Introduzione

Questa tesi si occupa del problema della fully abstraction per il linguaggio PCF. È noto che esiste un modello categoriale fully abstract, definito da Abramsky, Jagadeesan e Malacaria in [1] e da Hyland e Ong in [2]. Quello che si vuole definire è un modello "numerico" secondo le idee di Kleene e Gandy su come dovrebbero essere definiti i modelli.

Nel capitolo 1 è descritto il linguaggio PCF, mentre nel capitolo 2 è presentato il problema di trovare un modello fully abstract per PCF. Nel capitolo 3 questo problema sarà risolto per i tipi 0, 1 e 2; e inoltre, si farà notare, con dei controesempi, la difficoltà nel generalizzare al tipo 3 l'approccio seguito. Il capitolo 4 conclude la tesi illustrando brevemente le caratteristiche dei linguaggi funzionali, in particolare dei linguaggi ML e Haskell.

Ringraziamenti

Ringrazio il prof. Pani per avermi seguito durante lo svolgimento di questa tesi.

Ringrazio la mia famiglia per avermi dato la possibilità di studiare e per non avermi mai fatto mancare niente di dui avevo bisogno.

Ringrazio l'amico Marco Magno per le sue innumerevoli idee che mi hanno profondamente colpito e ispirato in varie occasioni, anche per la scrittura di questa tesi, e per i consigli e gli incoraggiamenti che da lui ho ricevuto nel corso di questi cinque anni di studi universitari.

Indice

Capitolo 1

Il linguaggio PCF

Il sistema formale PCF (Programs for Computable Functions) è stato definito da Scott [3] come un "calcolo logico" per studiare la computabilità e altre proprietà logiche dei programmi usando la teoria dei tipi. La sintassi di PCF è abbastanza semplice: è essenzialmente il lambda calcolo tipato con l'aggiunta della ricorsione generale nella forma di un operatore di punto fisso per ogni tipo e dell'aritmetica di base.

In [4], Plotkin ha presentato il PCF esplicitamente come un linguaggio di programmazione e ha studiato la relazione tra la sua semantica operazione e quella denotazionale, basata sul modello dello spazio delle funzioni continue di Scott. Un linguaggio di programmazione è un sistema di riduzioni con alcune carat-

teristiche aggiuntive, come una specifica strategia di riduzione. Un sistema di riduzioni è semplicemente un linguaggio formale con delle regole di riscrittura. Il linguaggio PCF si ottiene dal sistema di riduzioni $\lambda^{\rightarrow}_{\mathrm{Y}}(\mathbf{A})$ definendo una specifica strategia di riduzione. Il sistema di riduzioni $\lambda^{\rightarrow}_{\mathrm{Y}}(\mathbf{A})$ è il lambda calcolo tipato con gli operatori di punto fisso e un insieme $\mathbf{A}$ di operazioni aritmetiche di base.

1.1 Sintassi

1.1.1 Tipi

Useremo le meta-variabili σ, τ e υ per denotare i tipi, i quali sono definiti nel seguente modo:

$$\begin{array}{rcll} \sigma & ::= & \iota & \text{numeri naturali} \\ & | & o & \text{booleani} \\ & | & \sigma \Rightarrow \sigma & \text{freccia o tipo funzione} \end{array}$$

Useremo la meta-variabile β per denotare i tipi di base ι e o. $\sigma \Rightarrow \tau \Rightarrow \upsilon$ si legge come $\sigma \Rightarrow (\tau \Rightarrow \upsilon)$. Si noti che, per $n \geq 0$, ogni tipo può essere espresso unicamente come

$$\sigma_1 \Rightarrow \sigma_2 \cdots \Rightarrow \sigma_n \Rightarrow \beta$$

che abbrevieremo come $(\sigma_1, \sigma_2, \ldots, \sigma_n, \beta)$. Il *rank* (rango, ordine) di un tipo è definito come segue:

$$\begin{aligned} \text{rank}(\beta) &\stackrel{def}{=} 0, \\ \text{rank}(\sigma \Rightarrow \tau) &\stackrel{def}{=} \max(\text{rank}(\sigma)+1, \text{rank}(\tau)). \end{aligned}$$

Un tipo di base β ha rango 0. Se $\text{rank}(\sigma) = 1$, allora σ è un tipo del primo ordine. In generale, per $n \geq 0$, diremo che σ è un tipo di ordine n se $\text{rank}(\sigma) = n$.

1.1.2 Termini

Per ogni tipo σ, fissiamo un insieme $\{x^\sigma\}$ di variabili. Sia $\mathbf{C} = \{c^\sigma\}$ un insieme di costanti tipate. Le espressioni semplici del linguaggio $\lambda_{\stackrel{\rightarrow}{\text{Y}}}(\mathbf{C})$ parametrizzate sull'insieme $\mathbf{C}$ di costanti sono così definite:

M	::=	Ω^σ	termine indefinito
	|	c^σ	costante
	|	x^σ	variabile
	|	$(M \cdot M)$	applicazione
	|	$(\lambda x^\sigma . M)$	λ-astrazione
	|	$\mathbf{Y}^\sigma(M)$	Y-termine

L'applicazione $(M \cdot N)$ è scritta semplicemente MN. $MN_1 \cdots N_n$ è una abbreviazione per $(\cdots((MN_1)N_2)\cdots N_n)$, in quanto l'applicazione associa a sinistra. Un termine non tipato è una espressione semplice senza le etichette di tipo.

L'insieme delle variabili libere di un termine M, che denoteremo con $\mathsf{FV}(M)$, è così definito induttivamente:

$$\begin{aligned}
\mathsf{FV}(\Omega^\sigma) &\stackrel{def}{=} \emptyset \\
\mathsf{FV}(c^\sigma) &\stackrel{def}{=} \emptyset \\
\mathsf{FV}(x^\sigma) &\stackrel{def}{=} \{x^\sigma\} \\
\mathsf{FV}(MN) &\stackrel{def}{=} \mathsf{FV}(M) \cup \mathsf{FV}(N) \\
\mathsf{FV}(\lambda x^\sigma.M) &\stackrel{def}{=} \mathsf{FV}(M) \setminus \{x^\sigma\} \\
\mathsf{FV}(\mathbf{Y}^\sigma(M)) &\stackrel{def}{=} \mathsf{FV}(M)
\end{aligned}$$

Un termine M è *chiuso* se $\mathsf{FV}(M) = \emptyset$.

1.1.3 Regole di assegnazione dei tipi

Data una collezione $\mathbf{C}$ di costanti tipate, il sistema di riduzioni $\lambda^{\rightarrow}_{\mathrm{Y}}(\mathbf{C})$ consiste di espressioni semplici che sono ben tipate. La frase $M : \sigma$ asserisce che il tipo del termine M è σ. Il tipo di un

termine è definito induttivamente dalle seguenti regole:

$$x^\sigma : \sigma \qquad \Omega^\sigma : \sigma$$

$$c^\sigma : \sigma \qquad \frac{M : \sigma \Rightarrow \sigma}{\mathbf{Y}^\sigma(M) : \sigma}$$

$$\frac{M : \sigma \Rightarrow \tau \quad N : \sigma}{(M \cdot N) : \tau} \qquad \frac{M : \tau}{(\lambda x^\sigma . M) : \sigma \Rightarrow \tau}$$

1.1.4 Operazioni aritmetiche

Fissiamo un insieme **A** di operazioni aritmetiche di base con i rispettivi tipi:

n	:	ι	numerali, per ogni $n \geq 0$
t, f	:	o	booleani, vero e falso
succ	:	$\iota \Rightarrow \iota$	costante successore
pred	:	$\iota \Rightarrow \iota$	costante predecessore
$\mathsf{zero?}$	:	$\iota \Rightarrow o$	test per la costante zero
cond^ι	:	$o \Rightarrow \iota \Rightarrow \iota \Rightarrow \iota$	condizionale intero
cond^o	:	$o \Rightarrow o \Rightarrow o \Rightarrow o$	condizionale booleano

1.1.5 Regole di riduzione

Scriviamo l'operazione di sostituzione di termini come $M[N/x^\sigma]$ che significa "in M, sostituisci il termine N ad ogni occorrenza libera di x^σ", avendo cura di rinominare le variabili vincolate quando necessario per evitare un conflitto di variabili.

I contesti sono termini con dei "buchi" (holes) come sotto-termini, che possono essere riempiti con termini di tipo compatibile. Le meta-variabili X_i^σ, Y_j^τ denotano dei buchi; le etichette per i tipi saranno omesse quando possibile. La meta-variabile C denota invece un contesto. I contesti sono definiti come segue:

$$C \ ::= \ X^\sigma \mid \Omega^\sigma \mid c^\sigma \mid x^\sigma \mid (\lambda x^\sigma.C) \mid (C.C) \mid (\mathbf{Y}^\sigma(C)).$$

Scriveremo $C[X_1, \ldots, X_n]$ per denotare un contesto con i buchi $X_1, \ldots, X_n$. La sostituzione all'interno di un contesto è una operazione definita ricorsivamente come segue: per i termini $P_1, \ldots, P_n$ che sono di tipo compatibile rispettivamente con i buchi $X_1^{\sigma_1}, \ldots, X_n^{\sigma_n}$, $C[P_1, \ldots, P_n]$ è così definita:

$$\begin{cases} C & \text{se } C \text{ è una variabile} \\ & \text{o costante} \\ P_i & \text{se } C \text{ è il buco } X_i^{\sigma_i} \\ (C_1[P_1, \ldots, P_n] \cdot C_2[P_1, \ldots, P_n]) & \text{se } C = (C_1 \cdot C_2) \\ (\lambda x^\sigma.C'[P_1, \ldots, P_n]) & \text{se } C = (\lambda x^\sigma.C') \\ (\mathbf{Y}^\sigma(C'[P_1, \ldots, P_n])) & \text{se } C = (\mathbf{Y}^\sigma(C')) \end{cases}$$

È importante notare la differenza tra la sostituzione nei contesti e la sostituzione nei termini: le variabili possono diventare vincolate dopo la sostituzione in un contesto. Per esempio,

sia $C[X]$ il contesto $\lambda x.Xx$; allora $C[xy]$ è $\lambda x.(xy)x$. Invece, $C[X][(xy)/X]$ è $\lambda z.(xy)z$; si noti che la variabile vincolata x in $\lambda x.X.x$ è stata rinominata in z per evitare il conflitto.

La riduzione o riscrittura nel sistema $\lambda^{\rightarrow}_{\mathrm{Y}}(\mathbf{A})$ è rappresentata come una relazione binaria $\rightarrow$ sui termini chiusi. $M \rightarrow N$ si legge come "il termine M si riduce in un passo a N". Il sistema di riduzioni $\lambda^{\rightarrow}_{\mathrm{Y}}(\mathbf{A})$ così come presentato nel seguito non è deterministico: dato che non è legato a nessuna strategia di riduzione, esso non è ancora un linguaggio di programmazione. La relazione $\rightarrow$ è definita ricorsivamente dai seguenti insiemi di schemi di assiomi e di regole:

- **λ-riduzioni:**

  $(\lambda x^{\sigma}.M)N \rightarrow M[N/x^{\sigma}]$ $\quad$ $\mathbf{Y}^{\sigma}(M) \rightarrow M(\mathbf{Y}^{\sigma}(M))$

  $\Omega^{\sigma} \rightarrow \Omega^{\sigma}$

- **δ-riduzioni:** sia β un tipo di base,

  $\mathsf{cond}^{\beta}\mathsf{t} \rightarrow \lambda x^{\beta}.(\lambda y^{\beta}.x)$ $\quad$ $\mathsf{cond}^{\beta}\mathsf{f} \rightarrow \lambda x^{\beta}.(\lambda y^{\beta}.y)$

  $\mathsf{succ}n \rightarrow n+1$ $\quad$ $\mathsf{pred}n+1 \rightarrow n$

  $\mathsf{pred}0 \rightarrow 0$ $\quad$ $\mathsf{zero?}n+1 \rightarrow \mathsf{f}$

  $\mathsf{zero?}0 \rightarrow \mathsf{t}$ $\quad$ $\Omega^{\alpha} \rightarrow \Omega^{\alpha}$

- **chiusura compatibile:**

$$\frac{M \to M'}{C[M] \to C[M']}$$

La parte sinistra di una istanza di uno schema di assiomi di λ-riduzione è chiamato λ-*redex*; analogamente si definisce un δ-*redex*. La riduzione "sotto una lambda" è permessa nel sistema di riduzioni (ma non nel linguaggio di programmazione PCF come vedremo in seguito). Per esempio, $\lambda x^{\iota}.\mathsf{succ}1 \to \lambda x^{\iota}2$ è derivabile. Se nessun sottotermine di M è un λ-redex allora chiamiamo M una λ-*forma normale*; analogamente definiamo una δ-*forma normale*. Un termine M è una $\to$-*forma normale* (o semplicemente una forma normale) se non esiste nessun termine N tale che $M \to N$. Quando $M \to M'$ è una λ-riduzione scriveremo $M \to_{\lambda} M'$, mentre quando $M \to M'$ è una δ-riduzione scriveremo $M \to_{\delta} M'$.

Un $\lambda^{\to}_{Y}(\mathbf{A})$-*programma* (o semplicemente un programma) è un $\lambda^{\to}_{\mathrm{Y}}(\mathbf{A})$-termine chiuso di un tipo di base. I *valori di base* sono i numerali e i booleani. Useremo la meta-variabile V per denotare un valore di base. Da una semplice analisi sintattica dei casi, se un programma è in $\to$-forma normale, allora è un valore di base. Questo significa che quando la computazione di

un programma termina, si ha la garanzia che il risultato sia un dato significativo, piuttosto che un pezzo di codice insignificante come Ω^ι. Per questa ragione ammettiamo la riduzione $\Omega^\sigma \to \Omega^\sigma$.

Definiamo $\twoheadrightarrow$ come la chiusura riflessiva e transitiva di $\to$. Una relazione binaria $\mathbf{R}$ su un insieme S si dice che soddisfa la *proprietà del diamante* se quando $s\mathbf{R}s_1$ e $s\mathbf{R}s_2$, allora esiste un t tale che $s_1\mathbf{R}t$ e $s_2\mathbf{R}t$.

Lemma 1.1.1.

i. La relazione binaria $\twoheadrightarrow_\lambda$ soddisfa la proprietà del diamante.

ii. La relazione binaria $\twoheadrightarrow_\delta$ soddisfa la proprietà del diamante.

iii. Le relazioni binarie $\twoheadrightarrow_\lambda$ e $\twoheadrightarrow_\delta$ commutano; cioè, quando $M \twoheadrightarrow_\lambda M_1$ e $M \twoheadrightarrow_\delta M_2$, esiste un N tale che $M_1 \twoheadrightarrow_\delta N$ e $M_2 \twoheadrightarrow_\lambda N$.

Usando questi tre risultati si può dimostrare:

Proposizione 1.1.2 (Church-Rosser). *La riduzione $\to$ su $\lambda^{\to}_{Y}(\mathbf{A})$ è Church-Rosser, cioè se $M \twoheadrightarrow M_1$ e $M \twoheadrightarrow M_2$, allora esiste un N tale che $M_1 \twoheadrightarrow N$ e $M_2 \twoheadrightarrow N$.*

1.2 Semantica operazionale

La semantica operazionale di un linguaggio di programmazione si riferisce ad una specifica macchina astratta con cui è possibile eseguire un programma del linguaggio. Il significato di un programma è dato dalla sequenza di passi che la macchina compie per eseguirlo.

La semantica operazionale del linguaggio PCF si ottiene dal sistema di riduzioni $\lambda^{\rightarrow}_{\mathrm{Y}}(\mathbf{A})$ specificando come strategia di riduzione quella che sceglie la riduzione più a sinistra; i β-redex sono contratti secondo la modalità call-by-name (piuttosto che con la call-by-value). La strategia permette solo riduzioni *deboli*, cioè non sono consentite riduzioni "sotto una lambda". La semantica operazionale è presentata in termini di contesti di valutazione. Definiamo un contesto di valutazione come

$$E \ ::= \ X^{\sigma} \mid (E \cdot M) \mid (f \cdot E)$$

dove X^{σ} denota un buco di tipo σ, mentre M e f sono meta-variabili che denotano rispettivamente i termini PCF e le costanti dell'insieme $\mathbf{A}$. Si noti che il buco X occorre precisamente una volta in ognuno di tali contesti. La *riduzione di un passo* $>$ sui termini PCF chiusi è definita induttivamente secondo i seguenti

assiomi e regole:

$$(\lambda x^\sigma.M)N > M[N/x^\sigma] \qquad \mathbf{Y}^\sigma(M) > M(\mathbf{Y}^\sigma(M))$$

$$\Omega^\sigma > \Omega^\sigma \qquad \frac{M > M'}{E[M] > E[M']}$$

$$\mathsf{cond}^\beta \mathsf{t} MN > M \qquad \mathsf{cond}^\beta \mathsf{f} MN > N$$

$$\mathsf{succ} n > n+1 \qquad \mathsf{pred} n+1 > n$$

$$\mathsf{pred} 0 > 0 \qquad \mathsf{zero?} n+1 > f$$

$$\mathsf{zero?} 0 > t$$

Inoltre definiamo:

$$\begin{array}{lcl} \gg & \stackrel{def}{=} & \text{chiusura riflessiva e transitiva di } >, \\ M \gg_n M' & \stackrel{def}{\iff} & \exists\{M_0, \ldots, M_n\}.M \equiv M_0, M_n \equiv M' \\ & & \& \ M_i > M_{i+1}, \text{ per ogni } 0 \leq i < n, \\ M \Downarrow_n V & \stackrel{def}{\iff} & M \gg_n V, \text{ per qualche } n \geq 0, \\ M \Downarrow V & \stackrel{def}{\iff} & M \Downarrow_n V, \text{ per qualche } n \geq 0. \end{array}$$

Ricordiamo che i programmi PCF sono termini chiusi di tipi di base. I valori sono le λ-astrazioni e le costanti; useremo la meta-variabile V per denotare un valore. Seguendo il paradigma funzionale, eseguire un programma in PCF significa valutarlo.

Possiamo presentare la semantica operazione del PCF in modo equivalente, in termini di una relazione di valutazione. Formalmente, definiamo ricorsivamente una relazione $\Downarrow$ tra termini

chiusi e valori tramite le seguenti regole (leggiamo $M \Downarrow V$ come "il valore del termine chiuso M è V"):

$$V \Downarrow V \qquad \frac{M \Downarrow n}{\mathsf{succ}M \Downarrow n+1} \qquad \frac{M \Downarrow n+1}{\mathsf{pred}M \Downarrow n}$$

$$\frac{M \Downarrow 0}{\mathsf{pred}M \Downarrow 0} \qquad \frac{M \Downarrow \mathsf{t} \quad P \Downarrow V}{\mathsf{cond}^{\beta}MPQ \Downarrow V} \qquad \frac{M \Downarrow \mathsf{f} \quad Q \Downarrow V}{\mathsf{cond}^{\beta}MPQ \Downarrow V}$$

$$\frac{M \Downarrow 0}{\mathsf{zero?}M \Downarrow t} \qquad \frac{M \Downarrow n+1}{\mathsf{zero?}M \Downarrow \mathsf{f}} \qquad \frac{P[N/x] \Downarrow V}{(\lambda x.P)N \Downarrow V}$$

$$\frac{M\mathbf{Y}^{\sigma}(M) \Downarrow V}{\mathbf{Y}^{\sigma}(M) \Downarrow V} \qquad \frac{M \Downarrow V \quad VN \Downarrow V'}{MN \Downarrow V'}$$

Naturalmente le due nozioni di $\Downarrow$ coincidono.

1.2.1 Lemma del Contesto

Esaminiamo una proprietà operazionale del PCF chiamata lemma del contesto, secondo il quale l'"equivalenza osservazionale" tra termini è determinata dal solo comportamento applicativo.

Diciamo che due pezzi di programma M e N sono *osservazionalmente equivalenti*, e scriveremo $M \approx N$, se per ogni contesto $C[X]$ tale che sia $C[M]$ sia $C[N]$ sono programmi, e per ogni valore V, $C[M] \Downarrow V$ se e solo se $C[N] \Downarrow V$. Scriveremo invece $M \precsim N$ per indicare che, per ogni contesto $C[X]$ tale che sia $C[M]$ sia $C[N]$ sono programmi, e per ogni valore V, se

$C[M] \Downarrow V$ allora $C[N] \Downarrow V$. Naturalmente, nelle definizioni assumiamo che M e N siano dello stesso tipo, e consideriamo solo quei contesti $C[X]$ i cui buchi sono dello stesso tipo di M e N.

Il lemma del contesto dice che se M e N sono osservazionalmente inequivalenti, allora esiste un particolare tipo di contesto, detto *applicativo*, che ci mostra le differenze tra i due termini. M e N sono osservazionalmente inequivalenti nel caso in cui esiste un contesto $C[X]$ che li distingue: il programma $C[M]$ converge a un valore di base, diciamo V, mentre $C[N]$ no (cioè $C[N]$ converge a un valore diverso da V o non converge), o viceversa. Un contesto applicativo è un contesto che ha precisamente un buco, il quale occorre nella posizione più a sinistra. Più precisamente un contesto applicativo è un contesto del tipo $C[X] \equiv XM_1 \cdots M_n$ dove gli M_i sono termini PCF.

Teorema 1.2.1 (Lemma del contesto). *Siano M e N termini* PCF *chiusi dello stesso tipo σ, che scriviamo come $(\sigma_1, \ldots, \sigma_n, \beta)$. Se M e N soddisfano la seguente condizione:*

> *Per ogni n-pla di termini* PCF *chiusi $P_1 : \sigma_1, \ldots, P_n : \sigma_n$, e per ogni valore V, se $MP_1 \cdots P_n \Downarrow V$, allora anche $NP_1 \cdots P_n \Downarrow V$;*

allora $M \precsim N$.

Il significato del lemma del contesto è il seguente: i termini del linguaggio PCF sono determinati dal loro comportamento applicativo.

1.3 Semantica denotazionale

Secondo la semantica denotazionale, il significato di un programma è un elemento di una struttura matematica scelta come universo del discorso. Questa struttura rappresenta il modello denotazionale. Una semantica o funzione di valutazione associa ai programmi delle denotazioni che sono elementi del modello. Una caratteristica importante dell'approccio denotazionale è la definizione di una funzione di valutazione tramite induzione strutturale. Questo comporta una interpretazione *composizionale* dei programmi: il significato di un programma è definito in termini dei rispettivi significati dei suoi componenti.

Dato che PCF è un linguaggio funzionale tipato, una interpretazione del linguaggio deve mettere a disposizione un sistema di "domini" che danno una interpretazione ai tipi semplici, un dominio per ogni tipo. La denotazione di un termine PCF è data da

una funzione di valutazione che rispetta il tipo, la quale prende un termine PCF e un ambiente che assegna delle denotazioni alle variabili che occorrono libere nel termine, per produrre una denotazione del termine come un elemento del dominio appropriato. Seguendo lo stile delle semantiche denotazionali, la funzione di valutazione deve essere definita composizionalmente.

Una caratteristica importante del PCF è la ricorsione generale nella forma di un operatore di punto fisso per ogni tipo funzione $\sigma \Rightarrow \sigma$.

Una *collezione di domini di valori* per PCF comprende:

- una famiglia $\{D^\sigma\}$ di CPO (la definizione è riportata in seguito), uno per ogni tipo σ,
- per ogni coppia di tipi σ e τ, una operazione $\cdot : D^{\sigma \Rightarrow \tau} \times D^\sigma \to D^\tau$. Dati $d \in D^{\sigma \Rightarrow \tau}$ e $e \in D^\sigma$, spesso scriveremo $d \cdot e$ semplicemente come de.

Un *ordinamento parziale completo* (CPO, complete partial order) è un insieme parzialmente ordinato $\langle D, \sqsubseteq \rangle$ tale che:

i. Esiste l'elemento minimo (denoteremo l'elemento minimo di D^σ con $\bot^\sigma$, o semplicemente con $\bot$);

ii. Ogni sottoinsieme diretto X ha un minimo estremo superiore (lub, least upper bound) in D (che denoteremo con $\bigsqcup X$).

Un sottoinsieme X di un insieme parzialmente ordinato $\langle D, \sqsubseteq \rangle$ è *diretto* se ogni coppia di elementi di X ha un estremo superiore in X.

Una *interpretazione* del linguaggio $\lambda^{\rightarrow}_{\mathrm{Y}}(\mathbf{A})$ è una coppia $\langle \{D^\sigma\}, \mathcal{A} \rangle$, dove $\{D^\sigma\}$ è una collezione di domini di valori per PCF e la funzione

$$\mathcal{A} : \mathbf{A} \longrightarrow \bigcup_\sigma D^\sigma$$

rispetta i tipi, cioè se $c \in \mathbf{A}$ è una costante di tipo σ, allora $\mathcal{A}(c)$ è un elemento di D^σ. Tale interpretazione $\mathcal{A}$ (di $\mathbf{A}$) è detta essere *standard* se D^ι è il CPO piatto (flat) dei numeri naturali, ordinati come in figura 1.1, e D^o è il CPO piatto dei booleani ordinati come in figura 1.2; inoltre deve valere la seguente interpretazione

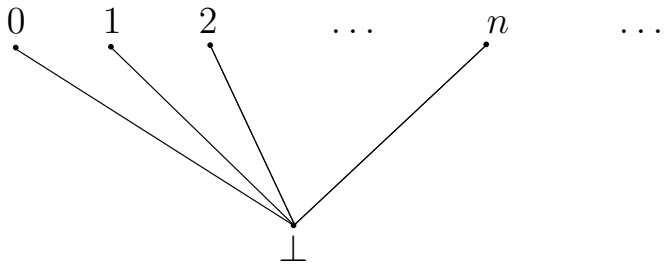

Figura 1.1: Il CPO piatto dei numeri naturali.

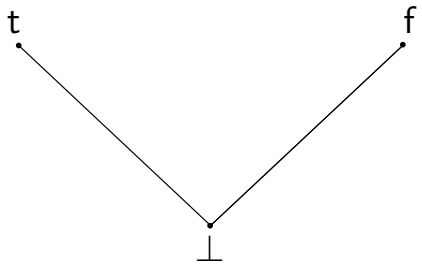

Figura 1.2: Il CPO piatto dei valori booleani.

delle costanti: per $d, e \in D^\beta, n \in D^\iota, b \in D^o$,

$$\mathcal{A}(\mathsf{t}) = \mathsf{t} \qquad \mathcal{A}(\mathsf{f}) = \mathsf{f}$$

$$\mathcal{A}(\mathsf{cond}^\beta)bde = \begin{cases} d & \text{se } b = \mathsf{t} \\ e & \text{se } b = \mathsf{f} \\ \bot & \text{se } b = \bot \end{cases} \qquad \mathcal{A}(\mathsf{zero?})n = \begin{cases} \mathsf{t} & \text{se } n = 0 \\ \mathsf{f} & \text{se } n > 0 \\ \bot & \text{se } n = \bot \end{cases}$$

$$\mathcal{A}(\mathsf{succ})n = \begin{cases} n+1 & \text{se } n \geq 0 \\ \bot & \text{se } n = \bot \end{cases} \qquad \mathcal{A}(\mathsf{pred})n = \begin{cases} n-1 & \text{se } n \geq 1 \\ 0 & \text{se } n = 0 \\ \bot & \text{se } n = \bot \end{cases}$$

$$\mathcal{A}(n) = n \quad (n \geq 0).$$

Abbiamo bisogno ora di un insieme Env di *ambienti*. Un ambiente è una funzione, che rispetta i tipi, dall'insieme delle variabili a $\cup_\sigma D^\sigma$. Denoteremo gli ambienti con la meta-variabile ρ. L'operazione di modifica è definita come segue: per $d \in D^\sigma$,

$$\rho[x^\sigma \mapsto d](y) \stackrel{def}{=} \begin{cases} d & \text{se } y = x^\sigma \\ \rho(x) & \text{altrimenti.} \end{cases}$$

L' ambiente *indefinito* $\perp$ manda ogni variabile x^σ in $\perp^\sigma$.

Data una interpretazione $\langle \{D^\sigma\}, \mathcal{A} \rangle$, un *modello continuo del minimo punto fisso* di PCF è una funzione semantica che rispetta i tipi

$$\mathcal{A}[\![-]\!](-) : \lambda^{\rightarrow}_{\mathrm{Y}}(\mathbf{A}) \longrightarrow (\mathrm{Env} \longrightarrow \bigcup_\sigma D^\sigma)$$

e che soddisfa i seguenti criteri:

Semantica composizionale del minimo punto fisso Siano M e N termini PCF e $d \in D^\sigma$,

$$\mathcal{A}[\![x^\sigma]\!](\rho) = \rho(x^\sigma), \tag{1.1}$$

$$\mathcal{A}[\![\Omega^\sigma]\!](\rho) = \perp^\sigma, \tag{1.2}$$

$$\mathcal{A}[\![c^\sigma]\!](\rho) = \mathcal{A}(c^\sigma), \tag{1.3}$$

$$\mathcal{A}[\![MN]\!](\rho) = \mathcal{A}[\![M]\!]\rho \cdot \mathcal{A}[\![N]\!](\rho), \tag{1.4}$$

$$\mathcal{A}[\![\lambda x^\sigma.M]\!](\rho) \cdot d = \mathcal{A}[\![M]\!](\rho[x^\sigma \mapsto d]), \tag{1.5}$$

$$\mathcal{A}[\![\mathbf{Y}^\sigma(M)]\!](\rho) = \mathsf{fix}_\sigma(\mathcal{A}[\![M]\!](\rho)); \tag{1.6}$$

dove $\mathsf{fix}_\sigma(f)$ è una abbreviazione di $\bigsqcup\{f^n(\bot^\sigma) : n \in \omega\}$, per ogni $f \in D^{\sigma\Rightarrow\sigma}$. ($f^0(\bot)$ sta per $\bot$ e $f^n(\bot)$ sta per $\underbrace{f(\cdots(f}_{n}(\bot))\cdots)$.)

L'interpretazione dell'operatore di punto fisso $\mathbf{Y}^\sigma(-)$ si basa sul teorema del punto fisso di Knaster-Tarski. Per applicare il teorema è necessario che la denotazione di ogni termine PCF chiuso di tipo $\sigma \Rightarrow \sigma$ (che già sappiamo essere un elemento di $D^{\sigma\Rightarrow\sigma}$) sia una funzione continua da D^σ a D^σ. Una funzione $f : D \longrightarrow E$, tra i CPO D e E, è *continua* se $f(\bigsqcup X) = \bigsqcup f(X)$, per ogni sottoinsieme diretto X di D.[1] Questo ci costringe a porre un ulteriore vincolo sulla collezione dei domini di valori: per ogni tipo funzione $\sigma \Rightarrow \tau$, il dominio $D^{\sigma\Rightarrow\tau}$ deve essere un sottoCPO del CPO $[D^\sigma \Rightarrow D^\tau]$ delle funzioni continue da D^σ a D^τ.

Vincoli strutturali Siano M e N termini ben tipati e $\rho, \rho' \in$

[1] L'esistenza di $\bigsqcup f(X)$ è garantita se f è monotona, cioè se $\forall d, d' \in D$: $d \sqsubseteq_D d' \Longrightarrow f(d) \sqsubseteq_E f(d')$, in quanto l'immagine di un insieme diretto, data da una funzione che conserva l'ordine, è un insieme diretto.

Env,

(Env) se $\rho(x^\sigma) = \rho'(x^\sigma)$ per ogni $x^\sigma \in \mathsf{FV}(M)$,
allora $\mathcal{A}[\![M]\!](\rho) = \mathcal{A}[\![M]\!](\rho')$;

(Subst) $\mathcal{A}[\![M[N/x^\sigma]]\!](\rho) = \mathcal{A}[\![M]\!](\rho[x^\sigma \mapsto \mathcal{A}[\![N]\!](p)])$;

(Cont) se $\mathcal{A}[\![M]\!](\rho) = \mathcal{A}[\![N]\!](\rho)$ per ogni ρ,
allora per ogni contesto di tipo compatibile $C[X]$
abbiamo che $\mathcal{A}[\![C[M]]\!](\rho) = \mathcal{A}[\![C[N]]\!](\rho)$ per ogni ρ.

Secondo il primo vincolo strutturale, (Env), la valutazione di un termine deve chiaramente essere invariante sugli ambienti che concordano su tutte le variabili che occorrono libere nel termine. Per questa ragione, se un termine è chiuso (non ha variabili libere), non c'è pericolo nello scrivere la denotazione di M come $\mathcal{A}[\![M]\!](\bot)$, dove $\bot$ è l'ambiente che associa $\bot$ a ogni variabile; si può anche scrivere $\mathcal{A}[\![M]\!]$.

Il secondo vincolo strutturale, (Subst), insieme ai precedenti vincoli semantici del minimo punto fisso rende valida la β-equivalenza; questo significa che, per ogni coppia di termini PCF $M : \tau$ e $N : \sigma$, e per ogni ambiente ρ,

$$\mathcal{A}[\![(\lambda x^\sigma.M)N]\!](\rho) = \mathcal{A}[\![M[N/x^\sigma]]\!](\rho)$$

Infatti:

$$\begin{aligned}
\mathcal{A}[\![(\lambda x^{\sigma}.M)N]\!](\rho) &= \mathcal{A}[\![\lambda x^{\sigma}.M]\!](\rho) \cdot \mathcal{A}[\![N]\!](\rho) && \text{per 1.4} \\
&= \mathcal{A}[\![M]\!](\rho[x^{\sigma} \mapsto \mathcal{A}[\![N]\!](\rho)]) && \text{per 1.5} \\
&= \mathcal{A}[\![M[N/x^{\sigma}]]\!](\rho) && \text{per (Subst)}
\end{aligned}$$

Infine, il terzo vincolo strutturale, (Cont), asserisce che la valutazione di termini PCF è una operazione "libera da contesto".

Correttezza operazionale Se $M \to N$, allora $\mathcal{A}[\![M]\!](\rho) = \mathcal{A}[\![N]\!](\rho)$ per ogni ρ.

Assumendo che l'interpretazione dei tipi di base è standard, questo vincolo assicura che quando un programma converge a un valore, la sua denotazione coincide con esso.

Inoltre il modello è detto essere di *ordine estensionale* se, per ogni $f, g \in D^{\sigma \Rightarrow \tau}$, $f \sqsubseteq g$ se e solo se $f \cdot d \sqsubseteq g \cdot d$, per ogni $d \in D^{\sigma}$. Il modello è *estensionale* se, per ogni $f, g \in D^{\sigma \Rightarrow \tau}$, $f = g$ se e solo se $f \cdot d = g \cdot d$, per ogni $d \in D^{\sigma}$. La condizione di estensionalità è soltanto un altro modo di dire che per ogni tipo funzione $\sigma \Rightarrow \tau$, il dominio di valori $D^{\sigma \Rightarrow \tau}$ è una collezione di funzioni da D^{σ} a D^{τ}. Ovviamente un modello è estensionale se è di ordine estensionale.

Capitolo 2

Il problema della full abstraction per PCF

Secondo la semantica operazionale, due programmi sono equivalenti se la macchina che li esegue si comporta allo stesso modo. Al contrario, secondo la semantica denotazionale, due programmi sono equivalenti se denotano lo stesso oggetto nel modello denotazionale. Dato che i due stili di semantica sono diversi concettualmente, non c'è nessuna ragione per cui le due teorie di equivalenza debbano coincidere. Quindi è naturale chiedersi come una teoria di equivalenza di programmi è correlata all'altra.

Lo scopo primario della semantica denotazionale è quello di fornire una definizione canonica dei significati dei programmi.

Una definizione canonica è importante perché non solo documenta il progetto di un linguaggio, ma stabilisce anche uno standard matematico per l'implementazione del linguaggio sui calcolatori. Senza delle fondamenta sicure per i significati dei programmi, non ci possono essere delle basi per ragionare sulla correttezza e su altre proprietà di programmi. Al contrario, la semantica operazionale è adatta a catturare le intuizioni di un linguaggio di programmazione a un livello più vicino alla macchina piuttosto che al programmatore. E' ideale per descrivere implementazioni a basso livello piuttosto che specifiche ad alto livello di un linguaggio di programmazione. Riassumento, ogni stile di semantica è adatto per qualcosa. Si possono quindi usare entrambi, cosicché uno complementi l'altro. Questo ci porta ad un punto cruciale: affinché sia possibile beneficiare di *entrambi* gli stili di semantica, è necessario sapere come uno stile è correlato all'altro.

Esaminiamo più da vicino ognuno dei due stili di equivalenza di programmi. Cosa significa che due programmi (o pezzi di programmi) hanno lo stesso comportamento a livello di osservazioni quando sono eseguiti sulla stessa macchina? I programmatori comprendono bene questa nozione di equivalenza: due

pezzi di programma sono equivalenti se possono sempre essere *intercambiati* senza influenzare il risultato visibile o osservabile della computazione. Questo criterio di equivalenza, chiamato equivalenza osservazionale, è formalmente espresso in termini di invarianza del risultato osservabile sotto tutti i contesti di programma. Come abbiamo visto nella sezione 1.2.1,

> Due pezzi di programma M e N sono detti essere *osservazionalmente equivalenti*, e scriveremo $M \approx N$, se per ogni contesto di tipo compatibile $C[X]$ tale che $C[M]$ e $C[N]$ sono programmi, e per ogni valore V, $C[M] \Downarrow V$ se e solo se $C[N] \Downarrow V$.

In [3], Scott ha introdotto quello che sarebbe divenuto il modello dello spazio delle funzioni continue di Scott, per i programmi PCF. Ha usato i CPO piatti standard per interpretare i booleani e i numeri naturali, e lo spazio delle funzioni continue di Scott per interpretare i tipi funzione. Chiameremo questo modello il *modello standard continuo* (o semplicemente, il *modello standard*) del PCF. L'aggettivo "continuo" qualifica il modo in cui sono interpretati gli operatori di punto fisso, cioè in modo standard come i minimi estremi superiori delle ω-catene

delle successive iterate.[1] Il modello standard è inoltre di ordine estensionale.

Scrivendo $\mathcal{A}[\![M]\!]$ per denotare il significato del termine M (trascurando l'ambiente), diciamo che la semantica denotazionale $\mathcal{A}[\![-]\!]$ è *adeguata* se per ogni coppia di termini M e N, di tipo compatibile,

$$\mathcal{A}[\![M]\!] = \mathcal{A}[\![N]\!] \qquad \Longrightarrow \qquad M \approx N$$

Se, in aggiunta, è valida anche l'implicazione inversa, e cioè

$$\mathcal{A}[\![M]\!] = \mathcal{A}[\![N]\!] \qquad \Longleftrightarrow \qquad M \approx N$$

allora la semantica denotazionale è detta essere *fully abstract* per il linguaggio.

Adeguatezza e full abstraction ci dicono come i punti di vista operazionale e denotazionale sull'equivalenza di programmi sono correlati l'uno con l'altro. Esse sono indicazioni di quanto affidabile o di quanto "adatto" sia il modello denotazionale rispetto al linguaggio. Più specificamente, l'adeguatezza ci assicura che il modello è abbastanza affidabile per affermare l'equivalenza osservazionale tra due termini, dato che per farlo è sufficiente avere l'uguaglianza denotazionale; ma il modello non

[1] Vedi sezione 1.3.

è abbastanza affidabile per rifiutare l'equivalenza, per cui abbiamo bisogno della full abstraction. L'adeguatezza è di solito relativamente semplice da dimostrare, ma non è così per la full abstraction. Di solito un modello non è fully abstract perché è, in qualche senso, una struttura troppo ricca per il linguaggio: contiene oggetti semantici che "non possono essere computati" dal linguaggio di programmazione. Al contrario, un modello che è fully abstract per un linguaggio fornisce un caratterizzazione molto soddisfacente del linguaggio e della sua equivalenza osservazionale.

Plotkin in [4] ha dimostrato che il modello standard è adeguato ma non fully abstract per PCF. Egli ha anche indicato la causa del fallimento della full abstraction. Essa può essere spiegata, in poche parole, dal fatto che mentre i programmi PCF corrispondono a algoritmi "sequenziali", il modello standard dello spazio delle funzioni continue di Scott contiene funzioni "parallele", o più precisamente funzioni che possono essere implementate solo da algoritmi paralleli.

2.1 Requisiti per un modello fully abstract

Ora, nonostante esista un consenso ampiamente diffuso sul fatto che il problema della full abstraction sia difficile da risolvere, non esiste però una idea ugualmente accettata su cosa costituisca una soluzione al problema. La questione è di natura filosofica e riguarda lo stabilire cosa è un buon modello: un buon modello illumina, dà una nuova prospettiva sul comportamento o semantica operazionale del linguaggio in questione. Un buon modello è astratto e sintetico. Un modello astratto è un modello costruito senza ricorrere alla sintassi o alla semantica operazionale del linguaggio. Più il modello è computazionalmente neutrale nella sua concezione, più è adatto come soluzione al problema. Per modello sintetico si intende invece un modello con una descrizione costruttiva e assiomatica dello spazio delle funzioni che interpreta i tipi del PCF (in termini delle rispettive interpretazioni delle componenti).

Consideriamo il modello standard dello spazio delle funzioni continue di Scott: questo chiaramente soddisfa il criterio di costruzione astratta, in quanto CPO e funzioni continue sono oggetti semantici computazionalmente neutrali. Anche la descri-

zione sintetica fornita dal modello è soddisfacente: i domini di Scott e le funzioni continue costituiscono una categoria cartesiana chiusa. Il modello dello spazio delle funzioni continue fallisce però nel caratterizzare la sequenzialità ad alti tipi del PCF perché "troppo grande"; infatti, alcuni algoritmi paralleli, che non sono definibili in PCF, sono denotati da funzioni continue.

2.2 Full abstraction e sequenzialità

Dato che il punto cruciale del problema della full abstraction è la caratterizzazione delle computazioni *sequenziali*, possiamo riformulare il problema della full abstraction per PCF come il problema di trovare una caratterizzazione astratta e sintetica dei funzionali sequenziali definibili in PCF. Con questa formulazione evidenziamo le difficoltà inerenti al problema, dato che non abbiamo una definizione adatta di sequenzialità ad alti tipi. Infatti, non è chiaro se esistono diverse nozioni non equivalenti di sequenzialità di ad alti tipi, ognuna delle quali ugualmente attraente o, come la nozione di computabilità effettiva, ne esiste essenzialmente una sotto possibili forme differenti.

Sebbene non ci sia ancora un consenso sul fatto che PCF sia

un esempio canonico di linguaggio di programmazione sequenziale ad alti tipi, c'è molta evidenza che PCF calcola solo funzioni sequenziali. Naturalmente l'aggettivo sequenziale è usato in un senso informale. Espressa in termini tecnologicamente più precisi, l'intuizione generale è che un linguaggio è sequenziale "se può essere implementato senza dividere il tempo tra più processi di controllo". Sebbene l'idea di sequenzialità è intuitivamente chiara, è molto difficile formularla in termini matematici: la sequenzialità del primo ordine si può caratterizzare abbastanza facilmente; la reale difficoltà è definire la sequenzialità ad alti tipi.

Sembra abbastanza chiaro che il PCF come linguaggio di programmazione "definisce" un modello di funzioni computabili sequenziali di alto tipo. Chiamiamo questo modello di computazione sequenziale (formalmente definito in termini della collezione di funzioni numeriche di alto tipo definibili in PCF) *sequenzialità* PCF . Ora, un modello fully abstract di PCF che soddisfa i due criteri di astrazione e di descrizione sintetica sarebbe una buona caratterizzazione di questo particolare modello di sequenzialità ad alti tipi. Quello che è meno chiaro a questo punto è se la sequenzialità PCF così definita è in qualche senso *canonica.*

Detto diversamente, la sequenzialità PCF è universale, o in qualche senso inevitabile? O è solo una nozione *ad hoc*? C'è qualche evidenza per supportare la seguente proposizione che può essere chiamata la tesi di Church-Turing per la sequenzialità ad alti tipi?

> **Tesi di Church-Turing per la sequenzialità** *La collezione di funzioni computabili sequenziali di alto tipo, intuitivamente chiara ma definita informalmente, è data dalle funzioni numeriche di alto tipo che sono definibili in* PCF.

Come la tesi di Church-Turing, anche questa tesi non ammette nessuna dimostrazione matematica nel senso convenzionale. Un modo per ottenere la sua accettazione, come è avvenuto per la tesi di Church-Turing, è quello di costruire un certo numero di modelli, intuitivamente attraenti, di computazioni sequenziali di alto tipo, ognuno dei quali computa precisamente la classe delle funzioni definibili in PCF.

2.3 Adeguatezza e Adeguatezza Debole

Come abbiamo visto, una semantica denotazionale è adeguata per un linguaggio di programmazione se l'uguaglianza denotazionale implica l'equivalenza osservazionale. Questa implicazione è spesso la più semplice delle due direzioni da provare. Se vale anche l'implicazione nell'altra direzione, allora la semantica denotazionale è detta essere fully abstract per il linguaggio.

Il comportamento operazionale di un linguaggio di programmazione può essere correlato al modello denotazionale in un senso ancora più debole: diciamo che la semantica denotazionale è *debolmente adeguata* se per ogni programma M e per ogni valore V

$$\mathcal{A}[\![M]\!](\bot) = \mathcal{A}[\![V]\!](\bot) \qquad \Longleftrightarrow \qquad M \Downarrow V.$$

L'adeguatezza debole esprime una corrispondenza debole tra il comportamento denotazionale ed operazionale dei programmi. Se non valesse per la semantica denotazionale in questione, probabilmente quest'ultima non sarebbe utile. L'adeguatezza debole è quindi una condizione indispensabile nel valutare quanto una semantica denotazionale sia adatta ad un linguaggio di programmazione.

Se la funzione di valutazione è composizionale, o più precisamente, se il vincolo strutturale (Cont)[2] è soddisfatto, allora l'adeguatezza debole implica l'adeguatezza. Si consideri una coppia di termini PCF M e N, e si supponga che $\mathcal{A}[\![M]\!](\rho) = \mathcal{A}[\![N]\!](\rho)$ per ogni ambiente ρ. Dato un contesto $C[X]$ tale che $C[M]$ e $C[N]$ siano programmi, si supponga che $C[M] \Downarrow V$ per qualche valore V. Dall'adeguatezza debole, $\mathcal{A}[\![C[M]]\!](\bot) = \mathcal{A}[\![V]\!](\bot)$. Dal vincolo strutturale (Cont), abbiamo che $\mathcal{A}[\![C[M]]\!](\bot) = \mathcal{A}[\![C[N]]\!](\bot)$. Quindi, deduciamo che $\mathcal{A}[\![C[N]]\!](\bot) = \mathcal{A}[\![V]\!](\bot)$ per ogni contesto $C[X]$. Di nuovo dall'adeguatezza debole, abbiamo che $C[N] \Downarrow V$. Esattamente nello stesso modo, possiamo far vedere che $C[N] \Downarrow V$ implica $C[M] \Downarrow V$.

Teorema 2.3.1 (Adeguatezza Debole). *Sia $\mathcal{A}[\![-]\!](-)$ un modello continuo di* PCF *che segue l'interpretazione standard. Per ogni programma M e ogni valore di base V, se $\mathcal{A}[\![M]\!](\bot) = \mathcal{A}[\![V]\!](\bot)$ allora $M \Downarrow V$.*

La strategia di dimostrazione si basa sul cosiddetto *argomento di computabilità* [4]. Essa illustra una caratteristica saliente del ragionamento induttivo nel lambda calcolo tipato: spesso è

[2]Vedi sezione 1.3.

necessario dimostrare una proprietà uniformemente per tutti i termini quando siamo interessati alla sua validità solo per un sottoinsieme proprio dei termini.

Si usi V per denotare un valore di base. Un termine chiuso di un tipo di base (cioè un programma) è computabile se e solo se soddisfa l'asserzione del teorema sulla adeguatezza debole. La nozione di computabilità si estende ai termini di alto tipo e a quelli che contengono variabili libere:

- Se $M : \beta$ è chiuso, allora M è computabile se $\mathcal{A}[\![M]\!] = \mathcal{A}[\![V]\!]$ implica $M \Downarrow V$.
- Se $M : \sigma \Rightarrow \tau$ è chiuso, allora M è computabile se MN è computabile per ogni termine chiuso computabile $N : \sigma$.
- Se $M : \sigma$ ha variabili libere $\{x_1^{\sigma_1}, \ldots, x_n^{\sigma_n}\}$, allora M è computabile se per ogni sostituzione θ che mappa ogni $x_i^{\sigma_i}$ a un termine chiuso computabile N_i, il termine $M_\theta \equiv M[N_1/x_1^{\sigma_1}, \ldots, N_n/x_n^{\sigma_n}]$ è computabile. (Si noti che stiamo considerando la sostituzione come una funzione dalle variabili ai termini che rispetta i tipi.)

Combinando i tre casi, osserviamo che un termine $M : (\sigma_1, \ldots, \sigma_n, \beta)$ è *computabile* se e solo se

- per ogni sostituzione θ che associa alle variabili libere di M dei termini chiusi computabili, e
- per ogni n-pla di termini chiusi computabili $N_1 : \sigma_1, \ldots, N_n : \sigma_n$,

il termine $M_\theta N_1 \cdots N_n : \beta$ è computabile.

Proposizione 2.3.2. *Ogni termine* PCF *è computabile.*

Per la dimostrazione vedere [4] o [5].

Se le proprietà matematiche di un modello denotazionale sono ben comprese, allora l'approccio denotazionale porta ad un modo "algebrico" di ragionare sulle proprietà globali di programmi. Comunque, affinché le fondamenta semantiche fornite dal modello denotazionale siano rilevanti per il linguaggio di programmazione in questione, il modello deve essere adatto al comportamento operazionale dei programmi. L'adeguatezza fornisce un criterio di fondamentale importanza per valutare i modelli denotazionali.

2.4 Risultati sulla full abstraction

Definizione 2.4.1 (Elemento compatto). *Un elemento d di un* CPO *D è* compatto *se, per ogni sottoinsieme diretto X di D,*

se $d \sqsubseteq \bigsqcup X$ *allora esiste* $x \in X$ *tale che* $d \sqsubseteq x$.

Definizione 2.4.2 (Cpo algebrico, Cpo ω-algebrico). *Un* CPO D *è* algebrico *se per ogni* $d \in D$*, il sottoinsieme* $\{x \sqsubseteq d : x$ *è compatto*$\}$ *di* D *è diretto e il suo minimo estremo superiore è* d*. Inoltre,* D *è* ω-algebrico *se ha un insieme infinito numerabile di elementi compatti.*

Definizione 2.4.3 (Cpo consistentemente completo). *Un* CPO D *è* consistentemente completo *se ogni coppia di elementi di* D *che ha un estremo superiore in* D *ha anche un minimo estremo superiore in* D.

Definizione 2.4.4 (Dominio di Scott). *Un* dominio di Scott *è un* CPO *ω-algebrico, consistentemente completo.*

Supponiamo che una collezione $\{D^\sigma\}$ di CPO come domini di valori per PCF costituisca un modello continuo di ordine estensionale per PCF. Quali proprietà possiamo inferire sulla struttura del sistema di CPO $\{D^\sigma\}$? Il seguente risultato afferma che tali domini di valori sono domini di Scott. Inoltre esiste una importante condizione necessaria e sufficiente che ci dice quando un modello continuo e di ordine estensionale è fully abstract.

Teorema 2.4.5 (Definibilità). *Sia* $\mathbb{D} = \langle \{D^\sigma\}, \mathcal{A} \rangle$ *un modello di* PCF *continuo e di ordine estensionale, che segue l'interpretazione standard. Allora, per ogni tipo* σ*, il dominio* D^σ *è un dominio di Scott. Inoltre,* $\mathbb{D}$ *è fully abstract se e solo se ogni elemento compatto di ogni dominio di valori* D^σ *è definibile.*

Vedere [5] per una dimostrazione.

In [4], Plotkin ha dimostrato che il modello standard dello spazio delle funzioni continue di Scott non è fully abstract per PCF, ma lo è invece per PCF esteso con il costrutto "condizionale parallelo".

Proposizione 2.4.6. *Il modello standard dello spazio delle funzioni continue di Scott non è fully abstract per* PCF*.*

Dimostrazione. Dall'adeguatezza abbiamo che l'equivalenza semantica implica l'equivalenza operazionale. Affinché il modello sia fully abstract dovrebbe essere vero anche l'inverso. Facciamo vedere che ciò non è vero con un controesempio: definiamo due termini M_0 e M_1 tali che $M_0 \approx M_1$ ma non vale $\mathcal{A}[\![M_0]\!] = \mathcal{A}[\![M_1]\!]$. Per $i = 0, 1$,

$$M_i = \lambda f.\mathsf{cond}^\iota(f\,\mathsf{t}\,\Omega^o)(\mathsf{cond}^\iota(f\,\Omega^o\,\mathsf{t})(\mathsf{cond}^\iota(f\,\mathsf{f}\,\mathsf{f})\Omega^\iota n_i)\Omega^\iota)\Omega^\iota,$$

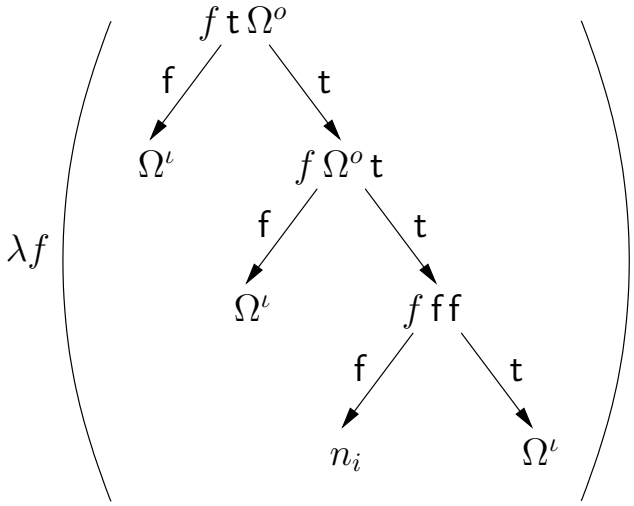

Figura 2.1: I termini M_i.

in cui il tipo di f è (o, o, o), mentre n_0 e n_1 sono due numerali distinti. Per rendere più comprensibili i termini, scriviamoli in forma diagrammatica (figura 2.1).

Definiamo V in $D^{o \Rightarrow o \Rightarrow o}$ mediante la seguente tabella:

V	$\bot$	f	t
$\bot$	$\bot$	$\bot$	t
f	$\bot$	f	t
t	t	t	t

Applicando M_i a V abbiamo che $\mathcal{A}[\![M_i]\!](\bot)(V) = n_i$. Non abbiamo quindi l'equivalenza semantica tra i due termini. Dimostriamo ora che $M_0 \approx M_1$.

Definizione 2.4.7 (Sottoprogramma attivo). *Il sottopro-*

gramma attivo *in un programma M, se esiste, è definito come segue:*

1. *Se M ha una delle forme $\lambda x.M_1 \cdots$, $\mathbf{Y}^\sigma \cdots$, $\mathsf{succ}c$, $\mathsf{pred}c$, $\mathsf{zero?}c$ o $\mathsf{cond}^\beta c \cdots$, allora M è il programma attivo in M.*

2. *Se M ha una delle forme $\mathsf{succ}M_1$, $\mathsf{pred}M_1$, $\mathsf{zero?}M_1$ o $\mathsf{cond}^\beta M_1 \cdots$, in cui M_1 non è una costante, il programma attivo in M, se esiste, è quello in M_1, se esiste.*

Notiamo che se una programma non ha un programma attivo allora esso è una costante o un termine indefinito; inoltre, se un programma termina ed ha un programma attivo, allora anche il programma attivo termina.

Lemma 2.4.8 (Activity Lemma). *Supponiamo che $C[M_1, \ldots, M_m]$ sia un programma terminante con valore c, contenente i termini chiusi $M_1, \ldots, M_m$. Allora o $C[M'_1, \ldots, M'_m]$ termina con valore c per tutti i termini chiusi $M'_1, \ldots, M'_m$ di tipo appropriato oppure esistono un contesto $D[-]$, un intero i, $1 \leq i \leq m$, e degli interi $d_1, \ldots, d_k$ tali che per tutti i termini chiusi $M'_1, \ldots, M'_m$ di tipo appropriato risulta che $C[M'_1, \ldots, M'_m] \gg D[M'_{d_1}, \ldots, M'_{d_k}]$ ed il programma attivo in $D[M'_{d_1}, \ldots, M'_{d_k}]$ esiste ed è il programma attivo in un termine*

della forma $M_i' \cdots$, *oppure ha una delle seguenti forme* $\mathsf{succ}M_i'$, $\mathsf{pred}M_i'$, $\mathsf{zero?}M_i'$ *o* $\mathsf{cond}^\beta M_i' \cdots$.

Dimostriamo che $M_0 \approx M_1$ ragionando per assurdo. Supponiamo che $C[M_0]$ e $C[M_1]$ siano programmi, che $C[M_0]$ termini e che $C[M_1]$, se termina, lo faccia con un valore diverso. Per l'activity lemma applicato a $C[M_0]$ avremo che $C[M_0] \gg M$ e che il programma attivo in M esiste e termina; inoltre, poiché M_0 è di tipo $((o,o,o),o)$, il programma attivo ha la forma $M_0 L$. Dalla definizione di M_0 segue che i programmi $L\,\mathsf{t}\,\Omega^o$, $L\,\Omega^o\,\mathsf{t}$ e $L\,\mathsf{f}\,\mathsf{f}$ terminano rispettivamente con i valori t, t e f. Applicando l'activity lemma a $C'[\mathsf{t},\Omega^o] = L\,\mathsf{t}\,\Omega^o$, poiché Ω^o non termina, o $LM_0'N_0' \Downarrow \mathsf{t}$ per ogni M_0', N_0' oppure per ogni M_0', N_0' risulta che $LM_0'N_0' \gg M$, dove M è un programma il cui programma attivo è il programma attivo in M_0' oppure ha la forma $\mathsf{cond}^\beta M_0' \cdots$ e termina se $LM_0'N_0'$ termina. La prima possibilità è da escludere poiché $L\,\mathsf{f}\,\mathsf{f} \Downarrow \mathsf{f}$, la seconda perché $L\,\Omega^o\,\mathsf{t}$ termina ma Ω^o no. La contraddizione prova che $M_0 \precsim M_1$; analogamente si dimostra che $M_1 \precsim M_0$. □

Vedere [5] per un'altra dimostrazione di questo teorema che utilizza questa volta il teorema di definibilità.

Come la precedente dimostrazione ha chiaramente indicato, il modello standard non è fully abstract per PCF perché è una struttura troppo ricca per il linguaggio. La causa della non corrispondenza è che esistono funzioni, come l'or parallelo (la funzione V della precedente dimostrazione), che possono essere calcolate solo da algoritmi paralleli, mentre PCF è un linguaggio sequenziale.

Ci sono due approcci che possono essere seguiti per ottenere la full abstraction: il primo è quello di mantenere lo stesso modello e di modificare il linguaggio; il secondo è quello di non cambiare il linguaggio ma di restringere il modello. Il primo approccio è detto *espansivo*, il secondo è detto *restrittivo*.

Nell'approccio espansivo il PCF viene esteso con un condizionale parallelo o con l'or parallelo. Sebbene questi costrutti abbiano un comportamento operazionale molto semplice, rendono il linguaggio non sequenziale. Essi aggiungono una potenza espressiva a PCF tale che tutti gli elementi compatti del modello dello spazio delle funzioni continue di Scott siano definibili nel linguaggio esteso. Per il teorema di definibilità, il modello standard è quindi fully abstract per il PCF così esteso.

L'or parallelo, p-or, è una costante del primo ordine, di tipo

$o \Rightarrow o \Rightarrow o$. Per ogni termine B di tipo o,

$$\begin{aligned} \mathsf{p\text{-}or}\,\mathsf{t}\,B &> \mathsf{t} \\ \mathsf{p\text{-}or}\,B\,\mathsf{t} &> \mathsf{t} \\ \mathsf{p\text{-}or}\,\mathsf{f}\,\mathsf{f} &> \mathsf{f} \end{aligned}$$

L'interpretazione standard viene estesa nel seguente modo, per includere l'or parallelo: per $b_1, b_2 \in \mathbb{B}_\perp$,[3]

$$\mathcal{A}[\![\mathsf{p\text{-}or}]\!](\perp)b_1b_2 = \begin{cases} \mathsf{t} & \text{se } b_1 = \mathsf{t} \text{ o } b_2 = \mathsf{t}, \\ \mathsf{f} & \text{se } b_1 = b_2 = \mathsf{f}, \\ \perp & \text{altrimenti.} \end{cases}$$

Il condizionale parallelo, pif, è invece una costante del primo ordine di tipo $o \Rightarrow \beta \Rightarrow \beta \Rightarrow \beta$ le cui regole di riduzione sono le seguenti:

$$\begin{aligned} \mathsf{pif}^\beta BMM &\to M \\ \mathsf{pif}^\beta \mathsf{t}MN &\to M \\ \mathsf{pif}^\beta \mathsf{f}MN &\to N. \end{aligned}$$

Una intepretazione del condizionale parallelo è standard se soddisfa la seguente proprietà: per ogni $b \in \mathbb{B}_\perp$ e per ogni $d_1, d_2 \in$

[3] $\mathbb{B}_\perp$ è il CPO piatto dei booleani.

$\mathbb{N}_\perp$,[4]

$$\mathcal{A}[\![\mathsf{pif}^\iota]\!](\perp)bd_1d_2 \;=\; \begin{cases} d_1 & \text{se } b = \mathsf{t}, \\ d_2 & \text{se } b = \mathsf{f}, \\ d_1 & \text{se } b = \perp \text{ e } d_1 = d_2, \\ \perp & \text{altrimenti.} \end{cases}$$

L'or parallelo può essere simulato dal condizionale parallelo con il termine

$$\lambda x^o.\lambda y^o.\mathsf{pif}^o x\mathsf{t}y.$$

Infatti, l'or parallelo e il condizionale parallelo sono interdefinibili.

Teorema 2.4.9. *Il modello continuo standard è fully abstract per* PCF *esteso con l'or parallelo o con il condizionale parallelo.*

Dato che il modello standard non caratterizza PCF, esiste un modello che lo fa? Più precisamente esiste un modello di ordine estensionale, continuo e fully abstract per PCF? La risposta a questa domanda è stata data da Milner [6], che ha costruito un tale modello e ha fatto vedere che è unico a meno di isomorfismo.

Teorema 2.4.10. *Esiste un unico (a meno di isomorfismo) mo-*

[4] $\mathbb{N}_\perp$ è il CPO piatto dei naturali.

dello continuo, di ordine estensionale inequazionalmente fully abstract.

Quello costruito da Milner è però essenzialmente un term model e non aggiunge alla nostra comprensione di PCF più di quanto abbiamo acquisito dalla sintassi del linguaggio. Un modo per riformulare il problema della full abstraction per PCF è quindi il seguente:

> **Problema della full abstraction per PCF** *Trovare una descrizione astratta e sintetica dell'unico modello continuo, di ordine estensionale e inequazionalmente fully abstract di* PCF *come identificato da Milner.*

Capitolo 3

Un modello per PCF

3.1 Elementi parziali ereditariamente consistenti

Sia $\mathbb{N}$ l'insieme dei numeri naturali. Definiamo induttivamente i tipi nel seguente modo:

1. $\mathbb{N}$ è un tipo;

2. se α e β sono tipi, lo è anche $\alpha \to \beta$.

In particolare, denotiamo con 0 il tipo $\mathbb{N}$ e, per ogni $n \geq 0$, denotiamo con $n+1$ il tipo $n \to 0$.

Diamo ora delle definizioni per induzione sul tipo. Per il tipo 0 definiamo $\mathrm{HC}_0 = \mathbb{N} \cup \{\bot^0\}$ ($\bot^0$ è chiamato *bottom* o elemento

vuoto del tipo 0). Per ogni $a, b \in \mathrm{HC}_0$, definiamo le seguenti operazioni e relazioni:

i. $a \subseteq_0 b \overset{def}{\iff} a = b \vee a = \perp^0$ (Inclusione)

ii. $a \uparrow_0 b \overset{def}{\iff} a = b \vee a = \perp^0 \vee b = \perp^0$ (Compatibilità)

iii. $a \cup_0 b \overset{def}{=} \begin{cases} a & \text{se } a = b \vee b = \perp^0 \\ b & \text{se } a = b \vee a = \perp^0 \\ \perp^0 & \text{altrimenti} \end{cases}$ (Unione)

iv. $a \downarrow_0 b \overset{def}{\iff} a = b$ (Compatibilità verso il basso)

v. $a \cap_0 b \overset{def}{=} \begin{cases} a & \text{se } a = b \\ \perp^0 & \text{altrimenti} \end{cases}$ (Intersezione)

Per il tipo $\sigma = \alpha \to \beta$ definiamo HC_σ come l'insieme delle funzioni parziali continue e monotone da HC_α a $\mathrm{HC}_\beta - \{\perp^\beta\}$. La funzione $A \in \mathrm{HC}_\sigma$ che a $B_1, \ldots, B_n \in \mathrm{HC}_\alpha$ associa rispettivamente $A(B_1), \ldots, A(B_n) \in \mathrm{HC}_\beta$ sarà denotata con

$$\begin{pmatrix} A(B_1) & \ldots & A(B_n) \\ B_1 & \ldots & B_n \end{pmatrix}$$

La funzione ovunque indefinita, l'elemento vuoto del tipo σ, sarà denotata con $\perp^\sigma$. Per ogni $A \in \mathrm{HC}_\sigma$, $Dom(A)$ indica il dominio di A. Per ogni $A_1, A_2 \in \mathrm{HC}_\sigma$, definiamo le seguenti operazioni e relazioni:

i. (Inclusione)

$$A_1 \subseteq_\sigma A_2 \overset{def}{\iff} \forall B_1 \in Dom(A_1)\ \exists B_2 \in Dom(A_2).\ B_2 \subseteq_\alpha B_1 \wedge A(B_1) \subseteq_\beta A(B_2)$$

ii. (Compatibilità)

$$A_1 \uparrow_\sigma A_2 \overset{def}{\iff} \forall B_1 \in Dom(A_1)\ \forall B_2 \in Dom(A_2):\ B_1 \uparrow_\alpha B_2 \Rightarrow A_1(B_1) \uparrow_\beta A_2(B_2)$$

iii. (Unione)

$$A_1 \cup_\sigma A_2 \overset{def}{=} \begin{cases} \text{l'unione delle relazioni di } A_1 \text{ e } A_2 & \text{se } A_1 \uparrow A_2 \\ A_2 & \text{se } A_1 \subseteq_\sigma A_2 \\ \bot^\sigma & \text{altrimenti} \end{cases}$$

iv. (Compatibilità verso il basso)

$$A_1 \downarrow_\sigma A_2 \overset{def}{\iff} \exists B_1 \in Dom(A_1)\ \exists B_2 \in Dom(A_2).\ B_1 \uparrow_\alpha B_2 \wedge A_1(B_1) \downarrow_\beta A_2(B_2)$$

v. (Intersezione)

$$A_1 \cap_\sigma A_2 \overset{def}{=} \left\{ \begin{pmatrix} A_1(B_1) \cap_\beta A_2(B_2) \\ B_1 \cup_\alpha B_2 \end{pmatrix} \;\middle|\; B_1 \in Dom(A_1),\ B_2 \in Dom(A_2),\ B_1 \uparrow_\alpha B_2\ \&\ A_1(B_1) \downarrow_\beta A_2(B_2) \right\}$$

vi. (Differenza)

$$A_1 -_\sigma A_2 \stackrel{def}{=} \begin{cases} A_1 & \text{se } A_2 = \perp^\sigma \\ \text{la differenza tra le relazioni di } A_1 \text{ e } A_2 & \text{se } A_2 \subseteq_\sigma A_1 \\ \perp^\sigma & \text{altrimenti} \end{cases}$$

Quando sarà chiaro dal contesto, ometteremo l'indicazione del tipo dalle operazioni. Dati $A_1, A_2 \in \mathrm{HC}_\sigma$, dire che $A_1 \uparrow A_2$ significa che ha senso fare la loro unione. In tal caso $A_1 \cup A_2$ è il minimo estremo superiore tra A_1 e A_2, secondo la relazione d'ordine $\subseteq$. $A_1 \subseteq A_2$ indica che A_1 è meno definito o uguale a A_2. $\langle \mathrm{HC}_\sigma, \subseteq_\sigma \rangle$ è un CPO, il cui elemento minimo è $\perp_\sigma$. $A_1 \cap A_2$ rappresenta invece il massimo estremo inferiore tra A_1 e A_2, e la compatibilità verso il basso indica quando l'intersezione è diversa da $\perp$.

Useremo i simboli $\not\subseteq$, $\not\uparrow$ e $\not\downarrow$ per indicare rispettivamente i complementi delle relazioni $\subseteq$, $\uparrow$ e $\downarrow$. Useremo, se non indicato espressamente in modo diverso, le lettere minuscole (a, b, ...), con o senza indici, per denotare elementi del tipo 0, mentre useremo lettere greche minuscole (α, β, ...) o lettere maiuscole (A, B, ...), con o senza indici, per gli elementi del tipo 1, lettere maiuscole (F, G, ...), con o senza indici, per gli elementi del tipo 2 e lettere greche maiuscole (Δ, Γ, ...), con o senza indici, per gli

elementi del tipo 3. Spesso useremo lo stesso nome per denotare un funzionale e il programma che lo definisce.

Esempio 1 (Inclusione).

a. Siano $A_1, A_2 \in \mathrm{HC}_1$, $A_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $A_2 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$.
$Dom(A_1) = \{0\}$, $Dom(A_2) = \{0, 1\}$. Per $B_1 = 0$ esiste $B_2 = 0$ tale che $B_2 \subseteq B_1$ e $A_1(B_1) \subseteq A_2(B_2)$. Quindi $A_1 \subseteq A_2$.

b. Siano $A_1, A_2 \in \mathrm{HC}_1$, $A_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$, $A_2 = \begin{pmatrix} 0 \\ \bot \end{pmatrix}$.
$Dom(A_1) = \{0, 1\}$, $Dom(A_2) = \mathrm{HC}_0$. Per $B_1 = 0$ esiste $B_2 = \bot$ tale che $B_2 \subseteq B_1$ e $A_1(B_1) \subseteq A_2(B_2)$. Per $B_1 = 1$ esiste $B_2 = \bot$ tale che $B_2 \subseteq B_1$ e $A_1(B_1) \subseteq A_2(B_2)$. Quindi $A_1 \subseteq A_2$.

c. Siano $A_1, A_2 \in \mathrm{HC}_1$, $A_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $A_2 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$.
$Dom(A_1) = \{0\}$, $Dom(A_2) = \{0, 1\}$. Per $B_1 = 0$ non esiste un $B_2 \in Dom(A_2)$ tale che $B_2 \subseteq B_1$ e $A_1(B_1) \subseteq A_2(B_2)$. Infatti per $B_2 = 0$ abbiamo che $A_1(B_1) \not\subseteq A_2(B_2)$, mentre per $B_2 = 1$ abbiamo che $B_2 \not\subseteq B_1$. Quindi $A_1 \not\subseteq A_2$.

d. Siano $A_1, A_2 \in \mathrm{HC}_2$, $A_1 = \begin{pmatrix} 1 \\ \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \end{pmatrix}$, $A_2 = \begin{pmatrix} 1 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{pmatrix}$.

$$Dom(A_1) = \left\{ B \in \mathrm{HC}_1 \;\middle|\; \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \subseteq B \right\},$$

$$Dom(A_2) = \left\{ B \in \mathrm{HC}_1 \;\middle|\; \begin{pmatrix} 0 \\ 0 \end{pmatrix} \subseteq B \right\}.$$

Per ogni $B_1 \in Dom(A_1)$ esiste $B_2 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ tale che $B_2 \subseteq B_1$ e $A_1(B_1) \subseteq A_2(B_2)$. Quindi $A_1 \subseteq A_2$.

e. Siano $A_1, A_2 \in \mathrm{HC}_2$, $A_1 = \left\langle \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \right\rangle$, $A_2 = \left\langle \begin{pmatrix} 0 \\ \bot \end{pmatrix} \right\rangle$.

$$Dom(A_1) = \left\{ B \in \mathrm{HC}_1 \;\middle|\; \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \subseteq B \right\},$$

$Dom(A_2) = \left\{ \begin{pmatrix} 0 \\ \bot \end{pmatrix} \right\}$. $\begin{pmatrix} 0 \\ \bot \end{pmatrix} \not\subseteq \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ perché $0 \not\subseteq \bot$ e $1 \not\subseteq \bot$. Quindi $A_1 \not\subseteq A_2$.

Esempio 2 (Compatibilità).

a. Siano $A_1, A_2 \in \mathrm{HC}_1$, $A_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$, $A_2 = \begin{pmatrix} 0 \\ \bot \end{pmatrix}$.
$Dom(A_1) = \{0, 1\}$, $Dom(A_2) = \mathrm{HC}_0$. Per $B_1 = 0$ e $B_2 = \bot$ abbiamo che $B_1 \uparrow B_2$ e $A_1(B_1) \uparrow A_2(B_2)$. Per $B_1 = 1$ e $B_2 = \bot$ abbiamo che $B_1 \uparrow B_2$ e $A_1(B_1) \uparrow A_2(B_2)$. Lo stesso vale per $B_1 = 0$ e $B_2 = 0$ e per $B_1 = 1$ e $B_2 = 1$. Quindi $A_1 \uparrow A_2$.

b. Siano $A_1, A_2 \in \mathrm{HC}_1$, $A_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $A_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. $Dom(A_1) = \{0\}$, $Dom(A_2) = \{1\}$. $0 \not\uparrow 1$, quindi $A_1 \not\uparrow A_2$.

c. Siano $A_1, A_2 \in \mathrm{HC}_2$, $A_1 = \begin{pmatrix} 1 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{pmatrix}$, $A_2 = \begin{pmatrix} 1 \\ \begin{pmatrix} 0 \\ \bot \end{pmatrix} \end{pmatrix}$. $A_1 \uparrow A_2$, poiché $0 \uparrow \bot$, $0 \uparrow 0$ e $1 \uparrow 1$.

Esempio 3 (Incompatibilità).

a. Siano $A_1, A_2 \in \mathrm{HC}_1$, $A_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$, $A_2 = \begin{pmatrix} 1 \\ \bot \end{pmatrix}$. $Dom(A_1) = \{0, 1\}$, $Dom(A_2) = \mathrm{HC}_0$. Per $B_1 = 0$ e $B_2 = \bot$ abbiamo che $B_1 \uparrow B_2$ ma $A_1(B_1) \not\downarrow A_2(B_2)$. Quindi $A_1 \not\downarrow A_2$.

b. Siano $A_1, A_2 \in \mathrm{HC}_2$, $A_1 = \begin{pmatrix} 1 \\ \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \end{pmatrix}$, $A_2 = \begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{pmatrix}$. $A_1 \not\downarrow A_2$, poiché $\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \uparrow \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ ma $1 \not\downarrow 0$.

Esempio 4 (Unione).

a. Siano $A_1, A_2 \in \mathrm{HC}_1$, $A_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $A_2 = \begin{pmatrix} 0 & 1 & 2 \\ 0 & 1 & 2 \end{pmatrix}$. $A_1 \cup A_2 = A_2$, poiché $A_1 \subseteq A_2$.

b. Siano $A_1, A_2 \in \mathrm{HC}_1$, $A_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, $A_2 = \begin{pmatrix} 2 \\ 1 \end{pmatrix}$. $A_1 \cup A_2 = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$, poiché $A_1 \uparrow A_2$.

c. Siano $A_1, A_2 \in \mathrm{HC}_1$, $A_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, $A_2 = \begin{pmatrix} 1 \\ \bot \end{pmatrix}$. $A_1 \cup A_2 = A_2$, poiché $A_1 \subseteq A_2$.

d. Siano $A_1, A_2 \in \mathrm{HC}_1$, $A_1 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$, $A_2 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$. $A_1 \cup A_2 = \perp$, poiché $A_1 \not\upharpoonright A_2$.

e. Siano $A_1, A_2 \in \mathrm{HC}_2$, $A_1 = \left(\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \right)$, $A_2 = \left(\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \right)$. $A_1 \cup A_2 = \left(\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \right)$, poiché $A_1 \uparrow A_2$.

f. Siano $A_1, A_2 \in \mathrm{HC}_2$, $A_1 = \left(\begin{pmatrix} 0 \\ \perp \end{pmatrix} \right)$, $A_2 = \left(\begin{pmatrix} 0 \\ 1 \end{pmatrix} \right)$. $A_1 \cup A_2 = A_2$, poiché $A_1 \subseteq A_2$.

g. Siano $A_1, A_2 \in \mathrm{HC}_2$, $A_1 = \left(\begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \right)$, $A_2 = \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix} \right)$. $A_1 \cup A_2 = \perp$, poiché $A_1 \not\upharpoonright A_2$.

Esempio 5 (Compatibilità verso il basso). Siano $A_1, A_2 \in \mathrm{HC}_1$, $A_1 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$, $A_2 = \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}$. $Dom(A_1) = \{0, 1\}$, $Dom(A_2) = \{0, 1\}$. Esistono $B_1 \in Dom(A_1)$ e $B_2 \in Dom(A_2)$, $B_1 = 0$ e $B_2 = 0$, tali che $B_1 \uparrow B_2$ e $A_1(B_1) \downarrow A_2(B_2)$. Quindi $A_1 \downarrow A_2$.

Esempio 6 (Incompatibilità verso il basso). Siano $A_1, A_2 \in \mathrm{HC}_1$, $A_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$, $A_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. $Dom(A_1) = \{0\}$,

$Dom(A_2) = \{0,1\}$. Per $B_1 = 0$ e $B_2 = 0$ abbiamo che $A_1(B_1) \not\downarrow A_2(B_2)$, mentre per $B_1 = 0$ e $B_2 = 1$ abbiamo che $B_1 \not\uparrow B_2$. Quindi $A_1 \not\downarrow A_2$.

Esempio 7 (Intersezione). Siano $A_1, A_2 \in \mathrm{HC}_1$, $A_1 = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}$, $A_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. $A_1 \cap A_2 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$, poiché esistono $B_1 \in Dom(A_1)$ e $B_2 \in Dom(A_2)$, $B_1 = 0$ e $B_2 = 0$, tali che $B_1 \uparrow B_2$ e $A_1(B_1) \downarrow A_2(B_2)$.

Un elemento A di tipo n, con $n > 0$, è *finito* se è formato da un insieme finito di elementi, la cui unione è uguale ad A. Al tipo 0 tutti gli elementi sono finiti. Un elemento finito $A = \left\{ \begin{pmatrix} a_i \\ B_i \end{pmatrix} \;\middle|\; 1 \leq i \leq n \right\}$ di tipo n $(n > 0)$ rappresenta una funzione finita così definita:

$$A(B) \cong a \quad \text{se esiste un } i,\ 1 \leq i \leq n, \text{ tale che } B_i \subseteq B \text{ e } a_i = a$$

Se si considerano solo elementi finiti, $\cup$ e $\cap$ sono funzioni ricorsive, e $\subseteq$, $\uparrow$ e $\downarrow$ sono relazioni ricorsive.

Definizione 3.1.1 (Elemento non ridondante). *Ogni elemento di tipo* 0 *è* non ridondante. *Un elemento finito* $A = \left\{ \begin{pmatrix} a_i \\ B_i \end{pmatrix} \;\middle|\; 1 \leq i \leq n \right\}$ *di tipo* $n+1$ *è* non ridondante *se, per* $1 \leq i, j \leq n$, *si ha che*

$$B_i \subseteq B_j \Longrightarrow i = j$$

e, per ogni i, B_i è un elemento finito non ridondante di tipo n.

Esempio 8. L'elemento $\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$ è non ridondante. L'elemento $\begin{pmatrix} 0 & 0 \\ 0 & \bot \end{pmatrix}$ è ridondante.

Ogni elemento A ha un'unica rappresentazione non ridondante, che denotiamo con $Irr(A)$, che si ottiene eliminando da A tutti gli elementi $\begin{pmatrix} a_i \\ B_i \end{pmatrix}$ per i quali esiste un elemento $\begin{pmatrix} a_j \\ B_j \end{pmatrix} \in A$, con $i \neq j$, tale che $B_j \subseteq B_i$. Si noti che $Irr(A)$ rappresenta la stessa funzione rappresentata da A.

Esempio 9. La rappresentazione non ridondante dell'elemento $A = \begin{pmatrix} 0 & 0 \\ 0 & \bot \end{pmatrix}$ è $Irr(A) = \begin{pmatrix} 0 \\ \bot \end{pmatrix}$.

Definizione 3.1.2 (Elemento chiuso rispetto all'input). *Un elemento $A = \left\{ \begin{pmatrix} a_i \\ B_i \end{pmatrix} \;\middle|\; 1 \leq i \leq n \right\}$ di tipo $n+1$ è* chiuso rispetto all'input *se per ogni i, $1 \leq i \leq n$, e per ogni $B \in HC_n$ si ha che*

$$B_i \subseteq B \Longrightarrow \begin{pmatrix} a_i \\ B \end{pmatrix} \in A.$$

Esempio 10. L'elemento $\begin{pmatrix} 1 \\ 0 \end{pmatrix}$ è chiuso rispetto all'input. L'elemento $\begin{pmatrix} 1 \\ \bot \end{pmatrix}$ non è chiuso rispetto all'input.

Ogni elemento A ha un'unica rappresentazione chiusa rispetto all'input, che denotiamo con $Closed(A)$, che si ottiene aggiungendo ad A tutti gli elementi $\begin{pmatrix} a_i \\ B_i \end{pmatrix}$ per i quali esiste un elemento $\begin{pmatrix} a_j \\ B_j \end{pmatrix} \in A$ tale che $B_j \subseteq B_i$ e $a_i = a_j$. Si noti che $Closed(A)$ rappresenta la stessa funzione rappresentata da A.

Esempio 11. La rappresentazione chiusa rispetto all'input dell'elemento $A = \begin{pmatrix} 1 \\ \bot \end{pmatrix}$ è $Closed(A) = \begin{pmatrix} 1 & 1 & 1 & \cdots \\ \bot & 0 & 1 & \cdots \end{pmatrix}$.

3.2 Elementi sequenziali

Definizione 3.2.1 (Eliminatore).
Siano $F = \left\{ \begin{pmatrix} a_i \\ B_i \end{pmatrix} \;\middle|\; 1 \leq i \leq n \right\}$ *un elemento di tipo* 2 *e* $S = \left(\begin{pmatrix} b_i \\ c_i \end{pmatrix} \right)_{1 \leq i \leq n}$ *una famiglia di elementi di tipo* 1 *tale che per ogni* i, $1 \leq i \leq n$, *si ha che* $\begin{pmatrix} b_i \\ c_i \end{pmatrix} \in B_i$. *Per ogni* $a \in \mathbb{N}$ *definiamo l'eliminatore di* S *da* F *rispetto ad* a *nel seguente modo:*

$$Elim_a(F, S) \stackrel{def}{=} \left\{ \begin{pmatrix} a_i \\ B_i - \begin{pmatrix} b_i \\ c_i \end{pmatrix} \end{pmatrix} \;\middle|\; 1 \leq i \leq n \wedge b_i \uparrow a \right\}$$

Esempio 12. Siano

$$F = \left\{ \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \end{pmatrix} , \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \end{pmatrix} , \left(\begin{pmatrix} 1 \\ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} \right\}$$

e $S = (S_i)_{1 \leq i \leq 3}$ tale che

$$S_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \; S_2 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \text{ e } S_3 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Allora

$$\begin{aligned} Elim_0(F, S) &= \left\{ \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} , \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 1 \\ 1 \end{pmatrix} \end{pmatrix} \right\} \\ Elim_1(F, S) &= \left\{ \begin{pmatrix} 1 \\ \perp^1 \end{pmatrix} \right\} \end{aligned}$$

Proposizione 3.2.2. *Tutti gli elementi finiti di tipo* 0 *e* 1 *sono sequenziali.*

Proposizione 3.2.3. *Sia A un elemento finito di tipo* 2*. A è sequenziale se e solo se $A = \perp^2$ oppure, dato*

$$Irr(A) = \left\{ \begin{pmatrix} a_i \\ B_i \end{pmatrix} \;\middle|\; 1 \leq i \leq n \right\},$$

è verificata una delle seguenti due condizioni:

1. *$Irr(A)$ è un funzionale costante (cioè del tipo $\begin{pmatrix} a \\ \perp^1 \end{pmatrix}$);*
2. *Esiste $c \in \prod_{1 \leq i \leq n} Dom(B_i)$, $c = (c_1, \ldots, c_n)$, tale che*

(a) $c_1 \uparrow \cdots \uparrow c_n$*;*

(b) dato $S_c = \left(\begin{pmatrix} B_i(c_i) \\ c_i \end{pmatrix} \right)_{1 \leq i \leq n}$ *si ha che, per ogni* $1 \leq i \leq n$*,* $Elim_{B_i(c_i)}(Irr(A), S_c)$ *è sequenziale.*

Esempio 13 (Elemento sequenziale). Sia

$$F = \left\{ \begin{pmatrix} 0 \\ \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \end{pmatrix}, \begin{pmatrix} 0 \\ \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \end{pmatrix}, \begin{pmatrix} 1 \\ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} \right\}$$

Verifichiamo se F è sequenziale. Per $c = (0, 0, 0)$ abbiamo che $S_c = (S_{ci})_{1 \leq i \leq 3}$ è tale che

$$S_{c1} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \; S_{c2} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \text{ e } S_{c3} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

Allora

$$\begin{aligned} Elim_0(F, S_c) &= \left\{ \begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix}, \begin{pmatrix} 0 \\ \begin{pmatrix} 1 \\ 1 \end{pmatrix} \end{pmatrix} \right\} \\ Elim_1(F, S_c) &= \left\{ \begin{pmatrix} 1 \\ \perp^1 \end{pmatrix} \right\} \end{aligned}$$

$Elim_1(F, S_c)$ è sequenziale. Verifichiamo se lo è anche $Elim_0(F, S_c)$. Per $c' = (1, 1)$ abbiamo che $S_{c'} = (S_{c'i})_{i=1,2}$ è tale che

$$S_{c'1} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \text{ e } S_{c'2} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Allora

$$\begin{aligned} Elim_0(Elim_0(F, S_c), S_{c'}) &= \\ Elim_1(Elim_0(F, S_c), S_{c'}) &= \left\{ \begin{pmatrix} 0 \\ \perp^1 \end{pmatrix} \right\} \end{aligned}$$

che è sequenziale. Quindi F è sequenziale.

Esempio 14 (Elemento sequenziale). Sia

$$F = \left\{ \begin{pmatrix} 1 \\ \begin{pmatrix} 1 & 2 & 3 \\ 0 & 1 & 2 \end{pmatrix} \end{pmatrix}, \begin{pmatrix} 0 \\ \begin{pmatrix} 0 & 2 \\ 0 & 1 \end{pmatrix} \end{pmatrix}, \begin{pmatrix} 2 \\ \begin{pmatrix} 0 & 2 \\ 1 & 2 \end{pmatrix} \end{pmatrix} \right\}$$

Verifichiamo se F è sequenziale. Per $c = (1, 1, 1)$ abbiamo che $S_c = (S_{ci})_{1 \leq i \leq 3}$ è tale che

$$S_{c1} = \begin{pmatrix} 2 \\ 1 \end{pmatrix}, \; S_{c2} = \begin{pmatrix} 2 \\ 1 \end{pmatrix} \text{ e } S_{c3} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

Allora

$$\begin{aligned} Elim_0(F, S_c) &= \left\{ \begin{pmatrix} 2 \\ \begin{pmatrix} 2 \\ 2 \end{pmatrix} \end{pmatrix} \right\} \\ Elim_2(F, S_c) &= \left\{ \begin{pmatrix} 1 \\ \begin{pmatrix} 1 & 3 \\ 0 & 2 \end{pmatrix} \end{pmatrix}, \begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{pmatrix} \right\} \end{aligned}$$

$Elim_0(F, S_c)$ è sequenziale poiché, per $c' = 2$ e $S_{c'} = (S_{c'1})$ tale che $S_{c'1} = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$, $Elim_2(Elim_0(F, S_c), S_{c'}) = \begin{pmatrix} 2 \\ \perp^1 \end{pmatrix}$, che

è sequenziale. Verifichiamo se lo è anche $Elim_2(F, S_c)$. Per $c'' = (0,0)$ abbiamo che $S_{c''} = (S_{c''i})_{i=1,2}$ è tale che

$$S_{c''1} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ e } S_{c''2} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

Allora

$$\begin{aligned} Elim_0(Elim_2(F, S_c), S_{c''}) &= \begin{pmatrix} 0 \\ \perp^1 \end{pmatrix} \\ Elim_1(Elim_2(F, S_c), S_{c''}) &= \left\{ \begin{pmatrix} 1 \\ \begin{pmatrix} 3 \\ 2 \end{pmatrix} \end{pmatrix} \right\} \end{aligned}$$

che sono entrambi sequenziali. Quindi F è sequenziale.

Esempio 15 (Elemento non sequenziale). Sia

$$F = \left\{ \begin{pmatrix} 1 \\ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix}, \begin{pmatrix} 1 \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix}, \begin{pmatrix} 1 \\ \perp^1 \end{pmatrix} \right\}.$$

F non è sequenziale perché $0 \not\gamma 1$.

Esempio 16 (Elemento non sequenziale). Sia

$$F = \left\{ \begin{pmatrix} 0 \\ \begin{pmatrix} 0 & 0 \\ 0 & 2 \end{pmatrix} \end{pmatrix}, \begin{pmatrix} 0 \\ \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \end{pmatrix}, \begin{pmatrix} 2 \\ \begin{pmatrix} 2 \\ 0 \end{pmatrix} \end{pmatrix} \right\}$$

Verifichiamo se F è sequenziale. Per $c = (0,0,0)$ abbiamo che $S_c = (S_{ci})_{1 \leq i \leq 3}$ è tale che

$$S_{c1} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, S_{c2} = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \text{ e } S_{c3} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}.$$

Allora

$$Elim_0(F, S_c) = \left\{ \begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 2 \end{pmatrix} \end{pmatrix}, \begin{pmatrix} 0 \\ \begin{pmatrix} 1 \\ 1 \end{pmatrix} \end{pmatrix} \right\}$$

$$Elim_2(F, S_c) = \left\{ \begin{pmatrix} 2 \\ \perp^1 \end{pmatrix} \right\}$$

$Elim_2(F, S_c)$ è sequenziale ma $Elim_0(F, S_c)$ non lo è perché $2 \not\gamma 1$. Quindi F non è sequenziale.

3.3 Elementi definibili

Un elemento A è definibile in PCF se esiste un programma PCF la cui semantica è A.

Proposizione 3.3.1. *Tutti gli elementi sequenziali dei tipi* 0, 1 *e* 2 *sono definibili in* PCF.

Dimostrazione. Per il tipo 0 è banale.

Tipo 1. Sia A un elemento sequenziale di tipo 1. Se $A = \perp^1$, allora è definito dal programma $\lambda x.\Omega$. Altrimenti si consideri $Irr(A)$, il quale può assumere una delle seguenti forme:[1]

1. $\begin{pmatrix} a \\ \perp \end{pmatrix}$, l'elemento costante, che è definito dal programma

$$\lambda x.a$$

[1]Si ricordi che $Irr(A)$ e A rappresentano la stessa funzione.

2. $\begin{pmatrix} a \\ b \end{pmatrix}$, con $b \neq \perp$, definito dal programma

$$\lambda x.\mathsf{if}\ x = b\ \mathsf{then}\ a;$$

3. $\begin{pmatrix} a_1 & a_2 & \cdots & a_m \\ b_1 & b_2 & \cdots & b_m \end{pmatrix}$, con $b_i \neq \perp$ $(1 \leq i \leq m)$, definito dal programma

$$\begin{aligned} &\lambda x.\mathsf{if}\ x = b_1\ \mathsf{then}\ a_1 \\ &\quad \mathsf{else\ if}\ x = b_2\ \mathsf{then}\ a_2 \\ &\quad \vdots \\ &\quad \mathsf{else\ if}\ x = b_m\ \mathsf{then}\ a_m \end{aligned}$$

Tipo 2. Sia F un elemento sequenziale di tipo 2. Se $F = \perp^2$, il programma che lo definisce è $\lambda\alpha.\Omega$. Altrimenti consideriamo $Irr(F) = \left\{ \begin{pmatrix} a_i \\ B_i \end{pmatrix} \;\middle|\; 1 \leq i \leq n \right\}$. Se $Irr(F)$ è un funzionale costante, $\begin{pmatrix} a \\ \perp^1 \end{pmatrix}$, allora esso è definito dal programma $\lambda\alpha.a$. Altrimenti consideriamo l'elemento $c \in \prod_{1 \leq i \leq n} Dom(B_i)$, $c = (c_1, \ldots, c_n)$, tale che $c_1 \uparrow \cdots \uparrow c_n$, esistente in base alla proposizione 3.2.3, e poniamo $x = c_1 \cup \cdots \cup c_n$. Definiamo $S_c = \left(\begin{pmatrix} B_i(c_i) \\ c_i \end{pmatrix} \right)_{1 \leq i \leq n}$, come nella proposizione 3.2.3, e $T_c = \{B_i(c_i) \mid 1 \leq i \leq n\}$. Siano $b_1, \ldots, b_m$ gli elementi di T_c e, per ogni i, $1 \leq i \leq m$, sia P_{x,b_i} il programma che definisce $Elim_{b_i}(F, S_c)$. Il programma che definisce F sarà allora il

seguente:

$$\begin{aligned}
&\lambda\alpha.\textsf{if } \alpha(x) = b_1 \textsf{ then } P_{x,b_1}(\alpha)\\
&\quad \textsf{else if } \alpha(x) = b_2 \textsf{ then } P_{x,b_2}(\alpha)\\
&\quad \vdots\\
&\quad \textsf{else if } F\alpha(x) = b_m \textsf{ then } P_{x,b_m}(\alpha)
\end{aligned}$$

□

Esempio 17. Sia

$$F = \left\{ \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \end{pmatrix} , \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} \end{pmatrix} , \left(\begin{pmatrix} 1 \\ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} \right\} .$$

Nell'esempio 13 è stato visto che F è sequenziale. Il programma che lo definisce è il seguente:

$$\begin{aligned}
&\lambda\alpha.\textsf{if } \alpha(0) = 0 \textsf{ then if } \alpha(1) = 0 \textsf{ then } 0\\
&\qquad\qquad\qquad\qquad \textsf{else if } \alpha(1) = 1 \textsf{ then } 0\\
&\quad \textsf{else if } \alpha(0) = 1 \textsf{ then } 1
\end{aligned}$$

Esempio 18 (Or parallelo). Sia $F = \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{pmatrix} \begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} \right)$. F non è sequenziale in base alla proposizione 3.2.3, in quanto $0 \not\gamma 1$. Un programma che lo definisce sarebbe

$$P_F = \lambda\alpha.\textsf{if } \alpha(0) = 0 \vee \alpha(1) = 0 \textsf{ then } 0,$$

ma l'or parallelo ($\vee$) non è definibile in PCF. Se si sceglie di calcolare prima $\alpha(0)$ si ha che, se $\alpha(0) \uparrow$ allora $P_F(\alpha) \uparrow$, anche se $\alpha(1) = 0$. Lo stesso accade se si calcola prima $\alpha(1)$.

Esempio 19. Sia

$$F = \left\{ \left(\begin{pmatrix} & 1 & \\ 1 & 2 & 3 \\ 0 & 1 & 2 \end{pmatrix} \right), \left(\begin{pmatrix} & 0 & \\ 0 & & 2 \\ 0 & & 1 \end{pmatrix} \right), \left(\begin{pmatrix} & 2 & \\ 0 & & 2 \\ 1 & & 2 \end{pmatrix} \right) \right\}.$$

Nell'esempio 14 è stato visto che F è sequenziale. Il programma che lo definisce è il seguente:

$\lambda\alpha$.if $\alpha(1) = 0$ then if $\alpha(2) = 2$ then 2
else if $\alpha(1) = 2$ then if $\alpha(0) = 1$ then if $\alpha(2) = 3$ then 1
else if $\alpha(0) = 0$ then 0

3.4 Tipo 3

Proviamo a definire per il tipo 3 un procedimento per determinare se un dato elemento è definibile o meno, analogamente a quanto fatto per il tipo 2.

Definizione 3.4.1 (Eliminatore).
Siano $\Delta = \left\{ \begin{pmatrix} a_i \\ F_i \end{pmatrix} \;\middle|\; 1 \leq i \leq n \right\}$ *un elemento di tipo* 3 *e* $S = \left(\begin{pmatrix} b_i \\ C_i \end{pmatrix} \right)_{1 \leq i \leq n}$ *una famiglia di elementi di tipo* 2 *tale che per*

ogni i, $1 \leq i \leq n$, *si ha che* $\begin{pmatrix} b_i \\ C_i \end{pmatrix} \in F_i$. *Per ogni* $a \in \mathbb{N}$ *definiamo l'eliminatore di* S *da* Δ *rispetto ad* a *nel seguente modo:*

$$Elim_a(\Delta, S) \stackrel{def}{=} \left\{ \left(\begin{array}{c} a_i \\ F_i - \begin{pmatrix} b_i \\ C_i \end{pmatrix} \end{array} \right) \;\middle|\; 1 \leq i \leq n \wedge b_i \uparrow a \right\}$$

Esempio 20. Siano

$$\Delta = \left\{ \left(\begin{array}{c} 0 \\ \begin{pmatrix} 0 & 0 \\ \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{pmatrix} \end{array} \right), \left(\begin{array}{c} 1 \\ \begin{pmatrix} 1 & 1 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{pmatrix} \end{array} \right) \right\}$$

e $S = (S_i)_{1 \leq i \leq 2}$ tale che

$$S_1 = \begin{pmatrix} 0 \\ \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \end{pmatrix} \text{ e } S_2 = \begin{pmatrix} 1 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{pmatrix}.$$

Allora

$$Elim_0(\Delta, S) = \left\{ \left(\begin{array}{c} 0 \\ \begin{pmatrix} 0 \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{pmatrix} \end{array} \right) \right\}$$

$$Elim_1(\Delta, S) = \left\{ \left(\begin{array}{c} 1 \\ \begin{pmatrix} 1 \\ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{pmatrix} \end{array} \right) \right\}$$

Sia Δ un elemento di tipo 3. Se $\Delta = \perp^3$, allora è definito dal programma $\lambda F.\Omega$. Altrimenti consideriamo $Irr(\Delta) = \left\{ \begin{pmatrix} a_i \\ F_i \end{pmatrix} \;\middle|\; 1 \leq i \leq n \right\}$. Se $Irr(\Delta)$ è un funzionale costante, $\begin{pmatrix} a \\ \perp^2 \end{pmatrix}$, allora esso è definito dal programma $\lambda F.a$. Se invece $Irr(\Delta)$ non è un funzionale costante, ogni F_i è sequenziale, ed esiste $C \in \prod_{1 \leq i \leq n} Dom(F_i)$, $C = (C_1, \ldots, C_n)$, tale che $C_1 \uparrow \cdots \uparrow C_n$, seguendo il seguente procedimento si può ottenere un programma che definisce $Irr(\Delta)$ (e Δ). Definiamo $\alpha = C_1 \cup \cdots \cup C_n$ (denotiamo con α sia l'elemento sia il programma che lo definisce), $S_C = \left(\begin{pmatrix} F_i(C_i) \\ C_i \end{pmatrix} \right)_{1 \leq i \leq n}$ e $T_C = \{F_i(C_i) \mid 1 \leq i \leq n\}$. Siano $b_1, \ldots, b_m$ gli elementi di T_C e, per ogni i, $1 \leq i \leq m$, sia P_{α,b_i} il programma, se esiste, che definisce $Elim_{b_i}(\Delta, S_C)$. Il programma che definisce Δ sarà allora il seguente:

$$
\begin{aligned}
&\lambda F.\textsf{if } F(\alpha) = b_1 \textsf{ then } P_{\alpha,b_1}(F) \\
&\quad \textsf{else if } F(\alpha) = b_2 \textsf{ then } P_{\alpha,b_2}(F) \\
&\quad \vdots \\
&\quad \textsf{else if } F(\alpha) = b_m \textsf{ then } P_{\alpha,b_m}(F)
\end{aligned}
$$

Esempio 21. Sia $\Delta = \left\{ \left(\begin{pmatrix} 3 \\ \begin{pmatrix} 0 & 1 \\ \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}2\\0\end{pmatrix}\end{pmatrix}\end{pmatrix} \right), \left(\begin{pmatrix} 2 \\ \begin{pmatrix} 2 & 1 \\ \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}3\\0\end{pmatrix}\end{pmatrix}\end{pmatrix} \right) \right\}$.

Verifichiamo se Δ è definibile. Sia $\begin{pmatrix} 0 & 1 \\ \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}2\\0\end{pmatrix}\end{pmatrix}$ sia $\begin{pmatrix} 2 & 1 \\ \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}3\\0\end{pmatrix}\end{pmatrix}$ sono sequenziali. Siano $C = (C_1, C_2) = \left(\begin{pmatrix}1\\0\end{pmatrix}, \begin{pmatrix}1\\0\end{pmatrix} \right)$ e $\alpha = C_1 \cup C_2 = \begin{pmatrix}1\\0\end{pmatrix}$. Abbiamo che $S_C = (S_{Ci})_{i=1,2}$ è tale che

$$S_{C1} = \begin{pmatrix} 0 \\ \begin{pmatrix}1\\0\end{pmatrix}\end{pmatrix} \text{ e } S_{C2} = \begin{pmatrix} 2 \\ \begin{pmatrix}1\\0\end{pmatrix}\end{pmatrix}$$

e quindi

$$Elim_0(\Delta, S_C) = \left\{ \left(\begin{pmatrix} 3 \\ \begin{pmatrix}1\\ \begin{pmatrix}2\\0\end{pmatrix}\end{pmatrix}\end{pmatrix} \right) \right\}$$

$$Elim_2(\Delta, S_C) = \left\{ \left(\begin{pmatrix} 2 \\ \begin{pmatrix}1\\ \begin{pmatrix}3\\0\end{pmatrix}\end{pmatrix}\end{pmatrix} \right) \right\}$$

Per $Elim_0(\Delta, S_C)$ definiamo $C' = \alpha' = \begin{pmatrix}2\\0\end{pmatrix}$. Abbiamo che

$S_{C'} = (S_{C'1})$ è tale che $S_{C'1} = \left\langle \begin{pmatrix} 1 \\ \begin{pmatrix} 2 \\ 0 \end{pmatrix} \end{pmatrix} \right\rangle$ e

$$Elim_1(Elim_0(\Delta, S_C), S_{C'}) = \left\{ \begin{pmatrix} 3 \\ \perp^2 \end{pmatrix} \right\}$$

Per $Elim_2(\Delta, S_C)$ definiamo $C'' = \alpha'' = \begin{pmatrix} 3 \\ 0 \end{pmatrix}$. Abbiamo che $S_{C''} = (S_{C''1})$ è tale che $S_{C''1} = \left\langle \begin{pmatrix} 1 \\ \begin{pmatrix} 3 \\ 0 \end{pmatrix} \end{pmatrix} \right\rangle$ e

$$Elim_1(Elim_2(\Delta, S_C), S_{C''}) = \left\{ \begin{pmatrix} 2 \\ \perp^2 \end{pmatrix} \right\}$$

Il programma che definisce Δ è quindi il seguente:

$$\lambda F.\mathsf{if}\ F(\alpha) = 0\ \mathsf{then\ if}\ F(\alpha') = 1\ \mathsf{then}\ 3$$
$$\mathsf{else\ if}\ F(\alpha) = 2\ \mathsf{then\ if}\ F(\alpha'') = 1\ \mathsf{then}\ 2$$

dove

$$\begin{aligned} \alpha &= \lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 1 \\ \alpha' &= \lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 2 \\ \alpha'' &= \lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 3 \end{aligned}$$

Esempio 22. Sia $\Delta = \left\{ \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 1 & 1 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} \end{pmatrix} \right), \left(\begin{pmatrix} 1 \\ \begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{pmatrix} \end{pmatrix} \right) \right\}$. Δ non è definibile perché $\begin{pmatrix} 1 & 1 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix}$ non è sequenziale.

Esempio 23. Sia $\Delta = \left\{ \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 1 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{pmatrix} \end{pmatrix} \right), \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} \end{pmatrix} \right) \right\}$. Δ non è definibile perché $\begin{pmatrix} 0 \\ 0 \end{pmatrix} \not\asymp \begin{pmatrix} 1 \\ 0 \end{pmatrix}$.

3.4.1 Un controesempio

Ci sono dei casi in cui il procedimento descritto non porta ad ottenere un programma che definisce un funzionale Δ del tipo 3, anche se Δ è definibile. Ad esempio, sia

$$\Delta = \left\{ \left(\begin{pmatrix} 1 \\ \begin{pmatrix} 0 & 2 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} \end{pmatrix} \right), \left(\begin{pmatrix} 3 \\ \begin{pmatrix} 1 & 2 \\ \begin{pmatrix} 1 \\ 1 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} \end{pmatrix} \right) \right\}$$

Verifichiamo se Δ è definibile. Sia $F_1 = \begin{pmatrix} 0 & 2 \\ \begin{pmatrix}0\\0\end{pmatrix} & \begin{pmatrix}1\\0\end{pmatrix} \end{pmatrix}$ sia $F_2 = \begin{pmatrix} 1 & 2 \\ \begin{pmatrix}1\\1\end{pmatrix} & \begin{pmatrix}0\\1\end{pmatrix} \end{pmatrix}$ sono sequenziali. Siano

$$C = (C_1, C_2) = \left(\begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix} \right)$$
$$\alpha = C_1 \cup C_2 = \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix}$$

Abbiamo che $S_C = (S_{Ci})_{i=1,2}$ è tale che

$$S_{C1} = \begin{pmatrix} 0 \\ \begin{pmatrix}0\\0\end{pmatrix} \end{pmatrix} \text{ e } S_{C2} = \begin{pmatrix} 1 \\ \begin{pmatrix}1\\1\end{pmatrix} \end{pmatrix}$$

e quindi

$$Elim_0(\Delta, S_C) = \left\{ \begin{pmatrix} 1 \\ \begin{pmatrix} 2 \\ \begin{pmatrix}1\\0\end{pmatrix} \end{pmatrix} \end{pmatrix} \right\}$$

$$Elim_1(\Delta, S_C) = \left\{ \begin{pmatrix} 3 \\ \begin{pmatrix} 2 \\ \begin{pmatrix}0\\1\end{pmatrix} \end{pmatrix} \end{pmatrix} \right\}$$

Per $Elim_0(\Delta, S_C)$ definiamo $C' = \alpha' = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. Abbiamo che $S_{C'} = (S_{C'1})$ è tale che $S_{C'1} = \begin{pmatrix} 2 \\ \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix}$ e

$$Elim_2(Elim_0(\Delta, S_C), S_{C'}) = \left\{ \begin{pmatrix} 1 \\ \perp^2 \end{pmatrix} \right\}$$

Per $Elim_1(\Delta, S_C)$ definiamo $C'' = \alpha'' = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$. Abbiamo che $S_{C''} = (S_{C''1})$ è tale che $S_{C''1} = \begin{pmatrix} 2 \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix}$ e

$$Elim_2(Elim_1(\Delta, S_C), S_{C''}) = \left\{ \begin{pmatrix} 3 \\ \perp^2 \end{pmatrix} \right\}$$

Il programma che dovrebbe definire Δ è quindi il seguente:

$$\begin{aligned} P_\Delta = \lambda F.\mathsf{if}\ F(\alpha) = 0\ \mathsf{then\ if}\ F(\alpha') = 2\ \mathsf{then}\ 1 \\ \mathsf{else\ if}\ F(\alpha) = 1\ \mathsf{then\ if}\ F(\alpha'') = 2\ \mathsf{then}\ 3 \end{aligned}$$

dove

$$\begin{aligned} \alpha &= \lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 0 \\ &\qquad \mathsf{else\ if}\ x = 1\ \mathsf{then}\ 1 \\ \alpha' &= \lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 1 \\ \alpha'' &= \lambda x.\mathsf{if}\ x = 1\ \mathsf{then}\ 0 \end{aligned}$$

Si noti però che P_Δ è definito anche per i seguenti funzionali $F_3, F_4 \notin Dom(\Delta)$:

$$F_3 = \begin{pmatrix} 0 & 2 \\ \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} & \begin{pmatrix} 1 \\ 0 \end{pmatrix} \end{pmatrix} \quad F_4 = \begin{pmatrix} 1 & 2 \\ \begin{pmatrix} 0 & 1 \\ 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix}$$

Si ha infatti che $P_\Delta(F_3) = 1$ e $P_\Delta(F_4) = 3$. Il problema è dato dal fatto che, nel programma che definisce Δ, α ha una *sottocomputazione* che dipende da F. Abbiamo, quindi, che α non è funzione solo di x, ma anche di F:

$$\alpha = \lambda F.\lambda x.\mathsf{if}\ x = 0 \wedge F(\lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 1) = 2\ \mathsf{then}\ 0$$
$$\mathsf{else\ if}\ x = 1 \wedge F(\lambda x.\mathsf{if}\ x = 1\ \mathsf{then}\ 0) = 2\ \mathsf{then}\ 1$$

Il funzionale Δ sarà allora definito dal seguente programma:

$$P'_\Delta = \lambda F.\mathsf{if}\ F(\alpha(F)) = 0\ \mathsf{then}\ 1$$
$$\mathsf{else\ if}\ F(\alpha(F)) = 1\ \mathsf{then}\ 3$$

Si noti che $P'_\Delta(F_1) = 1$ in quanto $\alpha(F_1) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ e $F_1(\alpha(F_1)) = 0$, mentre $P'_\Delta(F_2) = 3$ in quanto $\alpha(F_2) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ e $F_2(\alpha(F_2)) = 1$. Si noti inoltre che questa volta $P'_\Delta(F_3) \uparrow$ e $P'_\Delta(F_4) \uparrow$. Infatti $\alpha(F_3) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ e $F_3(\alpha(F_3)) \uparrow$; analogamente $\alpha(F_4) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ e $F_4(\alpha(F_4)) \uparrow$.

Vediamo ora altri esempi in cui ci sono sottocomputazioni.

Esempio 24. Sia

$$\Delta = \left\{ \left(\begin{array}{cc} & 2 \\ 0 & 1 \\ \begin{pmatrix}0\\0\end{pmatrix} & \begin{pmatrix}2\\0\end{pmatrix} \end{array} \right), \left(\begin{array}{cc} & 2 \\ 1 & 2 \\ \begin{pmatrix}1\\1\end{pmatrix} & \begin{pmatrix}2\\1\end{pmatrix} \end{array} \right), \left(\begin{array}{cc} & 1 \\ 2 & 2 \\ \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}2\\0\end{pmatrix} \end{array} \right) \right\}$$

I funzionali $F_1 = \left(\begin{array}{cc} 0 & 1 \\ \begin{pmatrix}0\\0\end{pmatrix} & \begin{pmatrix}2\\0\end{pmatrix} \end{array} \right)$, $F_2 = \left(\begin{array}{cc} 1 & 2 \\ \begin{pmatrix}1\\1\end{pmatrix} & \begin{pmatrix}2\\1\end{pmatrix} \end{array} \right)$ e $F_3 = \left(\begin{array}{cc} 2 & 2 \\ \begin{pmatrix}1\\0\end{pmatrix} & \begin{pmatrix}2\\0\end{pmatrix} \end{array} \right)$ sono sequenziali. Siano

$$C = (C_1, C_2, C_3) = \left(\begin{pmatrix}2\\0\end{pmatrix}, \begin{pmatrix}1\\1\end{pmatrix}, \begin{pmatrix}2\\0\end{pmatrix} \right)$$

$$\alpha = C_1 \cup C_2 \cup C_3 = \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix}$$

Abbiamo che $S_C = (S_{Ci})_{i=1,2,3}$ è tale che

$$S_{C1} = \left(\begin{array}{c} 1 \\ \begin{pmatrix}2\\0\end{pmatrix} \end{array} \right), \; S_{C2} = \left(\begin{array}{c} 1 \\ \begin{pmatrix}1\\1\end{pmatrix} \end{array} \right) \text{ e } S_{C2} = \left(\begin{array}{c} 2 \\ \begin{pmatrix}2\\0\end{pmatrix} \end{array} \right).$$

Allora

$$Elim_1(\Delta, S_C) = \left\{ \left(\begin{array}{c} 2 \\ 0 \\ \begin{pmatrix}0\\0\end{pmatrix} \end{array} \right), \left(\begin{array}{c} 2 \\ 2 \\ \begin{pmatrix}2\\1\end{pmatrix} \end{array} \right) \right\}$$

$$Elim_2(\Delta, S_C) = \left\{ \left(\begin{array}{c} 1 \\ 2 \\ \begin{pmatrix}1\\0\end{pmatrix} \end{array} \right) \right\}$$

Per $Elim_1(\Delta, S_C)$ definiamo $C' = (C'_1, C'_2) = \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ 1 \end{pmatrix} \right)$ e $\alpha' = C'_1 \cup C'_2 = \begin{pmatrix} 0 & 2 \\ 0 & 1 \end{pmatrix}$. Abbiamo che $S_{C'} = (S_{C'i=1,2})$ è tale che $S_{C'1} = \left(\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \right)$ e $S_{C'2} = \left(\begin{pmatrix} 2 \\ 2 \\ 1 \end{pmatrix} \right)$. Allora

$$Elim_0(Elim_1(\Delta, S_C), S_{C'}) = \left\{ \begin{pmatrix} 2 \\ \perp^2 \end{pmatrix} \right\}$$

$$Elim_2(Elim_1(\Delta, S_C), S_{C'}) = \left\{ \begin{pmatrix} 2 \\ \perp^2 \end{pmatrix} \right\}$$

Per $Elim_2(\Delta, S_C)$ definiamo $C'' = \alpha'' = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. Abbiamo che $S_{C''} = (S_{C''1})$ è tale che $S_{C''1} = \left(\begin{pmatrix} 2 \\ 1 \\ 0 \end{pmatrix} \right)$ e

$$Elim_2(Elim_0(\Delta, S_C), S_{C''}) = \left\{ \begin{pmatrix} 1 \\ \perp^2 \end{pmatrix} \right\}$$

Il programma che dovrebbe definire Δ è quindi il seguente:

$$P_\Delta = \lambda F.\mathsf{if}\ F(\alpha) = 1\ \mathsf{then\ if}\ F(\alpha') = 0\ \mathsf{then}\ 2$$
$$\mathsf{else\ if}\ F(\alpha') = 2\ \mathsf{then}\ 2$$
$$\mathsf{else\ if}\ F(\alpha) = 2\ \mathsf{then\ if}\ F(\alpha'') = 2\ \mathsf{then}\ 1$$

dove

$$\begin{aligned}
\alpha &= \lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 2 \\
&\qquad \mathsf{else\ if}\ x = 1\ \mathsf{then}\ 1 \\
\alpha' &= \lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 0 \\
&\qquad \mathsf{else\ if}\ x = 1\ \mathsf{then}\ 2 \\
\alpha'' &= \lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 1
\end{aligned}$$

Si noti però che P_Δ è definito anche per i seguenti funzionali $F_i \notin Dom(\Delta)$ $(i = 4, \ldots, 10)$:

$$F_4 = \begin{pmatrix} 0 & 1 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} \end{pmatrix} \qquad F_5 = \begin{pmatrix} 0 & 1 \\ \begin{pmatrix} 0 & 2 \\ 0 & 1 \end{pmatrix} & \begin{pmatrix} 2 \\ 0 \end{pmatrix} \end{pmatrix}$$

$$F_6 = \begin{pmatrix} 0 & 1 \\ \begin{pmatrix} 0 & 2 \\ 0 & 1 \end{pmatrix} & \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} \end{pmatrix} \qquad F_7 = \begin{pmatrix} 1 & 2 \\ \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} & \begin{pmatrix} 2 \\ 1 \end{pmatrix} \end{pmatrix}$$

$$F_8 = \begin{pmatrix} 1 & 2 \\ \begin{pmatrix} 1 \\ 1 \end{pmatrix} & \begin{pmatrix} 0 & 2 \\ 0 & 1 \end{pmatrix} \end{pmatrix} \qquad F_9 = \begin{pmatrix} 1 & 2 \\ \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 2 \\ 0 & 1 \end{pmatrix} \end{pmatrix}$$

$$F_{10} = \begin{pmatrix} 2 & 2 \\ \begin{pmatrix} 1 \\ 0 \end{pmatrix} & \begin{pmatrix} 2 & 1 \\ 0 & 1 \end{pmatrix} \end{pmatrix}$$

Si ha infatti che $P_\Delta(F_4) = \ldots = P_\Delta(F_9) = 2$ e $P_\Delta(F_{10}) = 1$.

Considerando le sottocomputazioni abbiamo che

$$\alpha = \lambda F.\lambda x.\textsf{if } x = 0 \textsf{ then if } F(\lambda x.\textsf{if } x = 0 \textsf{ then } 2) = 1 \textsf{ then } 0$$
$$\textsf{else if } F(\lambda x.\textsf{if } x = 0 \textsf{ then } 2) = 2 \textsf{ then } 1$$
$$\textsf{else if } x = 1 \wedge F(\lambda x.\textsf{if } x = 1 \textsf{ then } 2) = 2 \textsf{ then } 1$$

Il funzionale Δ sarà allora definito dal seguente programma:

$$P'_\Delta = \lambda F.\textsf{if } F(\alpha(F)) = 0 \textsf{ then } 2$$
$$\textsf{else if } F(\alpha(F)) = 1 \textsf{ then } 2$$
$$\textsf{else if } F(\alpha(F)) = 2 \textsf{ then } 1$$

Si noti che $P'_\Delta(F_1) = 2$ in quanto $\alpha(F_1) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ e $F_1(\alpha(F_1)) = 0$, $P'_\Delta(F_2) = 2$ in quanto $\alpha(F_2) = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ e $F_2(\alpha(F_2)) = 1$, mentre $P'_\Delta(F_3) = 1$ in quanto $\alpha(F_3) = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ e $F_3(\alpha(F_3)) = 2$. Si noti

inoltre che questa volta $P'_\Delta(F_i) \uparrow$ per $4 \leq i \leq 10$. Infatti

$$\begin{aligned}
&\alpha(F_4) = \perp^2 \text{ e } F_4(\alpha(F_4)) \uparrow,\\
&\alpha(F_5) = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \text{ e } F_5(\alpha(F_5)) \uparrow,\\
&\alpha(F_6) = \perp^2 \text{ e } F_6(\alpha(F_6)) \uparrow,\\
&\alpha(F_7) = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \text{ e } F_7(\alpha(F_7)) \uparrow,\\
&\alpha(F_8) = \perp^2 \text{ e } F_8(\alpha(F_8)) \uparrow,\\
&\alpha(F_9) = \perp^2 \text{ e } F_9(\alpha(F_9)) \uparrow,\\
&\alpha(F_{10}) = \perp^2 \text{ e } F_{10}(\alpha(F_{10})) \uparrow.
\end{aligned}$$

Esempio 25. Sia

$$\Delta = \left\{ \begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{pmatrix} \end{pmatrix}, \begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} \end{pmatrix} \right\}$$

I funzionali $F_1 = \begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{pmatrix}$ e $F_2 = \begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix}$ sono ovviamente sequenziali. Siano

$$\begin{aligned}
C = (C_1, C_2) &= \left(\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right)\\
\alpha = C_1 \cup C_2 &= \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix}
\end{aligned}$$

Abbiamo che $S_C = (S_{Ci})_{i=1,2}$ è tale che

$$S_{C1} = \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{pmatrix} \right) \text{ e } S_{C2} = \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} \right)$$

e quindi

$$Elim_0(\Delta, S_C) \ = \ \left\{ \begin{pmatrix} 0 \\ \perp^2 \end{pmatrix} \right\}$$

Il programma che dovrebbe definire Δ è quindi il seguente:

$$P_\Delta = \lambda F.\mathsf{if}\ F(\alpha) = 0\ \mathsf{then}\ 0$$

dove

$$\alpha = \lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 0$$
$$\mathsf{else\ if}\ x = 1\ \mathsf{then}\ 0$$

Si noti però che P_Δ è definito anche per il seguente funzionale $F_3 \notin Dom(\Delta)$:

$$F_3 = \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \end{pmatrix} \right)$$

Si ha infatti che $P_\Delta(F_3) = 0$. In questo caso le sottocomputazioni sono "nascoste": si osservi che possiamo riscrivere Δ come

$$\Delta = \left\{ \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 0 & 0 \\ \begin{pmatrix} 0 \\ 0 \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \end{pmatrix} \end{pmatrix} \end{pmatrix} \right), \left(\begin{pmatrix} 0 \\ \begin{pmatrix} 0 & 0 \\ \begin{pmatrix} 0 \\ 1 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \end{pmatrix} \end{pmatrix} \end{pmatrix} \right) \right\}$$

e definire α nel seguente modo:

$$\alpha = \lambda F.\lambda x.\mathsf{if}\ x = 0 \wedge F(\lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 0) = 0\ \mathsf{then}\ 0$$
$$\mathsf{else\ if}\ x = 1 \wedge F(\lambda x.\mathsf{if}\ x = 1\ \mathsf{then}\ 0) = 0\ \mathsf{then}\ 0$$

Il funzionale Δ sarà allora definito dal seguente programma:

$$P'_\Delta = \lambda F.\mathsf{if}\ F(\alpha(F)) = 0\ \mathsf{then}\ 0$$

Si noti che $P'_\Delta(F_1) = 0$ in quanto $\alpha(F_1) = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$ e $F_1(\alpha(F_1)) = 0$, $P'_\Delta(F_2) = 0$ in quanto $\alpha(F_2) = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ e $F_2(\alpha(F_2)) = 0$, mentre $P'_\Delta(F_3) \uparrow$ perché $\alpha(F_3) = \perp^2$ e $F_3(\alpha(F_3)) \uparrow$.

Capitolo 4

I linguaggi funzionali

La programmazione funzionale è un paradigma di programmazione che tratta la computazione come la valutazione di una funzione matematica. Essa enfatizza la definizione e l'applicazione di funzioni, piuttosto che l' implementazione di macchine a stati. Questo è in contrasto con la programmazione imperativa o procedurale, la quale enfatizza la sequenzialità dei comandi in esecuzione. I valori nei linguaggi imperativi sono costruiti con assegnazioni che trasformano lo stato del programma; un programma funzionale, al contrario, piuttosto che modificare lo stato per produrre valori, costruisce nuovi stati a partire da altri stati. L'iterazione, un costrutto fondamentale della programmazione imperativa, è sostituito dalla più generale

ricorsione.

Il più vecchio esempio di linguaggio funzionale è il Lisp, sebbene né il Lisp originale né i moderni Lisp come Common Lisp e Scheme siano funzionali puri. Gli esempi canonici moderni sono Haskell e i membri della famiglia ML, fra cui SML e OCaml. Altri esempi sono Erlang, Clean, e Miranda. Altri linguaggi, come Tcl, Perl, Python e Ruby, possono anche essere usati in uno stile funzionale, dato che dispongono di funzioni di alto ordine, astrazioni e altro.

I programmi funzionali puri non hanno *effetti secondari* (side-effects) e sono quindi automaticamente thread-safe. Una funzione produce un effetto secondario quando, oltre a ritornare un valore, modifica uno stato. Per esempio, una funzione può modificare una variabile globale, uno dei suoi argomenti, scrivere dei dati sul display o in un file, o ricevere dei dati da un'altra funzione con effetti secondari. Gli effetti secondari spesso rendono il comportamento di un programma più difficile da comprendere. La programmazione imperativa fa molto uso di effetti secondari; quella funzionale al contrario li minimizza. Una funzione con effetti secondari è detta essere *referentially opaque*, e una senza è detta *referentially transparent*. Una funzione referentially trans-

parent è una funzione che, dati in input gli stessi parametri, dà sempre lo stesso risultato.

Come esempio, consideriamo due funzioni, una referentially opaque (rq) e l'altra referentially transparent (rt):

```
globalValue = 0;
integer function rq(integer x)
begin
  globalValue = globalValue + 1;
  return x + globalValue;
end

integer function rt(integer x)
begin
  return x + 1;
end
```

rt è referentially transparent, il che significa che $rt(x) = rt(x)$, purché x sia lo stesso valore. Ad esempio, $rt(6) = rt(6) = 7$, $rt(4) = rt(3+1) = 5$, e così via. Però, non possiamo dire niente di simile su rq perché usa una variabile globale e allo stesso tempo la modifica. Ora, qual è il problema? Bene, diciamo

che vogliamo fare qualche ragionamento sul seguente pezzo di codice:

```
integer p = rq(x) + rq(y) * (rq(x) - rq(x));
```

Uno potrebbe essere tentato a semplificare questa linea di codice con `integer p = rq(x)`, poiché

$$\begin{aligned} rq(x) + rq(y) * (rq(x) - rq(x)) &= rq(x) + rq(y) * 0 \\ &= rq(x) + 0 \\ &= rq(x) \end{aligned}$$

Però questo non funziona con rq poiché $rq(x)$ non è uguale a $rq(x)$! Si ricordi che il valore di ritorno di rq dipende da una variabile globale che non è passata come parametro e che è modificata ad ogni chiamata. Questo è contro il senso comune che qualsiasi cosa meno se stessa deve essere uguale a zero. Al contrario questo ragionamento è corretto per rt perché è una funzione referentially transparent.

Le funzioni possono essere manipolate in vari modi in un linguaggio funzionale. Esse sono trattate come *valori di prima classe*; ciò significa che possono essere parametri o valori di ritorno di altre funzioni. Ad esempio la funzione `map` in Haskell ha

come parametri una funzione e una lista, e applica la funzione a ogni elemento della lista. Alle funzioni possono essere assegnati dei nomi, come in altri linguaggi, oppure possono essere definite anonime usando una lambda astrazione. È possibile creare funzioni durante l'esecuzione del programma e usare funzioni come valori in altre funzioni.

In C non è possibile creare nuove funzioni a runtime, mentre ciò è consentito con altri tipi di oggetti. Per questo le funzioni in C non sono oggetti di prima classe; a volte sono chiamate oggetti di seconda classa perché possono essere manipolate tramite puntatori. Analogamente, in Fortran le stringhe non sono di prima classe perché non è possibile assegnarle a variabili.

Una funzione di *alto ordine* (higher-order) è una funzione che ha una funzione come parametro o che ritorna una funzione di alto ordine. Ad esempio, le due seguenti funzioni Haskell, `squareList` e `squareListNoHof`, elevano al quadrato gli elementi di una lista; la prima usa la funzione di alto ordine `map`, la seconda no:

```
squareList list = map (^2) list
squareListNoHof [] = []
```

```
squareListNoHof (head:tail)
    = (head^2):(squareListNoHof tail)
```

`map` prende la funzione `(^2)` (si noti che manca il primo argomento di `(^2)`; questo indica ad Haskell di sostituire gli elementi della lista come primo argomento) e la lista `list`, e fa il quadrato di ogni elemento. `map` "mappa" una funzione su una lista, e cioè, applica la funzione su ogni elemento della lista. Per eseguire `squareList` e `squareListNoHof` con una lista che contiene 1, 2 e 3 scriviamo ("-" indica il prompt, la linea successiva il risultato):

```
- squareList [1, 2, 3]
[1,4,9]
- squareListNoHof [1, 2, 3]
[1,4,9]
```

Il precedente era un esempio di funzione di alto ordine che riceveva come argomento una funzione, ma non ritornava una funzione come risultato. Ci sono però funzioni standard di alto ordine che lo fanno, come la funzione ".", che rappresenta la composizione di funzioni. Le due seguenti funzioni Haskell

calcolano la funzione $\cos(\ln\sqrt{3x+2})$ con e senza la funzione ".":

```
Composite x = (cos.log.sqrt) (3*x+2)
CompositeNoHof x = cos (log (sqrt (3*x+2)))
```

La funzione "." riceve due funzioni come argomenti e ritorna una funzione che rappresenta la loro composizione, cioè $(f.g)x = f(g(x))$. Nel precedente esempio `(cos.log.sqrt) (3*x+2)` si legge come `(cos.(log.sqrt)) (3*x+2)`.

Un esempio di funzione di alto ordine definita dall'utente è la seguente funzione SML `d` che calcola la derivata di una funzione `f` nel punto `x`:

```
- fun d (delta, f, x) =
      (f (x + delta) - f (x - delta)) / (2.0 * delta);
val d = fn : real * (real -> real) * real -> real
```

Questa funzione richiede di specificare un piccolo valore `delta`[1]. Per calcolare la derivata della funzione $x^3 - x - 1$ nel punto 3 scriviamo:

```
- d (1E~8, fn x => x * x * x - x - 1.0, 3.0);
val it = 25.9999996644 : real
```

[1]Una buona scelta è la radice quadrata del epsilon della macchina.

Si noti che la funzione $x^3 - x - 1$ è definita "al volo" con il parametro `fn x => x * x * x - x - 1.0`.

I linguaggi funzionali permettono anche di applicare il *currying* alle funzioni. Il currying è una tecnica di riscrittura di una funzione con più parametri in una funzione con un solo parametro che restituisce un'altra funzione con un paramatro e così via, finché non sono esauriti tutti i parametri. La funzione così ottenuta (curried) può essere applicata a un sottoinsieme dei suoi parametri originali. Il risultato è una funzione in cui i parametri di questo sottoinsieme sono fissati come costanti, e il valore dei restanti parametri è ancora non specificato. Questa nuova funzione può essere applicata ai restanti parametri per ottenere il valore finale della funzione.

Per esempio, alla funzione $sum(x, y) = x + y$ può essere applicato il currying cosicché il valore di $sum(1)$ (si noti che non è specificato il parametro y) sia una funzione anonima tale che $sum1(y) = 1 + y$. Questa nuova funzione ha un solo parametro, al quale aggiunge 1.

In SML abbiamo:

```
- fun sum (x, y) = x + y;
```

```
val sum = fn : int * int -> int
- sum (1, 2);
val it = 3 : int
- fun carried_sum x y = x + y;
val carried_sum = fn : int -> int -> int
- carried_sum 1 2;
val it = 3 : int
- val sum1 = carried_sum 1;
val sum1 = fn : int -> int
- sum1 2;
val it = 3 : int
```

Si noti che si può definire una funzione di alto ordine *curry* con cui applicare il currying ad una funzione, ad esempio a `sum`:

```
- fun carry f x y = f (x, y);
val carry = fn : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
- val carried_sum = carry sum;
val carried_sum = fn : int -> int -> int
```

Possiamo riscrivere la funzione Haskell `squareList` tenendo presente che la funzione di alto ordine `map` è curried:

```
squareList = map (^2)
```

Qui, non viene specificato il secondo argomento di `map`, e `map (^2)` è quindi una funzione che ha come parametro una lista, sulla quale applica la funzione `(^2)`.

Riprendendo la funzione SML `d` che calcola la derivata, abbiamo che la sua versione carried è:

```
- fun d delta f x =
      (f (x + delta) - f (x - delta)) / (2.0 * delta);
val d = fn : real -> (real -> real) -> real -> real
```

Possiamo ora specificare un valore per `delta`, da usare tutte le volte che vogliamo calcolare una derivata:

```
- val d = d 1E~8;
val d = fn : (real -> real) -> real -> real
```

Un concetto presente in alcuni linguaggi funzionali come ML e Haskell è il *polimorfismo parametrico*, un particolare tipo di polimorfismo. Una funzione polimorfa è una funzione che può essere applicata a tipi diversi e dare come risultato valori di tipi diversi. L'idea di polimorfismo si estende anche ai tipi di dati: un tipo di dati polimorfo è un tipo di dati che può contenere elementi di differenti tipi.

Ci sono fondamentalmente due diversi tipi di polimorfismo. Se l'insieme dei tipi che possono essere usati è finito e le varie combinazioni devono essere specificate prima dell'uso, il polimorfismo è detto *ad-hoc*. Se invece tutto il codice è scritto senza nessun riferimento a particolari tipi, e può quindi essere usato trasparentemente con ogni tipo, allora il polimorfismo è detto parametrico.

La programmazione che usa questo secondo tipo di polimorfismo è detta *programmazione generica*, specialmente nella comunità object-oriented. I sostenitori della programmazione object-oriented spesso citano il polimorfismo come uno dei maggiori benefici di quel paradigma rispetto agli altri. I sostenitori della programmazione funzionale rifiutano questa rivendicazione per il semplice fatto che il polimorfismo parametrico è intrinsecamente presente in alcuni linguaggi funzionali staticamente tipati.

Con il polimorfismo parametrico, ad esempio, si può definire una funzione *append* che concatena due liste e che non dipende dal tipo degli elementi: può funzionare con liste di interi, liste di reali, liste di stringhe e così via. Se denotiamo con a il tipo degli elementi, il tipo di *append* è $[a] \times [a] \rightarrow [a]$, dove $[a]$ è il tipo "lista di elementi di tipo a". Il tipo di *append* è quindi

parametrizzato da a. Ogni volta che *append* viene applicata, viene stabilito un valore per a. (Si noti che *append* non può essere applicata a una qualsiasi coppia di liste: esse, così come la lista risultante, devono avere lo stesso tipo di elementi.)

Un altro esempio di polimorfismo parametrico lo abbiamo con la seguente funzione Haskell, `length`, che calcola la lunghezza di una lista:

```
length [] = 0
length (first:rest) = 1 + length rest
```

È evidente che questa funzione gestisce liste di qualsiasi tipo e che il caso base della ricorsione ritorna un intero. Possiamo quindi assegnarli il tipo con `length :: [a] -> Int` (in realtà non c'è bisogno di far ciò in quanto è il compilatore a farlo automaticamente). `length` è in effetti una funzione polimorfa:

```
- length [1, 2, 3, 4, 5]
5
length ["Haskell", "polimorfismo", "funzione",
        "tipo", "length"]
5
```

Il polimorfismo parametrico è stato identificato da Christo-

pher Strachey nel 1967 ed è stato il primo tipo di polimorfismo ad apparire in un linguaggio di programmazione, ML, nel 1976. Alcuni ritengono che i *template* debbano essere considerati un esempio di polimorfismo parametrico sebbene le implementazioni, invece di produrre codice generico, generano codice specifico per tutti i tipi usati.

Il polimorfismo parametrico è un modo per rendere un linguaggio più espressivo, pur mantenendo una sicurezza statica sui tipi. Esso è quindi irrilevante nei linguaggi dinamicamente tipati, dato che essi per definizione sono privi di sicurezza statica sui tipi. Ad ogni funzione dinamicamente tipata con n argomenti può essere assegnato un tipo statico usando il polimorfismo parametrico: $p_1 \times \ldots \times p_n \rightarrow r$, in cui $p_1, \ldots, p_n$ e r sono parametri tipo. Ovviamente questo tipo è privo di sostanza e quindi essenzialmente inutile. Per questo nei linguaggi dinamicamente tipati il tipo degli argomenti e del valore di ritorno è controllato a run-time, per verificare che sia rispettato dalle operazioni effettuate su di essi.

I sistemi dei tipi con polimorfismo parametro possono essere classificati in predicativi e non-predicativi; la differenza sta nel modo in cui i parametri possono essere istanziati. Per esempio,

si consideri la funzione *append* descritta precedentemente, il cui tipo è $[a] \times [a] \rightarrow [a]$. In un sistema non predicativo il tipo sostituito ad a può essere uno qualsiasi, anche un tipo esso stesso polimorfo; quindi *append* può essere applicata ad una coppia di liste di elementi di qualsiasi tipo, anche a liste di funzioni polimorfe come la stessa *append*. In un sistema predicativo, invece, i tipi istanziati non possono essere polimorfi. Questa restrizione rende la differenza tra tipi polimorfi e non polimorfi molto importante.

Il polimorfismo in ML è predicativo. Questo perché la predicatività, insieme ad altre restrizioni, rende il sistema dei tipi abbastanza semplice da consentire l'inferenza dei tipi. Nei linguaggi in cui sono necessarie le annotazioni sui tipi quando si applica una funzione polimorfa, la restrizione sulla predicatività è meno importante; per questo tali linguaggi sono generalmente non predicativi. Haskell effettua l'inferenza sui tipi senza la predicatività, ma con alcune complicazioni.

4.1 ML

ML è un linguaggio di programmazione funzionale general-purpose sviluppato da Robin Milner e altri alla fine degli anni '70 all'Università di Edinburgo. È famoso per l'uso dell'algoritmo di inferenza di Hindley-Milner, che può inferire i tipi di molti valori senza richiedere estensive annotazioni. ML è un linguaggio funzionale impuro, perché permette gli effetti secondari, e quindi la programmazione imperativa, contrariamente a linguaggi funzionali puri come Haskell. Le caratteristiche principali di ML sono: strategia di valutazione call-by-value, polimorfismo parametrico, tipaggio statico, tipi di dati algebrici, pattern matching e gestione delle eccezioni.

Oggi ci sono diversi linguaggi nella famiglia ML; i due dialetti più importanti sono SML (Standard ML) e OCaml. Le idee di ML hanno influenzato altri linguaggi, tra cui Haskell. ML è molto popolare tra i ricercatori che si occupano di linguaggi di programmazione e tra gli sviluppatori di compilatori e di dimostratori di teoremi. SML è l'unico tra i liguaggi ampiamente diffusi ad avere una specifica formale, data da regole di tipaggio e da una semantica operazionale [7][8].

Molte funzioni matematiche possono essere rappresentate facilmente con un linguaggio funzionale. La seguente funzione ricorsiva implementa il fattoriale in SML:

```
- fun factorial x =
      if x = 0 then 1 else x * factorial (x-1);
val factorial = fn : int -> int
- factorial 4;
val it = 24 : int
```

Ad un compilatore SML è richiesto di inferire il tipo `int -> int` di questa funzione senza annotazioni sui tipi fornite dall'utente. Cioè, deve dedurre che `x` è usata solo in espressioni intere e deve quindi essere essa stessa un intero, e che tutte le espressioni nella funzione sono interi.

La stessa funzione può essere espressa con una definizione clausale (*pattern matching*) in cui il condizionale *if-then-else* è sostituito da una sequenza di template della funzione valutati per specifici valori, separati da "|", che sono analizzati uno alla volta secondo l'ordine in cui sono scritti finché non viene trovato un *match*:

```
- fun factorial 0 = 1
```

```
    | factorial n = n * factorial (n - 1);
val factorial = fn : int -> int
```

Questa funzione può essere riscritta, usando una funzione locale, con una *tail recursion*:[2]

```
- fun factorial x =
      let
          fun tail_fact p 0 = p
            | tail_fact p n = tail_fact (p * n) (n - 1)
      in
          tail_fact 1 x
      end
val factorial = fn : int -> int
```

Il valore di una espressione *let* è l'espressione tra *in* e *end*.

4.2 Haskell

Haskell è un linguaggio funzionale lazy puro. È così chiamato dal nome del logico Haskell Curry, ed è stato creato da una

[2]Una tail recursion è un particolare tipo di ricorsione che può essere valutato efficientemente come una iterazione; una volta calcolato il risultato, e ciò avviene, in una tail recursion, nella chiamata ricorsiva più "profonda", non c'è bisogno di "srotolare" lo stack delle chiamate.

commissione formata nel 1987 con lo specifico scopo di definire questo linguaggio. Il diretto predecessore di Haskell è Miranda, del 1985; l'ultimo standard semi-ufficiale del linguaggio è Haskell 98. Il linguaggio evolve continuamente, e le due principali implementazioni Hugs e GHC (Glasgow Haskell Compiler) rappresentano lo standard corrente de facto.

Haskell è uno dei pochi linguaggi *non-stretti* (non-strict) in circolazione; quasi tutti i linguaggi di uso comune sono attualmente stretti. Un linguaggio di programmazione stretto è un linguaggio in cui possono essere definite solo funzioni strette. Secondo la semantica denotazionale dei linguaggi di programmazione, una funzione f è stretta se $f(\perp) = \perp$; dove $\perp$, che si legge "bottom", denota il "valore" di una espressione che non ritorna un valore. Operazionalmente, una funzione stretta è una funzione che valuta sempre il suo argomento. Funzioni con più parametri possono essere strette o non-strette in ciascun parametro indipendentemente, così come possono essere strette o non-strette in più parametri simultaneamente.

Alcune caratteristiche di Haskell sono: pattern matching, list comprehensions, guards, operatori definibili, tipi di dati algebrici. Altri concetti, come monads e type classes, sono presenti

solo nel linguaggio Haskell. La combinazione di tali funzionalità permette la definizione di funzioni che sarebbero difficili da scrivere in linguaggi procedurali.

Haskell è il linguaggio funzionale lazy su cui viene effettuata la maggior parte della ricerca. Sono state sviluppate diverse varianti: versioni parallele al MIT e a Glasgow, entrambe chiamate Parallel Haskell; versioni parallele e distribuite chiamate Distributed Haskell e Eden; versioni object oriented, Haskell++, O'Haskell e Mondrian; una versione eager, Eager Haskell.

Il fattoriale, già visto con ML, si scrive in Haskell nel seguente modo:

```
fac 0 = 1
fac n | n > 0 = n * fac (n-1)
```

Anche Haskell è in grado in inferire il tipo di `fac`, che è `Integer -> Integer`. Usando il pattern matching, quando l'argomento della funzione è 0 il risultato è 1; negli altri casi viene considerata la linea di codice successiva, che contiene il *guard* `n > 0`, il quale impedisce che `fac` sia invocata con numeri negativi.

`fac` può essere scritta anche nel seguente modo, usando la composizione "." e la funzione `enumFromTo` del *Prelude* (un

insieme di funzioni, analogo alla libreria standard del linguaggio C):

```
fac = product . enumFromTo 1
```

Si noti che non è specificato il parametro di `fac`; questa definizione è simile ad una definizione matematica del tipo $f = g \circ h$.

Bibliografia

[1] S. Abramsky, R. Jagadeesan e P. Malacaria. Full Abstraction for PCF. *Information and Computation*, 163:409–470, 2000.

[2] J. M. E. Hyland e C.-H. L. Ong. On Full Abstraction for PCF: I. Models, observables and the full abstraction problem, II. Dialogue games and innocent strategies, III. A fully abstract and universal game model. In *Information and Computation*, 163:285–408, 2000.

[3] D. S. Scott. A type-theoretical alternative to CUCH, ISWIM and OWHY. *Theoretical Computer Science* (Böhm, Festschrift, Edizione Speciale), 121:411–440, 1993. Questo articolo è stato ampiamente diffuso come un manoscritto non pubblicato dal 1969.

[4] G. D. Plotkin. LCF as a programming language. *Theoretical Computer Science*, 5:223–225, 1977.

[5] C.-H. L. Ong. Correspondence between Operational and Denotational Semantics. In *Handbook of Logic in Computer Science, Volume 4*, a cura di S. Abramsky, D. Gabbay e T. S. E. Maibaum, Oxford University Press, pagine 269–356, 1995.

[6] R. Milner. Fully abstract models of typed lambda-calculus. *Theoretical Computer Science*, 4:1–22, 1977.

[7] R. Milner, M. Tofte, R. Harper. *Definition of Standard ML*. MIT Press, 1990.

[8] R. Milner, M. Tofte, R. Harper, D. MacQueen. *Definition of Standard ML (Revised)*. MIT Press, 1997.

[9] G. Pani. A simply definition of games for PCF. To appear.

[10] P. Bungaro. Il problema del modello fully abstract per il linguaggio funzionale PCF. Tesi di laurea. Università degli Studi di Bari, A.A. 1988–89.

[11] D. Palmitessa. Analisi degli elementi booleani di tipo 0, 1 e 2. Tesi di laurea. Università degli Studi di Bari, 1998.

[12] Wikipedia contributors. Functional programming. *Wikipedia, The Free Encyclopedia*, 2005. <http://en.wikipedia.org/wiki/Functional_programming>