più interna per poter avanzare con dimostrazione e ricostruirlo per usare l'ipotesi induttiva, che andrà poi un'altra volta ri-espansa rendendo la prova più tediosa. Tuttavia, occorre osservare che il risultato generico sul Crumble può anche essere espresso come un corollario del risultato sugli Environment, dal momento che per il primo lemma di matching in 7.11 vale quanto espresso in 7.8, dove $\beta(\cdot)$, $\gamma(\cdot)$ sono le funzioni descritte in 7.1.3.

$$\begin{gathered}(([x \leftarrow b]e)^{\alpha}_{n})^{\alpha}_{m}) = \\ ([x \leftarrow b]e)^{\alpha}_{m} \\ ([x' \leftarrow \gamma(b, \beta(e, m))](e)^{\alpha}_{m} = \\ x'\langle (b, e)^{\alpha}_{m}\rangle \end{gathered} \tag{7.8}$$

Tutti i passi di riscrittura sono giustificati da lemmi tecnici relativi alla scomposizione di `alpha*`, `beta*` e `gamma*` rispetto ai costruttori di tipo, alle funzioni di manipolazione e a quelle di plugging già dimostrate formalmente come lemmi tecnici per provare il lemma 5.5.4. Dunque, la dimostrazione formale del lemma 7.9, ci ha permesso di concludere che l'invariante del lemma 5.5.3 vale "up to $\cdot^{\alpha}$" nel caso specifico della regola $\mapsto_{\beta_v}$: non solo abbiamo rimosso l'utilizzo del remark 4.1.3, ma abbiamo formulato la dimostrazione di una proprietà più forte di quella richiesta.

7.2.4 Lemma 5.5.4

Questo risultato è uno degli invarianti legate alla correttezza dell'implementazione della disciplina di valutazione CbV dalla Crumble GLAM. Come abbiamo visto, esso non è formulato in maniera forte come il lemma analogo per il caso iniziale, poiché l'applicazione della $\mapsto_{\beta_v}$ (le cosiddette "overhead transitions") può rompere questo vincolo localmente. Tuttavia, la situazione non è patologica poiché il vincolo viene sempre ricostituito per mezzo di applicazioni di transizioni di overhead prima dell'applicazione della regola $\mapsto_{\beta_v}$. La dimostrazione di questo lemma proposta in [5] faceva riferimento al remark 4.1.3, per questa ragione lo studio che se ne è fatto non è volto a formalizzare un risultato preesistente, quanto più a proporre una strada alternativa alla dimostrazione fallace che ne è data in [5].

Riprendiamo l'enunciato del lemma e la sua dimostrazione al fine di studiare qual è il ruolo nella dimostrazione del remark 4.1.3 e dalla funzione $\cdot^\alpha$.

Lemma 7.2.2
Tutte le decomposizioni $C\langle(b, e_v)\rangle$ *di crumble raggiungibili dalla Crumble GLAM sono tali che tutti i prefissi* C' *di* C*, se decodificati siano right-v evaluation context.*

Dimostrazione. ... Nel caso dell'applicazione della regola $\rightarrow_{\beta_v}$, il claim deriva dal fatto che, per il remark 4.1.3 ed il 5.5.3, ogni c^α sia un sottotermine del termine di partenza a meno di operazioni di ridenomina e dunque dal fatto che la ridenomina non vìoli i right-v evaluation contexts, dal momento che tutti i sotto-termini di partenza si decodificano in right-v evaluation context per il lemma del contextual decoding. ...

Tale prova, simbolicamente, si può riassumere come segue:

$$\begin{gathered} (c^\alpha)_\downarrow = \\ ((c_0)^\alpha)_\downarrow = \\ ((C_0\langle(b,e)\rangle)^\alpha)_\downarrow = \\ (C_0^\alpha\langle(b,e)^\alpha\rangle)_\downarrow = \\ (C^\alpha_{0\downarrow}\langle(b,e)^\alpha_\downarrow\rangle) \end{gathered} \tag{7.9}$$

Dove:

- La prima riscrittura è giustificata da un'applicazione del lemma 5.5.3 che è giustificata dalla dimostrazione che ne abbiamo dato nella sezione precedente.
- La seconda riscrittura non è altro che l'espansione del crumble c_0 sotto forma di plugging.
- La terza riscrittura è una conseguenza della definizione della funzione $\cdot^\alpha$ che tuttavia non viene dimostrata in [5], ma è stata dimostrata vera formalmente come un lemma tecnico per dimostrare la 7.10
- L'ultima riscrittura è dimostrato da un apposito lemma tecnico in [5].

A questo punto, però la dimostrazione comincia ad utilizzare risultati impropri:

$$\begin{gathered} C^\alpha_{0\downarrow} = \\ (C_0\{x_{a_1} \leftarrow y_{b_1}\}\ldots\{x_{a_1} \leftarrow y_{b_1}\})_\downarrow = \\ (\underline{R}\{x_{a_1} \leftarrow y_{b_1}\}\ldots\{x_{a_1} \leftarrow y_{b_1}\})_\downarrow = \\ (\underline{R\{x_{a_1} \leftarrow y_{b_1}\}\ldots\{x_{a_1} \leftarrow y_{b_1}\}})_\downarrow \end{gathered} \tag{7.10}$$

Dove:

- La prima riscrittura è un'espansione della chiamata di $\cdot^\alpha$ dove a_i e b_i sono due successioni di nomi di variabili, dove vale che b_i è composta da soli nomi freschi.
- La seconda riscrittura è impropria: in tutto [5] non si fa menzione dell'esistenza di una funzione di underline su contesti, ma possiamo assumere che esista e si possa definire in maniera abbastanza naturale. R è $C_{0\downarrow}$.
- La terza riscrittura è l'applicazione del remark 4.1.3.

A questo punto la conclusione dovrebbe derivare dall'osservazione che la ridenomina di un right-v context è ancora un right-v context, anche se questo lemma, che intuitivamente sembra vero, non è né dimostrato né enunciato in [5]. Nonostante ciò, il problema da risolvere si trova più a monte dell'uso di tale lemma: anziché procedere con il lemma 4.1.3, si potrebbe pensare più propriamente di applicare la commutabilità della funzione alpha rispetto alla read-back, ma purtroppo, come si legge dal confronto della (7.11) e della (7.12), l'operazione $\cdot^\alpha$ commuta con la $\cdot_\downarrow$ solo modulo α-conversioni.

$$\begin{aligned}(\lambda x.y, [y \leftarrow x])_\downarrow =& (\lambda x.y)_\downarrow \{y \leftarrow x\} = \\ & (\lambda x.y)\{y \leftarrow x\} = \lambda x'.x\end{aligned} \tag{7.11}$$

$$\begin{aligned}(\lambda x.y, [y \leftarrow x])^\alpha_\downarrow = (\lambda x.z, [z \leftarrow x])_\downarrow =& (\lambda x.z)_\downarrow \{z \leftarrow x\} = \\ & (\lambda x.z)\{z \leftarrow x\} = \lambda x''.x\end{aligned} \tag{7.12}$$

La differenza è che le variabili x' e x'' sono generate dalla funzione di sostituzione su λ_p-termini. Il fatto di aver rinominato la variabile y sotto l'astrazione λx con una variabile fresca fa sì che l'α-conversione conseguente la sostituzione possa scegliere valori diversi con cui ridenominare la x legata. Per questa ragione, anche l'ipotetica dimostrazione alternativa che sfrutta la commutazione della funzione $\cdot^\alpha$ con la read-back non è percorribile. Per questa ragione abbiamo scelto di dimostrare direttamente che la funzione `alpha` rispetta i right-v evaluation contexts. L'enunciato, molto simile a quello del lemma 4.5.5 (cfr. 6.12), è stato formulato come in 7.10. Una volta dimostrato tale lemma, la conclusione della 5.5.4 potrà essere derivata banalmente dall'osservazione 5.5.3, per la quale tutti i prefissi raggiungibili dalla Crumble GLAM sono esprimibili come un sottotermine iniziale a cui è stata applicata una sola chiamata alla funzione `alpha` e con il lemma 7.10 concludere la tesi.

```
lemma four_dot_five_dot_five_alpha: (∀t, s, c, C, l. ∀(Hm :
    (fresh_var_t t ≤ s)).
∀H'.
(pi1 Crumble ? (alpha_c (fst … (underline t s)) l H'))  =
    plug_c C c
  → rv_context (cc_read_back C) ) ∧
(∀v:pValue.True).
```

Codice 7.10: Enunciato del lemma che dimostra che la funzione alpha rispetta i right-v evaluation contexts

La dimostrazione di questo risultato è stata impostata analogamente alla dimostrazione del lemma 4.5.5 descritta in 6.3.4: ancora una volta procediamo per induzione e scomponiamo l'Environment per ricostruire i CrumbledContext necessari alle due ipotesi induttive per poter concludere che le chiamate ricorsive hanno generato TermContext che soddisfano il predicato `rv_context`. Tuttavia la dimostrazione è stata appesantita da un notevole overhead dovuto al fatto che la funzione `alpha` richieda esplicitamente la dimostrazione di condizioni di freschezza dei parametri per poter essere invocata, inoltre i lemmi di matching degli Enviroment che sono stati dimostrati per il lemma 4.5.5 sono dovuti essere ri-dimostrati ibridati alla funzione `alpha`. Prendiamo come esempio il caso del lemma 6.13: formularlo direttamente come per la dimostrazione dell'enunciato non ibridato sarebbe stato troppo complesso, per tale motivo si è preferito riformularlo mediante i due lemmi in 7.11, i quali combinati permettono di effettuare matching simili a quello di 6.13.

```
lemma alpha_push: ∀ e, y, b, n, H.
 pi1 … (alpha_e (push e [y←b]) n H) =push (pi1 … (alpha_e e n
     (alpha_push_aux1 … H)))
  [ν(n+e_len e)←pi1 … (gamma_b b (beta_e e n) (alpha_push_aux2
      … H))].

lemma alpha_e_concat: ∀f, e, n, H.
 pi1 … (alpha_e (concat e f) n H) =
 concat (pi1 … (gamma_e (pi1 … (alpha_e e (n+(e_len f))
     (alphae_concat_aux2 f e n H))) (beta_e f n)
     (alpha_e_concat_aux3 … H)))
        (pi1 … (alpha_e f n (alpha_e_concat_aux2 … H))).
```

Codice 7.11: Lemmi di matching per gli ambienti ibridati alla funzione alpha

Per giungere alla prova del lemma sono stati dimostrati anche molti altri risultati di matching degli ambienti, di decomposizione del plugging e di conversione delle funzioni della famiglia `alpha*` con quelle della famiglia `beta*`. Modificando alcuni dettagli implementativi delle funzioni della famiglia `ss*` è stato verificato che, complessivamente, è stato raggiunto un livello di astrazione tale da permettere di concludere tesi complesse come il lemma in questione senza fare riferimento ai dettagli implementativi della funzione di sostituzione semplice. Nonostante l'alto livello di astrazione, la gestione delle ipotesi per invocare la funzione $\cdot^{\alpha}$ (e le analoghe per la funzione `beta`) hanno fatto esplodere la taglia della dimostrazione dalle 450 richieste per il lemma 4.5.5 non ibridato alle quasi 1000 righe per la dimostrazione del lemma ibridato, misurate senza contare l'espansione dei lemmi tecnici che vengono richiamati durante la dimostrazione.

Capitolo 8

Conclusioni

Nel presente lavoro ho formalizzato, per mezzo del dimostratore interattivo Matita, parte della teoria che soggiace alle crumbling abstract machines: un'implementazione ragionevole del λ-calcolo presentata in [5].

In particolare, data l'assenza di lavori precedenti in quest'ambito, mi sono dedicato alla formalizzazione della fase di inizializzazione della prima macchina descritta in [5]: la Crumble GLAM. Tale lavoro ha comportato la definizione formale di tipi di dati astratti in grado di rappresentare i dati su cui lavora la macchina, la formalizzazione di diverse funzioni di manipolazione di tali dati nonché delle funzioni di traduzione da termini del λ_p-calcolo in crumbled form e la sua inversa.

Per far ciò ho dovuto introdurre *ex-novo* meccanismi e strumenti per la gestione formale dei nomi di variabile, un aspetto che in [5] era particolarmente carente, anche a causa dell'alto livello di astrazione con cui di norma questo ambito viene trattato in letteratura.

L'implementazione di meccanismi formali di gestione dei nomi di variabile, mi ha permesso di riscontrare alcune falle nella teoria descritta in [5], in particolare mi riferisco al secondo ed al terzo punto del remark 6.3.6 (che in [5] prende il nome di "remark 4.1").

I problemi indotti dall'incorrettezza del remark 4.1.2 non avevano particolari influenze negative sulla stabilità dell'intera teoria, tanto che mi è bastato riformulare tale risultato con un enunciato più debole ma che fosse sufficientemente espressivo per permettere di utilizzarlo al posto del precedente.

Al contrario, i problemi indotti dall'incorrettezza del remark 4.1.3 si riverberavano negativamente anche su parti più sottili della teoria, che riguardavano il comportamento dinamico della macchina Crumble GLAM; così, per poter proporre soluzioni alternative alle dimostrazioni che facevano uso del risultato in questione, ho deciso di formalizzare anche parte del comportamento dinamico della macchina Crumble GLAM.

Nello specifico, gli aspetti dinamici che sono stati oggetto di formalizzazione sono stati quelli riguardanti la funzione $\cdot^{\alpha}$, che ha lo scopo di ridenominare le variabili del dominio di una specifica crumbled form, utilizzata in fase di

valutazione delle applicazioni al fine di evitare conflitti di nomi di variabile. Sulla funzione $\cdot^{\alpha}$, oltre al rispetto degli invarianti della Crumble GLAM, sono stati dimostrati formalmente anche le sotto-derivazioni di teoremi in cui veniva fatto uso specifico del remark 4.1.3 per provare risultati specifici sul suo comportamento.

In conclusione, dunque, mi è stato possibile risolvere formalmente tutti i problemi causati dalle fallacie che ho potuto individuare formalizzando gli aspetti di inizializzazione della macchina Crumble GLAM, nonché di verificare formalmente la correttezza di tutti gli altri risultati a cui mi sono dedicato.

Appendice A

Matita: uso avanzato

A.1 Introduzione ai proof assistants

Come inteso dal titolo di questa sezione, lo scopo della presente trattazione è quello di prendere in esame alcuni aspetti peculiari dell'uso di Matita per cui, al fine di affrontare alcune parti di questa appendice, è necessaria una discreta conoscenza del proof-assistant. Una lettura consigliata a questo fine può essere per esempio [6].

Per dimostrare formalmente la correttezza della teoria abbiamo scelto di utilizzare il dimostratore interattivo di teoremi Matita. Molti applicativi di questo tipo sfruttano i risultati offerti dall'isomorfismo di Curry-Howard-Kolmogorov per permettere la costruzione di dimostrazioni corrette. Secondo tale corrispondenza, esiste una biezione fra λ-termini ben tipati in un determinato sistema di tipi e derivazioni corrette di enunciati di una determinata logica associata a tale sistema di tipi. Per esempio, al sistema di tipi del λ-calcolo tipato semplice corrisponde la logica minimale. Arricchendo il sistema di tipi applicato ai termini, aumenta il potere espressivo della logica isomorfa al calcolo in questione. Per l'isomorfismo di Curry-Howard-Kolmogorov, i costruttori di tipo possono essere visti come regole di introduzione di connettivi o quantificatori, mentre i "distruttori" (il pattern matching ad esempio) come regole di eliminazione di tali costruttori. Per questo motivo, i dimostratori assistiti rappresentano dati e le funzioni mediante λ-termini tipati, dimodoché, con un proof-assistant, per dimostrare un enunciato P basti costruire un λ-termine con tipo P.

È facile rendersi conto che i λ-termini che codificano prove di enunciati spesso possano essere difficilmente costruibili manualmente per ragioni di dimensioni, di complessità e di scarsa naturalezza del procedimento stesso. Per agevolare la costruzione di questi termini (i.e delle prove), spesso i proof assistants mettono a disposizione dell'utente una serie di tactics: operatori che permettono di maneggiare in maniera automatica il termine della dimostrazione mediante passi che spesso emulano quelli che verrebbero adottati in una dimostrazione informale.

Esempio A.1.1. Si immagini di avere una formula logica nella forma $P \to Q$ dove P e Q sono predicati di complessità arbitraria. Per dimostrarla informalmente si procederebbe mediante l'assunzione dell'ipotesi P e la successiva costruzione di una dimostrazione di Q entro la quale può eventualmente comparire P. Per sintetizzare questo comportamento, i più diffusi proof assistants mettono a disposizione un apposito tactic: in Matita la sintassi per questa operazione è semplicemente `#H` dove `H` è il nome che l'utente sceglie di dare all'ipotesi assunta. Il tactic `#` agisce sul termine di prova costruendo una λ-astrazione della forma $\lambda x : P.t$ in cui t è il termine che dimostra il subgoal Q e $\lambda x : P.$ rappresenta l'assunzione dell'ipotesi x di tipo P. La corrispondenza fra le due operazioni (a livello logico e a livello del calcolo) è verificabile semplicemente: informalmente, dopo aver assunto P, procederemmo alla costruzione di una dimostrazione del nuovo goal Q mediante l'eventuale uso dell'ipotesi P, mentre a livello del λ-termine ci riduciamo alla costruzione di un termine $t : Q$ entro il quale possiamo usare il termine $x : P$.

Nonostante la possibilità di certificare la correttezza di dimostrazioni sia estremamente vantaggiosa, l'utilizzo avanzato dei proof assistants costringe l'utente ad affrontare numerose difficoltà che in un contesto di dimostrazione informale non avrebbe dovuto affrontare. Il fine di questa sezione è quello di riassumere le problematiche che sono state affrontate nella formalizzazione qui presentata, dimodoché sia fornita una giustificazione a scelte implementative o dimostrative apparentemente controintuitive. Inoltre, si spera che la presente trattazione possa essere di riferimento per i lettori interessati ad approfondire l'uso alcune tecniche diffuse nell'ambito della dimostrazione formale per mezzo di proof assistants, ma che di rado hanno un'adeguata documentazione.

Per fa ciò, tratteremo in principio il caso dell'implementazione della funzione di sostituzione su λ_p-termini. In questo modo campioneremo le principali difficoltà che siamo stati costretti ad affrontare durante la formalizzazione. In seguito aggiungeremo alcune trattazioni di carattere più generale o che semplicemente non trovano riscontro nella funzione di sostituzione. Infine, trattare in questa sede tale funzione ci consentirà di rimandare a questa sezione eventuali riferimenti ad essa in altre sezioni del lavoro.

A.2 p_Subst

La definizione canonica di sostituzione su termini, come riportata da [7], è la seguente:

$$\begin{aligned} x\{x \leftarrow t\} &:= t \\ x\{y \leftarrow t\} &:= x \quad \text{se } x \neq y \\ (t_1 t_2)\{x \leftarrow t\} &:= (t_1\{x \leftarrow t\})(t_2\{x \leftarrow t\}) \\ (\lambda z.s)\{x \leftarrow t\} &:= \lambda z.(s\{x \leftarrow t\}) \end{aligned} \tag{A.1}$$

Riguardo all'ultimo caso è necessario specificare che la sostituzione avviene in questo modo se e solo se $x \neq y \wedge y \notin FV(t)$. Nel caso di $(\lambda x.s)\{x \leftarrow t\}$, siccome

nel termine non appaiono occorrenze libere di x, l'esito della sostituzione sarà semplicemente $(\lambda x.s)$. Mentre, nel caso più complesso di $y \in FV(t)$, l'atteggiamento della letteratura non è uniforme e spesso, nel contesto di trattazioni informali, le definizioni date per questo caso sono deboli da un punto di vista algoritmico: per esempio sempre Barendregt in [7] liquida entrambi i casi in esame imponendo che in ogni termine l'insieme delle variabili legate e quello delle variabili libere siano disgiunti, ma questo vincolo non era presente in [5], dunque non avrebbe avuto senso imporlo. Un approccio più generico, sempre suggerito in [7], può essere quello di imporre che i λ-termini siano i rappresentanti delle loro classi di equivalenza modulo alpha-conversione ed implementare la funzione di sostituzione dimodoché fosse una funzione fra classi di equivalenza. In [5], pur non essendo stata definita alcuna funzione di sostituzione, si tratta quest'ultima come se fosse una funzione `pTerm` → `Variable` → `pTerm` → `pTerm`, inoltre, è espressamente scritto che tutte le uguaglianze fra λ-termini presenti nell'articolo vadano intese in senso stretto. È stato dunque scelto di adottare un approccio largamente diffuso nella letteratura: quello di α-convertire la λ-astrazione $\lambda x.s$, qualora s contenga occorrenze libere della variabile x in modo che x non compaia in t.

A questo punto della trattazione, la funzione p_Subst da implementare, avrebbe dovuto avere la seguente forma:

$$\begin{aligned} x\{x \leftarrow t\} &\coloneqq t \\ x\{y \leftarrow t\} &\coloneqq x \quad \text{se } x \neq y \\ (t_1 t_2)\{x \leftarrow t\} &\coloneqq (t_1\{x \leftarrow t\})(t_2\{x \leftarrow t\}) \\ (\lambda x.s)\{x \leftarrow t\} &\coloneqq (\lambda x.s) \\ (\lambda z.s)\{x \leftarrow t\} &\coloneqq \lambda z.(s\{x \leftarrow t\}) \quad \text{se } z \notin FV(t) \\ (\lambda z.s)\{x \leftarrow t\} &\coloneqq \lambda k.(s\{z \leftarrow k\}\{x \leftarrow t\}) \quad \text{se } z \in FV(t) \end{aligned} \tag{A.2}$$

Con k fresca contemporaneamente in $\lambda z.s$ e in t.

Ora, nonostante nella (A.2) si celi ancora un'imprecisione grave, proviamo a tradurre alla lettera tale funzione in Matita e commentiamo il risultato.

```
1   let rec p_subst_t t x (s:pTerm) on t :=
2    match t with
3    [ val_to_term v ⇒ match v with
4      [ pvar y ⇒ match veqb x y with
5        [ true ⇒ s
6        | false ⇒ val_to_term (pvar y)
7        ]
8      | abstr y t ⇒ val_to_term (match fvb_t y s with
9        [ true ⇒ let k :=  (ν(max (S match x with [variable nx⇒ nx])
10                              (max (fresh_var_t s) (fresh_var_t s)))) in
11                abstr k (p_subst_t (p_subst_t t y (val_to_term (pvar k))) x s)
12       | false ⇒ abstr y (p_subst_t t x s)
13       ])
14     ]
15   | appl t1 t2 ⇒ appl (p_subst_t t1 x s) (p_subst_t t2 x s)
16   ]
17  .
```

Codice A.1: Prima bozza di implementazione della funzione `p_subst`

Dall'esempio di codice riportato qui sopra possiamo possiamo trarre una serie di osservazioni:

- Seguendo la consuetudine già vista per esempio in 6.3, siccome i tipi pTerm e pValue sono mutui fra loro, avremmo potuto definire la funzione `p_subst` mediante due funzioni mutualmente ricorsive: una che effettua le sostituzioni su `pTerm` ed una che effettua le sostituzioni su `pValue`. Questo approccio non sarebbe stato sbagliato e avrebbe prodotto, come effetto collaterale, la definizione di una funzione di sostituzione di termini su valori. Entrambe le definizioni sono equivalenti nella pratica, ma per economia di definizioni, ci siamo limitati senza perdere di generalità, a definire una sola funzione di sostituzione con signature `pTerm` → `Variable` → `pTerm` → `pTerm`. In questo modo abbiamo prodotto una definizione di funzione che nella sua forma ricorda molto di più la funzione canonica di sostituzione.

- Per α-convertire il termine `t` è una buona idea quella di utilizzare nuovamente la funzione di sostituzione: in tal modo si riduce il numero di funzioni definite e qualora si approcci una dimostrazione di una proprietà della `p_subst` non è necessario introdurre lemmi tecnici per l'analisi di un'ipotetica funzione di α-conversione, ma possono tornare utili quelli già dimostrati sulla `p_subst`.

- La funzione mostrata è corretta e calcolabile, ma non sarebbe definibile in Matita per una ragione strettamente legata alla teoria della calcolabilità. Come è noto, quello della terminazione non è un problema decidibile, tuttavia è necessario che le funzioni definite in Matita siano terminanti. Per garantire ciò, Matita implementa un'approssimazione da dentro di questa proprietà la quale prevede che tutte le funzioni definite debbano avere un numero di costruttori di tipo annidati sul termine sul quale si procede per ricorsione strettamente decrescente rispetto a quelle del chiamate. Tale vincolo non è rispettato dalla funzione che abbiamo mostrato poiché, per α-convertire il termine `t`, abbiamo annidato due chiamate ricorsive alla `p_subst_t`, rompendo il vincolo appena enunciato.

Per porre rimedio a questo problema, data anche la necessità di avere una funzione che lavorasse su termini anziché su classi di equivalenza, le principali soluzioni erano le seguenti: generalizzare la `p_subst` in modo che ammettesse la possibilità di effettuare sostituzioni simultanee e rappresentare l' α-conversione come una nuova sostituizione, oppure quella di definire la `p_subst` per induzione sulla taglia del termine anziché sul termine stesso.

Una funzione di sostituzione simultanea avrebbe richiesto l'utilizzo di un insieme di sostituzioni (per maggior semplicità avrebbe potuto anche essere implementato per mezzo di una lista) da passare in rassegna ogniqualvolta la `p_subst` avesse raggiunto una variabile. Dunque, tale approccio avrebbe introdotto la necessità di definire meccanismi di memoization delle variabili astratte nel sottotermine corrente (le quali andrebbero rimosse da tale insieme), nonché

la definizione di una funzione ausiliaria che scorresse la lista delle sostituzioni ed eventualmente ne estraesse quella da operare tenendo conto delle entry rimosse a causa di λ-astrazioni. Se inoltre avessimo voluto definire una funzione general purpose, sarebbe stata un'ottima pratica quella di aggiungere fra gli argomenti di tale funzione un ipotesi che in grado di garantire che tutte le variabili nel dominio delle sostituzioni fossero distinte. Al contrario, senza tale ipotesi sarebbe stato necessario definire una nozione coerente di precedenza fra le sostituzioni.

Una funzione di sostituzione singola per induzione sulla taglia, invece, avrebbe reso necessario introdurre la nozione di taglia di un λ-termine che però, come abbiamo visto, è già presente in [5] dal momento che se ne fa uso per l'enunciato e la dimostrazione del Size Lemma (6.4). Di contro, questo approccio avrebbe reso necessario modificare il tipo di ritorno della `p_subst` a causa della necessaria introduzione di Σ-tipi, nonché la dimostrazione di un considerevole numero di lemmi secondari volti a garantire la correttezza della funzione. Data la simile complessità delle due implementazioni ipotizzate, ma visto anche che la funzione di taglia di un termine sarebbe sicuramente tornata utile in seguito, la scelta è ricaduta su questa seconda alternativa. Per giustificare le precedenti considerazioni, prendiamo in esame una funzione di sostituzione che non fa uso di Σ-tipi. Una possibile implementazione naïve di una funzione di questo tipo avrebbe potuto essere la seguente:

```
1  let rec p_subst_t t x (s:pTerm) on t :=
2   match t with
3   [ val_to_term v ⇒ match v with
4     [ pvar y ⇒ match veqb x y with
5       [ true ⇒ s
6       | false ⇒ val_to_term (pvar y)
7       ]
8     | abstr y t ⇒ val_to_term (match fvb_t y s with
9       [ true ⇒ let k :=  (ν(max (S match x with [variable nx⇒ nx])
10                               (max (fresh_var_t s) (fresh_var_t s)))) in
11                abstr k (p_subst_t (p_subst_t t y (val_to_term (pvar k))) x s)
12      | false ⇒ abstr y (p_subst_t t x s)
13      ])
14    ]
15  | appl t1 t2 ⇒ appl (p_subst_t t1 x s) (p_subst_t t2 x s)
16  ]
17 .
```

Codice A.2: Prototipo della `p_subst` per induzione sulla taglia

In questa definizione, infatti, assumiamo che `n` sia la taglia del termine `t` e produciamo per induzione su di esso. Il caso d'uso tipico di una funzione di questo tipo sarebbe il seguente: `p_subst_size (t_size t) y t' t`, ove `t_size` è una funzione che calcola la taglia di `pTerm` definita come in figura A.3. Chiaramente, però `n` (i.e. `t_size t`) non deve essere un valore qualsiasi, poiché se tale valore si esaurisse prima che fosse esaurito il termine, la soluzione risultante sarebbe solo parziale. Questo vincolo però non è catturato dalla funzione `p_subst` qui riportata; per procedere opportunamente dovremmo, ogniqualvolta chiamiamo questa funzione, produrre (*o richiedere*) una dimostrazione del tipo `t_size t` $\leq$ `n`.

```
1  let rec t_size t on t :=
```

```
2    match t with
3     [ val_to_term v ⇒ v_size v
4     | appl t1 t2 ⇒ S ((t_size t1) + (t_size t2))
5     ]
6
7   and v_size v on v :=
8    match v with
9     [ pvar v ⇒ 1
10    | abstr x t ⇒ S (t_size t)
11    ]
12  .
```

Codice A.3: Definizione della funzione `t_size`

Dal momento che per l'isomorfismo di Curry-Howard-Kolmogorov le prove sono particolari λ-termini, per fare questo, basterebbe richiedere fra gli argomenti della `p_subst` anche questo termine. In questo modo, per poter chiamare la funzione di definizione, il suo ipotetico utilizzatore deve dapprima garantire che il parametro `n` sia sufficientemente elevato. Ma dal momento che la funzione `p_subst` è ricorsiva, anche quando viene effettuata una chiamata ricorsiva su di un termine, deve essere costruita una dimostrazione che garantisce la coerenza dei parametri attuali, cosicché la funzione assumerebbe una formula simile a quella riportata in A.4.

```
1   let rec p_subst_size (n: nat) y t' t (H: t_size t ≤ n) on n :=
2    match n with
3     [ O ⇒ ?
4     | S m ⇒ match t with
5        [ val_to_term v ⇒ match v return λv. pTerm with
6         [ pvar x ⇒  match veqb y x  with
7             [ true ⇒ t'
8             | false ⇒ (val_to_term (pvar x))
9             ]
10        | abstr x t1 ⇒ match veqb y x with
11            [ true ⇒ (val_to_term (abstr x t1))
12            | false ⇒ match fvb_t y (val_to_term (abstr x t1)) with
13              [ true ⇒ let z := (max (S match y with [variable n ⇒ n]) (max (S match
                     x with [variable nx⇒ nx]) (max (fresh_var_t t1) (fresh_var_t t'))))
14                  in (val_to_term (abstr (ν(z)) ((p_subst_size m y t'
                         (p_subst_size m x (val_to_term (pvar ν(z))) t1 ?) ?))))
15              | false ⇒ (val_to_term (abstr x t1))
16              ]
17            ]
18        ]
19       | appl t2 u ⇒  (appl ((p_subst_size m y t' t2 ?)) ((p_subst_size m y t' u ?)))
20       ]
21    ]
22  .
```

Codice A.4: Prototipo della `p_subst` per induzione sulla taglia e con richiesta di dimostrazione `H : t_size t` $\leq$ `n`

A riguardo di tale definizione, occorre introdurre il termine `?`. Come riportato da [6], il termine `?` rappresenta un argomento implicito che deve essere inferito dal sistema. Il sistema di inferenza di Matita sfrutta bi-direzionalmente sia il tipo atteso che quello inferito per il termine `?`, in modo da poterlo ricostruire. Qualora questa operazione non fosse possibile, allora viene automaticamente aperto un goal in cui è compito dell'utente costruire i termini che il

sistema non è stato in grado di inferire. Nel caso di A.4, il sistema non è in grado di inferire nessuna delle prove le prove sostituite da `?`, così diventa compito dell'utente dimostrarle dopo il corpo della funzione. Una generalizzazione del termine `?` è il termine ... che apparirà in seguito e che rappresenta una sequenza arbitrariamente lunga di `?`.

Sempre in riferimento alla A.4, possiamo osservare che alla riga 3 compare solo il termine `?`. Questa scelta non è casuale: per la definizione A.3 nessun termine può avere taglia nulla, quindi ci aspettiamo che la situazione descritta alla riga 3 conduca ad un assurdo da cui è possibile costruire qualsiasi tipo e dunque anche un `pTerm`. Infatti il `?` di riga 3 apre un goal del tipo `pTerm`, per concluderlo ci possiamo limitare ad applicare il principio di eliminazione di `False` per poi procedere per casi su `t` e far vedere come in ogni caso l'ipotesi H sia assurda.

Tuttavia, nemmeno questo terzo raffinamento della funzione sarà quello definitivo: come si vede dall'intestazione, il tipo dell'ipotesi `H` è `t_size t` $\leq$ `n`: ciò significa che quando andiamo a fare match su `t` al fine di introdurne i due costruttori di tipo ed i sottotermini immediati, tale operazione non ha effetto su `H` che, essendo dichiarata nell'intestazione, continuerà ad avere tipo `t_size t` $\leq$ `n` nonostante `t` sia già stato eliminato. Per fare ciò osserviamo il tipo che Matita attribuisce alla funzione `p_subst`:

$$\forall \texttt{n} : \texttt{nat}.\texttt{Variable} \rightarrow \texttt{pTerm} \rightarrow \forall \texttt{t} : \texttt{pTerm}.\texttt{t_sizet} \leq \texttt{n} \rightarrow \texttt{pTerm} \qquad \text{(A.3)}$$

Ciò che è interessante riguardo a questa definizione è che, come era da aspettarsi, Matita rappresenta le funzioni con più parametri nel modo convenzionale del λ-calcolo: mediante un numero di λ-astrazioni annidate pari al numero di parametri della funzione da rappresentare. Dunque il tipo risultante della funzione è dato dal tipo di ciascun parametro, seguito da $\rightarrow$ (indotto dalla λ-astrazione), seguito infine dal tipo del corpo della funzione. Per questo motivo, all'interno del corpo della funzione i tipi di tutti i parametri sono fissati. Come abbiamo detto però, noi non vogliamo che il tipo di `H` sia fissato, ma vorremmo che il sistema di tipi di Matita fosse in grado di attribuire ad ogni ramo del `match t with` il tipo opportuno ad `H`. Per farlo possiamo quindi spostare il tipo di `H` verso il tipo di ritorno: in questo modo il tipo complessivo della `p_subst` risulta invariato, ma, con l'aiuto del tipo di ritorno esplicito accanto al costrutto `match`, siamo in grado di indurre il sistema di tipi di Matita a riconoscere i tipi specifici di `H` nei diversi rami. Applicando questa costruzione, il corpo della `p_subst` diventa come quello riportato in figura A.5.

```
1   let rec p_subst_size (n: nat) y t' t on n: t_size t ≤ n → pTerm :=
2    match n return λn. t_size t ≤ n → pTerm with
3    [ O ⇒ λH. ?
4    | S m ⇒ match t return λt.  t_size t ≤ S m → pTerm with
5       [ val_to_term v ⇒ match v return λv.  t_size (val_to_term v) ≤ S m → pTerm
             with
6        [ pvar x ⇒  λH. match veqb y x  with
7            [ true ⇒ t'
8            | false ⇒ (val_to_term (pvar x))
9            ]
10       | abstr x t1 ⇒ λH. match veqb y x with
11          [ true ⇒ (val_to_term (abstr x t1))
```

```
12          | false ⇒ match fvb_t x t' with
13            [ true ⇒ let z := (max (S match y with [variable n ⇒ n]) (max (S match
                  x with [variable nx⇒ nx]) (max (fresh_var_t t1) (fresh_var_t t'))))
14                    in (val_to_term (abstr (ν(z)) ((p_subst_size m y t'
                           (p_subst_size m x (val_to_term (pvar ν(z))) t1 ?) ?))))
15            | false ⇒  (val_to_term (abstr x ((p_subst_size m y t' t1 ?))))
16            ]
17          ]
18       ]
19      | appl t2 u ⇒ λH. (appl ((p_subst_size m y t' t2 ?)) ((p_subst_size m y t' u
           ?)))
20      ]
21   ]
22  .
```

Codice A.5: Prototipo della `p_subst` con tipo di ritorno `t_sizet` $\leq$ `n` $\rightarrow$ `pTerm`

Si osserva che le dimostrazioni da costruire per la A.5 sono in molti casi molto semplici, come evidenzia l'esempio qui sotto:

Esempio A.2.1. Si prenda per il caso in cui $\mathtt{t} = \mathtt{t_1 t_2}$ (come avviene alla riga 22 della figura A.5), fornire una dimostrazione del fatto che `t_size` $\mathtt{t_i} \leq \mathtt{n}$ sapendo che `t_size t` $\leq$ `n` è molto semplice: basta osservare che, per la definizione della funzione `t_size`, `t_size` $\mathtt{t_i} \leq$ `t_size t` e dunque `t_size` $\mathtt{t_i} \leq \mathtt{n}$.

Tuttavia, sempre in riferimento al prototipo di `p_subst` dato in figura A.5, si può osservare che alla riga 15 si trovano due chiamate annidate della `p_subst`. Per la chiamata più interna si può sempre garantire la proprietà in maniera simile a quella riportata dall'esempio A.2.1, mentre per la chiamata più esterna ciò non è possibile: durante la definizione della funzione `p_subst` le uniche informazioni che si hanno sul comportamento della funzione sono quelle riguardanti la sua signature che è riportata in A.3. Per questa ragione, la chiamata più esterna dovrebbe essere in grado di dedurre la taglia di `(p_subst_size m x (val_to_term (pvar ν(z))) t1 ?)` solo dalla A.3, ma affinché ciò sia possibile è necessario fare in modo di arricchire il tipo di ritorno della `p_subst`. Una soluzione possibile a questo problema è quella di utilizzare un Σ-tipo. Per capire in che modo i Σ-tipi possano aiutarci a risolvere il problema della A.5, riportiamo prima la definizione che viene data nella libreria `basics/types.ma` del tipo `Sig`.

```
record Sig (A:Type[0]) (f:A→Prop) : Type[0] := {
    pi1: A
  ; pi2: f pi1
  }.
```

Codice A.6: Definizione del tipo Sig in Matita

Ricorsivamente, per essere in grado di interpretare la definizione in A.6, occorre prima introdurre una definizione del costrutto `record`: come chiarito in [6], esso non è altro che zucchero sintattico per la costruzione di un tipo algebrico con un solo costruttore che racchiude un numero finito di campi (in questo caso un costruttore `mk_Sig` con due campi: `pi1` di tipo `A` e `pi2` di tipo `f pi1`). I nomi dei campi di un record sono automaticamente utilizzati per definire

le funzioni di proiezione sui campi che in questo caso sono `pi1` e `pi2`. Alla luce di ciò è abbastanza semplice capire in che cosa consistano i Σ-tipi: essi sono coppie $\langle a, h \rangle$ dove `a : A` è un elemento di tipo `A` e `h : a` → `Prop` è una prova che la proprietà `h` è valida per `a`. En passant, come fatto notare sia da [6] e da [11], una visione suggestiva di questo tipo di dati può essere quella per cui un elemento `x : Sig A P` sia l'insieme $\{x \in A \mid P\}$.

A questo punto è stato necessario trovare una `P` che esprimesse la taglia dei termini restituiti dalla `p_subst`. Un semplice modo per farlo, detto `u` il termine restituito dalla `p_subst` è il seguente:

$$\texttt{t_size u} = (\texttt{t_size t}) + (\texttt{free_occ_t y t}) * (\texttt{t_size t}' - 1) \tag{A.4}$$

Tuttavia, questa definizione da sola non è ancora sufficiente: per poterla utilizzare è necessario riuscire a contare le occorrenze di `y` in `t` (che nel caso specifico di riga 15 è `(p_subst_size m x (val_to_term (pvar` $_\nu$`(z))) t1 ?))`. Ancora una volta, è necessario sapere qualcosa in più sul comportamento della `p_subst`. Il modo più naturale per farlo, dunque, è quello di rendere ancora più espressiva la proposizione `P` mediante l'introduzione di un'altra proprietà da congiungere alla (A.4). Un modo per contare le occorrenze libere di una variabile qualsiasi `z` in `u` è formulare la P nel seguente modo:

$$\begin{aligned}
&\texttt{t_size u} = (\texttt{t_size t}) + (\texttt{free_occ_t y t}) \cdot (\texttt{t_size t}' - 1) \land \\
&\forall \texttt{z. free_occ_t z u} = \texttt{match veqb y z with} \\
&[\,\texttt{true} \rightarrow (\texttt{free_occ_t z t}) * (\texttt{free_occ_t z t}') \\
&\mid \texttt{false} \rightarrow (\texttt{free_occ_t y t}) * (\texttt{free_occ_t z t}') + \\
&\quad (\texttt{free_occ_t z t}) \\
&]
\end{aligned} \tag{A.5}$$

Il secondo congiunto della (A.5) non contiene riferimenti ad altre funzioni che analizzano `u`. Oltre la `free_occ` stessa, ci aspettiamo dunque che non ci sia bisogno di modificare ulteriormente il Σ-tipo di ritorno della `p_subst`. A questo punto sembrerebbe che le operazioni di raffinamento effettuate sinora sulla `p_subst` abbiano prodotto il risultato cercato: purtroppo, però non è così: come sappiamo, la A.5 non è altro che un'implementazione della A.2, la quale però, nonostante sembrasse corretta, soffriva di un'imprecisione dovuta alla sua definizione troppo poco computazionale. Per rendercene conto, prendiamo in esame l'esempio che segue:

Esempio A.2.2. Supponiamo di effettuare la sostituzione del termine $(\lambda y.\lambda z.w)$ $\{x \leftarrow ((\lambda x.x)z)\}$ con $w, k \in Var$. Il risultato che otterremo con la definizione di sostituzione che troviamo in A.6 sarà simile al seguente:

$$\begin{gathered}
(\lambda y.\lambda z.w)\{x \leftarrow ((\lambda x.x)z)\} \\
\lambda y.((\lambda z.w)\{x \leftarrow ((\lambda x.x)z)\}) \\
\lambda y.\lambda z'.(w\{x \leftarrow ((\lambda x.x)z)\}) \\
\lambda y.\lambda z'.w
\end{gathered}$$

Dove z' è una variabile fresca inserita al posto di z. Però, dal momento che la variabile x non occorre all'interno del termine $(\lambda y.\lambda z.w)$, rinominare la variabile z in z' è un'operazione che da un punto di vista computazionale è stata completamente inutile ed ha anzi α-convertito un termine senza che ce ne fosse bisogno.

Se dal punto di vista di una trattazione teorica del lambda calcolo il fenomeno evidenziato in A.2.2 non ha conseguenze negative, nel contesto di [5], dove le uguaglianze fra termini devono sempre essere interpretate in senso stretto, causa problemi. Ad esempio, come è già stato evidenziato in 6.3.1, per poter provare il lemma `stronger_aux_read_back3`, è necessario evitare α-conversioni superflue. Per far ciò è stato scelto di modificare la funzione di sostituzione nel seguente modo:

$$
\begin{aligned}
s\{x \leftarrow t\} &:= s \quad \text{se } x \notin FV(t) \\
x\{x \leftarrow t\} &:= t \\
x\{y \leftarrow t\} &:= x \quad \text{se } x \neq y \\
(t_1 t_2)\{x \leftarrow t\} &:= (t_1\{x \leftarrow t\})(t_2\{x \leftarrow t\}) \\
(\lambda x.s)\{x \leftarrow t\} &:= (\lambda x.s) \\
(\lambda z.s)\{x \leftarrow t\} &:= \lambda z.(s\{x \leftarrow t\}) \quad \text{se } z \notin FV(t) \\
(\lambda z.s)\{x \leftarrow t\} &:= \lambda k.(s\{z \leftarrow k\}\{x \leftarrow t\}) \quad \text{se } z \in FV(t)
\end{aligned}
\tag{A.6}
$$

A questo punto la specifica descritta in A.6 è quella adottata dall'implementazione della `p_subst` che è stata utilizzata per il lavoro di normalizzazione di [5], quindi la riportiamo qui sotto senza però trascrivere le costruzioni che è stato necessario introdurre: molte di queste sono poco significative in relazione al lavoro complessivo di formalizzazione. Preferiamo introdurre la formulazione e la dimostrazione di un semplice lemma sulla `p_subst` in modo da mostrare come l'utilizzo di Σ-tipi influenzi la costruzione di dimostrazioni.

```
1   let rec p_subst_sig (n: nat) y t': Πt. (t_size t ≤ n) →
2    Σu: pTerm. ((t_size u = (t_size t)+ (free_occ_t y t) * (t_size t' - 1) ∧
3     (∀z. free_occ_t z u = match veqb y z with
4      [ true ⇒ (free_occ_t z t) * (free_occ_t z t')
5      | false ⇒ (free_occ_t y t) * (free_occ_t z t') + (free_occ_t z t)
6      ]))) :=
7    match n return λn. Πt. (t_size t ≤ n) → Σu: pTerm. ((t_size u = (t_size t)+
         (free_occ_t y t) * ((t_size t') - 1)) ∧
8     (∀z. free_occ_t z u = match veqb y z with
9      [ true ⇒ (free_occ_t z t) * (free_occ_t z t')
10     | false ⇒ (free_occ_t y t) * (free_occ_t z t') + (free_occ_t z t)
11     ]))
12    with
13   [ O ⇒ λt.?
14   | S m ⇒ λt. match t return λt.  t_size t ≤ S m → Σu: pTerm. ((t_size u = (t_size
         t)+ (free_occ_t y t) * ((t_size t') - 1)) ∧
15      (∀z. free_occ_t z u = match veqb y z with
16      [ true ⇒ (free_occ_t z t) * (free_occ_t z t')
17      | false ⇒ (free_occ_t y t) * (free_occ_t z t') + (free_occ_t z t)
18      ]))
19     with
```

```
20      [ val_to_term v ⇒ match v return λv.  t_size (val_to_term v) ≤ S m → Σu:
            pTerm. (t_size u = (t_size (val_to_term v))+ (free_occ_v y v) * ((t_size
            t') - 1) ∧
21        (∀z. free_occ_t z u = match veqb y z with
22         [ true ⇒ (free_occ_v z v) * (free_occ_t z t')
23         | false ⇒ (free_occ_v y v) * (free_occ_t z t') + (free_occ_v z v)
24         ]))
25        with
26       [ pvar x ⇒  match veqb y x return λb.  veqb y x = b → t_size (val_to_term
             (pvar x)) ≤ S m → Σu: pTerm. (t_size u = (t_size (val_to_term (pvar
             x)))+ (free_occ_v (match (psubst y t') with [psubst x t ⇒ x]) (pvar x))
             * ((t_size (match (psubst y t') with [psubst x t ⇒ t])) - 1) ∧
27          (∀z. free_occ_t z u = match veqb (match (psubst y t') with [psubst y t' ⇒
                y]) z with
28            [ true ⇒ (free_occ_v z (pvar x)) * (free_occ_t z match (psubst y t')
                 with [psubst y t' ⇒ t'])
29            | false ⇒ (free_occ_v (match (psubst y t') with [psubst y t' ⇒ y])
                 (pvar x)) * (free_occ_t z match (psubst y t') with [psubst y t' ⇒
                 t']) + (free_occ_v z (pvar x))
30            ]))
31          with
32           [true ⇒λH.λp.mk_Sig … t' ? | false ⇒ λH.λp.mk_Sig … (val_to_term (pvar
                x)) ?] (refl …)
33       | abstr x t1 ⇒ match veqb y x return λb. veqb y x = b → t_size (val_to_term
             (abstr x t1)) ≤ S m → Σu: pTerm. (t_size u = (t_size (val_to_term
             (abstr x t1)))+ (free_occ_v y (abstr x t1)) * ((t_size t') - 1) ∧
34          (∀z. free_occ_t z u = match veqb y z with
35            [ true ⇒ (free_occ_v z (abstr x t1)) * (free_occ_t z t')
36            | false ⇒ (free_occ_v y (abstr x t1)) * (free_occ_t z t') + (free_occ_v
                 z (abstr x t1))
37            ]))
38         with
39          [ true ⇒ λH.λp. mk_Sig …  (val_to_term (abstr x t1)) ?
40          | false ⇒ λH. match fvb_t x t' return λb. fvb_t x t'=b → t_size
               (val_to_term (abstr x t1)) ≤ S m → Σu: pTerm. (t_size u = (t_size
               (val_to_term (abstr x t1)))+ (free_occ_v y (abstr x t1)) * ((t_size
               t') - 1) ∧
41            (∀z. free_occ_t z u = match veqb y z with
42              [ true ⇒ (free_occ_v z (abstr x t1)) * (free_occ_t z t')
43              | false ⇒ (free_occ_v (y) (abstr x t1)) * (free_occ_t z t') +
                   (free_occ_v z (abstr x t1))
44              ]))
45           with
46            [ true ⇒ λHH. match fvb_t y (val_to_term (abstr x t1)) return λHfvb.
                 fvb_t y (val_to_term (abstr x t1)) = Hfvb →  t_size (val_to_term
                 (abstr x t1)) ≤ S m → Σu: pTerm. ((t_size u = (t_size (val_to_term
                 (abstr x t1)))+ (free_occ_t y (val_to_term (abstr x t1))) *
                 ((t_size t') - 1)) ∧
47              (∀z. free_occ_t z u = match veqb y z with
48                 [ true ⇒ (free_occ_t z (val_to_term (abstr x t1))) * (free_occ_t z
                      t')
49                 | false ⇒ (free_occ_t y (val_to_term (abstr x t1))) * (free_occ_t
                      z t') + (free_occ_t z (val_to_term (abstr x t1)))
50                 ]))
51               with
52              [ true ⇒ λHfv.λp. let z := (max (S match y with [variable n ⇒ n])
                   (max (S match x with [variable nx⇒ nx]) (max (fresh_var_t t1)
                   (fresh_var_t t'))))
53                   in match (p_subst_sig m x (val_to_term (pvar ν(z))) t1 ?) with
54                 [ mk_Sig a h ⇒ mk_Sig …  (val_to_term (abstr (ν(z)) (pi1 …
                     (p_subst_sig m y t' a ?)))) ?]
55              | false ⇒ λHfv. λp. mk_Sig pTerm ? (val_to_term (abstr x t1)) ?
56              ] (refl …)
57            | false ⇒ λHH. λp. mk_Sig … (val_to_term (abstr x (pi1 … (p_subst_sig m
                 y t' t1 ?)))) ?
58            ] (refl …)
59          ] (refl …)
60       ]
```

```
61       | appl t2 u ⇒  λp. mk_Sig … (appl (pi1 …(p_subst_sig m y t' t2 ? )) (pi1 …
             (p_subst_sig m y t' u ?))) ?
62       ]
63   ]
64 .
```

Codice A.7: Implementazione della funzione `p_subst_sig`

Per quanto riguarda la A.8, osserviamo che, mentre nel prototipo A.5 il numero di prove da produrre era uguale al numero di chiamate ricorsive, con l'utilizzo dei Σ-tipi, il numero di prove da inserire nel corpo della `p_subst` è uguale al numero di chiamate ricorsive sommato al numero di diversi possibili valori di ritorno della funzione: per ognuno di essi, oltre a dover fornire un `u : pTerm` su cui è stata effettuata la sostituzione, bisogna anche dimostrare che la proprietà regge per quel termine.

In linea di principio, potrebbe sembrare che inserire tutte le prove richieste dalla `p_subst` all'interno del corpo della funzione stessa o definire dei lemmi ad hoc [1] ed inserire chiamate a questi lemmi per costruire le proposizioni necessarie siamo due approcci equivalenti. In realtà non è così: il secondo approccio rende il λ-termine che implementa la `p_subst` molto più semplice, dal momento che esso non contiene direttamente tutto il termine di prova, ma la sua rappresentazione più compatta per mezzo dell'invocazione del lemma con le rispettive ipotesi. Nonostante possa sembrare un'ottimizzazione di poco conto, in realtà non è così: se si operano delle politiche di riduzione mirate (i.e. non generalizzate), si possono evitare fastidiosi ritardi di elaborazione fra l'applicazione di ciascuno dei vari passi della dimostrazione formale, come evidenziato in A.4. Mentre se si optasse per l'altro approccio, molto spesso lo sforzo di elaborazione necessario ad un comune elaboratore commerciale sarebbe tale da rendere impraticabile la dimostrazione in tempi accettabili.

Inoltre, occorre notare che le dimostrazioni delle proprietà prodotte dalla `p_subst` sono state introdotte al solo scopo di renderla definibile in Matita. Ma dal momento che la funzione è stata definita, si potrebbe pensare di fare in modo che la funzione si comporti come se fosse stata definita senza l'utilizzo di sigma tipi. Per fare ciò occorre innanzitutto modificare il tipo della `p_subst` in modo che corrisponda al tipo `pTerm → Variable → pTerm → pTerm` che comunemente ci si aspetterebbe. Per farlo basta introdurre la seguente definizione:

```
1  definition p_subst := t.s.
2    pi1 … (p_subst_sig (t_size t) match s with [psubst y t'  y] match s with
          [psubst y t'  t'] t ?). // qed.
```

Codice A.8: Definizione della funzione `p_subst` attorno alla funzione `p_subst_sig`

Rimane da rimuovere il problema legato alla complessità dei termini della `p_subst`: se la definizione data in A.8 fa sì che la signature della funzione sia

[1] Ricordiamo che per Matita i lemmi sono funzioni che, date le opportune ipotesi, restituiscono la tesi.

`pTerm → pSubst → pTerm` (quindi sostanzialmente equivalente a quello visto sopra), tuttavia non risolve il problema della possibile esplosione della dimensione del termine in seguito a politiche di riduzione generalizzate. Per evitare questo ultimo problema è stato necessario introdurre alcuni di lemmi in grado di sintetizzare il comportamento della `p_subst` in presenza di determinate precondizioni (per esempio nel caso in cui la variabile da sostituire non apparisse libera nel sostituendo); in questo modo, anziché ridurre la `p_subst`, in presenza delle opportune precondizioni, sarà possibile applicare in un passo il lemma specifico. Si guardi il prossimo esempio:

```
lemma no_subst0: ∀n,x,y,t',H. veqb x y = false → pi1 pTerm ?
    (p_subst_sig n x t' (val_to_term (pvar y)) H) =
    (val_to_term (pvar y)).
@nat_ind
[ #x #y #t' #H @False_ind lapply H /2/ ]
#n #HI #y #x #t #H1 #H
whd in match (p_subst_sig (S n) y t (val_to_term (pvar x))
    (?));
cut (∀P1.∀P2. eq pTerm (pi1 pTerm ?
 (match veqb y x
 return λb. veqb y x = b → 1 ≤ S n →
    Σu: pTerm. ?
    with
     [ true ⇒ λH: veqb y x = true.
        λp: 1 ≤ S n.
         «t, P1 H p»
     | false ⇒ λH: veqb y x = false.
        λp: 1 ≤ S n.
         «val_to_term (pvar x), P2 H p»
     ] (refl bool (veqb y x )) (H1)))  (val_to_term (pvar x)))
[3: #UU @(UU …) | skip ] #P1 #P2 >H in P1 P2 ⊢ %; #P1' #P2'
    normalize // qed.
```

Codice A.9: Enunciato e dimostrazione del lemma `no_subst0`

Già osservando la lunghezza della dimostrazione del lemma A.9, si può notare che la sintesi di quello che dovrebbe essere un banale passo di semplificazione ha richiesto un certo impegno. Per sommi capi la dimostrazione procede nel seguente modo: si applica il principio di induzione su `n`, nel caso `n = 0` si raggiunge un assurdo poiché nessun termine può avere taglia nulla, come evidenziato dall'applicazione del principio di eliminazione del falso `False_ind`. Per il caso `n = S m` è possibile usare il tactic `whd` per operare una semplificazione mirata del goal in modo da riscrivere il corpo della `p_subst_sig` in *weak-head form*; in questo caso, siccome l'argomento in testa è `S m`, la semplificazione consiste con l'esecuzione del primo passo induttivo. In casi come questo, infatti, non vogliamo adottare tecniche di semplificazione troppo agressive poiché, altrimenti, a causa delle prove contenute nei temini, le dimensioni del goal potrebbero crescere a tal punto da rendere il goal stesso poco interpretabile ed i tempi di

elaborazione dei tactic troppo elevati per rendere possibile la dimostrazione in tempi accettabili.

A questo punto, se la funzione non contenesse le prove della proprietà `P` per i due possibili valori di ritorno `t'` e `val_to_term (pvar x)`, sarebbe possibile, mediante il tactic `>H`, sostituire `veqb x y` con il lato destro dell'uguaglianza nell'ipotesi `H : veqb x y = false`, semplificare e chiudere il goal. Tuttavia, la presenza delle due prove fa sì che se provassimo a sostiture in questo modo, creeremmo un mismatch di tipo: infatti, le due prove hanno tipo dipendente rispettivamente dalle ipotesi `H: veqb x y = true` e `H: veqb x y = false`. Se sostituissimo, il primo `H` acquisitebbe tipo `H: false = true` ed il secondo `H: false = false`, mentre i teoremi usati per costruire le prove, se non espansi, continuerebbero ad avere il loro tipo di partenza. Una delle possibili soluzioni a questo problema prevede che, mediante il tactic `cut`, siano generalizzate le due prove della proprietà `P` con i termini `P1` e `P2` in modo da sostituire i teoremi con due prove locali delle quali è possibile manipolare il tipo. Dopo aver effettuato questa operazione, risulta possibile riscrivere `veqb x y` anche all'interno dei tipi delle prove appena introdotte così da evitare il mismatch di tipo di cui sopra. Infine, per concludere il goal basta normalizzare e Matitia è in grado di chiudere automaticamente la prova.

Avendo dimostrato il lemma A.9, è possibile usarlo per dimostrare il seguente corollario:

```
lemma no_subst: ∀y,x,t'.
 veqb x y = false →
  (p_subst (val_to_term (pvar y)) (psubst x t') ) =
      (val_to_term (pvar y)).
#y #x #t' @no_subst0 qed.
```

Codice A.10: Enunciato e dimostrazione del lemma `no_subst`

Ora, ogniqualvolta si debba studiare il comportamento della funzione in situazioni simili a quelle descritte dalle ipotesi del lemma A.10, sarà possibile applicarlo direttamente senza dover maneggiare il corpo della `p_subst_sig` e le dimostrazioni in esso contenute, alleggerendo il carico di lavoro del dimostratore assistito e rendendo più semplici le dimostrazioni di lemmi più complessi.

Allo stesso modo del lemma A.10 sono stati dimostrati altri risultati, fra cui riportiamo gli enunciati dei seguenti risultati notevoli:

```
lemma p_subst_bound_irrelevance0: ∀n.
 (∀t, n', y, t', H, H'.
   p_subst_sig n y t' t H = p_subst_sig n' y t' t H').

lemma p_subst_distro0:
 ∀n1, n2, t1, t2, y, t', H, H1, H2.
   pi1 … (p_subst_sig (S (n1 + n2)) y t' (appl t1 t2) H) =
    appl (pi1 … (p_subst_sig n1 y t' t1 H1)) (pi1 …
        (p_subst_sig n2 y t' t2 H2)).

lemma atomic_subst: ∀x, t. (p_subst (val_to_term (pvar x))
    (psubst x t)) = t.
```

```
lemma abstr_step_subst: ∀x, y, t1, t'.
 veqb y x =false →
  fvb_t x t' = false →
   p_subst (val_to_term (abstr x t1)) (psubst y t') =
    (val_to_term (abstr x (p_subst t1 (psubst y t')))).

lemma no_subst5:
 (∀t,y,t'. fvb_t (νy) t =false → p_subst t (psubst (νy) t')=t).

lemma p_subst_distro:
 ∀t1, t2, s. p_subst (appl t1 t2) s = appl (p_subst t1 s)
     (p_subst t2 s).
```

Codice A.11: Lemmi notevoli riguardanti la `p_subst`

Per la dimostrazione dell'ultimo lemma elencato in A.11 è stato necessario introdurre il lemma ausiliario lemma ausiliario A.12.

```
lemma p_subst_bound_irrelevance0: ∀n.
 (∀t, n', y, t', H, H'.
   p_subst_sig n y t' t H = p_subst_sig n' y t' t H').
```

Codice A.12: Lemma di bound irrelevance per la `p_subst_sig`

Il lemma A.12 è necessario poiché, per come è definita la funzione `p_subst` (i.e. la `p_subst_sig`), la chiamata induttiva sui sottotermini di un'applicazione, che in A.8 possiamo trovare alla riga 61, viene effettuata su `m`, il predecessore del parametro formale, mentre le chiamate alla `p_subst_sig` dell'espansione di `appl (p_subst t1 s) (p_subst t2 s)` vengono effettuate con parametro `t_size t1` e `t_size t2`. Un fenomeno di questo tipo è quello mostrato dal seguente esempio:

Esempio A.2.3. Siano definiti `t1`, `t2`, `s` tali che `t_size t1 = 5`, `t_size t2 = 7` e `s = psubst y t'`, se istanziassimo la tesi del teorema `p_subst_distro` su questi dati e la espandessimo otterremmo qualcosa di simile a:

```
1  pi1 ... appl (pi1 ... (p_subst_sig 11 y t' t1 P)) (pi1 ... (p_subst_sig 11 y t'
   ↪  t2 Q)) =
2  appl (pi1 ... (p_subst_sig 5 y t' t1 R) (p_subst_sig 7 y t' t1 S))
```

Dove `P`, `Q`, `R`, `S` sono termini che dimostrano le condizioni sulle taglie di `t1` e `t2`.

Il ruolo del lemma in A.12 è quello di poter asserire che il bound sulla taglia dei termini è irrilevante ai fini della sostituzione (a patto che sia possibile dimostrare che effettivamente sia un bound superiore alla taglia del termine). Una volta applicato, allora la conclusione è banale.

Abbiamo visto come la sintetizzazione di un passo di riduzione della `p_subst` abbia richiesto uno sforzo non banale a causa della presenza di tipi dipendenti dovuta alla costruzione di prove all'interno della funzione. La dimostrazione del lemma A.12 ha richiesto di agire similmente ogniqualvolta nella definizione della `p_subst_sig` vi fosse un branching di scelta e non di esplorazione del tipo.

Infatti, dal momento che generalmente le proprietà che determinano il branching sono necessarie per costruire le prove di A.5, allora in tali casi avremo di certo casi di tipi dipendenti da risolvere, come nel caso del lemma A.9, mediante, ad esempio, generalizzazione ed introducendo un significativo overhead nella dimostrazione di enunciati che intuitivamente sembrano triviali.

A.3 Principi di induzione

Nel contesto di una dimostrazione informale, uno dei metodi più semplici per produrre dimostrazioni di proprietà riguardanti tipi di dato algebrici è quello di ricorrere all'uso dell'induzione strutturale sul tipo di dato. Molti dimostratori assistiti implementano meccanismi per rendere questa operazione semplice e trasparente. In Matita, questo comportamento è reso possibile mediante il tactic `elim x` ove `x` è il termine su cui procedere per induzione. Questa operazione, altro non è che zucchero sintattico per l'espressione @(`type_ind` ... `x`), ove `type` è il tipo di `x`.

Per rendere questa operazione possibile, dunque, è necessario che sia definito un opportuno teorema (i.e. principio, funzione) d'induzione da applicare. In Matita, la generazione dei principi di induzione può essere automatica o manuale a seconda del tipo di dato che viene definito: nel caso di tipi di dato enumerativi o ricorsivi il sistema è in grado di generare autonomamente il principio di induzione, mentre per il caso di tipi di dato mutualmente ricorsivi questa funzionalità non è implementata. Nel caso del nostro lavoro, come evidenziato nel capitolo 4, i tipi di dato più importanti, sia i pTerm che i Crumble, sono mutualmente ricorsivi. Per questo motivo è stato dunque necessario intraprendere la scrittura di un numero adeguato di principi di induzione grazie ai quali rendere possibile la dimostrazione di lemmi per induzione strutturale.

A questo punto ci sembra opportuno presentare al lettore l'implementazione del principio di induzione per antonomasia: quello sui numeri naturali, poiché riteniamo che in questo modo questi possa essere facilitato nella comprensione, per analogia, delle definizioni che daremo dei principi d'induzione sui tipi di dato pTerm e Crumble.

Il principio di induzione sui naturali a seconda del formalismo in cui è definito o derivato recita più o meno quanto segue:

$$\forall P.P(0) \to (\forall n.P(n) \to P(S\ n)) \to \forall n.P(n) \tag{A.7}$$

Dunque, per implementarlo in λ-calcolo, è sufficiente produrre un termine `nat_ind` che abbia tipo:

$$\forall(\mathtt{P} : \mathtt{nat} \to \mathtt{Prop}).\mathtt{P}\ \mathtt{0} \to (\forall(\mathtt{n} : \mathtt{nat}).\mathtt{P}\ \mathtt{n} \to \mathtt{P}\ (\mathtt{S}\ \mathtt{n})) \to \forall \mathtt{n}.\mathtt{P}\ \mathtt{n}$$

Ma in virtù dell'isomorfismo di Curry-Howard-Kolmogorov, possiamo guardare il principio di induzione sui numeri naturali come una funzione che presi in input una qualsiasi proposizione `P`, una dimostrazione di $P(0)$ (dunque un termine del tipo `P 0`) ed una prova di $(\forall n.P(n) \to P(S\ n))$ (dunque un termine del tipo

$(\forall(\texttt{n}:\texttt{nat}).\texttt{P n} \rightarrow \texttt{P (S n)}))$ è in grado di produrre una dimostrazione di $\forall n.P(n)$, dunque un termine del tipo $\forall\texttt{n}.\texttt{P n}$.

Alla luce di ciò si osserva che una funzione di questo tipo è implementabile senza troppe difficoltà se si procede come segue:

```
let rec n_ind (P : nat → Prop) (H1: P 0) (H2: ∀m. P m → P (S
    m)) n on n: P n :=
 match n with
 [ O ⇒ H1
 | S m ⇒ (H2 m (n_ind P H1 H2 m))
 ] .
```

Codice A.13: Definizione del principio di induzione sul tipo nat.

Se il type-checking del termine non è sufficiente per convincersi che il dato appena definito sia veramente il principio di induzione, il lettore può anche pensare che applicando questo lemma *(o funzione)* durante la prova di un teorema, il sistema di inferenza di Matita sarà in grado di dedurre il tipo di `P` e quindi di ridurre il goal da dimostrare ai due subgoals `P 0` e $(\forall(\texttt{n}:\texttt{nat}).\texttt{P n} \rightarrow \texttt{P (S n)}))$. Ciò che abbiamo realizzato, dunque, è veramente il principio di induzione sui numeri naturali.

Questo procedimento abbastanza suggestivo, se generalizzato al caso di funzioni mutualmente ricorsive, ci permette di definire principi di induzione per tipi mutui. Nel nostro caso, era necessario definire il principio di induzione su pTerm e quello su Crumble; da un punto di vista teorico le difficoltà inerenti la definizione di un principio piuttosto che l'altro sono le medesime per cui, per ragioni di economia di spazio, tratteremo solo il caso del principio di induzione su pTerm.

In riferimento all'esempio precedente, osserviamo che, da un punto di vista computazionale, un principio di induzione per il tipo `t` è un caso particolare di funzione ricorsiva che ha fra i suoi argomenti un termine `P : t` $\rightarrow$ `Prop` ed un termine `x : t`, mentre come tipo di ritorno ha `P x`. Da questo punto di vista, le ipotesi induttive sono possibili chiamate ricorsive della funzione (nel caso di A.13, `(n_ind P H1 H2 m)`) sui sottotermini immediati del costruttore di tipo.

Si deduce quindi che per un medesimo tipo, possono essere definibili principi di induzione multipli, ognuno dei quali fa uso o meno di determinate ipotesi induttive e, per questo motivo, può essere più o meno adatto alla dimostrazione di un certo tipo di enunciati. Inoltre, si intuisce come possa esistere una sorta di "principio di induzione generale" in grado di massimizzare il numero di ipotesi induttive fornite e quindi la potenza espressiva del principio stesso.

Se cerchiamo di applicare questo ragionamento ai tipi mutui pTerm e pValue otteniamo il seguente risultato:

```
let rec pTerm_ind (P: pTerm → Prop) (Q: pValue → Prop)
(H1: ∀v. Q v → P (val_to_term v))
(H2: ∀t1,t2. P t1 → P t2 → P (appl t1 t2))
(H3: ∀x. Q (pvar x))
(H4: ∀t, x. P t → Q (abstr x t))
(t: pTerm) on t: P t :=
```

```
match t return λt. P t with
 [ val_to_term v ⇒ H1 v (pValue_ind P Q H1 H2 H3 H4 v)
 | appl t1 t2 ⇒ H2 t1 t2 (pTerm_ind P Q H1 H2 H3 H4 t1)
     (pTerm_ind P Q H1 H2 H3 H4 t2)
 ]

and pValue_ind (P: pTerm → Prop) (Q: pValue → Prop)
(H1: ∀v. Q v → P (val_to_term v))
(H2: ∀t1,t2. P t1 → P t2 → P (appl t1 t2))
(H3: ∀x. Q (pvar x))
(H4: ∀t, x. P t → Q (abstr x t))
(v: pValue) on v: Q v :=
match v return λv. Q v with
 [ pvar x ⇒ H3 x
 | abstr x t ⇒ H4 t x (pTerm_ind P Q H1 H2 H3 H4 t)
 ]
 .
```

Codice A.14: Definizione del principio di induzione sui tipi mutui pTerm e pValue.

Dalla definizione in A.14 si osserva che la necessità di definire un principio di induzione generico per provare proposizioni nella forma `∀t. P t` ha indotto la necessità di inserire una proposizione `Q` corrispondente alla `P`, ma riferita ai pValue poiché questo, per esempio, era l'unico modo di poter sfruttare l'ipotesi induttiva sul sottotermine `v` nel ramo `val_to_term v` del costrutto `match t with`. Questo comportamento, non è anomalo, anzi è ciò ci si aspetterebbe quando si debbono formulare e dimostrare degli riguardanti funzioni mutualmente ricorsive tu tipi mutui: per ogni tipo mutualmente ricorsivo (e per ogni funzione mutua) è necessario avere (e dimostrare) degli invarianti. Uno di questi casi è per esempio quello che abbiamo visto per la dimostrazione del lemma 6.3.1 in cui la proprietà $\forall t.\underline{t}_{\downarrow} = t$ veniva formulata e dimostrata congiuntamente alla proprietà $\forall v.\overline{v}_{\downarrow} = v$ poiché, data la mutualità delle funzioni $\underline{\cdot}$ e $\overline{\cdot}$ nessuna delle due tesi era dimostrabile senza l'altra.

Molto spesso gli enunciati dei lemmi dimostrati utilizzando principi di induzione mutua sulla n-upla di tipi $\mathtt{T_1} \ldots \mathtt{T_n}$ sono formulati come una congiunzione di `n` proposizioni della forma $\forall\ \mathtt{x} : \mathtt{T_i}.\mathtt{P_i}$. Questa regolarità ci permette di semplificare l'applicazione dei principi di induzione mutua automatizzando il processo di inferenza delle proposizioni. Per esempio, se per dimostrare un goal `P` applicassimo direttamente il principio `pTerm_ind`, dovremmo specificare manualmente la proposizione `Q` necessaria per invocare il principio di induzione mutuo su pValue, ma dal momento che molto spesso gli enunciati di questi lemmi sono formati dalla congiunzione della `P` e della `Q`, possiamo fare in modo che il sistema di inferenza di tipi sia in grado di determinare autonomamente. Infatti, se definiamo il termine `pValueTerm_ind` come in A.15, osserviamo che applicandolo su di un goal della forma $\mathtt{P} \wedge \mathtt{Q}$, dove $\mathtt{P} : \mathtt{pTerm} \rightarrow \mathtt{Prop}$ e $\mathtt{Q} : \mathtt{pValue} \rightarrow \mathtt{Prop}$, tramite il principio di introduzione della congiunzione lo-

gica `conj`, vengono generati due subgoal indipendenti di tipo `P` e `Q`, sui quali vengono applicati rispettivamente `pTerm_ind` e `pValue_ind`.

```
lemma pValueTerm_ind: ∀P,Q,H1,H2,H3,H4.
 (∀t. P t) ∧ (∀v. Q v) :=
  λP,Q,H1,H2,H3,H4. conj … (pTerm_ind P Q H1 H2 H3 H4)
      (pValue_ind P Q H1 H2 H3 H4).
```

Codice A.15: Definizione del principio di induzione sui tipi mutui pTerm e pValue.

Dai tipi di `pTerm_ind` e `pValue_ind` il sistema è in grado di inferire `P` e `Q`. In questo modo, all'utente rimane solo da costruire i termini `H1, H2, H3, H4`, che equivale a fornire rispettivamente le dimostrazioni delle proposizioni riportate in A.8, mediante l'introduzione di altrettanti subgoal.

$$\begin{gathered}\forall \mathtt{v}.\ \mathtt{Q}\ \mathtt{v}\ \rightarrow \mathtt{P}\ (\mathtt{val_to_term}\ \mathtt{v})\\ \forall \mathtt{t1},\ \mathtt{t2}.\ \mathtt{P}\ \mathtt{t1} \rightarrow \mathtt{P}\ \mathtt{t2} \rightarrow \mathtt{P}\ (\mathtt{appl}\ \mathtt{t1}\ \mathtt{t2})\\ \forall \mathtt{x}.\ \mathtt{Q}\ (\mathtt{pvar}\ \mathtt{x})\\ \forall \mathtt{t},\ \mathtt{x}.\ \mathtt{P}\ \mathtt{t} \rightarrow \mathtt{Q}\ (\mathtt{abstr}\ \mathtt{x}\ \mathtt{t})\end{gathered} \tag{A.8}$$

Dunque, la definizione A.15 ci ha permesso di ridurre la dimostrazione formale di un goal $\mathtt{P} \wedge \mathtt{Q}$ nella dimostrazione dei delle proposizioni in A.8, come sarebbe avvenuto nel caso di una dimostrazione informale.

A volte, parte della potenza espressiva di un principio di induzione generico non è necessaria. Ciò può avvenire quando le proposizioni da dimostrare riguardano funzioni che non ricorrono su tutti i tipi mutui, o che per i costruttori di tipo che coinvolgono tipi mutui sono triviali. Uno di questi casi è stato per esempio quello della patch per il lemma 5.5.4 (7.10), nel quale al fine di dimostrare l'enunciato non erano necessarie proposizioni su pValue poiché, per come era definita la funzione `overline` nel caso `val_to_term`, la tesi era scontata. In situazioni di questo tipo si può agire in due modi: per esempio, nel caso specifico del lemma enunciato in (7.10), è stato possibile contenere il potere espressivo di `pTerm_ind` definendo una `Q` banale, nel caso specifico: `pValue` → `True`. Un'altra soluzione, sarebbe stata per esempio quella di definire principi di induzione semplificati che non facciano ricorso a principi di induzione su altri tipi mutui.

Una scelta di questo tipo è stata operata, per esempio, nel caso degli Environment: dal momento che sia le funzioni della famiglia della `alpha` e la funzione di `read_back`, sono definite prevalentemente per ricorsione sugli Environment[2], è stato utile definire numerosi lemmi sulle proprietà derivanti operazioni su Environment, per dimostrarli, se avessimo scelto di usare il principio di induzione

[2]Nel caso delle funzioni della famiglia `alpha` tutte le funzioni sono definite esclusivamente sull'Environment, mentre nel caso della `read_back` il ruolo della ricorsione sugli Environment è preminente per il ruolo della sostituzione su termini, mentre le altre funzioni mutualmente ricorsive, in ultima analisi, servono solo per cercare Environment su cui andare in ricorsione.

generico su Crumble, avremmo dovuto dare formulazioni pesanti e poco maneggevoli di lemmi relativamente semplici. Per esempio, prendiamo il seguente lemma:

```
lemma push_len: ∀e, s. e_len (push e s) = S (e_len e).
```

Tale lemma, che è stato veramente necessario definire e dimostrare ai fine della formalizzazione, se dimostrato mediante il principio `Environment_ind` avrebbe dovuto essere dimostrato come in A.16.

```
lemma push_len: ∀e, s. e_len (push e s) = S (e_len e).
@(Environment_ind (λx. True) (λx. True) ? (λx. True) (λx.
    True)) //
#e #s #H #_ #s1 normalize >H // qed.
```

Codice A.16: Enuciato e dimostrazione del lemma `push_len`

```
1   (Environment_ind
2   :∀P,Q,R,S,U.
3    .(∀b:Bite.∀e:Environment.Q b→R e→P ⟨b,e⟩)
4     →(∀v:Value.S v→Q (CValue v))
5      →(∀v:Value.∀w:Value.S v→S w→Q (AppValue v w))
6       →(∀x:Variable.S (var x))
7        →(∀x:Variable.∀c:Crumble.P c→S (λx.c))
8         →R Epsilon
9          →(∀e:Environment.∀s:Substitution.R e→U s→R (Snoc e s))
10          →(∀x:Variable.∀b:Bite.Q b→U [x←b])→∀e:Environment.R e)
```

Codice A.17: Tipo di `Environment_ind`

Come si osserva da A.17, il tipo di `Environment_ind` richiede che vengano specificati anche tutti i termini di tipo `P`, `Q`, `U`, `S`, ma che non sono necessari per la dimostrazione del lemma: l'unico apporto che danno in A.16 è un'ipotesi superflua `U s : True` che scartiamo col tactic `#_`. Questo comportamento, se confrontato con quello atteso nel contesto di una dimostrazione informale è piuttosto inaspettato: in linea teorica, l'applicazione del principio di induzione su di un ambiente dovrebbe essere più snello e non dovrebbe essere necessario fornire delle proposizioni triviali per i tipi mutui al tipo Environment, per poi scartarle. Per questo motivo è stato introdotto il principio di eliminazione in A.18, il quale, anziché dare lemmi triviali da provare ai principi di induzione mutui, semplicemente non è mutuo.

```
let rec Environment_simple_ind2 (P: Environment → Prop)
(H1: P Epsilon)
(H2: ∀e.∀s. P e → P (Snoc e s))
e on e :=
 match e return λe. P e with
 [ Epsilon ⇒ H1
 | Snoc e s ⇒ H2 e s (Environment_simple_ind2 P H1 H2 e)
 ].
```

Codice A.18: Principio di induzione su Environment

Infatti, se si confrontano A.17 e A.18, si può vedere come il caso base `Epsilon` a riga 8 in A.17 sia stato riportato identico nel tipo di `H1` modulo ridenominazione della `P` in `R`, mentre il caso induttivo a riga 9 di A.17 sia stato semplificato mediante la rimozione dell'ipotesi induttiva `U s`. Così facendo è stata spezzata la catena di dipendenze fra tipi che avrebbe reso necessaria l'introduzione delle proposizioni `P`, `Q`, ed `S`. In questo modo la dimostrazione del lemma `push_len` risulta più semplice e leggibile:

```
lemma push_len: ∀e, s. e_len (push e s) = S (e_len e).
@Environment_simple_ind2
[ //
|#e #s #H #s1 normalize >H //
] qed.
```

Codice A.19: Enuciato e dimostrazione del lemma `push_len`

A.4 Tecniche di riduzione in compendio

In molte parti del lavoro si è parlato della necessità di effettuare riduzioni mirate dei termini per evitare fenomeni di esplosione della taglia. Come abbiamo visto, infatti, tali fenomeni comporterebbero un rallentamento del dimostratore interattivo dovuto all'aumento dei tempi di applicazione dei tactic che, in alcuni casi, sarebbero tali da rendere impraticabile la dimostrazione in tempi accettabili. Per evitare questi fenomeni è necessario saper usare opportunamente i tactic di riduzione messi a disposizione da Matita.

Ogni tactic di riduzione, se non specificato altrimenti, viene applicato al goal corrente, mentre se seguito dalla *pattern* riducono solo in quei sottotermini del goal corrente (o delle ipotesi), che fanno match col *pattern*. Per i nostri fini non analizzeremo la sintassi e la semantica dei pattern, tuttavia una trattazione più accurata di tale argomento, rimandiamo il lettore alla consultazione di [6].

`normalize`

Il tactic `normalize` effettua la normalizzazione del termine su cui è chiamato; per far ciò tutte le definizioni (i.e. di lemmi e funzioni) vengono riscritti con il loro corpo che viene successivamente ridotto alla forma normale. Le condizioni di accettazione delle definizioni (per esempio come quelle previste per le funzioni ricorsive di cui abbiamo discusso in A.2) garantiscono che la riduzione ad un certo punto termini. Come abbiamo anticipato in A.2, l'uso di questo tactic è particolarmente aggressivo sul termine e può causare l'esplosione della sua taglia specialmente qualora nel corpo del *sotto*-termine ridotto siano presenti prove, poiché di solito le dimensioni di questi λ-termini sono spesso elevate. Quindi, in questi contesti, che nel lavoro di formalizzazione che abbiamo descritto erano la maggior parte dei casi, risulta più indicato l'uso dei tactic `whd` e `change with`.

Esempio A.4.1. Supponiamo di avere il goal `(3+5)+1 = 9`. L'applicazione del tactic `normalize` ridurrebbe il goal a `9=9`, mentre per esempio l'applicazione del tactic `normalize in match (3+5);` ridurrebbe il goal in `8+1 = 9`.

`whd`

Il tactic `whd` semplifica un termine riducendolo alla sua weak-head normal form, ossia in modo che il suo costruttore più esterno (la sua testa) non sia ulteriormente semplificabile.

Esempio A.4.2. Supponiamo di avere il goal `(3+5)+1 = 9`. L'applicazione del tactic `whd in match (3+5)` semplifica la somma in `S (2 + 5)` e, dal momento che il costruttore più esterno è `S`, l'espressione ha raggiunto la sua weak-head normal form ed il goal viene riscritto come `S (2+5)+1 = 9`. Similmente l'applicazione del tactic `normalize in match ((3+5)+1);` ridurrebbe il goal in `S (2+5)+1 = 9`, ma dal momento che la testa del sottotermine su cui è chiamato (in questo caso `+`) è ancora semplificabile, esso effettua un altro passo di riduzione del sottotermine in `S (2+5+1)`. A questo punto il termine ha raggiunto la sua weak-head normal form, per cui il goal viene riscritto in `S (2+5+1) = 9`.

`change with`

Il tactic `change with`, invece, è fortemente più espressivo dei precedenti: esso sostituisce il sottotermine `t` su cui è chiamato con qualsiasi altro termine `t'` che sia convertibile alla forma normale di `t`. Dal momento che ogni termine è convertibile alla propria forma normale o alla propria weak-head normal form, il tactic `change with` è espressivo almeno quanto i tactic `normalize` e `whd`. Inoltre può essere usato "a rovescio" per rifattorizzare alcune parti di un termine che sono state semplificate troppo per esempio da una `normalize`.

Esempio A.4.3. Supponiamo di avere il goal `(3+5)+1 = 9`. L'applicazione del tactic `change with (7+1) in match (3+5);` cerca nel goal tutti i sotto termini che si normalizzano come `(3+5)`, in questo caso `(3+5)` e `8`, sottotermine di `9`[3], e li riscrive come `7+1`, per cui il goal diventa `(7+1)+1 = S (7+1)`.

[3] Ricordiamo che i naturali sono espressi con il tipo induttivo $\mathbb{N} \coloneqq O : \mathbb{N} \mid S : \mathbb{N} \to \mathbb{N}$, per cui $9 \coloneqq S\ 8$.

Bibliografia

[1] Beniamino Accattoli. “The Complexity of Abstract Machines”. In: *Electronic Proceedings in Theoretical Computer Science* 235 (gen. 2017), pp. 1–15. ISSN: 2075-2180. DOI: 10.4204/eptcs.235.1. URL: http://dx.doi.org/10.4204/EPTCS.235.1.

[2] Beniamino Accattoli e Bruno Barras. “Environments and the Complexity of Abstract Machines”. In: *Proceedings of the 19th International Symposium on Principles and Practice of Declarative Programming*. PPDP ’17. Namur, Belgium: Association for Computing Machinery, 2017, pp. 4–16. ISBN: 9781450352918. DOI: 10.1145/3131851.3131855. URL: https://doi.org/10.1145/3131851.3131855.

[3] Beniamino Accattoli e Giulio Guerrieri. “Implementing Open Call-by-Value”. In: *7th International Conference on Fundamentals of Software Engineering (FSEN)*. A cura di Mehdi Dastani e Marjan Sirjani. Vol. LNCS-10522. Fundamentals of Software Engineering. Teheran, Iran: Springer International Publishing, apr. 2017, pp. 1–19. DOI: 10.1007/978-3-319-68972-2_1. URL: https://hal.archives-ouvertes.fr/hal-01675365.

[4] Beniamino Accattoli e Giulio Guerrieri. “Open Call-by-Value (Extended Version)”. In: *CoRR* abs/1609.00322 (2016). arXiv: 1609.00322. URL: http://arxiv.org/abs/1609.00322.

[5] Beniamino Accattoli et al. “Crumbling Abstract Machines”. In: *CoRR* abs/1907.06057 (2019). arXiv: 1907.06057. URL: http://arxiv.org/abs/1907.06057.

[6] Andrea Asperti, Wilmer Ricciotti e Claudio Sacerdoti Coen. “Matita Tutorial”. In: *Journal of Formalized Reasoning* 7.2 (dic. 2014), pp. 91–199. DOI: 10.6092/issn.1972-5787/4651. URL: https://jfr.unibo.it/article/view/4651.

[7] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*. Vol. 103. Studies in logic and the foundations of mathematics. North-Holland, 1985. ISBN: 978-0-444-86748-3.

[8] Hendrik Pieter Barendregt, Wil Dekkers e Richard Statman. *Lambda Calculus with Types*. Perspectives in logic. Cambridge University Press, 2013. ISBN: 978-0-521-76614-2. URL: http://www.cambridge.org/de/academic/subjects/mathematics/logic-categories-and-sets/lambda-calculus-types.

[9] Henk (Hendrik) Barendregt e E. Barendsen. "Introduction to lambda calculus". In: *Nieuw archief voor wisenkunde* 4 (gen. 1984), pp. 337–372.

[10] J. Barkley Rosser. "Haskell B. Curry and Robert Feys. Combinatory logic. Volume I. With two sections by William Craig. Studies in logic and the foundations of mathematics. North-Holland Publishing Company, Amsterdam1958, xvi 417 pp." In: *Journal of Symbolic Logic* 32.2 (1967), pp. 267–268. DOI: 10.1017/S0022481200114203.

[11] Yves Bertot e Pierre Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Snoctructions*. 1st. Springer Publishing Company, Incorporated, 2010. ISBN: 3642058809.

[12] Luca Cardelli. "Type Systems". In: *ACM Comput. Surv.* 28.1 (mar. 1996), pp. 263–264. ISSN: 0360-0300. DOI: 10.1145/234313.234418. URL: https://doi.org/10.1145/234313.234418.

[13] Oliver Danvy e Andrzex Filinski. "Representing Control: a Study of the CPS Transformation". In: *Mathematical Structures in Computer Science* 2.4 (1992), pp. 361–391. DOI: 10.1017/S0960129500001535.

[14] Davide Davoli e Claudio Sacerdoti Coen. *davidedavoli/Crumbling-Abstract-Machines v0.1.1*. Ver. v0.1.1. Dic. 2020. DOI: 10.5281/zenodo.4300328. URL: https://doi.org/10.5281/zenodo.4300328.

[15] Cormac Flanagan et al. "The Essence of Compiling with Continuations". In: *SIGPLAN Not.* 39.4 (apr. 2004), pp. 502–514. ISSN: 0362-1340. DOI: 10.1145/989393.989443. URL: https://doi.org/10.1145/989393.989443.

[16] Jan Martin Jansen. "Programming in the -Calculus: From Church to Scott and Back". In: (gen. 2013). DOI: 10.1007/978-3-642-40355-2_12.

[17] Andrew Kennedy. "Compiling with Continuations, Continued". In: *SIGPLAN Not.* 42.9 (ott. 2007), pp. 177–190. ISSN: 0362-1340. DOI: 10.1145/1291220.1291179. URL: https://doi.org/10.1145/1291220.1291179.

[18] G.D. Plotkin. "Call-by-name, call-by-value and the -calculus". In: *Theoretical Computer Science* 1.2 (1975), pp. 125–159. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(75)90017-1. URL: http://www.sciencedirect.com/science/article/pii/0304397575900171.