SAPIENZA
UNIVERSITÀ DI ROMA

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

CORSO DI LAUREA IN INFORMATICA

Tesi di Laurea Specialistica

Sottotipaggio Semantico per Linguaggi ad Oggetti

RELATORE	CANDIDATO
Prof. Daniele Gorla	Ornela Dardha (1049239)

Anno accademico 2009/2010

Indice

Introduzione

Un *sistema di tipi* per un linguaggio di programmazione è un insieme di regole che consentono di dare un tipo ad espressioni, comandi ed altri costrutti del linguaggio. Un linguaggio si dice *tipato* se per esso è definito un tale sistema; altrimenti si dice *non tipato*. Solitamente, i sistemi di tipi si basano su una relazione di sottotipaggio definita sull'insieme dei tipi sintattici. Un tipo T è sottotipo di un tipo S se, laddove ci si aspetta un valore di tipo S, possiamo usare correttamente un valore di tipo T. Possiamo formalizzare questa intuizione introducendo una relazione di sottotipaggio fra tipi, scritta $T \leq S$ e una regola di sussunzione (*subsumption*) nel sistema di tipi che, se $T \leq S$, permetta di assegnare ad ogni termine di tipo T anche il tipo S.

Si hanno due approcci per definire la relazione di sottotipaggio: quello *sintattico* e quello *semantico*. L'approccio sintattico è più diffuso e consiste nel definire la relazione di sottotipaggio tramite un sistema formale di regole deduttive. Si procede nel seguente modo: si definisce il linguaggio, l'insieme dei tipi sintattici e la relazione di sottotipaggio definita su questo insieme tramite delle regole di inferenza. Nell'approccio semantico, invece, si parte da un modello del linguaggio e un'interpretazione dei tipi come sottoinsiemi di questo modello. Si definisce la relazione di sottotipaggio tra due tipi come l'inclusione degli insiemi corrispondenti a questi tipi.

L'approccio semantico ha ricevuto meno attenzione rispetto a quello sintattico in quanto è più difficile definirlo: interpretare i tipi come sottoinsiemi di un modello non è banale; però riporta dei vantaggi che in un'algebra di tipi ricca, ad esempio in presenza di tipi booleani, sono veramente importanti. Vedremo quanto detto nel seguito della nostra trattazione.

Si ha una vasta letteratura per questo approccio, risalente a più di 15 anni fa, come [AW93, Dam94]. Un altro lavoro importante che usa questo approccio è quello di Hosoya e Pierce [HP01, HVP05, HP03] per il linguaggio XDuce. Questo è un linguaggio XML-*oriented* disegnato specificatamente per trasformare i documenti XML in altri documenti XML che soddisfano certe caratteristiche. I valori di questo linguaggio sono frammenti di documenti XML; i tipi sono insiemi di documenti, più precisamente sono insiemi di valori. Il sistema di tipi contiene tipi booleani, di cui parleremo in dettaglio in seguito, tipi prodotto e tipi ricorsivi. Mancano i tipi funzionali e il concetto di funzione nel linguaggio. La relazione di sottotipaggio è definita in modo semplice, in quanto i tipi denotano insiemi e il sottotipaggio viene definito semanticamente come inclusione di tali insiemi.

Castagna et al. in [CF05, Cas05, FCB08] estendono il linguaggio XDuce con funzioni *first-class* e con tipi freccia, dando luogo ad un linguaggio di ordine superiore (*higher-order language*), preservando l'approccio semantico al sottotipaggio. Il punto di partenza della loro trattazione è un $\lambda-$calcolo di ordine superiore in cui i tipi unione, intersezione e negazione hanno un'interpretazione insiemistica. In questo linguaggio viene studiato ampiamente il sottotipaggio semantico dando dei risultati teorici fondamentali. Questo è il punto di partenza del linguaggio $\mathbb{C}$Duce che loro hanno implementato.

Questo approccio al sottotipaggio applicato al $\lambda-$calcolo può essere applicato anche al $\pi-$calcolo [Mil99, SW03]. Infatti, in [CNV08] viene usata la stessa tecnica del sottotipaggio semantico per definire il linguaggio $\mathbb{C}\pi$, un $\pi-$calcolo aumentato con tipi booleani che vengono interpretati nel modo ovvio.

Quanto abbiamo appena visto è una panoramica veloce sul sottotipaggio semantico in letteratura. Il nostro contributo originale dato in questo lavoro di tesi è quello di definire il sottotipaggio semantico per un semplice linguaggio ad oggetti. Abbiamo considerato un *core-language*, il frammento funzionale di Java, e seguendo il lavoro fatto per il $\lambda-$calcolo

in [FCB08] abbiamo definito *from scratch* la relazione di sottotipaggio semantico in questo calcolo dando dei risultati teorici molto importanti. Per capire i vantaggi di questo approccio in un calcolo ad oggetti consideriamo il seguente esempio: supponiamo che stiamo lavorando con dei poligoni come triangoli, rettangoli, rombi ecc. Vogliamo definire un metodo che calcola la diagonale più lunga del poligono; questo ha senso solo se il poligono di cui vogliamo calcolare la diagonale non è un triangolo. In Java si hanno diversi modi per implementare questo problema, come in seguito:

```
class Polygon {
      ...
}

class Triangle extends Polygon {
            ...
}

class Other_Polygons extends Polygon {
                        ⋮
   real diagonal(Other_Polygons p) {...}
}

class Square extends Other_Polygons {
                        ...
}

class Rumble extends Other_Polygons {
                        ...
}
                        ⋮
```

Oppure, un altro modo per implementare il problema della diagonale è tramite l'utilizzo

delle interfacce come nel seguito:

```
public interface Diagonal {
   real diagonal(Polygon p);
}

class Polygon {
      ...
}

class Triangle extends Polygon {
            ...
}

class Square extends Polygon  implements Diagonal {
                              ...
}

class Rumble extends Polygon  implements Diagonal {
                              ...
}
                              ⋮
```

Supponiamo, invece, che la gerarchia delle classi abbia *Polygon* come classe padre e con tutte le altri classi di figure geometriche che estendono *Polygon*: assumiamo che questa implementazione sia data e non sia possibile modificarla. A questo punto, aggiungere un metodo che calcola la diagonale più lunga diventa più complicato. L'unico modo è quello di aggiungere questo metodo nella classe *Polygon* e utilizzare un **instanceof** (magari dentro un costrutto **try-catch**) che solleva un eccezione a tempo di esecuzione se l'argomento passato al metodo *diagonal* è un triangolo.

Con i tipi booleani, invece, basterebbe dichiarare un metodo con tipo di argomen-

to '*Polygon* AND NOT *Triangle*' (*Polygon* $\wedge \neg$ *Triangle*) che permette al *typechecker* di controllare staticamente a tempo di compilazione le restrizioni sui tipi. Una possibile implementazione è come segue:

```
class Polygon extends Object {
                ...
}

class Triangle extends Polygon {
                ...
}

class Square extends Polygon {
                ...
}

class Rumble extends Polygon {
                ...
}
                ⋮

class Diagonal extends Object {
   real diagonal(Polygon ∧ ¬Triangle p){...}
}
```

A questo punto, per calcolare la diagonale più lunga, basta invocare il metodo *diagonal* dalla classe *Diagonal*, che è una classe di servizio, e passare a questo metodo un argomento di tipo *Polygon*: se il poligono non è un *Triangle*, allora il metodo calcola la diagonale più lunga, come richiesto; altrimenti, se il poligono dato in input al metodo è un triangolo,

allora a tempo di compilazione si avrà un errore di tipo.

$$Polygon\ p = \textbf{new}\ Polygon();$$
$$p.diag = Diagonal.diagonal(p);$$

Un'altra linea di ricerca nella comunità scientifica, per quanto riguarda i paradigmi ad oggetti, è quella del sottotipaggio *nominale* vs. il sottotipaggio *strutturale*. In un linguaggio con sottotipaggio nominale, A è sottotipo di B se e solo se è dichiarato tale, ovvero se la classe A estende (o implementa) la classe (o l'interfaccia) B; queste relazioni devono essere dichiarate dal programmatore e sono basate solo sui nomi delle classi e delle interfacce coinvolte. Negli ultimi anni, è emerso un nuovo approccio al sottotipaggio, quello strutturale [GM08, MA08, MA09]. In questo approccio, la relazione di sottotipaggio viene stabilita solamente dall'analisi della struttura di una classe, i.e. dai campi e metodi dichiarati: una classe A è sottotipo di una classe B se i suoi campi e metodi sono sottoinsiemi dei campi e metodi di B. La definizione di sottotipaggio strutturale, come inclusione di sottoinsiemi corrispondenti a campi e metodi di una classe, ci rimanda alla definizione di sottotipaggio semantico, con il quale viene abbinato perfettamente. E infatti, nel nostro lavoro integriamo il sottotipaggio semantico e l'utilizzo dei tipi booleani con il sottotipaggio strutturale, sfruttando i benefici di questi due approcci.

Il presente elaborato è costituito da cinque capitoli, i cui contenuti sono descritti nel seguito.

Il primo capitolo descrive il sottotipaggio semantico, dando brevemente una procedura per definirlo partendo dal linguaggio di interesse e dai tipi sintattici. Il resto del capitolo si concentra sulla definizione di questa relazione in due linguaggi: il $\lambda-$calcolo e il $\pi-$calcolo.

Nel secondo capitolo presentiamo il sottotipaggio per classi ed oggetti. Si introduce in dettaglio il calcolo: la sintassi dei termini, i tipi, le regole di tipaggio e la semantica operazionale. Nel seguito, si definisce la relazione di sottotipaggio partendo dall'interpretazione insiemistica dei tipi e introducendo concetti fondamentali per la definizione di questa relazione come: modelli di tipi, modelli ben fondati, tipi come insiemi di valori ecc.

Nel terzo capitolo presentiamo i risultati teorici ottenuti in questo lavoro di tesi. Co-

minciamo con prove di proprietà che valgono per questo sottotipaggio, per concludere con le prove di *type soundness*, dati con i due lemmi standard di *subject reduction* e *progress*.

Il quarto capitolo presenta una discussione sul nostro calcolo. Mettiamo a confronto i due paradigmi di sottotipaggio per i linguaggi ad oggetti: nominale vs. strutturale. Presentiamo una variante *higher-order* del nostro linguaggio e concludiamo con degli esempi che fanno vedere il potere espressivo del linguaggio.

Infine, il quinto capitolo conclude l'elaborato, presentando le conclusioni cui il lavoro è giunto ed i possibili sviluppi futuri.

CAPITOLO 1

Sottotipaggio semantico

1.1 Introduzione al sottotipaggio semantico

Solitamente, i sistemi di tipi si basano su una relazione di sottotipaggio. Si hanno due approcci per definirlo: quello *sintattico* e quello *semantico*.

L'approccio sintattico è più diffuso e consiste nel definire la relazione di sottotipaggio tramite un sistema formale di regole deduttive. Si procede nel seguente modo: si definisce il linguaggio, l'insieme dei tipi sintattici e la relazione di sottotipaggio definita su questo insieme tramite delle regole di inferenza. In questo modo è definito il sottotipaggio in [IPW01], che costituirà il punto di partenza del nostro lavoro. Il sistema di tipi, che consiste nell'insieme dei tipi e nella relazione di sottotipaggio, viene affiancato al linguaggio tramite la relazione di tipaggio, solitamente definito con regole di tipaggio per induzione sui termini del linguaggio, più una regola di sussunzione (*subsumption*) che usa il sottotipaggio.

Nell'approccio semantico, invece, si parte con un modello del linguaggio e un'interpretazione dei tipi come sottoinsiemi di questo modello. Si definisce la relazione di sottotipaggio tra due tipi come l'inclusione degli insiemi corrispondenti a questi tipi. Quando la

relazione di sottotipaggio è decidibile, si sviluppa un'algoritmo di sottotipaggio partendo dalla definizione semantica della relazione.

Il sottotipaggio semantico ha dei vantaggi, come vedremo in seguito, ma dall'altra parte ha delle incombenze rispetto a quello sintattico. Ad esempio, trovare un'interpretazione che mappa i tipi in sottoinsiemi di un modello del linguaggio può essere difficile. Una soluzione a questo problema è stata proposta da Haruo Hosoya e Benjamin Pierce [HP01, HVP05, HP03] nel loro lavoro sul linguaggio XDuce. Questo è un linguaggio XML-*oriented* che trasforma documenti XML in altri documenti XML che soddisfano certe proprietà. L'idea proposta è la seguente: per definire la relazione di sottotipaggio semanticamente, non è necessario partire da un modello dell'intero linguaggio, quello che ci serve è un modello dei tipi. In particolare, loro propongono come modello di tipi l'insieme dei valori del linguaggio, dove i valori di un linguaggio tipato sono tutti i termini ben tipati, chiusi e in forma normale. Quindi, se denotiamo con $\mathcal{V}$ l'insieme dei valori del linguaggio, allora possiamo definire l'interpretazione semantica, o detto diversamente insiemistica, dei tipi nel seguente modo:

$$[\![\tau]\!]_{\mathcal{V}} = \{v \in \mathcal{V} \mid \vdash v : \tau\}$$

questa interpretazione induce la seguente relazione di sottotipaggio:

$$\tau_1 \leq_{\mathcal{V}} \tau_2 \iff [\![\tau_1]\!]_{\mathcal{V}} \subseteq [\![\tau_2]\!]_{\mathcal{V}}$$

Quanto appena stabilito funziona correttamente per il lavoro di Hosoya e Pierce perché l'insieme dei valori che loro considerano può essere definito indipendentemente dalla relazione di tipaggio. In generale, però, per stabilire quando un valore ha un dato tipo ($\vdash v : \tau$), si fa uso della relazione di tipaggio che a sua volta si basa sulla relazione di sottotipaggio. Questo fatto introduce un cerchio: stiamo costruendo un modello per interpretare i tipi e in seguito definire la relazione di sottotipaggio, ma la definizione del modello stesso fa uso della relazione di sottotipaggio che vogliamo stabilire. Per evitare questa dipendenza circolare, e comunque continuare ad interpretare i tipi come insiemi di valori, faremo uso di una tecnica che chiameremo la tecnica del *bootstrap*, come in [FCB08]. Intuitivamente, useremo un *bootstrap model* per definire la relazione di sottotipaggio e per rompere la dipen-

denza circolare che abbiamo precedentemente enunciata. Nel seguito, daremo brevemente i passi per definire la relazione di sottotipaggio semantico.

1.1.1 Sottotipaggio semantico in 5 passi

Il processo di definire il sottotipaggio semantico può essere brevemente ricapitolato nei seguenti 5 passi:

1. Si parte da un insieme di *costruttori* di tipo, ad esempio $\rightarrow, \times, ch \ldots$ che corrispondono rispettivamente, al costruttore di tipi freccia, al costruttore di tipi prodotto e ai canali nel $\pi-$calcolo, che introdurremo nella sezione 1.3. Si estende l'algebra dei tipi con le costanti, il vero $\mathbf{1}$, il falso $\mathbf{0}$ e i *connettivi booleani*: unione $\vee$, intersezione $\wedge$ e negazione $\neg$, dando luogo all'algebra dei tipi $\mathcal{T}$.

2. Si dà un *modello insiemistico* dei tipi, ovvero si definisce una funzione $[\![]\!]_D : \mathcal{T} \rightarrow \mathcal{P}(D)$, per un qualche insieme D. I connettivi booleani devono essere interpretati nel modo ovvio: $[\![\tau_1 \vee \tau_2]\!]_D = [\![\tau_1]\!]_D \cup [\![\tau_2]\!]_D$, $[\![\tau_1 \wedge \tau_2]\!]_D = [\![\tau_1]\!]_D \cap [\![\tau_2]\!]_D$ e $[\![\neg\tau]\!]_D = D \setminus [\![\tau]\!]_D$. Si noti che si possono avere diversi modelli, e ognuno di essi induce una relazione di sottotipaggio nell'algebra dei tipi. Per quanto riguarda i nostri obbiettivi, siamo interessati a far vedere che esiste almeno un tale modello; ne scegliamo uno, che chiameremo *bootstrap model*, e lo denotiamo con $[\![]\!]_{\mathcal{B}}$. La relazione di sottotipaggio indotta è come segue:

$$\tau_1 \leq_{\mathcal{B}} \tau_2 \stackrel{def}{\iff} [\![\tau_1]\!]_{\mathcal{B}} \subseteq [\![\tau_2]\!]_{\mathcal{B}}$$

   Questo sottotipaggio $\leq$ induce una relazione di equivalenza tra tipi, denotata con $\simeq$, come segue: $\tau_1 \simeq \tau_2 \iff \tau_1 \leq \tau_2 \wedge \tau_2 \leq \tau_1$.

3. Adesso che abbiamo definito la relazione di sottotipaggio, possiamo specificare un algoritmo di sottotipaggio che decide la relazione. Notiamo che questo passo non è obbligatorio nel nostro processo, ma è molto utile per utilizzare i nostri tipi in pratica.

4. A questo punto, ci possiamo concentrare sul linguaggio che vogliamo studiare. Consideriamo le regole di tipaggio e utilizziamo la relazione di sottotipaggio appena

definita per tipare i termini del linguaggio e derivare $\Gamma \vdash_{\mathcal{B}} e : \tau$. In particolare, utilizziamo la relazione di sottotipaggio $\leq_{\mathcal{B}}$ nella regola di sussunzione (*subsum*).

5. Le derivazioni di tipo ci permettono di dare una nuova interpretazione insiemistica dei tipi, ovvero, quella basata sui valori $[\![\tau]\!]_{\mathcal{V}} = \{v \in \mathcal{V} \mid \vdash_{\mathcal{B}} v : \tau\}$. Questa interpretazione induce una *nuova* relazione di sottotipaggio $\tau_1 \leq_{\mathcal{V}} \tau_2 \overset{def}{\iff} [\![\tau_1]\!]_{\mathcal{V}} \subseteq [\![\tau_2]\!]_{\mathcal{V}}$. Questa relazione $\leq_{\mathcal{V}}$ può essere diversa da quella da cui siamo partiti $\leq_{\mathcal{B}}$. Comunque, se le definizioni di modello, del linguaggio e le regole di tipaggio sono state scelte propriamente, queste due relazioni coincidono:

$$\tau_1 \leq_{\mathcal{B}} \tau_2 \iff \tau_1 \leq_{\mathcal{V}} \tau_2$$

   e questo chiude il cerchio di cui abbiamo parlato all'inizio della sezione. Quanto detto, non implica che le definizioni sono 'valide', ma semplicemente che sono mutuamente coerenti. Infatti, dobbiamo dimostrare i risultati di correttezza di tipi (*type soundness*). Nella nostra trattazione, faremo questo provando *subject reduction* e *progress*.

I passi appena introdotti danno un modo elegante per presentare il sottotipaggio semantico. In realtà, nello sviluppare questa teoria, il punto di partenza non è il modello dei tipi, ma il calcolo: uno comincia con il calcolo di interesse e i valori di questo calcolo e poi cerca di modificare i concetti in modo da formalizzarli e adattarli al sottotipaggio che verrà definito.

1.1.2 Vantaggi del sottotipaggio semantico

Il sottotipaggio semantico ha avuto meno attenzione rispetto a quello sintattico, in quanto è più difficile tecnicamente. Però, ha una serie di vantaggi:

1. Quando i costruttori di tipi hanno un'interpretazione naturale nel modello, allora il sottotipaggio semantico, dato come inclusione di sottoinsiemi, è completo per definizione ovvero, se $\tau_1 \not\leq \tau_2$ allora è possibile esibire un elemento dell'insieme corrispondente all'interpretazione di τ_2 che non appartiene all'insieme corrispondente a τ_1. Nell'approccio sintattico, invece, si può solo dire che il sistema formale non

prova che $\tau_1 \leq \tau_2$, ma non è chiaro quale regola formale ha reso possibile questo fatto. Questo argomento è particolarmente importante in un'algebra dei tipi ricca con diversi costruttori di tipi e connettivi booleani che interagiscono in maniera non banale. Nel sottotipaggio sintattico si devono introdurre delle regole, ad esempio, per le proprietà distributive come $(\tau_1 \vee \tau_2) \to \tau \simeq (\tau_1 \to \tau) \wedge (\tau_2 \to \tau)$. Dimenticare una di queste regole, ci dà un sistema di tipi che è corretto ma non completo rispetto alla semantica dei tipi.

2. Nell'approccio sintattico, derivare un algoritmo di sottotipaggio richiede una forte intuizione della relazione che è definita da un sistema formale, mentre nell'approccio semantico questo è una questione di 'aritmetica', in quanto basta semplicemente usare l'interpretazione insiemistica dei tipi e usare proprietà note dei connettivi booleani per decomporre i tipi e considerare il sottotipaggio di tipi più semplici, come vedremo in 3.2.

3. L'approccio sintattico richiede dimostrazioni complicate di proprietà formali che nell'approccio semantico sono una conseguenza della definizione del sottotipaggio; ad esempio, la proprietà transitiva è banale, in quanto l'inclusione di insiemi è transitiva. Questo fatto rende le dimostrazioni formali di queste proprietà inutili.

1.2 Il $\lambda-$ calcolo semantico

Come primo esempio di applicazione dell'approccio semantico al sottotipaggio, consideriamo il $\lambda-$ calcolo con i prodotti. Il lavoro teorico fatto su questo calcolo, presentato in [CF05, Cas05, FCB08], ha costituito le basi fondamentali per la definizione e l'implementazione di un linguaggio *general-purpose*, con funzioni di ordine superiore che lavora su documenti XML (i così detti XML *transformation language*), chiamato $\mathbb{C}$Duce . Cominciamo col presentare la sintassi del linguaggio.

1.2.1 Introduzione al calcolo

In questa sezione definiremo formalmente la sintassi dei tipi e delle espressioni del calcolo, per concludere con le regole di tipaggio.

Tipi. Sono prodotti dalla seguente grammatica:

$$t ::= \mathbf{0} \mid b \mid t \times t \mid t \to t \mid t \wedge t \mid t \vee t \mid \neg t$$

dove $\mathbf{0}$ corrisponde al tipo vuoto, b corrisponde ai tipi base: interi, reali ecc. $\times$ e $\to$ sono i costruttori di tipi prodotto e tipi funzionali, rispettivamente. Inoltre, l'algebra dei tipi viene aumentata con i tipi booleani che sono formati usando i connettivi booleani $\wedge, \vee$ e $\neg$ che hanno il loro significato intuitivo. In questo calcolo si considerano anche i tipi ricorsivi; quindi, i tipi sono gli alberi regolari potenzialmente infiniti generati dalla grammatica appena presentata e che soddisfanno la proprietà che ogni ramo infinito dell'albero ha un numero infinito di occorrenze di costruttori $\times$ e $\to$. Notiamo che questa proprietà esclude che espressioni della forma $(t \wedge (t \wedge (t \wedge (\ldots))))$, che non hanno senso, siano chiamati tipi. Questo stesso argomento lo vedremo anche nel seguito quando presenteremo il nostro calcolo ad oggetti.

Termini. Per definire i termi, scegliamo un insieme di costanti C e usiamo la meta-variabile c per denotare un elemento di questo insieme. Usiamo x per denotare una variabile. I termini del linguaggio sono detti espressioni e sono prodotti dalla seguente grammatica:

$$\begin{aligned} e ::= {} & c \mid x \mid (e, e) \mid \pi_i(e) \mid \mu f(t \to t; \ldots; t \to t).\lambda x.e \\ & \mid ee \mid (x = e \in t \,?\, e \mid e) \mid rnd(t) \end{aligned}$$

Tra le espressioni si hanno le coppie di espressioni (e, e) e le proiezioni $\pi_i(e)$ per $i = 1, 2$ che ci danno la prima e la seconda proiezione, rispettivamente. Si ha un costrutto esplicito per le funzioni ricorsive $\mu f(t \to t; \ldots; t \to t).\lambda x.e$ che combina una $\lambda-$astrazione con un operatore di punto fisso. Gli identificatori f e x sono legati (*bind*) all'interno del corpo della funzione. Questa $\lambda-$astrazione è accompagnata da una sequenza non vuota di tipi funzionali che vengono chiamati *interfaccia* della funzione: se più di un tipo funzionale è presente nell'interfaccia, allora siamo in presenza di una funzione *overloaded*. Da quanto abbiamo visto, per le $\lambda-$astrazioni si adotta uno stile *a là Church*: i tipi assegnati a queste espressioni si vedono nella segnatura, senza dover considerare il corpo della funzione.

Siccome stiamo considerando una variante del $\lambda-$calcolo, tra le espressioni si hanno anche le applicazioni. Inoltre, si ha un costrutto di *dynamic type dispatch* $x = e \in t\,?\,e \mid e$ che funziona come segue: se il tipo dell'espressione assegnata alla variabile x è t, allora si esegue il primo ramo, altrimenti il secondo. Concludendo, si ha un costrutto $rnd(t)$ che corrisponde alla scelta casuale di un'espressione arbitraria di tipo t; questo costrutto lo incontreremo anche nel nostro calcolo. A questo punto, dati i termini, possiamo introdurre i valori del linguaggio.

Definizione 1. *Un'espressione e è un* valore *se è chiusa (senza variabili libere), ben tipata ($\vdash e : t$ per qualche t) e prodotta dalla seguente grammatica:*

$$v ::= \; c \mid (v, v) \mid \mu f(\ldots).\lambda x.e$$

Regole di tipaggio. Per concludere la presentazione del $\lambda-$calcolo, diamo adesso le regole di tipaggio:

$$\frac{\Gamma \vdash e : t_1 \quad t_1 \leq t_2}{\Gamma \vdash e : t_2}\,(subsum) \qquad \frac{}{\Gamma \vdash c : b_c}\,(const) \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\,(var)$$

$$\frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash (e_1, e_2) : t_1 \times t_2}\,(pair) \qquad \frac{\Gamma \vdash e : t_1 \times t_2}{\Gamma \vdash \pi_i(e) : t_i}\,(proj)$$

$$\frac{\Gamma \vdash e_1 : t_1 \to t_2 \quad \Gamma \vdash e_2 : t_1}{\Gamma \vdash e_1 e_2 : t_2}\,(appl) \qquad \frac{}{\Gamma \vdash rnd(t) : t}\,(rnd)$$

$$\frac{\begin{array}{c} t = \bigwedge_{i=1\ldots n}(t_i \to s_i) \wedge \bigwedge_{j=1\ldots m} \neg(t'_j \to s'_j) \not\simeq \mathbf{0} \\ \forall i = 1 \ldots n,\ \Gamma, (f : t), (x : t_i) \vdash e : s_i \end{array}}{\Gamma \vdash \mu f(t_1 \to s_1; \ldots; t_n \to s_n).\lambda x.e : t}\,(abstr)$$

$$\frac{\Gamma \vdash e : t_0 \quad \begin{cases} t_0 \not\leq \neg t \Rightarrow \Gamma, (x : t_0 \wedge t) \vdash e_1 : s \\ t_0 \not\leq t \Rightarrow \Gamma, (x : t_0 \backslash t) \vdash e_2 : s \end{cases}}{\Gamma \vdash (x = e \in t\,?\,e_1 \mid e_2) : s}\,(case)$$

La regola (*subsum*) usa la relazione di sottotipaggio definito semanticamente, come vedremo in seguito, e fa sì che il sistema dei tipi dipenda da questa relazione. Le altre regole, tranne le ultime due, sono standard e semplici da capire. Ci concentriamo adesso sulla regola (*abstr*). L'interfaccia dell'astrazione contiene i tipi freccia $t_i \to s_i$ che sono delle restrizioni da rispettare, ovvero, sotto la condizione che la variabile x ha tipo t_i, il corpo della funzione ha tipo s_i. Inoltre, la variabile f ha tipo t, che dall'altra parte è il tipo assegnato all'intera funzione. Intuitivamente, il tipo deve essere l'intersezione dei tipi $t_i \to s_i$. Però, a questa intersezione si aggiunge un numero finito di tipi freccia complementati, a patto che il tipo t non diventi vuoto. La ragione per questa scelta è semplice: l'obbiettivo finale di questo studio è quello di interpretare i tipi come insiemi di valori in modo che i connettivi booleani hanno un'interpretazione insiemistica intuitiva. In particolare, l'unione di t e $\neg t$ deve essere equivalente ad $\mathbf{1}$; questo vuol dire che deve valere la seguente proprietà: $\forall v.\ \forall t.\ (\vdash v : t) \vee (\vdash v : \neg t)$ e siccome un'astrazione chiusa e ben tipata è un valore in questo calcolo, deve avere tipo $t \to s$ oppure $\neg(t \to s)$ per ogni tipo t ed s. Se il tipo $t \to s$ è un supertipo di $\bigwedge_{i=1\ldots n}(t_i \to s_i)$, allora usando la regola (*subsum*), si può derivare il tipo $t \to s$ per l'astrazione. Altrimenti, l'astrazione deve avere tipo $\neg(t \to s)$, e siccome non possiamo usare la sussunzione per inferire questo tipo, ci serve un modo alternativo, ovvero, aggiungendo tipi funzionali complementati al tipo t dell'astrazione.

Infine, commentiamo la regola (*case*). L'espressione e ha tipo t_0. Se questo tipo ha un'intersezione non vuota con il tipo t ($t_0 \not\leq \neg t$), allora si considera il primo ramo. Affinché l'intera espressione abbia tipo s, si controlla che e_1 ha tipo s sotto la condizione che x ha tipo $t_0 \wedge t$. Se viene soddisfatta la seconda condizione ($t_0 \not\leq t$), allora si considera il secondo ramo.

Da notare che questo costrutto, combinato con l'astrazione, dà luogo alle funzioni overloaded. Ad esempio, consideriamo l'astrazione $\mu f(b_1 \to b_1; b_2 \to b_2).\lambda x.(x \in b_1\ ?\ c_1 \mid c_2)$, con b_1 e b_2 due tipi base e c_1 e c_2 due costanti di tipo b_1 e b_2, rispettivamente. Se x ha tipo b_1 si esegue il ramo corrispondente a c_1 altrimenti, se x ha tipo b_2, si esegue il ramo di c_2. Questo fatto rispetta la caratteristica delle funzioni overloaded che, diversamente dalle funzioni polimorfe che lavorano su argomenti di tipi diversi eseguendo sempre lo stesso corpo di funzione, lavorano su argomenti di tipi diversi e ad ogni tipo di argomento corrisponde

un corpo di funzione da eseguire diverso.

1.2.2 Definizione del sottotipaggio semantico

Nella sezione precedente abbiamo introdotto una variante del $\lambda-$calcolo. Dati i tipi, in questa sezione consideriamo precisamente quali sono le caratteristiche di un'interpretazione insiemistica che deve, come prima cosa, rispettare la natura dei connettivi booleani. Sia $\mathcal{T}$ l'insieme dei tipi, D qualche insieme, e $[\![]\!]$ una funzione di interpretazione, ovvero, $[\![]\!] : \mathcal{T} \to \mathcal{P}(D)$. Le condizioni che $[\![]\!]$ deve soddisfare sono le seguenti:

$$
\begin{aligned}
&[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \\
&[\![t_1 \wedge t_2]\!] = [\![t_1]\!] \cap [\![t_2]\!] \\
&[\![\neg t]\!] = D \setminus [\![t]\!] \\
&[\![\mathbf{1}]\!] = D \\
&[\![\mathbf{0}]\!] = \varnothing \\
&[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!] \\
&[\![t \to s]\!] = ???
\end{aligned}
$$

Le prime sei condizioni rispondono all'intuizione che l'interpretazione deve essere insiemistica. A questo punto, si deve stabilire l'ultima condizione. Questa è più complicata delle altre ed, inoltre, dipende dal linguaggio e dalle funzioni che si vogliono implementare nel linguaggio. L'intuizione che si ha per gli spazi funzionali è la seguente: una funzione è di tipo $t \to s$ se, ogniqualvolta che si applica ad un valore di tipo t, restituisce un risultato di tipo s. Se interpretiamo le funzioni come relazioni binarie in D, allora $[\![t \to s]\!]$ è l'insieme delle relazioni binare nelle quali, se la prima proiezione sta nell'interpretazione di t, allora la seconda proiezione sta nell'interpretazione di s, ovvero:

$$\{f \subseteq D^2 \mid \forall (d_1, d_2) \in f.\ d_1 \in [\![t]\!] \Rightarrow d_2 \in [\![s]\!]\}$$

Notiamo che questo insieme si può anche scrivere come $\mathcal{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}})$, dove le linee indicano il complemento dell'insieme in questione. Quindi, vorremmo scrivere:

$$[\![t \to s]\!] = \mathcal{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}})$$

Quanto appena affermato è senza senso, in quanto si dovrebbe avere $\mathcal{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}}) \subseteq D$; ma questa inclusione è impossibile per ragioni di cardinalità. Dall'altra parte, l'obbiettivo di questa trattazione è quello di stabilire la relazione di sottotipaggio, quindi di capire le relazioni che gli elementi di D creano tra di loro. Detto diversamente, non si definisce l'interpretazione dei tipi per stabilire formalmente il significato dei tipi, ma semplicemente per stabilire le loro relazioni. Quindi, anche se non è possibile definire $[\![t \to s]\!]$ come $\mathcal{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}})$, per quanto riguarda i nostri obbiettivi è sufficiente richiedere che un'interpretazione insiemistica deve interpretare i tipi funzionali in maniera tale che la relazione di sottotipaggio che essa induce è la stessa di quella che gli insiemi delle parti indurrebbero, ovvero:

$$[\![t_1 \to s_1]\!] \subseteq [\![t_2 \to s_2]\!] \iff \mathcal{P}(\overline{[\![t_1]\!] \times \overline{[\![s_1]\!]}}) \subseteq \mathcal{P}(\overline{[\![t_2]\!] \times \overline{[\![s_2]\!]}})$$

Per stabilire formalmente questa equivalenza, si associa all'interpretazione insiemistica $[\![]\!]$ un'altra interpretazione insiemistica, detta estensionale $\mathbb{E}()$, che si comporta come $[\![]\!]$ tranne per i tipi funzionali, per i quali usiamo la condizione sopra come definizione:

$$\mathbb{E}(t \to s) = \mathcal{P}(\overline{[\![t]\!] \times \overline{[\![s]\!]}})$$

A questo punto, l'ultima condizione la si può scrivere come:

$$[\![t]\!] = \varnothing \iff \mathbb{E}(t) = \varnothing$$

che è equivalente a $[\![t_1]\!] \subseteq [\![t_2]\!] \iff \mathbb{E}(t_1) \subseteq \mathbb{E}(t_2)$, in quanto $[\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!] \setminus [\![t_2]\!] = \varnothing \iff [\![t_1 \wedge \neg t_2]\!] = \varnothing \iff \mathbb{E}(t_1 \wedge \neg t_2) = \varnothing \iff \mathbb{E}(t_1) \setminus \mathbb{E}(t_2) = \varnothing \iff \mathbb{E}(t_1) \subseteq \mathbb{E}(t_2)$. Questa condizione corrisponde alla nozione di modello di tipi. Nella sezione 2.2, quando presenteremo il nostro semplice calcolo ad oggetti, daremo con più precisione

le definizioni di interpretazione insiemistica, interpretazione estensionale e modello. Per concludere questa sezione, notiamo che la condizione per i tipi funzionali permette di avere nel calcolo funzioni che sono:

- *Non-deterministiche:* in quanto la condizione non impedisce che nell'interpretazione di un tipo funzionale sia inclusa una relazione con due coppie (d, d_1) e (d, d_2) con $d_1 \neq d_2$.

- *Non-terminanti:* ogni tipo funzionale è abitato, in quanto la relazione vuota appartiene all'interpretazione di ogni tale tipo. Quindi, anche il tipo $t \to \mathbf{0}$ per un qualsiasi t è abitato; ogni funzione di questo tipo non termina, perché se terminasse dovrebbe restituire un risultato di tipo $\mathbf{0}$, che non è possibile.

- *Overloaded:* in questo linguaggio, con il termine overloaded si intendono funzioni che possono essere applicate ad argomenti di tipi diversi e i tipi di ritorno dipendono dal tipo dell'argomento. Inoltre, come abbiamo già detto precedentemente, in combinazione con il costrutto type-case si eseguono anche espressioni diverse corrispondenti a tipi di argomenti diversi.

1.2.3 Chiudere il cerchio

Una volta stabilite le caratteristiche di un'interpretazione insiemistica e introdotta la nozione di modello, uno può concentrarsi su un particolare insieme $\mathcal{B}$ e $[\![]\!]_{\mathcal{B}}$ e da questo definire il resto. Supponiamo che esista una coppia $(\mathcal{B}, [\![]\!]_{\mathcal{B}})$ (in seguito di questa trattazione proveremo che almeno una tale coppia esiste); chiamiamola *bootstrap model*. Questo modello definisce una relazione di sottotipaggio in $\mathcal{T}$:

$$t \leq_{\mathcal{B}} s \iff [\![t]\!]_{\mathcal{B}} \subseteq [\![s]\!]_{\mathcal{B}}$$

Questo sottotipaggio verrà utilizzato nella regola (*subsum*) che insieme alle altre regole di tipaggio derivano giudizi di tipo (*typing judgement*) della forma $\Gamma \vdash e : t$, con e un'espressione del calcolo. Tra le espressioni, in particolare ci stanno anche i valori e quindi, per ogni valore è possibile derivare il suo tipo, ovvero $\vdash v : t$. Come abbiamo già detto

all'inizio del capitolo, queste derivazioni definiscono una nuova interpretazione, quella come insiemi di valori: $[\![t]\!]_{\mathcal{V}} = \{v \in \mathcal{V} \mid \vdash v : t\}$. Questa nuova interpretazione induce una 'nuova' relazione di sottotipaggio:

$$t \leq_{\mathcal{V}} s \iff [\![t]\!]_{\mathcal{V}} \subseteq [\![s]\!]_{\mathcal{V}}$$

A questo punto, possiamo ricominciare tutto, ovvero usare questa relazione di sottotipaggio nella regola di (*subsum*), derivare nuovi giudizi di tipi e un'altra ancora interpretazione dei tipi ecc. ecc. Tutto questo non è necessario, in quanto si può provare che le due relazioni di sottotipaggio coincidono, ovvero:

$$t \leq_{\mathcal{B}} s \iff t \leq_{\mathcal{V}} s \tag{1.1}$$

Questo è uno dei risultati cruciali di [FCB08]. Quindi, abbiamo chiuso il cerchio!

Abbiamo descritto in modo dettagliato il $\lambda-$calcolo e tutto lo studio del sottotipaggio semantico in quanto questo linguaggio è stato il nostro punto di partenza per lo sviluppo di questa teoria nel nostro semplice calcolo ad oggetti. Nella sezione che segue, per completezza, vogliamo introdurre in maniera più concisa il $\pi-$calcolo e lo studio del sottotipaggio semantico in questo linguaggio.

1.3 Il $\pi-$calcolo semantico

Per concludere questo primo capitolo, in questa sezione vogliamo presentare il sottotipaggio semantico applicato al $\pi-$calcolo. Lo faremo in modo sintetico, tralasciando i dettagli tecnici. Il $\pi-$calcolo è un modello di computazione di sistemi concorrenti. Esiste una vasta letteratura per questo linguaggio, possiamo citare, ad esempio [Mil99, SW03]. Le nozioni basilari sono quelle di *processo* e *canale*, che è un'astrazione di una linea di comunicazione (*communication link*) tra due processi. Nella seguente sezione daremo una breve introduzione al calcolo in modo da prepararci per la definizione del sottotipaggio del quale ci occuperemo nella sezione successiva.

1.3.1 Introduzione al calcolo

Tipi. Sono prodotti dalla seguente grammatica:

$$t ::= \; b \mid \mathbf{0} \mid \mathbf{1} \mid ch^{+}(t) \mid ch^{-}(t) \mid \neg t \mid t_1 \vee t_2 \mid t_1 \wedge t_2$$

Si noti che in questo linguaggio non si hanno i tipi ricorsivi. Seguono pochi commenti: $ch^{+}(t)$ è il tipo di un canale dal quale si possono ricevere valori di tipo t, $ch^{-}(t)$ è il tipo di un canale sul quale si possono inviare valori di tipo t, mentre $ch(t)$ è il tipo di un canale nel quale è possibile sia inviare che ricevere valori di tipo t. Sostanzialmente, questo è zucchero sintattico per $ch^{+}(t) \wedge ch^{-}(t)$.

Termini. Affinché il sottotipaggio semantico si adatti bene al linguaggio, il $\pi-$calcolo che andremo a definire riporta delle modifiche rispetto a quello standard, appunto per supportare tutte le novità che derivano dai tipi booleani, il sistema dei tipi e le varie inclusioni di tipi. Quanto affermato è argomentato in dettaglio in [Cas05, CNV08]. Chiameremo questo linguaggio $\mathbb{C}\pi$-calcolo. La sintassi del $\mathbb{C}\pi$-calcolo è simile a quella del $\pi-$calcolo asincrono, che è una variante del $\pi-$calcolo dove l'invio dei messaggi è non bloccante (*non-blocking*). Presentiamo in seguito la sua sintassi:

Canali	$\alpha ::= x$		variabile
		$\mid c^{t}$	costante di canale

Messaggi	$M ::= n$		costante base
		$\mid \alpha$	canale

Processi	$P ::= \overline{\alpha} M$		output
		$\mid \sum_{i \in I} \alpha(x : t) P_i$	input con condizioni sul tipo
		$\mid P_1 \| P_2$	composizione parallela
		$\mid (\nu c^t) P$	restrizione
		$\mid\ !P$	replicazione

Nella definizione sopra riportata, I è un insieme finito, eventualmente vuoto, di indici. Come di consueto, useremo la convenzione che la somma con $I = \varnothing$ è il processo *inerte*, denotato con 0. Dati i termini, possiamo a questo punto introdurre i valori. Sono generati dalla seguente grammatica:

$$v ::= \ n \mid c^t$$

I valori sono messaggi chiusi, ovvero, sono le costanti del linguaggio. Denotiamo con $\mathcal{V}$ l'insieme dei valori.

Dopo questa breve introduzione alla sintassi dei tipi e dei termini del linguaggio, ci concentreremo nel seguito nella definizione del sottotipaggio semantico per $\mathbb{C}\pi$.

1.3.2 Sottotipaggio semantico per $\mathbb{C}\pi$

L'intuizione insiemistica che abbiamo dei tipi di questo calcolo è quella che i tipi indicano insiemi di canali. Dall'altra parte, un canale può essere visto come una scatola (*box*) che può trasportare oggetti di un qualche tipo t; quindi, il concetto di canale è legato strettamente a quello del tipo di oggetti che trasporta. Ad esempio, $ch(t)$ è l'insieme di tutte le scatole che trasportano oggetti di tipo t. Quanto detto suggerisce la seguente interpretazione:

$$\begin{aligned}
[\![ch(t)]\!] &= \{c \mid \text{c è una scatola per oggetti in } [\![t]\!]\} \\
[\![ch^+(t)]\!] &= \{c \mid \text{c è una scatola per oggetti in } [\![s]\!] \text{ tale che } s \leq t\} \\
[\![ch^-(t)]\!] &= \{c \mid \text{c è una scatola per oggetti in } [\![s]\!] \text{ tale che } s \geq t\}
\end{aligned}$$

Data l'interpretazione insiemistica, dal punto di vista dei tipi, tutte le scatole c di un dato tipo t sono praticamente indistinguibili, in quanto o tutte appartengono all'interpretazione di t o nessuna vi appartiene. Questo implica che la relazione di sottotipaggio non viene influenzata dal numero di scatole di un certo tipo. Quindi, possiamo assumere che per ogni classe di equivalenza di tipi esiste una sola scatola che può essere semplicemente identificata con $[\![t]\!]$; abbiamo quanto segue:

$$[\![ch^+(t)]\!] = \{[\![s]\!] \mid s \leq t\} \qquad [\![ch^+(t)]\!] = \{[\![s]\!] \mid s \geq t\}$$

mentre il tipo di canale $ch(t)$ viene interpretato come $\{[\![t]\!]\}$. Notiamo che, nelle definizioni appena date c'è una circolarità in quanto viene utilizzata la relazione di sottotipaggio che è ancora da definire e per definirla siamo partiti dall'interpretazione insiemistica dei tipi sopra riportata. Quindi, per risolvere il problema invece di $s \leq t$ (rispettivamente, $s \geq t$) possiamo scrivere $[\![s]\!] \subseteq [\![t]\!]$ (rispettivamente, $[\![s]\!] \supseteq [\![t]\!]$).

Una volta definita l'interpretazione insiemistica dei tipi, si possono dedurre delle relazioni interessanti, come ad esempio quella presentata precedentemente:

$$ch(t) = ch^+(t) \wedge ch^-(t)$$

Oppure,

$$ch^-(s) \wedge ch^-(t) = ch^-(s \vee t)$$

$$ch^-(s) \vee ch^-(t) \leq ch^-(s \vee t)$$

$$ch^+(s) \vee ch^+(t) \leq ch^+(s \vee t)$$

e altre relazioni come possiamo vedere in [Cas05, CNV08].

A questo punto, come abbiamo fatto esattamente per il $\lambda-$calcolo, passiamo dall'interpretazione insiemistica, data precedentemente, all'interpretazione dei tipi come insiemi di valori e verifichiamo se queste due interpretazioni inducono la stessa relazione di sottotipaggio. Come ci aspettavamo, le cose stanno proprio così. In [CNV08] viene dimostrato che il *bootstrap model* e il modello dei valori induco la stessa relazione di sottotipaggio.

Questo chiude il cerchio che abbiamo introdotto.

Data questa panoramica del sottotipaggio semantico in letteratura, passando da XDuce a $\mathbb{C}$Duce per concludere con $\mathbb{C}\pi$, nel prossimo capitolo introdurremo un semplice linguaggio ad oggetti e definiremo per questo linguaggio la relazione di sottotipaggio dando anche delle proprietà e caratteristiche di questa relazione.

CAPITOLO 2

Sottotipaggio semantico per un semplice linguaggio ad oggetti

2.1 Il calcolo

In questa sezione presenteremo il nostro calcolo. Introdurremo i tipi, sui quali concentreremo la maggior parte del nostro studio, i termini del linguaggio, e poi, per assegnare tipi ai termini, daremo le regole di tipaggio. Concluderemo con la semantica operazionale.

2.1.1 I tipi

I tipi nel nostro calcolo sono prodotti dalla seguente sintassi:

$$\alpha ::= \mathbf{0} \mid \mathbb{B} \mid [\widetilde{l : \tau}] \mid \alpha \wedge \alpha \mid \neg\alpha$$
$$\mu ::= \alpha \to \alpha \mid \mu \wedge \mu \mid \neg\mu$$
$$\tau ::= \alpha \mid \mu$$

Chiameremo tipi α e tipi μ quelli prodotti, rispettivamente, da α e μ e quelli prodotti da τ li chiameremo semplicemente tipi. Indichiamo con $\mathcal{T}$ l'insieme dei tipi.

Il tipo $\mathbf{0}$ denota il tipo vuoto. Il tipo $\mathbb{B}$ denota i tipi base: interi, reali, booleani ecc. Il tipo $[\widetilde{l : \tau}]$ denota un tipo *record*. Indichiamo con $\widetilde{\cdot}$ una sequenza, eventualmente vuota, di elementi di tipo '$\cdot$'. Ovvero, $\widetilde{l : \tau}$ indica la sequenza $l_1 : \tau_1, \ldots, l_k : \tau_k$ per qualche $k \geq 0$. Quindi, un tipo record è una sequenza di etichette, appartenenti ad un insieme L infinito numerabile, associate a tipi. Quando sarà necessario, scriveremo un tipo record nel seguente modo: $[\widetilde{a : \alpha}, \widetilde{m : \mu}]$; questo per mettere in evidenza gli attributi (o campi) del record, denotati con le etichette $\widetilde{a}$, e i metodi del record, denotati con le etichette $\widetilde{m}$. Dato un tipo $\rho = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$ (ρ per record), indichiamo con $\rho(a_i)$ il tipo corrispondente all'attributo a_i e con $\rho(m_j)$ il tipo corrispondente al metodo m_j. In ogni tipo record deve valere $a_i \neq a_j$ per $i \neq j$ e $m_h \neq m_k$ per $h \neq k$.

Tra i tipi α appena descritti, non ci sono tipi funzionali (i così detti 'tipi freccia'). Siccome questi tipi sono necessari per tipare i metodi nel nostro calcolo, essi sono presentati da una categoria sintattica a parte, ovvero, da μ.

I connettivi booleani $\wedge$ e $\neg$ hanno il loro significato classico. Denotiamo con $\mathbf{1}$ il tipo $\neg\mathbf{0}$ che corrisponde al tipo universale. Useremo l'abbreviazione $\alpha \backslash \alpha'$ per $\alpha \wedge \neg\alpha'$ e useremo $\alpha \vee \alpha'$ per $\neg(\neg\alpha \wedge \neg\alpha')$. Lo stesso vale anche per i tipi μ.

Chiameremo tipi *atomici*, oppure semplicemente atomi, i tipi base $\mathbb{B}$, i tipi record $[\widetilde{l : \tau}]$ e i tipi funzionali $\alpha \to \alpha$. Questi sono gli atomi per i connettivi booleani.

Nel nostro insieme di tipi sono ammessi gli *alberi infiniti regolari* prodotti dalla sintassi dei tipi α e μ. Con regolare intendiamo che un albero, potenzialmente infinito, ha un numero finito di sottoalberi diversi (cioè, non isomorfi). Per semplicità di scrittura, useremo sistemi ricorsivi di tipi. Inoltre, siccome vogliamo che i tipi denotino insiemi, imporremo delle restrizioni per evitare tipi mal formati, come ad esempio: la soluzione di $\alpha = \alpha \wedge \alpha$ (rispettivamente $\mu = \mu \wedge \mu$) che non contiene nessun'informazione sull'insieme denotato da α (rispettivamente μ); oppure $\alpha = \neg\alpha$ ($\mu = \neg\mu$) che non può rappresentare nessun insieme. Quindi, chiameremo tipi solo quelli ben formati. Ovvero, diremo che un albero generato dalla sintassi α e μ è un tipo se non contiene nessun ramo infinito senza un atomo. Per

quanto appena detto, la relazione $\triangleright \subseteq \alpha^2$ definita come:

$$\alpha_1 \wedge \alpha_2 \triangleright \alpha_1$$
$$\alpha_1 \wedge \alpha_2 \triangleright \alpha_2$$
$$\neg\alpha \triangleright \alpha$$

non contiene catene infinite. Analogamente per i tipi μ. Questa relazione la utilizzeremo nella definizione di modelli ben fondati che introdurremo nella sezione 2.2.3.

2.1.2 I termini

Per definire i termini del nostro linguaggio ci siamo basati su *Featherweight Java* [IPW01]. FJ è un calcolo minimale. C'è una corrispondenza tra questo calcolo e la parte puramente funzionale di Java, nel senso che ogni programma in FJ è letteralmente un programma eseguibile di Java, come si afferma in [IPW01]. FJ ha una relazione con Java simile alla relazione che ha il lambda calcolo di Church [Bar84] con linguaggi funzionali come ML e Haskell. FJ è un calcolo leggermente più grande del calcolo ad oggetti di Abadi e Cardelli [AC94]. Una semplificazione fondamentale in FJ è la mancanza di assegnazione (e da qui l'uso del termine 'funzionale'). Sostanzialmente, i campi e i parametri dei metodi sono implicitamente marcati come **final**: assumiamo che i campi di un oggetto sono inizializzati dal suo costruttore e non vengono più modificati; detto diversamente, sono '*read only*'. Comunque, questo frammento è computazionalmente completo (si può codificare il lambda calcolo in FJ).

Passiamo adesso a considerare il nostro linguaggio. La sintassi è sostanzialmente la stessa di FJ, le differenze principali sono la mancanza del costrutto *cast* e l'utilizzo di tipi strutturali invece che di tipi nominali (parleremo di questi tipi in dettaglio nella sezione 4.1). Assumiamo un insieme numerabile di nomi, tra i quali abbiamo dei nomi riservati: $Object$, per indicare la classe situata alla radice della gerarchia delle classi, **this** per indicare la classe corrente e **super** per indicare la classe padre. Useremo le lettere $A, B, C, \ldots$ per indicare le classi, $a, b, \ldots$ per i campi, $m, n, \ldots$ per i metodi e $x, y, z, \ldots$ per indicare le variabili. Denoteremo con C l'insieme delle costanti del calcolo e useremo la meta-variabile

c per indicare gli elementi di questo insieme. Generalmente, soprattutto negli esempi, useremo nomi mnemonici per indicare classi, metodi ecc. Ad esempio, Point, print, val ecc. La sintassi del linguaggio è la seguente:

Dichiarazione di classi	$L ::= \textbf{class}\ C\ \textbf{extends}\ C\ \{\widetilde{\alpha\ a};\ K\ \widetilde{M}\}$
Costruttore	$K ::= C(\widetilde{\beta\ b}; \widetilde{\alpha\ a})\{\textbf{super}(\widetilde{b}); \textbf{this}.\widetilde{a} = \widetilde{a};\}$
Dichiarazione di metodi:	$M ::= \alpha\ m(\alpha\ a)\{\textbf{return}\ e;\}$
Espressioni:	$e ::= x \mid c \mid e.a \mid e.m(e) \mid \textbf{new}\ C(\widetilde{e}) \mid rnd\,(\alpha)$

Le espressioni del calcolo sono: variabili, costanti, accesso ad un campo, invocazione di un metodo, creazione di oggetti e scelta casuale non-deterministica di un'espressione di tipo α. La ragione per cui abbiamo introdotto quest'ultima espressione è la seguente: l'interpretazione dei tipi freccia, come abbiamo visto nel primo capitolo per il $\lambda-$calcolo e come vedremo in seguito nel calcolo proposto da noi, permette di avere funzioni che sono non-deterministiche, e senza questa espressione non si può avere un risultato di ritorno di una funzione che è non-deterministico. Inoltre, per semplicità, usiamo metodi unari, senza compromettere il potere espressivo del linguaggio. Consideriamo la seguente situazione: supponiamo di voler dichiarare un metodo che ha due argomenti, ad esempio, il metodo **int** *somma* (**int** x, **int** y)$\{\ldots\}$ che dati due interi in ingresso calcola la loro somma. Nel nostro linguaggio questo metodo lo possiamo scrivere come **int** *somma* ($[x : \textbf{int}, y : \textbf{int}]\ arg$)$\{\ldots\}$ dove $[x : \textbf{int}, y : \textbf{int}]$ e il tipo di una classe chiamata, ad esempio $Arguments\ \{\ldots\}$ che contiene due campi interi x, y. Quando invochiamo il metodo *somma* gli passiamo come argomento attuale un oggetto **new** $Arguments\ (i_1, i_2)$ e il risultato restituito dal metodo è la somma dei due interi i_1 e i_2 passati al costruttore della classe $Arguments\ \{\ldots\}$.

Da notare che nella sintassi appena introdotta non si ha un costrutto esplicito per i metodi overloaded. Tali metodi si possono ottenere ridefinendo il metodo di una classe nelle sue sottoclassi, come vedremo nel seguito della nostra trattazione. Inoltre, è importante notare che FJ non ammette funzioni overloaded. Questa è un'altra differenza tra FJ e il nostro calcolo.

Definiamo *programma* una coppia $(\widetilde{L}, e)$ formata da un insieme di dichiarazioni di classi

$\widetilde{L}$ in cui valutare l'espressione e. In questa trattazione, assumiamo le ovvie regole di ben-formatezza per $\widetilde{L}$, tipo 'non è possibile che la classe A estenda la classe B e B estenda A', oppure 'un costruttore chiamato A non può stare nella classe B' ecc.

2.1.3 Regole di tipaggio

Sia $\widetilde{L}$ una sequenza di dichiarazioni di classi e C una classe in $\widetilde{L}$. Per determinare il tipo della classe C dobbiamo esaminare la gerarchia di classi indotta da $\widetilde{L}$. Definiamo induttivamente la funzione $type(C)$ come segue. Tale funzione restituisce un tipo record:

- $type(Object) = [\,]$;
- $type(C) = \rho$, dove ρ è tale che:
  - C estende D in $\widetilde{L}$;
  - $type(D) = \rho'$;
  - per ogni campo a:
    - se $\rho'(a)$ è indefinito e $a \notin C$, allora $\rho(a)$ è indefinito,
    - se $\rho'(a)$ è indefinito e $a \in C$, allora $\rho(a) = \alpha''$, dove α'' è il tipo con cui a è dichiarato in C,
    - se $\rho'(a)$ è definito e $a \notin C$, allora $\rho(a) = \rho'(a)$,
    - se $\rho'(a)$ è definito e $a \in C$, allora $\rho(a) = \alpha''$ purché $\alpha'' \leq \rho'(a)$, altrimenti $type(C)$ è indefinito;
  - per ogni metodo m:
    - se $\rho'(m)$ è indefinito e $m \notin C$, allora $\rho(m)$ è indefinito,
    - se $\rho'(m)$ è indefinito e $m \in C$, allora $\rho(m) = \alpha \to \beta$, dove $\alpha \to \beta$ è il tipo con cui m è dichiarato in C,
    - se $\rho'(m)$ è definito e $m \notin C$, allora $\rho(m) = \rho'(m)$,
    - se $\rho'(m) = \bigwedge_{i=1}^{n} \alpha_i \to \beta_i$ e $m \in C$, allora $\rho(m) = \alpha \to \beta \ \wedge \bigwedge_{i=1}^{n} \alpha_i \setminus \alpha \to \beta_i$ purché $\rho(m) \leq \rho'(m)$, altrimenti $type(C)$ è indefinito.

È importante notare che la condizione $\rho(m) \leq \rho'(m)$ nella dichiarazione di un metodo in una classe è necessaria per garantire che il tipo di C sia un sottotipo del tipo di D; senza questa condizione sarebbe possibile avere una classe il cui tipo non è un sottotipo della classe padre. Se così fosse, la proprietà di *subject reduction*, che andremo a dimostrare nella sezione 3.6, non sarebbe garantita, e questo per la presenza di **this**. Consideriamo il seguente esempio:

class C **extends** $Object$ {
    real $m(x$: **real**$)$ {**return** x}
    real $F()$ {**return this**.$m(3)$}
}

class D **extends** C {
    compl $m(x$: **int**$)$ {**return** $x \times i$}
    real $G()$ {**return this**.$F()$}
}

A tempo di esecuzione, la funzione G restituisce un numero complesso, al posto di un reale. Il punto è che, quando si fa il sovraccaricamento di m, dobbiamo assicurarci che il tipo di ritorno non sia sovratipo del tipo originale altrimenti, per l'istanziazione dinamica di **this**, si possono generare errori di tipo.

Un argomento simile giustifica la condizione $\alpha'' \leq \rho'(a)$ nella dichiarazione dei campi di una classe.

Passiamo adesso a considerare le regole di tipaggio per il nostro calcolo. Le regole di tipaggio per le espressioni sono le seguenti:

$$\frac{\Gamma \vdash e : \alpha_1 \quad \alpha_1 \leq \alpha_2}{\Gamma \vdash e : \alpha_2}\ (subsum) \qquad \frac{c \in Val_{\mathbb{B}}}{\Gamma \vdash c : \mathbb{B}}\ (const)$$

$$\frac{}{\Gamma \vdash rnd(\alpha) : \alpha}\ (rnd) \qquad \frac{\Gamma \vdash e : [a : \alpha]}{\Gamma \vdash e.a : \alpha}\ (field)$$

$$\frac{\Gamma(x) = \alpha}{\Gamma \vdash x : \alpha}\ (var) \qquad \frac{\Gamma \vdash e_2 : [m : \alpha_1 \to \alpha_2] \quad \Gamma \vdash e_1 : \alpha_1}{\Gamma \vdash e_2.m(e_1) : \alpha_2}\ (method)$$

$$\frac{type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}] \quad \Gamma \vdash \widetilde{e} : \widetilde{\beta} \quad \widetilde{\beta} \leq \widetilde{\alpha} \qquad \rho = [\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[a'_i : \alpha'_i] \wedge \bigwedge_j \neg[m'_j : \mu'_j] \not\simeq \mathbf{0}}{\Gamma \vdash \mathbf{new}\ C(\widetilde{e}) : \rho}\ (new)$$

Le regole sono molto intuitive; pochi commenti seguono. La regola (*subsum*) permette di derivare per un'espressione e di tipo α_1 un tipo α_2 a patto che α_1 sia sottotipo di α_2. Da notare che, la relazione di sottotipaggio utilizzata in questa regola è quella indotta da un qualche modello di tipi che andremo a definire nella prossima sessione. Per adesso, supponiamo che questa relazione sia data. Passiamo alla regola (*const*): per ogni tipo base $\mathbb{B}$, assumiamo che esista un insieme fissato di espressioni base $Val_{\mathbb{B}} \in C$ (dove C è l'insieme delle costanti del nostro linguaggio), tale che gli elementi di questo insieme hanno tipo $\mathbb{B}$. Notiamo che per due tipi base $\mathbb{B}_1$ e $\mathbb{B}_2$ gli insiemi $Val_{\mathbb{B}_i}$ possono avere intersezione non vuota. La regola (*const*) afferma che è possibile derivare il tipo $\mathbb{B}$ per una costante c, a patto che c appartenga all'insieme $Val_{\mathbb{B}}$. La regola (*rnd*) è banale: all'espressione $rnd(\alpha)$ si assegna il tipo α. La regola (*var*), sotto l'ipotesi che la variabile x è di tipo α nell'ambiente di tipaggio Γ, afferma che si può derivare che x è di tipo α. Concentriamoci adesso sulle regole (*field*) e (*method*): la regola (*field*) afferma che se un'espressione e è di tipo record $[a : \alpha]$, allora e può accedere al campo a e il tipo dell'espressione $e.a$ è α; la regola (*method*) afferma che se un'espressione e_2 è di tipo $[m : \alpha_1 \to \alpha_2]$ e un'espressione e_1 è di tipo α_1, allora e_2 può invocare il metodo m con argomento e_1 e il tipo dell'espressione $e_2.m(e_1)$ è α_2. Notiamo che in queste regole i tipi record sono dei singoletti, in quanto ci basta che all'interno del record ci sia il campo o il metodo che vogliamo accedere o invocare. Se il record fosse più specializzato, ovvero se avesse altri campi o metodi, con la regola di sussunzione potevamo giungere al record singoletto in questione. Concludiamo con la regola (*new*): prima di tutto, siccome ci siamo concentrati sul frammento funzionale del calcolo, notiamo che la creazione di un oggetto può essere tipata con un tipo record che al suo interno ha come tipi associati ai campi quelli degli argomenti attuali passati al costruttore della classe. Ovviamente, se ci spostiamo nella parte del calcolo dove i campi possono essere modificati, procedere come appena detto non sarebbe corretto in quanto, durante la computazione, i campi possono essere aggiornati con valori appartenenti al tipo

dichiarato. Inoltre, in modo del tutto analogo alle astrazioni in [FCB08], a un oggetto possiamo assegnare un tipo che non è strettamente il record contente i campi e i metodi della classe che si istanzia, ma lo possiamo estendere con l'informazione su quali record non si possono assegnare a questo oggetto, a patto che non si abbia una contraddizione. Questo è semplicemente un trucco tecnico che assicura che qualsiasi tipo non vuoto, abbia un valore che lo abita (vedi Lemma 3.3.5).

A questo punto, presentiamo delle regole che controllano se le dichiarazioni sono accettabili. La regola per controllare quando la dichiarazione di un metodo è accettabile per una classe C è la seguente:

$$\frac{x : \alpha_1, \mathbf{this} : type(C) \vdash e : \alpha_2}{\alpha_2\ m\ (\alpha_1\ x)\ \{return\ e;\}\ OK(C)}\ (OK\ method)$$

La regola per controllare quando la dichiarazione di una classe è accettabile è la seguente:

$$\frac{type(D) = [\widetilde{b : \beta}, \widetilde{m : \mu}] \quad K = C(\widetilde{\beta\ b}; \widetilde{\alpha\ a})\{\mathbf{super}(\widetilde{b}); \mathbf{this}.\widetilde{a} = \widetilde{a}\} \quad \widetilde{M}\ OK(C)}{\mathbf{class}\ C\ \mathbf{extends}\ D\ \{\widetilde{\alpha\ a};\ K\ \widetilde{M}\}\ OK}\ (OK\ class)$$

Concludiamo questa sezione, nella quale abbiamo presentato le regole di tipaggio per il nostro calcolo, con la regola per controllare quando un programma $(\widetilde{L}, e)$ è ben tipato:

$$\frac{\widetilde{L}\ OK \qquad \vdash e : \alpha}{(\widetilde{L}, e)\ OK}\ (OK\ program)$$

2.1.4 Semantica operazionale

Prima di dare la semantica operazionale introduciamo i *valori* del nostro calcolo. Intuitivamente, i valori sono i risultati delle computazioni ben tipate di un linguaggio. Più formalmente, i valori di un linguaggio tipato sono tutti i termini ben tipati, chiusi e in forma normale. Come vedremo in seguito, i passi della computazione sono modellati con la relazione di riduzione che definisce la semantica operazionale. Denotiamo con $\mathcal{V}_\alpha$ l'insie-

me dei valori prodotti dalla seguente sintassi (cioè i valori a cui si può assegnare un tipo α; per ragioni tecniche, più avanti introdurremo anche $\mathcal{V}_\mu$):

$$u ::= c \mid \textbf{new } C(\widetilde{u})$$

A questo punto, siamo pronti per introdurre la semantica operazionale. Fissato l'insieme di dichiarazioni di classi $\widetilde{L}$, la semantica operazionale è una relazione binaria sulle espressioni, della forma $e \to e'$, detta relazione di riduzione. Abbiamo due regole di riduzione, una per l'accesso ai campi e l'altra per l'invocazione dei metodi.

La regola per l'accesso a un campo è semplice: se vogliamo accedere all'i-esimo campo di un oggetto, semplicemente ritorniamo l'i-esimo argomento passato al costruttore di quell'oggetto, ovvero:

$$\frac{type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]}{(\textbf{new } C(\widetilde{u})).a_i \to u_i}$$

In questa regola abbiamo usato la premessa $type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$ in quanto siamo interessati ai campi dell'oggetto che istanzia la classe C e la funzione $type()$ ci permette di fare quanto richiesto. Nel caso di invocazione di un metodo, la regola è la seguente:

$$\frac{body(m, u, C) = \lambda x.e}{(\textbf{new } C(\widetilde{u'})).m(u) \to e[{}^{u}/_{x}, {}^{\textbf{new } C(\widetilde{u'})}/_{\textbf{this}}]}$$

dove

$$body(m, u, C) = \begin{cases} \lambda x.e & \text{se } C \text{ contiene } \beta\, m(\alpha x)\{\textbf{return } e;\} \\ & \quad \text{e } \vdash u : \alpha, \\ body(m, u, D) & \text{se } C \text{ estende } D \text{ in } \widetilde{L}, \\ UNDEF & \text{altrimenti.} \end{cases}$$

La funzione $body(m, u, C)$ controlla se nella classe C si ha una dichiarazione del metodo m, ed inoltre se l'argomento u che viene passato ad m è del tipo giusto, ovvero, del tipo dell'argomento formale del metodo; altrimenti si controlla la classe D, che è la classe padre di C. Se il *look-up* delle classi non dà nessun risultato (cioè si arriva alla classe *Ob ject*, che

non contiene metodi né estende classi), allora la funzione $body(m, u, C)$ ritorna UNDEF e l'oggetto non è in grado di rispondere al metodo specificato col parametro attuale che gli è passato.

Per completare la definizione della semantica operazionale, introduciamo le seguenti regole strutturali:

$$\frac{e \to e'}{e.a \to e'.a} \qquad \frac{e' \to e''}{e'.m(e) \to e''.m(e)}$$

$$\frac{e' \to e''}{e.m(e') \to e.m(e'')}$$

$$\frac{e_i \to e_i'}{\textbf{new } C(e_1, \ldots, e_i, \ldots, e_k) \to \textbf{new } C(e_1, \ldots, e_i', \ldots, e_k)}$$

2.2 Sottotipaggio semantico

A questo punto della nostra trattazione, introdotto il calcolo ad oggetti sul quale ci vogliamo concentrare, siamo pronti per introdurre e definire il sottotipaggio semantico.

2.2.1 Interpretazione insiemistica dei tipi

Definizione 2. *Un'*interpretazione insiemistica *di $\mathcal{T}$ è data da un insieme D e una funzione $[\![]\!] : \mathcal{T} \to \mathcal{P}(D)$ tale che per ogni $\tau_1, \tau_2, \tau \in \mathcal{T}$:*

- $[\![\tau_1 \wedge \tau_2]\!] = [\![\tau_1]\!] \cap [\![\tau_2]\!]$
- $[\![\neg\tau]\!] = D \setminus [\![\tau]\!]$
- $[\![\mathbf{0}]\!] = \varnothing$

È importante notare che la definizione appena data implica quanto segue:

- $[\![\tau_1 \vee \tau_2]\!] = [\![\tau_1]\!] \cup [\![\tau_2]\!]$
- $[\![\tau_1 \backslash \tau_2]\!] = [\![\tau_1]\!] \setminus [\![\tau_2]\!]$
- $[\![\mathbf{1}]\!] = D$

Un'interpretazione insiemistica $[\![]\!] : \mathcal{T} \to \mathcal{P}(D)$ induce una relazione binaria $\leq_{[\![]\!]} \subseteq \mathcal{T}^2$ definita come segue:

$$\tau_1 \leq_{[\![]\!]} \tau_2 \iff [\![\tau_1]\!] \subseteq [\![\tau_2]\!]$$

Tale relazione è la relazione di sottotipaggio semantico. Siccome nei tipi in $\mathcal{T}$ è presente la negazione, il problema del sottotipaggio si riduce al problema del vuoto, ovvero, $[\![\tau_1]\!] \subseteq [\![\tau_2]\!] \iff [\![\tau_1]\!] \setminus [\![\tau_2]\!] = \varnothing \iff [\![\tau_1]\!] \cap (D \setminus [\![\tau_2]\!]) = \varnothing \iff [\![\tau_1 \wedge \neg\tau_2]\!] = \varnothing$.

2.2.2 *Modelli di tipi*

Data l'interpretazione insiemistica dei tipi, definiremo in seguito la nozione di modello. A tale scopo, assoceremo alla funzione di interpretazione un'altra funzione di interpretazione, chiamata *interpretazione estensionale*. A quel punto, affinché l'interpretazione $[\![]\!]$ sia un modello, chiederemo che si comporti come la sua interpretazione estensionale, questo per quanto riguarda il sottotipaggio.

In seguito, assumeremo che per ogni costante c è definito un tipo base $\mathbb{B}_c$ tale che $Val_{\mathbb{B}_c} = \{c\}$. Quindi, l'insieme delle costanti che abitano il tipo base $\mathbb{B}_c$ è costituito dalla singola costante c.

Per quanto riguarda un tipo record $\rho = [\widetilde{l : \tau}]$, la sua interpretazione estensionale è definita come l'insieme delle relazioni $R \subseteq L \times D$ tali che $\forall (r, r') \in R.\ \forall i\ r = l_i \implies r' \in [\![\tau_i]\!]$. Per come abbiamo dato questa definizione, il record $[a : \mathbf{0}]$ sarebbe interpretato come l'insieme delle relazioni le cui coppie sono tali che il primo elemento è diverso da a. Dall'altra parte, la nostra intuizione ci suggerisce che non può essere istanziato nessun oggetto di questo tipo. Quindi, avremmo voluto che l'interpretazione di questo tipo record fosse vuota. Per garantire quanto detto, aggiungiamo alla definizione sopra enunciata, la condizione $dom(R) \supseteq \{\widetilde{l}\}$.

Definizione 3. *Dato un tipo record* $[\widetilde{l : \tau}]$*, definiamo* $[\widetilde{l : [\![\tau]\!]}]$ *come :*

$$[\widetilde{l : [\![\tau]\!]}] = \{R \subseteq (L \times D) \mid dom(R) \supseteq \{\widetilde{l}\} \ \wedge\ \forall (r, r') \in R.\ \forall i.\ (r = l_i \implies r' \in [\![\tau_i]\!])\}$$

Per quanto riguarda un tipo funzionale $\alpha_1 \to \alpha_2$, la sua interpretazione estensionale dovrebbe essere l'insieme delle funzioni f tale che $\forall d \in D.\ d \in [\![\alpha_1]\!] \Rightarrow f(d) \in [\![\alpha_2]\!]$. È

importante osservare però, che il nostro calcolo può esprimere metodi che non terminano oppure che si comportano in modo non-deterministico. Questo fatto ci suggerisce di considerare relazioni binarie arbitrarie piuttosto che funzioni. Inoltre, il calcolo ha una nozione di errore di tipo (*type error*): non è possibile invocare un metodo arbitrario su un argomento arbitrario. Per questo, useremo Ω come un elemento speciale per denotare questo errore di tipo. Per quanto appena detto, interpreteremo il tipo $\alpha_1 \to \alpha_2$ come l'insieme delle relazioni binarie $Q \subseteq D \times D_\Omega$ (dove $D_\Omega = D \uplus \{\Omega\}$), tale che $\forall (q, q') \in Q.\ q \in [\![\alpha_1]\!] \Rightarrow q' \in [\![\alpha_2]\!]$.

Definizione 4. *Se D è un insieme e X, Y sono sottoinsiemi di D, scriviamo D_Ω per $D \uplus \{\Omega\}$ e definiamo $X \to Y$ come:*

$$X \to Y = \{Q \subseteq D \times D_\Omega \mid \forall (q, q') \in Q.\ q \in X \Rightarrow q' \in Y\}$$

Notiamo che, nella definizione appena data, se sostituissimo D_Ω con D, allora $X \to Y$ sarebbe sempre un sottoinsieme di $D \to D$. Come vedremo fra breve, questo implicherebbe che ogni tipo freccia sarebbe sottotipo di $\mathbf{1} \to \mathbf{1}$. A quel punto, usando la regola di sussunzione, l'invocazione di un qualsiasi metodo ben tipato su un qualsiasi argomento ben tipato, sarebbe a sua volta ben tipata. Chiaramente, questo romperebbe la proprietà di *type-safety* del calcolo. Con la Definizione 4, invece, si ha che $X \to Y \subseteq D \to D$ se e solo se $D = X$. Questo risulta vero, se consideriamo la covarianza delle frecce.

A questo punto, possiamo dare la definizione formale di interpretazione estensionale associata ad un'interpretazione insiemistica.

Definizione 5. *Sia $[\![]\!] : \mathcal{T} \to \mathcal{P}(D)$ un'interpretazione insiemistica di $\mathcal{T}$. Definiamo la sua* interpretazione estensionale *come l'unica interpretazione insiemistica $\mathbb{E}(\_) : \mathcal{T} \to \mathcal{P}(\mathbb{E}D)$ (dove $\mathbb{E}D = C \uplus \mathcal{P}(L \times D) \uplus \mathcal{P}(D \times D_\Omega)$) tale che:*

$$\begin{aligned}
\mathbb{E}(\mathbb{B}) &= Val_{\mathbb{B}} && \subseteq C \\
\mathbb{E}\left(\widetilde{[l : \tau]}\right) &= \widetilde{[l : [\![\tau]\!]]} && \subseteq \mathcal{P}(L \times D) \\
\mathbb{E}(\alpha_1 \to \alpha_2) &= [\![\alpha_1]\!] \to [\![\alpha_2]\!] && \subseteq \mathcal{P}(D \times D_\Omega)
\end{aligned}$$

Formalizziamo adesso il fatto che un'interpretazione insiemistica induce la stessa relazione di sottotipaggio della sua estensionale.

Definizione 6. *Un'interpretazione insiemistica* $[\![]\!] : \mathcal{T} \to \mathcal{P}(D)$ *è un* modello *se induce la stessa relazione di sottotipaggio della sua interpretazione estensionale:*

$$\forall \tau_1, \tau_2 \in \mathcal{T}. \quad [\![\tau_1]\!] \subseteq [\![\tau_2]\!] \iff \mathbb{E}(\tau_1) \subseteq \mathbb{E}(\tau_2)$$

Grazie all'osservazione fatta precedentemente sul problema del vuoto, la condizione sui tipi in $\mathcal{T}$ nella definizione di modello può essere riscritta come:

$$\forall \tau \in \mathcal{T}. \, [\![\tau]\!] = \varnothing \iff \mathbb{E}(\tau) = \varnothing$$

A questo punto, possiamo derivare molte proprietà per $\leq_{[\![]\!]}$, che seguono direttamente dal fatto che questa relazione sui tipi è indotta da un modello. Ad esempio, la covarianza e contro-varianza dei tipi freccia, oppure equivalenze come $(\alpha_1 \to \beta) \wedge (\alpha_2 \to \beta) \simeq (\alpha_1 \vee \alpha_2) \to \beta$, possono essere derivate direttamente dalla definizione di interpretazione estensionale. Lo studio della relazione di sottotipaggio e tipaggio si basa in modo cruciale su tante di queste proprietà. Con un approccio sintattico alla definizione di sottotipaggio (cioè, con un sistema di regole deduttive), queste proprietà andavano dimostrate, mentre con l'approccio semantico sono una conseguenza delle definizioni appena date. Questo è uno dei vantaggi dell'approccio semantico rispetto a quello sintattico.

2.2.3 Modelli ben fondati

In questa sezione introduciamo una nuova definizione di modello che cattura una proprietà molto importante, ovvero che i valori, precedentemente introdotti, sono degli alberi m-ari *finiti* (dove le foglie sono costanti). Ad esempio, consideriamo il tipo ricorsivo $\alpha = [a : \alpha]$. Intuitivamente, un valore u è di questo tipo se e solo se è un oggetto **new** $C(u')$ dove u' è anche esso di tipo α. Per costruire questo valore, dovremmo considerare un albero infinito, che è impossibile in quanto i valori sono risultati di una qualche computazione. Come conseguenza, il tipo α non contiene valori. Quanto appena affermato sembrerebbe restrittivo, in quanto siamo interessati a utilizzare i tipi ricorsivi nel nostro calcolo; si pensi

ad esempio alle liste. Nel Capitolo 4 vedremo che è possibile soddisfare la proprietà di finitezza sopra enunciata, e nel frattempo utilizzare i tipi ricorsivi e creare delle liste.

Introdurremo un nuovo criterio che cattura il fatto che non ci sono record annidati infinitamente. La seguente definizione afferma questo.

Definizione 7. *Un'interpretazione insiemistica* $[\![]\!] : \mathcal{T} \to \mathcal{P}(D)$ *è* strutturale*, se:*

- $\mathcal{P}_f(L \times D) \subseteq D$
- *per ogni* $\widetilde{\tau}$. $[\![\widetilde{[l : \tau]}]\!] = \widetilde{[l : [\![\tau]\!]]} \subseteq \mathcal{P}_f(L \times D)$
- *la relazione binaria su D definita come* $\{(l_1, d_1), \ldots, (l_n, d_n)\} \triangleright d_i$ *è Noetheriana.*

Dove Noetheriana vuol dire che la relazione non ammette catene infinite discendenti.

Definizione 8. *Un modello* $[\![]\!] : \mathcal{T} \to \mathcal{P}(D)$ *è* ben fondato *se induce la stessa relazione di sottotipaggio di un'interpretazione insiemistica strutturale.*

2.2.4 Interpretazione dei tipi come insiemi di valori

Per seguire la linea di [FCB08], a questo punto vogliamo dare un'altra interpretazione dei tipi, cioè come insiemi di valori. Intuitivamente, un tipo τ verrà interpretato come l'insieme di tutti quei valori che hanno tipo τ. Nella sezione 2.1.4 abbiamo presentato la grammatica che genera i valori del nostro calcolo e abbiamo denotato con $\mathcal{V}_\alpha$ l'insieme dei valori. Ricordiamo che i valori possono essere costanti oppure oggetti creati passando al loro costruttore altri valori. Come si può vedere, questi sono di tipo α (questo spiega il pedice di $\mathcal{V}$), le costanti sono di tipo base $\mathbb{B}$ e gli oggetti sono di tipo record ρ (vedremo in seguito come questa interpretazione si comporta rispetto ai connettivi booleani). A questo punto, ci vogliamo concentrare sull'interpretazione dei tipi μ come insiemi di valori. Siccome nel calcolo non si hanno valori di tipo μ, e questo perché nessuna computazione termina restituendo un metodo, dobbiamo dapprima estendere il concetto di valore. Denotiamo con $\mathcal{V}_\mu$ l'insieme dei *valori di tipo* μ (che ovviamente è diverso dall'insieme dei valori di tipo α; solo per analogia chiameremo queste espressioni valori). A questo punto, l'insieme dei valori sul quale interpreteremo i tipi sarà $\mathcal{V} = \mathcal{V}_\alpha \uplus \mathcal{V}_\mu$. Indicheremo l'interpretazione di

un tipo τ in $\mathcal{V}$ con $[\![\tau]\!]_{\mathcal{V}}$ ed in particolare, $[\![\alpha]\!]_{\mathcal{V}_\alpha}$ e $[\![\mu]\!]_{\mathcal{V}_\mu}$, quando τ sarà un tipo α e un tipo μ, rispettivamente. Denotiamo con $v \in \mathcal{V}$ un generico elemento di questo insieme.

Introduciamo le *forme normali*. Queste sono espressioni del calcolo che non si possono ulteriormente ridurre, dove la riduzione è la relazione che definisce la semantica operazionale. La sintassi è la seguente:

$$w ::= c \mid \textbf{new}\ \rho(\widetilde{w}) \mid x \mid x.a \mid x.m(w)$$

Notiamo che $\mathcal{V}_\alpha$, così com'è definito nella sezione 2.1.4, non conterebbe solo forme normali. Infatti, gli oggetti appartenenti a $\mathcal{V}_\alpha$ sono della forma **new** $C(\widetilde{u})$ dove u è a sua volta un valore e C è una classe dichiarata nella gerarchia di classi $\widetilde{L}$. Notiamo che nella sintassi delle forme normali abbiamo **new** $\rho(\widetilde{w})$, con ρ un tipo record. A questo punto, vale la pena spiegare perché abbiamo usato **new** $\rho(\widetilde{w})$ invece che **new** $C(\widetilde{u})$ per definire le forme normali. In realtà, ogni elemento **new** $C(\widetilde{u})$ può essere riscritto come **new** $type(C)(\widetilde{u'})$, dove $\widetilde{u'}$ è ottenuto da $\widetilde{u}$ propagando questo tipo di riscrittura. A questo punto, ogni elemento di $\mathcal{V}_\alpha$ riscritto in questo modo corrisponde ad una forma normale chiusa (senza variabili libere) e ben tipata ($\vdash w : \alpha$ per qualche α). L'insieme dei valori $\mathcal{V}_\alpha$, come abbiamo già detto, lo utilizzeremmo per interpretare i tipi α. In particolare, un qualsiasi tipo record ρ verrà interpretato come l'insieme degli oggetti ai quali è possibile assegnare tipo ρ. Gli oggetti sono istanziazioni delle classi appartenenti alla gerarchia di classi $\widetilde{L}$ dichiarata dal programmatore. A questi oggetti è possibile assegnare il tipo della classe che istanziano. Ma siccome le classi in $\widetilde{L}$ sono finite, questo vuol dire che riusciamo a interpretare (abitare con oggetti) soltanto un numero finito di tipi record, diversamente dal nostro obbiettivo, che è quello di abitare tutti i tipi record che possono essere abitati da oggetti. Per questa ragione, invece di specificare una classe C, abbiamo messo un tipo record ρ nell'espressione **new** $\rho(\widetilde{u})$, in quanto sarebbe possibile dichiarare una classe C di tipo ρ anche se il programmatore non l'ha fatto e quindi non appartiene ad $\widetilde{L}$. In questo modo, siamo indipendenti dal programmatore nello sviluppare la nostra teoria. Da notare che, in questo modo, riusciamo ad abitare solo record 'ben fatti', cioè record per cui è possibile istanziare (e quindi, creare un oggetto di) una classe corrispondente al record in questione. Ad esempio, **new** $[a : \mathbf{0}](u)$

non crea nessun oggetto, in quanto nessun valore di tipo $\mathbf{0}$ può essere passato al costruttore di questa classe (ovvero, la classe di tipo $[a : \mathbf{0}]$).

Una soluzione alternativa a questo problema era l'assumere che, per ogni tipo record ρ, esistesse in $\widetilde{L}$ una classe C_ρ tale che $type(C_\rho) = \rho$. Questa soluzione, però, ci avrebbe portato ad un insieme infinito di dichiarazioni di classi e quindi sarebbe stato soddisfacente solo sul piano teorico.

Definiamo adesso l'interpretazione di un tipo τ in $\mathcal{V}$, a seconda se τ è un tipo α o un tipo μ.

$$[\![\alpha]\!]_{\mathcal{V}_\alpha} = \{(\varnothing, w) \mid \varnothing \vdash w : \alpha\}$$

Per semplicità, scriveremo $[\![\alpha]\!]_{\mathcal{V}_\alpha} = \{w \mid \vdash w : \alpha\}$ e questa definizione ci riconduce alla definizione dei tipi come insiemi di valori in [FCB08]. Per i tipi freccia, invece, la situazione è più complessa:

$$[\![\mu]\!]_{\mathcal{V}_\mu} = \begin{cases} \varnothing \qquad \text{se } \mu \simeq \mathbf{0} \\ \text{altrimenti, se } \mu = \bigwedge_{i=1\cdots n} \alpha_i \to \beta_i \wedge \bigwedge_{j=1\cdots m} \neg\left(\alpha'_j \to \beta'_j\right) \\ \{(\alpha, z) \mid \forall i.\, \alpha \geq \alpha_i \wedge \\ \left[\left(z = \bot \wedge \forall j.\, \alpha \not\geq \alpha'_j\right) \vee (z = w \wedge fv(w) \subseteq \{x\} \wedge x : \alpha \vdash w : \beta_i)\right]\} \end{cases}$$

Notiamo in questa definizione la presenza di un pseudo-valore, $\bot$. La ragione per cui lo abbiamo introdotto è la seguente: originariamente volevamo interpretare i tipi freccia come insiemi di coppie (α, w) tale che, sotto le condizioni di contorno come sopra enunciate, era possibile assegnare alla forma normale w il tipo di ritorno del tipo freccia che volevamo interpretare. Facendo così, non sarebbe stato possibile interpretare tipi freccia come ad esempio $\alpha \to \mathbf{0}$: questo tipo avrebbe avuto un'interpretazione vuota. Dall'altra parte, vogliamo invece che la sua interpretazione sia diversa dall'insieme vuoto in quanto, questo è il tipo di un metodo che non termina, ed è possibile scrivere un tale metodo in un qualche programma. Il tipo μ è una composizione finita di tipi freccia. Si ha una congiunzione di tipi freccia $\alpha_i \to \beta_i$ congiunta ad una congiunzione di tipi freccia negati $\neg\left(\alpha'_j \to \beta'_j\right)$. Per come abbiamo dato questa definizione di interpretazione in $\mathcal{V}_\mu$ (simile alla regola di tipaggio (*abstr*) in [FCB08]), la parte rilevante sono i tipi freccia 'positivi', ovvero $\alpha_i \to \beta_i$

mentre la parte negativa può essere qualsiasi, a patto che il tipo $\mu \neq \mathbf{0}$. Quindi, un tipo freccia semplice $\alpha \to \beta$ sarebbe interpretato come:

$$[\![\alpha \to \beta]\!]_{\mathcal{V}_\mu} = \{(\alpha', z) \mid \alpha' \geq \alpha \wedge [z = \bot \vee (z = w \wedge fv(w) \subseteq \{x\} \wedge x : \alpha' \vdash w : \beta)]\}$$

Per quanto riguarda un tipo freccia negato $\neg(\alpha \to \beta)$, la sua interpretazione sarebbe formata dall'insieme delle coppie che non appartengono all'interpretazione di $\alpha \to \beta$. Questo ci suggerisce la seguente:

$$[\![\neg(\alpha \to \beta)]\!]_{\mathcal{V}_\mu} = \mathcal{V}_\mu \backslash [\![\alpha \to \beta]\!]_{\mathcal{V}_\mu}$$

Nella sezione 3.3 vedremo che l'interpretazione dei tipi in $\mathcal{V}$ è un'interpretazione insiemistica.

Notiamo come la condizione $\forall i.\ \alpha \geq \alpha_i$ soddisfa la contro-varianza dei tipi dei metodi. Lo facciamo vedere con un esempio. Consideriamo le seguenti dichiarazioni di classi:

```
class Persona extends Object {
    int età;
}

class Studente extends Persona {
    long matricola;
}

class StudLavoratore extends Studente {
    string contratto;
}
```

Consideriamo adesso, i seguenti due tipi freccia: $Studente \to \mathbf{long}$ e $StudLavoratore \to$

long. Per la contro-varianza delle frecce, si la seguente situazione:

$$\frac{\mathit{StudLavoratore} \leq \mathit{Studente}}{\mathit{Studente} \to \mathbf{long} \leq \mathit{StudLavoratore} \to \mathbf{long}}$$

Controlliamo a questo punto se la definizione di interpretazione di un tipo freccia in $\mathcal{V}_\mu$ rispetta la contro-varianza appena enunciata.

$$\begin{aligned}
[\![\mathit{Studente} \to \mathbf{long}]\!]_{\mathcal{V}_\mu} =& \{(\alpha, z) \mid \alpha \geq \mathit{Studente} \wedge \\
& \quad [z = \bot \vee (z = w \wedge fv(w) \subseteq \{x\} \wedge x : \alpha \vdash w : \mathbf{long})]\} \\
=& \{(\mathit{Object}, \bot), (\mathit{Studente}, \bot), (\mathit{Studente}, x.\mathit{matricola})\} \\
[\![\mathit{StudLavoratore} \to \mathbf{long}]\!]_{\mathcal{V}_\mu} =& \{(\alpha, z) \mid \alpha \geq \mathit{StudLavoratore} \wedge \\
& \quad [z = \bot \vee (z = w \wedge fv(w) \subseteq \{x\} \wedge x : \alpha \vdash w : \mathbf{long})]\} \\
=& \{(\mathit{Object}, \bot), (\mathit{Studente}, \bot), (\mathit{StudLavoratore}, \bot), \\
& \quad (\mathit{Studente}, x.\mathit{matricola}), (\mathit{StudLavoratore}, x.\mathit{matricola})\}
\end{aligned}$$

Notiamo che $[\![\mathit{Studente} \to \mathbf{long}]\!]_{\mathcal{V}_\mu} \subseteq [\![\mathit{StudLavoratore} \to \mathbf{long}]\!]_{\mathcal{V}_\mu}$ e questo ci da la contro-varianza.

CAPITOLO 3

Dimostrazioni

In questo capitolo daremmo i risultati teorici ottenuti per il nostro calcolo e le corrispondenti dimostrazioni.

3.1 Forme normali disgiuntive per i tipi

Nel seguito della nostra trattazione, indicheremo con $\mathbb{T}$ l'insieme dei tipi atomici e useremo la meta-variabile t per indicare gli elementi di questo insieme. Come abbiamo già detto, ci sono tre generi di tipi atomici, e anche di valori, che sono i tipi/valori base (**basic**), i tipi/valori record (**rec**) e i tipi/valori funzionali(**fun**). Diremo che un valore v e un tipo atomico t sono *compatibili* se sono dello stesso genere. Useremo la meta-variabile u per indicare gli elementi dell'insieme $U = \{\mathbf{basic}, \mathbf{rec}, \mathbf{fun}\}$. Scriveremo $\mathbb{T}_{\mathbf{basic}}$ per indicare i tipi base, $\mathbb{T}_{\mathbf{rec}}$ per i tipi record e $\mathbb{T}_{\mathbf{fun}}$ per indicare i tipi funzionali. Quindi, $\mathbb{T} = \mathbb{T}_{\mathbf{basic}} \uplus \mathbb{T}_{\mathbf{rec}} \uplus \mathbb{T}_{\mathbf{fun}}$. Ogni tipo può essere visto come una combinazione booleana di atomi. Quindi, è conveniente lavorare con i tipi in forma normale disgiuntiva.

Definizione 9. *Una* forma normale disgiuntiva *v è un insieme finito di coppie di insiemi finiti di atomi; ovvero, un elemento di* $\mathcal{P}_f\left(\mathcal{P}_f(\mathbb{T}) \times \mathcal{P}_f(\mathbb{T})\right)$.

Se $[\![]\!] : \mathcal{T} \to \mathcal{P}(D)$ *è un'arbitraria interpretazione insiemistica e* ν *una forma normale disgiuntiva, definiamo* $[\![\nu]\!]$ *come:*

$$[\![\nu]\!] = \bigcup_{(P,N)\in\nu} \left(\bigcap_{t\in P} [\![t]\!] \cap \bigcap_{t\in N} (D \setminus [\![t]\!]) \right)$$

Da notare che con la convenzione che l'intersezione su un insieme vuoto è uguale a D, si ha $[\![\nu]\!] \subseteq D$.

Per i tipi in forma normale disgiuntiva vale il seguente Lemma.

Lemma 3.1.1. *Per ogni* $\tau \in \mathcal{T}$*, è possibile computare una forma normale disgiuntiva* $\mathcal{N}(\tau)$ *tale che per ogni interpretazione insiemistica* $[\![]\!]$*,* $[\![\tau]\!] = [\![\mathcal{N}(\tau)]\!]$.

Siccome questo Lemma riguarda semplicemente i tipi booleani, senza prendere in considerazione gli atomi presenti nel calcolo, la dimostrazione è la stessa del $\lambda-$calcolo presentato in [FCB08]. Quindi, per i dettagli della dimostrazione si rimanda a questo articolo.

Come esempio, consideriamo il tipo $\tau = t_1 \wedge (t_2 \vee \neg t_3)$, dove t_1, t_2, t_3 sono tre atomi. Si ha $\mathcal{N}(\tau) = \{(\{t_1, t_2\}, \varnothing), (\{t_1\}, \{t_3\})\}$. Questo corrisponde al fatto che τ e $(t_1 \wedge t_2) \vee (t_1 \wedge \neg t_3)$ hanno la stessa interpretazione per ogni interpretazione insiemistica dell'algebra dei tipi.

Da notare che anche l'inverso risulta vero: per ogni forma normale disgiuntiva ν, possiamo trovare un tipo τ tale che $[\![\tau]\!] = [\![\nu]\!]$, per ogni interpretazione insiemistica. Quindi, le forme normali disgiuntive sono semplicemente un modo diverso, ma facile, di scrivere i tipi. In particolare, possiamo riformulare la definizione 6 di modello usando le forme normali disgiuntive; ovvero, $\forall \nu.\ [\![\nu]\!] = \varnothing \iff \mathbb{E}(\nu) = \varnothing$. Nella nostra trattazione, spesso confonderemo la nozione di tipo e di forma normale disgiuntiva e parleremo del tipo ν, considerando quest'ultimo come una forma canonica per rappresentare i tipi in $\mathcal{N}^{-1}(\nu)$.

3.2 La relazione di sottotipaggio

La definizione 6 di modello è una definizione *intensionale*. In questa sezione vogliamo dare una definizione più *estensionale* di modello. In particolare, useremo i tipi in forma normale disgiuntiva.

Sia $[\![]\!]$ un'interpretazione insiemistica. Siamo interessati a confrontare le asserzioni $[\![\nu]\!] = \varnothing$ e $\mathbb{E}(\nu) = \varnothing$ per una qualche forma normale disgiuntiva ν. Come per l'interpretazione dei tipi ν, l'interpretazione estensionale è definita come segue:

$$\mathbb{E}(\nu) = \bigcup_{(P,N)\in\nu} \left(\bigcap_{t\in P} \mathbb{E}(t) \cap \bigcap_{t\in N} (\mathbb{E}D \setminus \mathbb{E}(t)) \right)$$

Quindi:

$$\begin{gathered}
\mathbb{E}(\nu) = \varnothing \iff \\
\bigcup_{(P,N)\in\nu} \left(\bigcap_{t\in P} \mathbb{E}(t) \cap \bigcap_{t\in N} (\mathbb{E}D \setminus \mathbb{E}(t)) \right) = \varnothing \iff \\
\forall (P,N) \in \nu.\ \bigcap_{t\in P} \mathbb{E}(t) \cap \bigcap_{t\in N} \overline{\mathbb{E}(t)} = \varnothing \iff \\
\forall (P,N) \in \nu.\ \bigcap_{t\in P} \mathbb{E}(t) \cap \overline{\bigcup_{t\in N} \mathbb{E}(t)} = \varnothing \iff \\
\forall (P,N) \in \nu.\ \bigcap_{t\in P} \mathbb{E}(t) \setminus \bigcup_{t\in N} \mathbb{E}(t) = \varnothing \iff \\
\forall (P,N) \in \nu.\ \bigcap_{t\in P} \mathbb{E}(t) \subseteq \bigcup_{t\in N} \mathbb{E}(t)
\end{gathered}$$

Ricapitolando, abbiamo che

$$\mathbb{E}(\nu) = \varnothing \iff \forall (P,N) \in \nu.\ \bigcap_{t\in P} \mathbb{E}(t) \subseteq \bigcup_{t\in N} \mathbb{E}(t) \tag{3.1}$$

Scriviamo $\mathbb{E}^{\mathbf{basic}}D = C$, $\mathbb{E}^{\mathbf{rec}}D = \mathcal{P}(L \times D)$, $\mathbb{E}^{\mathbf{fun}}D = \mathcal{P}(D \times D_\Omega)$; quindi $\mathbb{E}D = \bigcup_{u\in U} \mathbb{E}^u D$ con $U = \{\mathbf{basic}, \mathbf{rec}, \mathbf{fun}\}$. A questo punto, possiamo riscrivere (3.1) come segue:

$$\forall u \in U.\ \forall (P,N) \in \nu.\ \bigcap_{t\in P} (\mathbb{E}(t) \cap \mathbb{E}^u D) \subseteq \bigcup_{t\in N} (\mathbb{E}(t) \cap \mathbb{E}^u D) \tag{3.2}$$

Siccome $\mathbb{E}(t) \cap \mathbb{E}^u D = \varnothing$ se $t \notin \mathbb{T}_u$ e $\mathbb{E}(t) \cap \mathbb{E}^u D = \mathbb{E}(t)$ se $t \in \mathbb{T}_u$, allora possiamo riscrivere

la (3.2) come:

$$\forall u \in U.\ \forall (P,N) \in \nu.\ (P \subseteq \mathbb{T}_u) \Longrightarrow \left(\bigcap_{t \in P} \mathbb{E}(t) \subseteq \bigcup_{t \in N \cap \mathbb{T}_u} \mathbb{E}(t) \right) \tag{3.3}$$

Per decomporre ulteriormente questi predicati, useremo l'interpretazione insiemistica della relazione di sottotipaggio semantico e ci verrano in aiuto due lemmi tecnici, uno per i tipi record e l'altro per i tipi funzionali. Prima di tutto, introduciamo una notazione che renderà le formule più chiare.

Notazione 1. *Siano S_1, S_2 due insiemi tali che $S_1 \subseteq S_2$. Scriviamo $\overline{S_1}^{S_2}$ per denotare il complemento di S_1 rispetto ad S_2, ovvero $S_2 \setminus S_1$.*

Il seguente lemma, preso da [FCB08], è un risultato tecnico utilizzato nelle dimostrazioni di lemmi valide nel nostro calcolo.

Lemma 3.2.1. *Siano $\{X_i\}_{i \in P}, \{X'_j\}_{j \in N}$ e $\{Y_i\}_{i \in P}, \{Y'_j\}_{j \in N}$ famiglie di sottoinsiemi di D. Allora,*

$$\left(\bigcap_{i \in P} X_i \times Y_i \right) \setminus \left(\bigcup_{j \in N} X'_j \times Y'_j \right) = \bigcup_{N' \subseteq N} \left(\bigcap_{i \in P} X_i \setminus \bigcup_{j \in N'} X'_j \right) \times \left(\bigcap_{i \in P} Y_i \setminus \bigcup_{j \in N \setminus N'} Y'_j \right)$$

Per i dettagli della dimostrazione del lemma appena enunciato, si rimanda a [FCB08]. Il lemma che segue è una generalizzazione del lemma 3.2.1.

Lemma 3.2.2. *Siano*

$$\begin{gathered} \left(\{X_{1_i}\}_{i \in P}\ ,\ \left\{X'_{1_j}\right\}_{j \in N} \right) \\ \left(\{X_{2_i}\}_{i \in P}\ ,\ \left\{X'_{2_j}\right\}_{j \in N} \right) \\ \vdots \\ \left(\{X_{n_i}\}_{i \in P}\ ,\ \left\{X'_{n_j}\right\}_{j \in N} \right) \end{gathered}$$

n coppie di famiglie di sottoinsiemi di D. Allora,

$$\left(\bigcap_{i\in P}[l_1 : X_{1_i}\ ,\ l_2 : X_{2_i}\ ,\ \cdots\ ,\ l_n : X_{n_i}]\right) \setminus \left(\bigcup_{j\in N}\left[l_1 : X'_{1_j}\ ,\ l_2 : X'_{2_j}\ ,\ \cdots\ ,\ l_n : X'_{n_j}\right]\right) =$$

$$\bigcup_{N=N_1\uplus N_2\uplus\cdots\uplus N_n}\left[l_1 : \bigcap_{i\in P} X_{1_i} \setminus \bigcup_{j\in N_1} X'_{1_j}\ ,\ l_2 : \bigcap_{i\in P} X_{2_i} \setminus \bigcup_{j\in N_2} X'_{2_j}\ ,\ \cdots\ ,\ l_n : \bigcap_{i\in P} X_{n_i} \setminus \bigcup_{j\in N_n} X'_{n_j}\right]$$

È importante osservare quanto segue: nel lemma appena enunciato, i tipi record sono tali che le loro etichette sono tutte uguali $l_1,\ l_2\ \cdots\ l_n$ ed, eventualmente, differiscono per i tipi associati alle etichette. Questa non è una limitazione, in quanto possiamo operare nel seguente modo: dato un insieme di tipi record qualsiasi, questi li possiamo riscrivere come dei record che ad ogni etichetta appartente a qualsiasi altro record nell'insieme associa il tipo $\mathbf{1}$ e i tipi associati alle etichette proprie rimangono invariati. Ad esempio, dati i seguenti tipi record: $[a : \alpha]$, $[b : \beta]$ e $[c : \gamma]$, li possiamo riscrivere come $[a : \alpha, b : \mathbf{1}, c : \mathbf{1}]$, $[a : \mathbf{1}, b : \beta, c : \mathbf{1}]$ e $[a : \mathbf{1}, b : \mathbf{1}, c : \gamma]$, rispettivamente. Questo modo di riscrivere i tipi record non cambia la relazione di sottotipaggio: se due tipi record sono in relazione di sottotipaggio, allora con la riscrittura questa relazione viene preservata. Nel seguito della nostra trattazione, faremmo riferimento ai tipi record riscritti in questa maniera. Dal lemma appena enunciato, segue:

Lemma 3.2.3. *Siano P, N due sottoinsiemi finiti di $\mathbb{T}_{\mathbf{rec}}$ contenenti record di lunghezza n. Allora:*

$$\bigcap_{t\in P}\mathbb{E}(t) \subseteq \bigcup_{t\in N}\mathbb{E}(t) \iff$$

$$\forall N = N_1 \uplus \cdots \uplus N_n.\ [\![\bigwedge_{\widetilde{[l:\tau]}\in P} \tau_1 \wedge \bigwedge_{\widetilde{[l:\tau']}\in N_1} \neg\tau'_1]\!] = \varnothing \vee \cdots \vee [\![\bigwedge_{\widetilde{[l:\tau]}\in P} \tau_n \wedge \bigwedge_{\widetilde{[l:\tau']}\in N_n} \neg\tau'_n]\!] = \varnothing$$

Dimostrazione. Siccome $P, N \subseteq \mathbb{T}_{\mathbf{rec}}$, allora ogni t appartenente a P o a N è della forma

$[\widetilde{l:\tau}]$. Per definizione,

$$\begin{aligned}\mathbb{E}\left([\widetilde{l:\tau}]\right) &= \{R \subseteq (L \times D) \mid dom(R) \supseteq \{\widetilde{l}\},\ \forall (r, r') \in R.\ (r = l_i \implies r' \in [\![\tau_i]\!])\} \\ &= [\widetilde{l:[\![\tau]\!]}] = [l_1 : [\![\tau_1]\!], \ldots, l_n : [\![\tau_n]\!]]\end{aligned}$$

Allora, si ha quanto segue:

$$\bigcap_{t\in P} \mathbb{E}\,(t) \subseteq \bigcup_{t\in N} \mathbb{E}\,(t) \iff$$

$$\bigcap_{[\widetilde{l:\tau}]\in P} \mathbb{E}\left([\widetilde{l:\tau}]\right) \subseteq \bigcup_{[\widetilde{l:\tau'}]\in N} \mathbb{E}\left([\widetilde{l:\tau'}]\right) \iff$$

$$\bigcap_{[\widetilde{l:\tau}]\in P} \mathbb{E}\left([\widetilde{l:\tau}]\right) \setminus \bigcup_{[\widetilde{l:\tau'}]\in N} \mathbb{E}\left([\widetilde{l:\tau'}]\right) = \varnothing \iff$$

$$\bigcap_{[\widetilde{l:\tau}]\in P} [\widetilde{l:[\![\tau]\!]}] \setminus \bigcup_{[\widetilde{l:\tau'}]\in N} [\widetilde{l:[\![\tau]\!]'}] = \varnothing \iff$$

$$\bigcap_{[\widetilde{l:\tau}]\in P} [l_1 : [\![\tau_1]\!], \cdots, l_n : [\![\tau_n]\!]] \setminus \bigcup_{[\widetilde{l:\tau'}]\in N} [l_1 : [\![\tau'_1]\!], \cdots, l_n : [\![\tau'_n]\!]] = \varnothing \overset{\text{lemma } 3.2.2}{\iff}$$

$$\bigcup_{N=N_1\uplus\cdots\uplus N_n} \left[l_1 : \bigcap_{[\widetilde{l:[\![\tau]\!]}]\in P} [\![\tau_1]\!] \setminus \bigcup_{[\widetilde{l:[\![\tau]\!]'}]\in N_1} [\![\tau'_1]\!]\ ,\ \cdots\ ,\ l_n : \bigcap_{[\widetilde{l:[\![\tau]\!]}]\in P} [\![\tau_n]\!] \setminus \bigcup_{[\widetilde{l:[\![\tau]\!]}]\in N_n} [\![\tau'_n]\!] \right] = \varnothing \iff$$

$$\forall N = N_1 \uplus \cdots \uplus N_n.\ \left[l_1 : \bigcap_{[\widetilde{l:[\![\tau]\!]}]\in P} [\![\tau_1]\!] \setminus \bigcup_{[\widetilde{l:[\![\tau]\!]'}]\in N_1} [\![\tau'_1]\!]\ ,\ \cdots\ ,\ l_n : \bigcap_{[\widetilde{l:[\![\tau]\!]}]\in P} [\![\tau_n]\!] \setminus \bigcup_{[\widetilde{l:[\![\tau]\!]'}]\in N_n} [\![\tau'_n]\!] \right] = \varnothing \iff$$

$$\forall N = N_1 \uplus \cdots \uplus N_n.\ [\![\bigwedge_{[\widetilde{l:\tau}]\in P} \tau_1 \setminus \bigvee_{[\widetilde{l:\tau'}]\in N_1} \tau'_1]\!] = \varnothing \vee \cdots \vee [\![\bigwedge_{[\widetilde{l:\tau}]\in P} \tau_n \setminus \bigvee_{[\widetilde{l:\tau'}]\in N_n} \tau'_n]\!] = \varnothing \iff$$

$$\forall N = N_1 \uplus \cdots \uplus N_n.\ [\![\bigwedge_{[\widetilde{l:\tau}]\in P} \tau_1 \wedge \bigwedge_{[\widetilde{l:\tau'}]\in N_1} \neg\tau'_1]\!] = \varnothing \vee \cdots \vee [\![\bigwedge_{[\widetilde{l:\tau}]\in P} \tau_n \wedge \bigwedge_{[\widetilde{l:\tau'}]\in N_n} \neg\tau'_n]\!] = \varnothing$$

□

A questo punto, per completezza, vogliamo stabilire un risultato simile per i tipi freccia. Come prima cosa, decomponiamo il costruttore $\to$ in operatori più semplici; ovvero, insieme delle parti, complemento e prodotto cartesiano.

Lemma 3.2.4. *Siano $X, Y \subseteq D$. Allora,*

$$X \to Y = \mathcal{P}\left(\overline{X \times \overline{Y}^{D_\Omega}}^{D \times D_\Omega}\right)$$

Dimostrazione.

$$\begin{aligned} X \to Y &= \{Q \subseteq D \times D_\Omega \mid \forall (q, q') \in Q.\ q \in X \Rightarrow q' \in Y\} \\ &= \{Q \subseteq D \times D_\Omega \mid \forall (q, q') \in Q.\ \neg (q \in X \wedge q' \notin Y)\} \\ &= \left\{Q \subseteq D \times D_\Omega \mid Q \cap X \times \overline{Y}^{D_\Omega} = \varnothing\right\} \\ &= \left\{Q \subseteq D \times D_\Omega \mid Q \subseteq \overline{X \times \overline{Y}^{D_\Omega}}^{D \times D_\Omega}\right\} \end{aligned}$$

□

Lemma 3.2.5. *Siano $\{X_i\}_{i \in P}$ e $\left\{X'_i\right\}_{i \in N}$ due famiglie di sottoinsiemi di D. Allora,*

$$\bigcap_{i \in P} \mathcal{P}(X_i) \subseteq \bigcup_{i \in N} \mathcal{P}(X'_i) \iff \exists i_0 \in N.\ \bigcap_{i \in P} X_i \subseteq X'_{i_0}$$

Dimostrazione. ($\Leftarrow$) Se $\bigcap_{i \in P} X_i \subseteq X'_{i_0}$ con $i_0 \in N$, allora $\bigcap_{i \in P} \mathcal{P}(X_i) = {}^1\mathcal{P}(\bigcap_{i \in P} X_i) \subseteq \mathcal{P}\left(X'_{i_0}\right) \subseteq \bigcup_{i \in N} \mathcal{P}\left(X'_i\right)$.
($\Rightarrow$) Assumiamo che $\bigcap_{i \in P} \mathcal{P}(X_i) \subseteq \bigcup_{i \in N} \mathcal{P}\left(X'_i\right)$. L'insieme $\bigcap_{i \in P} X_i$ appartiene a tutte le $\mathcal{P}(X_i)$ per $i \in P$. Quindi, appartiene all'intersezione di tutte le $\mathcal{P}(X_i)$ per $i \in P$. A questo punto, possiamo trovare un qualche $i_0 \in N$ tale che $\bigcap_{i \in P} X'_i \in \mathcal{P}(X_{i_0})$; questo conclude la dimostrazione. □

Lemma 3.2.6. *Siano P, N due sottoinsiemi finiti di $\mathbb{T}_{\mathbf{fun}}$. Allora:*

$$\bigcap_{t \in P} \mathbb{E}(t) \subseteq \bigcup_{t \in N} \mathbb{E}(t) \iff$$

$$\exists \alpha_0 \to \beta_0 \in N.\ \forall P' \subseteq P.\ [\![\alpha_0 \backslash \left(\bigvee_{\alpha \to \beta \in P'} \alpha\right)]\!] = \varnothing \vee [\![\left(\bigwedge_{\alpha \to \beta \in P \setminus P'} \beta\right) \backslash \beta_0]\!] = \varnothing$$

[1] $\mathcal{P}(A) \cap \mathcal{P}(B) \subseteq \mathcal{P}(A \cap B)$

Dimostrazione. Siccome $P, N \subseteq \mathbb{T}_{\mathbf{fun}}$, allora ogni t in P o N è della forma $\alpha \to \beta$. Per definizione, $\mathbb{E}(\alpha \to \beta) = [\![\alpha]\!] \to [\![\beta]\!]$ e per il lemma 3.2.4 si ha $\mathbb{E}(\alpha \to \beta) = \mathcal{P}\left(\overline{[\![\alpha]\!] \times \overline{[\![\beta]\!]}^{D_\Omega}}^{D \times D_\Omega}\right)$. Il risultato segue dai lemmi 3.2.5 e 3.2.1, notando inoltre che la condizione $P' \subseteq P$ è importante: se invece valesse $P' = P$, allora $\bigcap_{\alpha\to\beta\in P\setminus P'} [\![\beta]\!] \subseteq [\![\beta_0]\!]$ non sarebbe mai vera, in quanto l'intersezione su un insieme vuoto sarebbe, in questo caso, D_Ω, rendendo così l'inclusione impossibile.

$$\bigcap_{t\in P} \mathbb{E}(t) \subseteq \bigcup_{t\in N} \mathbb{E}(t) \iff$$

$$\bigcap_{\alpha\to\beta\in P} \mathcal{P}\left(\overline{[\![\alpha]\!] \times \overline{[\![\beta]\!]}}\right) \subseteq \bigcup_{\alpha\to\beta\in N} \mathcal{P}\left(\overline{[\![\alpha]\!] \times \overline{[\![\beta]\!]}}\right) \overset{lemma3.2.5}{\iff}$$

$$\exists \alpha_0 \to \beta_0 \in N.\ \bigcap_{\alpha\to\beta\in P} \left(\overline{[\![\alpha]\!] \times \overline{[\![\beta]\!]}}\right) \subseteq \left(\overline{[\![\alpha_0]\!] \times \overline{[\![\beta_0]\!]}}\right) \iff$$

$$\exists \alpha_0 \to \beta_0 \in N.\ \overline{\bigcup_{\alpha\to\beta\in P} \left([\![\alpha]\!] \times \overline{[\![\beta]\!]}\right)} \subseteq \left(\overline{[\![\alpha_0]\!] \times \overline{[\![\beta_0]\!]}}\right) \iff$$

$$\exists \alpha_0 \to \beta_0 \in N.\ [\![\alpha_0]\!] \times \overline{[\![\beta_0]\!]} \subseteq \bigcup_{\alpha\to\beta\in P} \left([\![\alpha]\!] \times \overline{[\![\beta]\!]}\right) \iff$$

$$\exists \alpha_0 \to \beta_0 \in N.\ \left([\![\alpha_0]\!] \times \overline{[\![\beta_0]\!]}\right) \setminus \bigcup_{\alpha\to\beta\in P} \left([\![\alpha]\!] \times \overline{[\![\beta]\!]}\right) = \varnothing \overset{lemma3.2.1}{\iff}$$

$$\exists \alpha_0 \to \beta_0 \in N.\ \bigcup_{P'\subseteq P} \left([\![\alpha_0]\!] \setminus \bigcup_{\alpha\to\beta\in P'} [\![\alpha]\!]\right) \times \left(\overline{[\![\beta_0]\!]} \setminus \bigcup_{\alpha\to\beta\in P\setminus P'} \overline{[\![\beta]\!]}\right) = \varnothing \iff$$

$$\exists \alpha_0 \to \beta_0 \in N.\ \forall P' \subseteq P.\ [\![\alpha_0 \setminus \left(\bigvee_{\alpha\to\beta\in P'} \alpha\right)]\!] = \varnothing \vee [\![\left(\bigwedge_{\alpha\to\beta\in P\setminus P'} \beta\right) \setminus \beta_0]\!] = \varnothing$$

Infatti

$$\begin{aligned}\overline{[\![\beta_0]\!]} \setminus \bigcup_{\alpha\to\beta\in P\setminus P'} \overline{[\![\beta]\!]} &= \overline{[\![\beta_0]\!]} \setminus \left(\overline{\bigcap_{\alpha\to\beta\in P\setminus P'} [\![\beta]\!]}\right) \\ &= \overline{[\![\beta_0]\!]} \cap \left(\bigcap_{\alpha\to\beta\in P\setminus P'} [\![\beta]\!]\right) \\ &= \left(\bigcap_{\alpha\to\beta\in P\setminus P'} [\![\beta]\!]\right) \setminus [\![\beta_0]\!]\end{aligned}$$

□

Lemma 3.2.6 indica come decomporre il sottotipaggio tra frecce. Ad esempio, possiamo dedurre dal lemma che $\mathbb{E}\,((\alpha_1 \to \beta_1) \wedge (\alpha_2 \to \beta_2)) \subseteq \mathbb{E}\,(\alpha \to \beta)$ se e solo se le seguenti proprietà valgono:

- $[\![\alpha]\!] = \varnothing$ oppure $[\![\beta_1 \wedge \beta_2]\!] \subseteq [\![\beta]\!]$
- $[\![\alpha]\!] \subseteq [\![\alpha_1]\!]$ oppure $[\![\beta_2]\!] \subseteq [\![\beta]\!]$
- $[\![\alpha]\!] \subseteq [\![\alpha_2]\!]$ oppure $[\![\beta_1]\!] \subseteq [\![\beta]\!]$
- $[\![\alpha]\!] \subseteq [\![\alpha_1 \vee \alpha_2]\!]$

Lemma 3.2.3 e lemma 3.2.6, insieme alla proprietà (3.3) suggeriscono la seguente definizione e danno immediatamente il risultato del teorema 3.2.7

Definizione 10. *Sia $\mathcal{S}$ un insieme arbitrario di tipi in forma normale. Definiamo un altro insieme di tipi in forma normale $\mathbb{E}\mathcal{S}$ come segue:*

$$\mathbb{E}\mathcal{S} = \left\{\nu \mid \forall u \in U.\ \forall (P,N) \in \nu.\ \left(P \subseteq \mathbb{T}_u \Rightarrow C_u^{P,N\cap\mathbb{T}_u}\right)\right\}$$

dove

$$C^{P,N}_{\mathbf{basic}} = C \cap \bigcap_{c\in P} \mathbb{E}\,(c) \subseteq \bigcup_{c\in N} \mathbb{E}\,(c)$$

$$C^{P,N}_{\mathbf{rec}} = \forall N = N_1 \uplus \cdots \uplus N_n. \begin{cases} \mathcal{N}\left(\bigwedge_{[\widetilde{l:\tau}]\in P} \tau_1 \wedge \bigwedge_{[\widetilde{l:\tau'}]\in N_1} \neg\tau'_1\right) \in \mathcal{S} \\ \vee \\ \vdots \\ \vee \\ \mathcal{N}\left(\bigwedge_{[\widetilde{l:\tau}]\in P} \tau_n \wedge \bigwedge_{[\widetilde{l:\tau'}]\in N_n} \neg\tau'_n\right) \in \mathcal{S} \end{cases}$$

$$C^{P,N}_{\mathbf{fun}} = \exists \alpha_0 \to \beta_0 \in N.\ \forall P' \subseteq P. \begin{cases} \mathcal{N}\left(\alpha_0 \setminus \left(\bigvee_{\alpha\to\beta\in P'} \alpha\right)\right) \in \mathcal{S} \\ \vee \\ \mathcal{N}\left(\left(\bigwedge_{\alpha\to\beta\in P\setminus P'} \beta\right) \setminus \beta_0\right) \in \mathcal{S} \end{cases}$$

Diremo che $\mathcal{S}$ è una simulazione *se:*

$$\mathcal{S} \subseteq \mathbb{E}\mathcal{S}$$

Abbiamo chiamato l'insieme $\mathcal{S}$ dei tipi in forma normale, precedentemente introdotto, *simulazione*, per seguire la linea di [FCB08]. L'intuizione che sta dietro questa definizione è la seguente: se i tipi in forma normale che appartengono all'insieme $\mathcal{S}$ sono vuoti e se consideriamo gli enunciati dei Lemmi 3.2.3 e 3.2.6 come regole di riscrittura da destra a sinistra ($\Longleftarrow$), allora l'insieme $\mathbb{E}\mathcal{S}$ contiene tutti i tipi che si possono dedurre a partire dai tipi in $\mathcal{S}$ e che sono vuoti. Detto diversamente, se consideriamo gli enunciati dei Lemmi 3.2.3 e 3.2.6 come regole di inferenza per determinare quando un tipo è vuoto, allora l'insieme $\mathbb{E}\mathcal{S}$ è l'insieme delle conseguenze dirette di $\mathcal{S}$. Inoltre, il concetto di simulazione è fondamentale nello stabilire la decidibilità della relazione di sottotipaggio, che è un risultato importante se vogliamo utilizzare i nostri tipi in pratica.

Teorema 3.2.7. *Sia $[\![\,]\!] : \mathcal{T} \to \mathcal{P}(D)$ un'interpretazione insiemistica. Definiamo un insieme di forme normali $\mathcal{S}$ come segue:*

$$\mathcal{S} = \{\nu \mid [\![\nu]\!] = \varnothing\}$$

Allora:

$$\mathbb{E}\mathcal{S} = \{\nu \mid \mathbb{E}(\nu) = \varnothing\}$$

Dimostrazione. La dimostrazione segue direttamente dai Lemmi 3.2.3 e 3.2.6 e dalle condizioni 3.1, 3.2 e 3.3. □

Per concludere, diamo come corollario il criterio affinché un'interpretazione insiemistica sia un modello.

Corollario 3.2.8. *Sia* $[\![]\!] : \mathcal{T} \to \mathcal{P}(D)$ *un'interpretazione insiemistica. Definiamo come sopra* $\mathcal{S} = \{v \mid [\![v]\!] = \varnothing\}$. *Allora* $[\![]\!]$ *è un modello se e solo se* $\mathcal{S} = \mathbb{E}\mathcal{S}$.

3.3 Tipi come insiemi di valori

In questa sezione ci concentreremo sull'interpretazione dei tipi come insiemi di valori e daremo le prove di certe proprietà valide per questa interpretazione. Fissato un *bootstrap model* $[\![]\!] : \mathcal{T} \to \mathcal{P}(D)$, scriviamo semplicemente $\leq$ per indicare la relazione di sottotipaggio indotta da questo modello e scriviamo $\simeq$ per indicare la relazione di equivalenza ad essa associata, ovvero $\tau_1 \simeq \tau_2 \iff [\![\tau_1]\!] \subseteq [\![\tau_2]\!]$ e $[\![\tau_2]\!] \subseteq [\![\tau_1]\!]$. Prima di dare le proprietà soddisfatte dall'interpretazione nel mondo dei valori, introduciamo dei lemmi che ci saranno utili per verificare queste proprietà.

Lemma 3.3.1. *Se* $\Gamma \vdash e : \alpha_1$ *e* $\Gamma \vdash e : \alpha_2$, *allora* $\Gamma \vdash e : \alpha_1 \wedge \alpha_2$.

Dimostrazione. Si procede per induzione sulla struttura delle due derivazioni.

Come prima cosa, consideriamo il caso in cui l'ultima regola usata in una delle due derivazioni è la regola *(subsum)*, ad esempio:

$$\dfrac{\dfrac{\dots}{\Gamma \vdash e : \beta} \quad \beta \leq \alpha_1}{\Gamma \vdash e : \alpha_1} \qquad \dfrac{\dots}{\Gamma \vdash e : \alpha_2}$$

Per ipotesi induttiva $\Gamma \vdash e : \beta \wedge \alpha_2$. Siccome $\beta \leq \alpha_1$ si ha $\beta \wedge \alpha_2 \leq \alpha_1 \wedge \alpha_2$. Applicando la *(subsum)* si ottiene $\Gamma \vdash e : \alpha_1 \wedge \alpha_2$.

Nei casi che seguono, le due derivazioni finiscono con l'applicazione della stessa regola che dipende dal costruttore di tipo di e.

Regole *(const)*, *(var)*, *(rnd)*: Queste regole danno un solo tipo α ad e, quindi banalmente si ha: $\Gamma \vdash e : \alpha \Longrightarrow \Gamma \vdash e : \alpha \wedge \alpha$ dove $\alpha \wedge \alpha \simeq \alpha$.

Regola *(field)*: La situazione è come segue:

$$\frac{\dfrac{\ldots}{\Gamma \vdash e : [a : \alpha_1]}}{\Gamma \vdash e.a : \alpha_1} \qquad \frac{\dfrac{\ldots}{\Gamma \vdash e : [a : \alpha_2]}}{\Gamma \vdash e.a : \alpha_2}$$

Per ipotesi induttiva si ha $\Gamma \vdash e : [a : \alpha_1] \wedge [a : \alpha_2]$. A questo punto, proviamo che $[a : \alpha_1] \wedge [a : \alpha_2] \simeq [a : \alpha_1 \wedge \alpha_2]$ e applicando la regola *(field)* si ottiene il risultato.

$$\begin{aligned}
&\mathbb{E}([a : \alpha_1] \wedge [a : \alpha_2]) = \mathbb{E}([a : \alpha_1]) \cap \mathbb{E}([a : \alpha_2]) \\
&= \{R \subseteq (L \times D) \mid dom(R) \supseteq \{a\},\ \forall (r, r') \in R. \\
&\qquad\qquad (r = a \Rightarrow r' \in [\![\alpha_1]\!]) \wedge (r = a \Rightarrow r' \in [\![\alpha_2]\!])\} \\
&= \{R \subseteq (L \times D) \mid dom(R) \supseteq \{a\},\ \forall (r, r') \in R. \\
&\qquad\qquad r = a \Rightarrow r' \in [\![\alpha_1]\!] \wedge r' \in [\![\alpha_2]\!]\} \\
&= \{R \subseteq (L \times D) \mid dom(R) \supseteq \{a\},\ \forall (r, r') \in R. \\
&\qquad\qquad r = a \Rightarrow r' \in [\![\alpha_1]\!] \cap [\![\alpha_2]\!]\} \\
&= \{R \subseteq (L \times D) \mid dom(R) \supseteq \{a\},\ \forall (r, r') \in R. \\
&\qquad\qquad r = a \Rightarrow r' \in [\![\alpha_1 \wedge \alpha_2]\!]\} \\
&= \mathbb{E}([a : \alpha_1 \wedge \alpha_2])
\end{aligned}$$

Regola *(method)*: La situazione è come segue:

$$\frac{\dfrac{\ldots}{\Gamma \vdash e : [m : \alpha_1 \to \beta_1]} \quad \dfrac{\ldots}{\Gamma \vdash e' : \alpha_1}}{\Gamma \vdash e.m(e') : \beta_1} \qquad \frac{\dfrac{\ldots}{\Gamma \vdash e : [m : \alpha_2 \to \beta_2]} \quad \dfrac{\ldots}{\Gamma \vdash e' : \alpha_2}}{\Gamma \vdash e.m(e') : \beta_2}$$

Per ipotesi induttiva $\Gamma \vdash e : [m : \alpha_1 \to \beta_1] \wedge [m : \alpha_2 \to \beta_2]$ e $\Gamma \vdash e' : \alpha_1 \wedge \alpha_2$.

Proviamo che $[m : \alpha_1 \to \beta_1] \wedge [m : \alpha_2 \to \beta_2] \simeq [m : \alpha_1 \to \beta_1 \wedge \alpha_2 \to \beta_2]$.

$$
\begin{aligned}
&\mathbb{E}([m : \alpha_1 \to \beta_1] \wedge [m : \alpha_2 \to \beta_2]) = \mathbb{E}([m : \alpha_1 \to \beta_1]) \cap \mathbb{E}([m : \alpha_1 \to \beta_1]) \\
&= \{R \subseteq (L \times D) \mid dom(R) \supseteq \{m\},\ \forall (r, r') \in R. \\
&\qquad\qquad (r = m \Rightarrow r' \in [\![\alpha_1 \to \beta_1]\!]) \wedge (r = m \Rightarrow r' \in [\![\alpha_2 \to \beta_2]\!])\} \\
&= \{R \subseteq (L \times D) \mid dom(R) \supseteq \{m\},\ \forall (r, r') \in R. \\
&\qquad\qquad r = m \Rightarrow r' \in [\![\alpha_1 \to \beta_1]\!] \wedge r' \in [\![\alpha_2 \to \beta_2]\!]\} \\
&= \{R \subseteq (L \times D) \mid dom(R) \supseteq \{m\},\ \forall (r, r') \in R. \\
&\qquad\qquad r = m \Rightarrow r' \in [\![\alpha_1 \to \beta_1]\!] \cap [\![\alpha_2 \to \beta_2]\!]\} \\
&= \{R \subseteq (L \times D) \mid dom(R) \supseteq \{m\},\ \forall (r, r') \in R. \\
&\qquad\qquad r = m \Rightarrow r' \in [\![\alpha_1 \to \beta_1 \wedge \alpha_2 \to \beta_2]\!]\} \\
&= \mathbb{E}([m : \alpha_1 \to \beta_1 \wedge \alpha_2 \to \beta_2])
\end{aligned}
$$

Quindi, abbiamo che $\Gamma \vdash e : [m : \alpha_1 \to \beta_1 \wedge \alpha_2 \to \beta_2]$. A questo punto, affinché possiamo applicare la regola *(method)* ed ottenere il risultato voluto, dobbiamo provare che $\alpha_1 \to \beta_1 \wedge \alpha_2 \to \beta_2 \leq (\alpha_1 \wedge \alpha_2) \to (\beta_1 \wedge \beta_2)$.

$$
\begin{aligned}
&\mathbb{E}(\alpha_1 \to \beta_1 \wedge \alpha_2 \to \beta_2) = \mathbb{E}(\alpha_1 \to \beta_1) \cap \mathbb{E}(\alpha_2 \to \beta_2) \\
&= \{Q \subseteq (D \times D_\Omega) \mid (q, q') \in Q. \\
&\qquad\qquad (q \in [\![\alpha_1]\!] \Rightarrow q' \in [\![\beta_1]\!]) \wedge (q \in [\![\alpha_2]\!] \Rightarrow q' \in [\![\beta_2]\!])\} \\
&\subseteq \{Q \subseteq (D \times D_\Omega) \mid (q, q') \in Q. \\
&\qquad\qquad (q \in [\![\alpha_1]\!] \cap [\![\alpha_2]\!]) \Rightarrow q' \in [\![\beta_1]\!] \cap [\![\beta_2]\!]\} \\
&= \mathbb{E}((\alpha_1 \wedge \alpha_2) \to (\beta_1 \wedge \beta_2))
\end{aligned}
$$

Regola *(new)*: Consideriamo due applicazioni della regola *(new)* che assegnano a **new** $C(\widetilde{e})$ i tipi ρ_1 e ρ_2 dove:

$$
\rho_1 = [\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_{i=1\ldots n} \neg[a'_i : \alpha'_i] \wedge \bigwedge_{j=1\ldots m} \neg[m'_j : \mu'_j]
$$

$$\rho_2 = [\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_{i=n+1\ldots n'} \neg[a'_i : \alpha'_i] \wedge \bigwedge_{j=m+1\ldots m'} \neg[m'_j : \mu'_j]$$

con $\rho_1 \not\simeq \mathbf{0}$, $\rho_2 \not\simeq \mathbf{0}$, $type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$, $\Gamma \vdash \widetilde{e} : \widetilde{\beta}$ e $\widetilde{\beta} \leq \widetilde{\alpha}$ in tutti e due i casi di derivazione di tipo. Definiamo ρ come segue:

$$\rho = [\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_{i=1\ldots n'} \neg[a'_i : \alpha'_i] \wedge \bigwedge_{j=1\ldots m'} \neg[m'_j : \mu'_j]$$

Si ha $\rho \simeq \rho_1 \wedge \rho_2 \not\simeq \mathbf{0}$. A questo punto, è facile vedere che applicando la regola *(new)* si deduce il tipo ρ per **new** $C(\widetilde{e})$.

□

Consideriamo a questo punto dei lemmi che sono specifici per l'interpretazione dei tipi nel mondo dei valori.

Lemma 3.3.2. *Se $\tau_1 \leq \tau_2$ allora $[\![\tau_1]\!]_{\mathcal{V}} \subseteq [\![\tau_2]\!]_{\mathcal{V}}$. In particolare, se $\tau_1 \simeq \tau_2$ allora $[\![\tau_1]\!]_{\mathcal{V}} = [\![\tau_2]\!]_{\mathcal{V}}$.*

Dimostrazione. Consideriamo i casi per τ. Notiamo che un tipo α non può essere sottotipo di un tipo μ né viceversa.

*τ_1 e τ_2 sono tipi α***:** Conseguenza della regola *(subsum)*.

*τ_1 e τ_2 sono tipi μ***:** Per semplicità, consideriamo il caso quando il tipo è una singola freccia; il caso per un tipo μ qualsiasi è una generalizzazione immediata di questa prova. Sia $\mu_1 = \alpha_1 \to \beta_1$ e $\mu_2 = \alpha_2 \to \beta_2$ tale che $\alpha_1 \to \beta_1 \leq \alpha_2 \to \beta_2$. Per la covarianza e la contro-varianza delle frecce, questo vuol dire che:

$$\alpha_2 \leq \alpha_1 \quad \beta_1 \leq \beta_2$$

Per come abbiamo definito l'interpretazione nel mondo dei valori di un tipo μ, si ha quanto segue:

$$[\![\alpha_1 \to \beta_1]\!]_{\mathcal{V}_\mu} = \{(\alpha, z) \mid \alpha \geq \alpha_1 \wedge [z = \bot \vee (z = w \wedge fv(w) \subseteq \{x\} \wedge x : \alpha \vdash w : \beta_1)]\}$$

A questo punto, possiamo notare che la condizione $\alpha \geq \alpha_1$ implica $\alpha \geq \alpha_2$ in quanto $\alpha_1 \geq \alpha_2$. Dall'altra parte, se $x : \alpha \vdash w : \beta_1$ applicando la regola (*subsum*), otteniamo che $x : \alpha \vdash w : \beta_2$. Quanto abbiamo appena detto prova che ogni $(\alpha, z) \in [\![\alpha_1 \to \beta_1]\!]_{\mathcal{V}_\mu}$ appartiene anche a $[\![\alpha_2 \to \beta_2]\!]_{\mathcal{V}_\mu}$ e questo implica che $[\![\alpha_1 \to \beta_1]\!]_{\mathcal{V}_\mu} \subseteq [\![\alpha_2 \to \beta_2]\!]_{\mathcal{V}_\mu}$.

□

Lemma 3.3.3. $[\![\mathbf{0}]\!]_{\mathcal{V}} = \varnothing$.

Dimostrazione. Per quanto riguarda $[\![\mathbf{0}]\!]_{\mathcal{V}_\alpha}$, basta provare la contronominale, cioè se $\vdash v : \alpha$ allora $\alpha \neq \mathbf{0}$. Procediamo per induzione sulla derivazione.

Caso $v = c$**:** Banale. Basta considerare la regola *(const)*.

Caso $v =$ **new** $C(\widetilde{u})$**:** Anche questo caso è banale. Considerando la regola *(new)*, notiamo nella premessa la condizione $\alpha \neq \mathbf{0}$.

Regola *(subsum)*: Sia $\vdash v : \alpha$ e $\beta \leq \alpha$. Per ipotesi induttiva $\beta \neq \mathbf{0}$. Applicando la regola *(subsum)* si ha che $\vdash v : \alpha$ e $\alpha \neq \mathbf{0}$ in quanto α è un sovratipo di β.

Per quanto riguarda $[\![\mathbf{0}]\!]_{\mathcal{V}_\mu}$, il risultato segue direttamente dalla definizione dell'interpretazione $[\![\mu]\!]_{\mathcal{V}_\mu}$ che è $\varnothing$ se $\mu \simeq \mathbf{0}$. □

Lemma 3.3.4. $[\![\tau_1 \wedge \tau_2]\!]_{\mathcal{V}} = [\![\tau_1]\!]_{\mathcal{V}} \cap [\![\tau_2]\!]_{\mathcal{V}}$.

Dimostrazione. Consideriamo i casi per τ. Anche in questo caso la congiunzione può avvenire solo tra due tipi α o due tipi μ.

tipi α**:** Lemma 3.3.2 ci dà $\forall i \ [\![\alpha_1 \wedge \alpha_2]\!]_{\mathcal{V}_\alpha} \subseteq [\![\alpha_i]\!]_{\mathcal{V}_\alpha}$ e quindi $[\![\alpha_1 \wedge \alpha_2]\!]_{\mathcal{V}_\alpha} \subseteq [\![\alpha_1]\!]_{\mathcal{V}_\alpha} \cap [\![\alpha_2]\!]_{\mathcal{V}_\alpha}$. Lemma 3.3.1 ci dà l'inclusione opposta.

tipi μ**:** Il risultato segue direttamente dalla definizione di $[\![\mu]\!]_{\mathcal{V}_\mu}$. Siano μ_1 e μ_2 due tipi freccia generici. Il tipo $\mu = \mu_1 \wedge \mu_2$ lo possiamo scrivere nella forma standard, come abbiamo fatto nella definizione di $[\![\mu]\!]_{\mathcal{V}_\mu}$, mettendo in congiunzione tutte le frecce positive di μ_1 e μ_2 e poi in congiunzione tutte le frecce negative di μ_1 e μ_2. È facile vedere che tutte le coppie che abitano il tipo μ abitano sia μ_1 che μ_2.

□

A questo punto, siamo pronti per dare l'interpretazione dei tipi atomici α nel mondo dei valori $\mathcal{V}$. Per fare questo useremo la seguente notazione: $\vdash \mathbf{new}\ \rho(\widetilde{u}) : [\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[a'_i : \alpha'_i] \wedge \bigwedge_j \neg[m'_j : \mu'_j]$ è un'abbreviazione per $\rho = [\widetilde{a : \alpha}, \widetilde{m : \mu}] \ \wedge\ \vdash \widetilde{u} : \widetilde{\beta} \ \wedge\ \widetilde{\beta} \leq \widetilde{\alpha} \ \wedge\ \left([\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[a'_i : \alpha'_i] \wedge \bigwedge_j \neg[m'_j : \mu'_j] \not\simeq \mathbf{0}\right)$.

Lemma 3.3.5.

$$\begin{aligned}
[\![\mathbb{B}]\!]_{\mathcal{V}} =& \{c \mid \mathbb{B}_c \leq \mathbb{B}\} \\
[\![[\widetilde{a : \alpha}, \widetilde{m : \mu}]]\!]_{\mathcal{V}} =& \{\mathbf{new}\ \rho(\widetilde{u}) \in \mathcal{V} \mid \ \vdash \mathbf{new}\ \rho(\widetilde{u}) : \\
& \qquad [\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[a'_i : \alpha'_i] \wedge \bigwedge_j \neg[m'_j : \mu'_j]\}
\end{aligned}$$

inoltre, se $v \in \mathcal{V}_\alpha$ e t è un tipo atomico non compatibile con v, allora $\vdash v : \neg t$.

Dimostrazione. Per le uguaglianze sopra riportate, l'inclusione $\supseteq$ è banale. Per provare l'inclusione opposta, facciamo prima un'osservazione generale. Una derivazione $\vdash v : \alpha$ può essere descritta come l'applicazione di un'istanza di una regola corrispondente al genere di v (ovvero, *(const)* per le costanti e *(new)* per la creazione di oggetti), seguita da zero o più istanze di *(subsum)*. Quindi, possiamo trovare un tipo $\alpha' \leq \alpha$ tale che $\vdash v : \alpha'$ è ottenuto da una diretta applicazione della regola di tipaggio corrispondente a v. Se v è una costante, α' è un tipo base; se v è un nuovo oggetto, α' è un tipo record intersecato con zero o più record negati. In tutti i casi, per ogni tipo atomico t, $\alpha' \cap t \simeq \mathbf{0}$, se t e v non sono dello stesso genere, ma siccome $\alpha' \leq t$, questo vuol dire che $\alpha' \simeq \mathbf{0}$ che è impossibile per il Lemma 3.3.3. Quindi, se α è un tipo atomico t, allora v è dello stesso genere di t. Inoltre, abbiamo provato l'osservazione finale dell'enunciato del Lemma: se t e v non hanno lo stesso genere, ovvero non sono compatibili, allora $\alpha' \leq \neg t$ e quindi $\vdash v : \neg t$.

Caso $\vdash v : \mathbb{B}$**:** Il valore è necessariamente una costante c tale che $\mathbb{B}_c \leq \mathbb{B}$.

Caso $\vdash v : [\widetilde{a : \alpha}, \widetilde{m : \mu}]$: Prima di dare la prova per questo caso, osserviamo nuovamente che abbiamo utilizzato un tipo record ρ nella creazione di un oggetto e non una classe C dichiarata in $\widetilde{L}$. Questo lo abbiamo fatto per le ragioni date precedentemente;

inoltre, lo possiamo fare in quanto i tipi record ρ, che sono ben fatti come abbiamo affermato precedentemente, sono in biiezione con le classi C che si possono eventualmente dichiarare, anche se il programmatore non le ha effettivamente dichiarate in $\widetilde{L}$, tale che $type(C) = \rho$. Torniamo alla nostra prova, il valore v in questione è necessariamente un oggetto **new** $\rho(\widetilde{u})$. Abbiamo una diretta applicazione della regola *(new)*. Il tipo che assegnamo tramite questa regola è della forma:

$$T = [\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[a'_i : \alpha'_i] \wedge \bigwedge_j \neg[m'_j : \mu'_j]$$

che è $T \not\simeq \mathbf{0}$ e $T \leq \rho = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$ in quanto si ha che:

$$[\widetilde{a : \beta}, \widetilde{m : \mu}] \leq [\widetilde{a : \alpha}, \widetilde{m : \mu}]$$

Notiamo che in questo lemma abbiamo dato l'interpretazione dei tipi base $\mathbb{B}$ e dei tipi record $[\widetilde{a : \alpha}, \widetilde{m : \mu}]$. L'interpretazione di un tipo freccia, $\alpha \to \beta$ o più in generale di un tipo qualsiasi μ, lo abbiamo definito precedentemente nella sezione 2.2.4 □

Diamo il seguente lemma, che utilizzeremo per completare l'interpretazione dei tipi nel mondo dei valori.

Lemma 3.3.6. *Sia v un valore di tipo α. L'insieme $\{\alpha \in \mathcal{T} \mid (\vdash v : \alpha) \vee (\vdash v : \neg\alpha)\}$ contiene $\mathbf{0}$ ed è chiuso rispetto a $\vee$ e $\neg$, quindi anche rispetto a $\wedge$. Inoltre, tale insieme coincide con $\mathcal{T}_\alpha$.*

Dimostrazione. Sia E l'insieme introdotto dal lemma. È facile vedere che tale insieme è chiuso rispetto a $\neg$. Inoltre, si ha $\vdash v : \mathbf{1} = \neg\mathbf{0}$ per la regola *(subsum)* e quindi $\mathbf{0} \in E$. Ci rimane da dimostrare che E è chiuso rispetto a $\vee$. Siano α_1 e α_2 due elementi in E. Se $\vdash v : \alpha_1 \vee \alpha_2$ allora, banalmente $\alpha_1 \vee \alpha_2 \in E$, altrimenti se $\nvdash v : \alpha_1 \vee \alpha_2$ allora, per la *(subsum)* si ha $\nvdash v : \alpha_1$ e $\nvdash v : \alpha_2$. Siccome α_1 e α_2 sono in E, si ha che $\vdash v : \neg\alpha_1$ e $\vdash v : \neg\alpha_2$. Lemma 3.3.1 ci da $\vdash v : \neg\alpha_1 \wedge \neg\alpha_2$ e $\neg\alpha_1 \wedge \neg\alpha_2 \simeq \neg(\alpha_1 \vee \alpha_2)$. Abbiamo dimostrato che $\vdash v : \neg(\alpha_1 \vee \alpha_2)$, quindi $\neg(\alpha_1 \vee \alpha_2) \in E$ da cui, per la chiusura rispetto a $\neg$, $\alpha_1 \vee \alpha_2 \in E$.

Mostriamo ora che $E = \mathcal{T}_\alpha$. Procediamo per induzione su v. Grazie a quanto provato fino ad ora sull'insieme E, possiamo assumere che α sia un tipo atomico t. Per il Lemma 3.3.5, se il valore v e il tipo t non sono compatibili allora, $\vdash v : \neg t$. Assumiamo che v ed a siano compatibili.

Caso $v = c$**:** si ha $\vdash c : \mathbb{B}_c$. L'insieme $\mathbb{E}(\mathbb{B}_c) = Val_{\mathbb{B}_c}$ è un *singleton*, ovvero $\{c\}$, quindi $\mathbb{E}(\mathbb{B}_c) \subseteq \mathbb{E}(t)$ oppure $\mathbb{E}(\mathbb{B}_c) \subseteq \mathbb{E}(\neg t)$. Quindi $\mathbb{B}_c \leq t$ oppure $\mathbb{B}_c \leq \neg t$ e per la regola *(subsum)* si ha $\vdash c : t$ oppure $\vdash c : \neg t$.

Caso $v = \mathbf{new}\ \rho(\widetilde{u})$, $t = [\widetilde{a : \beta}, \widetilde{m : \mu}]$: è facile vedere che $\vdash v : t$ se $type(C) = \rho = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$, $\vdash \widetilde{u} : \widetilde{\beta}$ con $\widetilde{\beta} \leq \widetilde{\alpha}$ ed inoltre $[\widetilde{a : \beta}, \widetilde{m : \mu}] \leq [\widetilde{a : \alpha}, \widetilde{m : \mu}]$, altrimenti $\vdash v : \neg t$.

□

Lemma 3.3.7. $[\![\neg\tau]\!]_{\mathcal{V}} = \mathcal{V} \setminus [\![\tau]\!]_{\mathcal{V}}$.

Dimostrazione. Consideriamo i casi per τ.

$\tau = \alpha$**:** Per i Lemmi 3.3.2, 3.3.4 e per il fatto che $(\alpha \wedge \neg\alpha) \simeq \mathbf{0}$ si ha $[\![\alpha]\!]_{\mathcal{V}_\alpha} \cap [\![\neg\alpha]\!]_{\mathcal{V}_\alpha} = [\![\alpha \wedge \neg\alpha]\!]_{\mathcal{V}_\alpha} = [\![\mathbf{0}]\!]_{\mathcal{V}_\alpha} = \varnothing$. A questo punto, per l'unicità del complemento, ci basta dimostrare che $[\![\alpha]\!]_{\mathcal{V}_\alpha} \cup [\![\neg\alpha]\!]_{\mathcal{V}_\alpha} = \mathcal{V}_\alpha$ ovvero:

$$\forall v.\forall\alpha(\vdash v : \alpha) \vee (\vdash v : \neg\alpha)$$

Questo risultato segue dal Lemma 3.3.6.

$\tau = \mu$**:** Questa parte è una generalizzazione diretta della definizione di $[\![\neg(\alpha \to \beta)]\!]_{\mathcal{V}_\mu}$.

□

Lemma 3.3.8. $[\![\tau_1 \vee \tau_2]\!]_{\mathcal{V}} = [\![\tau_1]\!]_{\mathcal{V}} \cup [\![\tau_2]\!]_{\mathcal{V}}$.

Dimostrazione. Usando i Lemmi 3.3.7, 3.3.4 e 3.3.2 si ha:
$[\![\tau_1 \vee \tau_2]\!]_{\mathcal{V}} = [\![\neg((\neg\tau_1) \wedge (\neg\tau_2))]\!]_{\mathcal{V}} = \mathcal{V} \setminus ([\![\neg\tau_1]\!]_{\mathcal{V}} \cap [\![\neg\tau_2]\!]_{\mathcal{V}}) = \mathcal{V} \setminus (\mathcal{V} \setminus [\![\tau_1]\!]_{\mathcal{V}} \cap \mathcal{V} \setminus [\![\tau_2]\!]_{\mathcal{V}}) =$
$\mathcal{V} \setminus (\mathcal{V} \setminus ([\![\tau_1]\!]_{\mathcal{V}} \cup [\![\tau_2]\!]_{\mathcal{V}})) = [\![\tau_1]\!]_{\mathcal{V}} \cup [\![\tau_2]\!]_{\mathcal{V}}$. □

I Lemmi 3.3.3, 3.3.7 e 3.3.8 mostrano che $[\![]\!]_{\mathcal{V}}$ è un'interpretazione insiemistica.

3.4 Costruzione di modelli

In questa sezione vogliamo presentare la costruzione di un modello. Un'idea naive sarebbe quella di cercare un insieme D tale che $D = \mathbb{E}D$. Sfortunatamente, un tale insieme non può esistere, in quanto la cardinalità di $\mathbb{E}^{\mathbf{rec}}D$ e $\mathbb{E}^{\mathbf{fun}}D$ e quindi anche di $\mathbb{E}D$, è strettamente maggiore della cardinalità di D. Questo problema di cardinalità può essere risolto considerando solo le parti finite, come vedremo in seguito. Inoltre, è importante notare che, facendo in questo modo, non cambia la relazione di sottotipaggio.

Per ogni insieme D, scriviamo $\mathbb{E}_f D = C \uplus \mathcal{P}_f(L \times D) \uplus \mathcal{P}_f(D \times D_\Omega)$, dove $\mathcal{P}_f$ denota le parti finite, ovvero l'insieme dei sottoinsiemi finiti di un insieme.

Definizione 11. *Un'interpretazione insiemistica* $[\![]\!] : \mathcal{T} \to \mathcal{P}(D)$ *è* finitamente estensionale *se:*

- $D = \mathbb{E}_f D$
- $[\![t]\!] = \mathbb{E}(t) \cap D$ *per ogni atomo* t.

Lemma 3.4.1. *Se* $[\![]\!]$ *è un'interpretazione insiemistica finitamente estensionale, allora* $[\![\tau]\!] = \mathbb{E}(\tau) \cap D$*, per ogni tipo* τ*, e* $[\![\nu]\!] = \mathbb{E}(\nu) \cap D$ *per ogni forma normale disgiuntiva* ν.

Dimostrazione. Induzione sul tipo τ. □

I seguenti lemmi affermano che, prendendo insiemi finiti come interpretazione estensionale per tipi record e tipi freccia, non cambia la relazione di sottotipaggio.

Lemma 3.4.2. *Siano* P, N *due insiemi finiti di tipi record. Allora:*

$$\bigcap_{t \in P} \mathbb{E}(t) \subseteq \bigcup_{t \in N} \mathbb{E}(t) \iff \mathcal{P}_f(L \times D) \cap \bigcap_{t \in P} \mathbb{E}(t) \subseteq \bigcup_{t \in N} \mathbb{E}(t)$$

Dimostrazione. Conseguenza diretta del Lemma 3.2.2, dove gli insiemi $[l_1 : X_1\ ,\ l_2 : X_2\ ,\ \cdots\ ,\ l_n : X_n]$ sono finiti. Ricordiamo che, con $[l_1 : X_1\ ,\ l_2 : X_2\ ,\ \cdots\ ,\ l_n : X_n]$ abbiamo denotato l'insieme delle relazioni $R \subseteq L \times D$ che alla etichetta l_i associano un elemento dell'insieme X_i, ovvero forziamo che il dominio delle relazioni contenga le etichette $l_1, \ldots, l_n$, e per quanto riguarda le altre etichette, associano un elemento qualsiasi. Quindi, queste relazioni sono infinite.

In particolare, se prendiamo delle relazioni finite, il risultato continua a valere. Questo lo possiamo fare in quanto, praticamente, il dominio delle relazioni è sempre finito. □

Lemma 3.4.3. *Siano P, N due insiemi finiti di tipi freccia. Allora:*

$$\bigcap_{t\in P}\mathbb{E}(t)\subseteq\bigcup_{t\in N}\mathbb{E}(t)\iff\mathcal{P}_f(D\times D_\Omega)\cap\bigcap_{t\in P}\mathbb{E}(t)\subseteq\bigcup_{t\in N}\mathbb{E}(t)$$

Dimostrazione. Per la dimostrazione di questo lemma si rimanda a [FCB08]. □

A questo punto siamo pronti a dimostrare il seguente lemma, che come ci aspettavamo, afferma che le interpretazioni finitamente estensionali sono dei modelli.

Lemma 3.4.4. *Ogni interpretazione finitamente estensionale è un modello.*

Dimostrazione. Siccome $[\![\nu]\!] = \mathbb{E}(\nu)\cap D$, per ogni forma normale disgiuntiva ν, dobbiamo provare che:

$$\mathbb{E}(\nu)=\varnothing\iff\mathbb{E}(\nu)\cap D=\varnothing$$

Scriviamo

$$\mathbb{E}(\nu)=\bigcup_{u\in U}\bigcup_{(P,N)\in\nu}\left(\mathbb{E}^uD\cap\bigcap_{t\in P}\mathbb{E}(t)\setminus\bigcup_{t\in N}\mathbb{E}(t)\right)$$

Quindi, dobbiamo provare che per ogni $u\in U$ e (P,N) due insiemi finiti di atomi, vale:

$$\mathbb{E}^uD\cap\bigcap_{t\in P}\mathbb{E}(t)\subseteq\bigcup_{t\in N}\mathbb{E}(t)\iff D\cap\mathbb{E}^uD\cap\bigcap_{t\in P}\mathbb{E}(t)\subseteq\bigcup_{t\in N}\mathbb{E}(t)$$

Se $u=$ **basic**, allora $\mathbb{E}^{\mathbf{basic}}D=C\subseteq D$ (si ricordi che $D=\mathbb{E}_fD$), e l'equivalenza è banale. I casi per $u=$ **rec** e $u=$ **fun** seguono dai Lemmi 3.4.2 e 3.4.3, rispettivamente. □

3.4.1 Un modello ben fondato

In questa sezione definiremo un modello strutturale e finitamente estensionale, quindi un modello ben fondato.

Per fare questo, dobbiamo costruire un insieme D^0 tale che $D^0=\mathbb{E}_fD^0$, ovvero una soluzione all'equazione $D^0=C\uplus\mathcal{P}_f(L\times D^0)\uplus\mathcal{P}_f(D^0\times D^0_\Omega)$. Concretamente, definiamo D^0 come l'insieme dei termini finiti generati dalla seguente grammatica:

$$d ::= c \mid \{(l, d), \cdots, (l, d)\} \mid \{(d, d'), \cdots, (d, d')\}$$
$$d' ::= d \mid \Omega$$

Adesso che abbiamo definito la grammatica che genera l'insieme D^0 che cercavamo, dobbiamo definire l'interpretazione insiemistica dei tipi in questo insieme, ovvero definiamo $[\![]\!]^0 : \mathcal{T} \to \mathcal{P}(D^0)$ tale che $[\![\tau]\!]^0 = \mathbb{E}(\tau)^0 \cap D^0$. Siccome D^0 ha una struttura induttiva, l'equazione per D^0 sopra introdotta definisce la funzione $[\![]\!]^0$. Per convincerci, procediamo nel seguente modo. Definiamo un predicato binario $(d : \tau)$ con $d \in D^0$ e $\tau \in \mathcal{T}$. Il valore di verità di $(d : \tau)$ viene definito per induzione sulla coppia (d, τ) usando la struttura induttiva degli elementi di D^0 e le proprietà dei connettivi booleani, per quanto riguarda i tipi. La definizione è la seguente:

$$\begin{aligned}
&(c : \mathbb{B}) && = c \in Val_{\mathbb{B}}\\
&(\{(l_1, d_1), \cdots, (l_n, d_n)\} : [l_1 : \tau_1, \cdots, l_n : \tau_n]) && = \forall i.\ (d_i : \tau_i)\\
&(\{(d_1, d_1'), \cdots, (d_n, d_n')\} : \alpha \to \beta) && = \forall i.\ (d_i : \alpha) \Rightarrow (d_i' : \beta)\\
&(d : \tau_1 \wedge \tau_2) && = (d : \tau_1) \wedge (d : \tau_2)\\
&(d : \neg\tau) && = \neg(d : \tau)\\
&(d : \tau) && = \texttt{false} \quad \textit{altrimenti}
\end{aligned}$$

A questo punto, definiamo $[\![\tau]\!]^0 = \left\{d \in D^0 \mid (d : \tau)\right\}$. Da questa definizione, è facile vedere che l'interpretazione $[\![]\!]^0$ è un'interpretazione insiemistica e anche strutturale (si ricordi la definizione di interpretazione strutturale). Chiaramente, tale interpretazione è finitamente estensionale quindi, per il Lemma 3.4.4, è anche un modello. Concludendo, abbiamo costruito un modello ben fondato.

3.5 Chiudere il cerchio

Nella sezione 1.2.3 abbiamo discusso ampiamente le due relazioni di sottotipaggio indotte dal *bootsrap model*, ovvero $\leq_{\mathcal{B}}$ e dal modello dei valori, ovvero $\leq_{\mathcal{V}}$, questo per quanto

riguarda il $\lambda-$calcolo studiato in [FCB08]. Per questo calcolo abbiamo visto che le due relazioni coincidono, come dato dall'equivalenza 1.1 e questo è un risultato teorico molto importante. A questo punto, ci concentriamo sul nostro linguaggio ad oggetti e ci chiediamo: che relazione c'è tra i due sottotipaggi che abbiamo definito per il nostro calcolo, ovvero quella indotto da $\mathcal{B}$ e quello indotto da $\mathcal{V}$? Come ci aspettavamo la 1.1 è valida; infatti, si ha il seguente Teorema che è il risultato fondamentale che abbiamo ottenuto in questo lavoro di tesi.

Teorema 3.5.1. *Se il* bootstrap model $[\![]\!]$ *è strutturale, allora induce la stessa relazione di sottotipaggio dell'interpretazione* $[\![]\!]_{\mathcal{V}}$.

Dimostrazione. Per definizione, un modello è ben fondato se induce la stessa relazione di sottotipaggio di un'interpretazione insiemistica strutturale. Quindi, se $[\![]\!]$ è strutturale, allora è anche ben fondato. Possiamo prendere come *bootstrap model* il modello $[\![]\!]^0$ definito nella sezione 3.4. Sotto l'ipotesi di strutturalità, ci basta dimostrare che, per ogni tipo τ,

$$[\![\tau]\!]_{\mathcal{V}} = \varnothing \Longleftrightarrow \tau \simeq \mathbf{0}$$

($\Leftarrow$) È dato dal Lemma 3.3.2 e 3.3.3.
($\Rightarrow$) Proviamo per induzione che, per tutti gli elementi $d \in D$, la seguente proprietà vale:

$$\forall \tau \in \mathcal{T}\,.\, d \in [\![\tau]\!] \Longrightarrow [\![\tau]\!]_{\mathcal{V}} \neq \varnothing$$

Consideriamo un tipo τ tale che $d \in [\![\tau]\!]$. Se $d = \{(l_1, d_1), \ldots, (l_n, d_n)\} \in \mathcal{P}_f(L \times D)$, allora d appartiene all'insieme:

$$[\![\tau]\!] \cap \mathcal{P}_f(L \times D) = \bigcup_{(P,N)\in\mathcal{N}(\tau)} \left(\mathcal{P}_f(L \times D) \cap \left(\bigcap_{t\in P} [\![t]\!] \setminus \bigcup_{t\in N} [\![t]\!] \right) \right)$$

Possiamo quindi, trovare una coppia $(P, N) \in \mathcal{N}(\tau)$ tale che $d \in \mathcal{P}_f(L\times D)\cap(\bigcap_{t\in P} [\![t]\!] \setminus \bigcup_{t\in N} [\![t]\!])$. Notiamo che, se t è un atomo diverso da un tipo record, allora $\mathcal{P}_f(L \times D) \cap [\![t]\!] = \varnothing$ (ricordiamo che $[\![]\!]$ è un'interpretazione strutturale). Quindi, assumiamo che $P \subseteq \mathbb{T}_{\mathbf{rec}}$ e abbiamo

che

$$d \in \bigcap_{[\widetilde{l:\tau}]\in P} [l_1 : [\![\tau_1]\!], \ldots, l_n : [\![\tau_n]\!]] \setminus \bigcup_{[\widetilde{l:\tau'}]\in N} [l_1 : [\![\tau'_1]\!], \ldots, l_n : [\![\tau'_n]\!]]$$

Se scriviamo $d = \{(l_1, d_1), \ldots, (l_n, d_n)\}$, allora il Lemma 3.2.2 ci da una partizione di $N = N_1 \uplus \cdots \uplus N_n$ tale che $\{(l_1, d_1), \ldots, (l_n, d_n)\} \in [l_1 : [\![\gamma_1]\!], \ldots, l_n : [\![\gamma_n]\!]]$, con $d_i \in [\![\gamma_i]\!]$ e

$$\begin{cases} \gamma_1 = \bigwedge_{[\widetilde{l:\tau}]\in P} \tau_1 \setminus \bigvee_{[\widetilde{l:\tau'}]\in N_1} \tau'_1 \\ \vdots \\ \gamma_n = \bigwedge_{[\widetilde{l:\tau}]\in P} \tau_n \setminus \bigvee_{[\widetilde{l:\tau'}]\in N_n} \tau'_n \end{cases}$$

L'ipotesi induttiva applicata a $d_1, \ldots, d_n$ ci dà $[\![\gamma_1]\!]_{\mathcal{V}} \neq \varnothing, \ldots, [\![\gamma_n]\!]_{\mathcal{V}} \neq \varnothing$ e quindi anche $[\![[l_1 : \gamma_1, \ldots, l_n : \gamma_n]]\!]_{\mathcal{V}} \neq \varnothing$. Per concludere, notiamo che $[l_1 : \gamma_1, \ldots, l_n : \gamma_n] \leq \tau$, in quanto il tipo τ è una disgiunzione di tipi ($\bigcup_{(P,N)\in\mathcal{N}(\tau)}$) e noi ne abbiamo preso una coppia (P, N).

A questo punto, possiamo assumere che $d \notin \mathcal{P}_f(L \times D) = [\![[]]\!]$ (ricordiamo che [] corrisponde al record vuoto; si ha $type(Object) = []$). Quindi, $d \in [\![\tau\backslash[]]\!]$ che implica $\tau\backslash[] \not\simeq \mathbf{0}$. Siccome $[\![]\!]$ è un modello, si ha che $\varnothing \neq \mathbb{E}(\tau\backslash[]) = \mathbb{E}(\tau) \cap (\mathbb{E}D \setminus \mathbb{E}^{\mathbf{rec}}D)$. Siamo in uno dei seguenti casi:

$\mathbb{E}(\tau) \cap C \neq \varnothing$**:** Sia $c \in \mathbb{E}(\tau) \cap C$. Abbiamo $\mathbb{E}(\mathbb{B}_c) = \{c\} \subseteq \mathbb{E}(\tau)$ e quindi $\mathbb{B}_c \leq \tau$. Concludiamo che $\vdash c : \tau$, usando la (*subsum*) e ricordando che c è un valore del nostro calcolo.

$\mathbb{E}(\tau) \cap \mathbb{E}^{\mathbf{fun}}D \neq \varnothing$**:** Abbiamo che

$$\mathbb{E}(\tau) \cap \mathbb{E}^{\mathbf{fun}}D = \bigcup_{(P,N)\in\mathcal{N}(\tau) t.c P\subseteq\mathbb{T}_{\mathbf{fun}}} \left(\mathbb{E}^{\mathbf{fun}}D \cap \left(\bigcap_{t\in P} [\![t]\!] \setminus \bigcup_{t\in N} [\![t]\!]\right)\right)$$

Questo insieme non è vuoto. Per ogni coppia $(P, N) \in \mathcal{N}(\tau)$ si ha che $P = \{\alpha_1 \to \beta_1, \ldots, \alpha_n \to \beta_n\}$, $N \cap \mathbb{T}_{\mathbf{fun}} = \left\{\alpha'_1 \to \beta'_1, \ldots, \alpha'_m \to \beta'_m\right\}$ e

$$\tau_{(P,N)} = \bigwedge_{i=1\ldots n} \alpha_i \to \beta_i \setminus \bigvee_{j=1\ldots m} \alpha'_j \to \beta'_j$$

Abbiamo che $\tau = \bigvee_{(P,N)\in\mathcal{N}(\tau)} \tau_{(P,N)}$ è una disgiunzione di tipi, tutti tipi freccia, tanti quante sono le coppie $(P,N) \in \mathcal{N}(\tau)$. Siccome $\tau_{(P,N)} \not\equiv \mathbf{0}$, per definizione abbiamo che:

$$[\![\tau_{(P,N)}]\!]_{\mathcal{V}_\mu} = \{(\alpha, z) \mid \forall i.\, \alpha \geq \alpha_i \wedge \left[\left(z = \bot \wedge \forall j.\, \alpha \not\geq \alpha'_j\right) \vee (z = w \wedge fv(w) \subseteq \{x\} \wedge x : \alpha \vdash w : \beta_i)\right]\} \neq \varnothing$$

Quindi, $[\![\tau]\!]_{\mathcal{V}} = \bigcup_{(P,N)\in\mathcal{N}(\tau)} [\![\tau_{(P,N)}]\!]$. Questo chiude il cerchio.

□

3.6 Type soundness

In questa sezione stabiliremo i risultati di *type soundness*. Faremo questo nel modo standard, tramite i lemmi di *subject reduction* e *progress*, come vedremo in seguito. Prima, daremo dei lemmi tecnici.

Lemma 3.6.1. *Se D è un antecedente di C nella gerarchia delle classi, allora $type(C) \leq type(D)$.*

Dimostrazione. Procediamo per induzione sulla distanza tra C e D. Il caso base è banale. Per il passo induttivo, si ha che C estende C' che ha come suo antecedente D. Per ipotesi induttiva, $type(C') \leq type(D)$; quindi, ci basta provare che $type(C) \leq type(C')$. Questo segue dalla definizione di *type*. □

Lemma 3.6.2. *Se $\Gamma \vdash e : \alpha$, $\Gamma = \Gamma', x : \alpha'$ e $\Gamma' \vdash e' : \alpha'$, allora $\Gamma' \vdash e[^{e'}/_x] : \alpha$.*

Dimostrazione. Procediamo per induzione sulla struttura di e. Per il caso base, assumiamo $e = x$; in questo caso il risultato segue banalmente dal fatto che $\alpha = \alpha'$ e $e[^{e'}/_x] = e'$. Altrimenti, se $e \neq x$, allora $e[^{e'}/_x] = e$ e il risultato vale. Per il passo induttivo consideriamo i vari casi per e:

$e = e''.a$: per la regola *(field)* si ha $\Gamma \vdash e'' : [a : \alpha]$. Per ipotesi induttiva $\Gamma' \vdash e''[^{e'}/_x] : [a : \alpha]$ e quindi $\Gamma' \vdash (e''.a)[^{e'}/_x] : \alpha$ dato che $a \neq x$.

$e = e''.m(e_1)$**:** per la regola *(method)* si ha $\Gamma \vdash e'' : [m : \alpha_1 \to \alpha_2]$ e $\Gamma \vdash e_1 : \alpha_1$. Per ipotesi induttiva $\Gamma \vdash e''[^{e'}/_x] : [m : \alpha_1 \to \alpha_2]$ e $\Gamma \vdash e_1[^{e'}/_x] : \alpha_1$ e quindi $\Gamma \vdash (e''.m(e_1))[^{e'}/_x] : \alpha_2$ dato che $m \neq x$.

$e =$ **new** $C(\widetilde{e})$**:** per la regola *(new)* si ha $type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$, $\Gamma \vdash \widetilde{e} : \widetilde{\beta}$ con $\widetilde{\beta} \leq \widetilde{\alpha}$ e $\rho = [\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[a'_i : \alpha'_i] \wedge \bigwedge_j \neg[m'_j : \mu'_j] \not\simeq \mathbf{0}$. Per ipotesi induttiva $\Gamma' \vdash \widetilde{e}[^{e'}/_x] : \widetilde{\beta}$ e quindi $\Gamma' \vdash$ (**new** $C(\widetilde{e})$)$[^{e'}/_x] : \rho$ dato che $C \neq x$.

□

Teorema 3.6.3 (Subject reduction)**.** *Se* $\vdash e : \alpha$ *e* $e \to e'$*, allora* $\vdash e' : \alpha'$ *con* $\alpha' \leq \alpha$*.*

Dimostrazione. La dimostrazione è per induzione sulla lunghezza dell'inferenza di $e \to e'$. Ci sono due casi base.

$e =$ (**new** $C(\widetilde{u})$)$.a_i$ **e** $e' = u_i$**,** dove $type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$. Dalla regola *(field)* si ha $\vdash$ **new** $C(\widetilde{u}) : [a_i : \alpha]$. Dalla la regola *(new)* si ha $\vdash$ **new** $C(\widetilde{u}) : [\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[a'_i : \alpha'_i] \wedge \bigwedge_j \neg[m'_j : \mu'_j]$ e $\vdash \widetilde{u} : \widetilde{\beta}$, con $\widetilde{\beta} \leq \widetilde{\alpha}$. Per la regola *(subsum)*, le due derivazioni di tipo per **new** $C(\widetilde{u})$ sono compatibili solo se $[\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[a'_i : \alpha'_i] \wedge \bigwedge_j \neg[m'_j : \mu'_j] \leq [a_i : \alpha]$, i.e. $\beta_i \leq \alpha$. Quindi, $\vdash e' : \beta_i$, come voluto.

$e =$ (**new** $C(\widetilde{u})$)$.m(v)$ **e** $e' = e''[^{v}/_x, ^{\textbf{new } C(\widetilde{u})}/_{\textbf{this}}]$, dove $body(m, v, C) = \lambda x.e''$. Siccome e è tipabile, si deve avere che $\vdash$ **new** $C(\widetilde{u}) : [m : \gamma_1 \to \gamma_2]$ e $\vdash v : \gamma_1$. La prima derivazione di tipo comporta che $\vdash$ **new** $C(\widetilde{u}) : \rho$, con $[m : \gamma_1 \to \gamma_2] \geq \rho = [\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[a'_i : \alpha'_i] \wedge \bigwedge_j \neg[m'_j : \mu'_j] \not\simeq \mathbf{0}$; inoltre, $type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$ e $\vdash \widetilde{u} : \widetilde{\beta}$, con $\widetilde{\beta} \leq \widetilde{\alpha}$. Sia $C = D_0$ **extends** D_1 **extends** ... **extends** $D_n = Object$ la sequenza delle classi nella gerarchia da C a $Object$ per qualche $n \geq 0$. Ora, $body(m, v, C) = \lambda x.e''$ comporta che esiste $i \in \{0, \ldots, n-1\}$ tale che D_i contiene la definizione di metodo $\beta_i\ m(\alpha_i\ x)\{$**return** $e''; \}$, con $\vdash v : \alpha_i$, e D_i è la classe più vicina a C nella gerarchia che soddisfa questo fatto. Per la regola che controlla quando una dichiarazione di metodo è accettabile in una classe, si ha che $\beta_i\ m(\alpha_i\ x)\{$**return** $e''\}OK(D_i)$ e quindi, $x : \alpha_i$, **this** $: type(D_i) \vdash e'' : \beta_i$. Siccome $\rho \leq type(C)$, dal Lemma 3.6.1 e dal Lemma 3.6.2 applicato a **this** e a x, abbiamo

che $\vdash e''[{}^{v}/_{x}, {}^{\mathbf{new}\ C(\widetilde{u})}/_{\mathbf{this}}] : \beta_i$. Per concludere, ci basta far vedere che $\beta_i \leq \gamma_2$. Questo vale in quanto $\rho \leq [m : \gamma_1 \to \gamma_2]$ e il tipo μ assegnato a m in $type(C)$ contiene $(\alpha_i \setminus \alpha_{i-1}) \setminus \cdots \setminus \alpha_0 \to \beta_i$, dove $\alpha_j \to \beta_j$ è il tipo dichiarato per m in D_j.

Per il passo induttivo, ragioniamo per casi sull'ultima regola usata nell'inferenza. Abbiamo quattro possibili casi.

$e = e_1.a$**,** $e_1 \to e_2$ **e** $e' = e_2.a$**.** Dalla regola *(field)* si ha $\vdash e_1 : [a : \alpha]$ e, per ipotesi induttiva, $\vdash e_2 : \beta$ per qualche $\beta \leq [a : \alpha]$. Dalla *(subsum)* $\vdash e_2 : [a : \alpha]$ e quindi, di nuovo per la regola *(field)*, $\vdash e' : \alpha$.

$e = e_1.m(v)$**,** $e_1 \to e_2$ **e** $e' = e_2.m(v)$**.** Dalla regola *(method)* si ha $\vdash e_1 : [m : \alpha_1 \to \alpha_2]$ e $\vdash v : \alpha_1$. Per ipotesi induttiva, $\vdash e_2 : \beta$, per qualche $\beta \leq [m : \alpha_1 \to \alpha_2]$. Di nuovo, per la *(subsum)* e per la regola *(method)*, $\vdash e' : \alpha_2$.

$e = e_0.m(e_1)$**,** $e_1 \to e_1'$ **e** $e' = e_0.m(e_1')$**.** Dalla regola *(method)* si ha $\vdash e_0 : [m : \alpha_1 \to \alpha_2]$ e $\vdash e_1 : \alpha_1$. Per ipotesi induttiva, $\vdash e_1' : \alpha_1'$, per qualche $\alpha_1' \leq \alpha_1$. Di nuovo, per la *(subsum)* e per la regola *(method)*, $\vdash e' : \alpha_2$.

$e = \mathbf{new}\ C(e_1, \ldots, e_i, \ldots, e_k)$**,** $e_i \to e_i'$ **e** $e' = \mathbf{new}\ C(e_1, \ldots, e_i', \ldots, e_k)$**.** Dalla regola *(new)* si ha $\vdash e_i : \alpha_i$. Per ipotesi induttiva, $\vdash e_i' : \alpha_i'$, per qualche $\alpha_i' \leq \alpha_i$. Di nuovo, per la *(subsum)* e per la regola *(new)*, è derivabile che e' ha lo stesso tipo di e.

□

Teorema 3.6.4 (Progress)**.** *Se* $\vdash e : \alpha$ *con* e *un'espressione chiusa, allora* o e *è un valore oppure esite un* e' *tale che* $e \to e'$.

Dimostrazione. La dimostrazione è per induzione sulla struttura di e. Siccome, e è un'espressione chiusa, l'unico caso base è quando e è un valore base, in questo caso l'affermazione segue banalmente. Per il passo induttivo, procediamo per casi sulla forma di e.

$e = e_0.a$**.** Dalla regola *(field)*, $\vdash e_0 : [a : \alpha]$; per ipotesi induttiva, e_0 è un valore (e in questo caso si deve avere che $e_0 = \mathbf{new}\ C(\widetilde{u})$, per qualche C e $\widetilde{u}$) oppure $e_0 \to e_0'$, per qualche

e'_0. Nel secondo caso, facilmente concludiamo che $e \to e'$, con $e' = e'_0.a$. Nel primo caso, dalla regola *(new)* si ha, $\vdash$ **new** $C(\widetilde{u})$: $[\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[a'_i : \alpha'_i] \wedge \bigwedge_j \neg[m'_j : \mu'_j]$, con $type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$, $\vdash \widetilde{u} : \widetilde{\beta}$ e $\widetilde{\beta} \leq \widetilde{\alpha}$. Inoltre, $[\widetilde{a : \beta}, \widetilde{m : \mu}] \wedge \bigwedge_i \neg[a'_i : \alpha'_i] \wedge \bigwedge_j \neg[m'_j : \mu'_j] \leq [a : \alpha]$ implica che esiste un i tale che $a = a_i$; quindi, $e \to e'$, con $e' = u_i$.

$e = e_0.m(e_1)$**.** Dalla regola *(method)*, $\vdash e_0 : [m : \gamma_1 \to \gamma_2]$. Come nel caso precedente, la parte interessante è quando e_0 è un valore (in particolare, $e_0 =$ **new** $C(\widetilde{u})$, per qualche C e $\widetilde{u}$); con un ragionamento simile, possiamo dire che $type(C) = [\widetilde{a : \alpha}, \widetilde{m : \mu}]$, $\vdash \widetilde{u} : \widetilde{\beta}$ con $\widetilde{\beta} \leq \widetilde{\alpha}$ e $m = m_i$ per qualche i.

Come prima cosa, consideriamo il caso in cui e_1 non è un valore. Per ipotesi induttiva, esiste un e'_1 tale che $e_1 \to e'_1$; quindi, $e \to e'$, con $e' = e_0.m(e'_1)$.

Assumiamo adesso che $e_1 = v$. Per definizione della funzione *type*, si deve avere che $\mu_i = \alpha \to \beta \ \wedge \bigwedge_{i=1}^{n} \alpha_i \setminus \alpha \to \beta_i \leq \bigwedge_{i=1}^{n} \alpha_i \to \beta_i = T'(m_i)$, dove C estende D in $\widetilde{L}$ e $type(D) = T'$. Inoltre, dal sottotipaggio si ha $\mu_i \leq \gamma_1 \to \gamma_2$. A questo punto, ci basta provare che $body(m, v, C) = \lambda x.e''$, per qualche e''. È da notare che C non può essere *Object*, altrimenti $\vdash e : \alpha$ non può valere per nessun α. Adesso, procediamo con una seconda induzione sulla distanza tra D e *Object* nella gerarchia delle classi definita da $\widetilde{L}$. Il caso base si ha quando D è *Object*: $\mu_i = \alpha \to \beta$ e quindi, $\gamma_1 \leq \alpha$. Per la regola *(subsum)*, $\vdash v : \alpha$ e quindi $body(m, v, C) = \lambda x.e''$, dove C contiene la dichiarazione di metodo $\beta\, m(\alpha\, x)\{$**return** $e''; \}$. Per il passo induttivo, se $T'(m_i) = \mathbf{0} \to \mathbf{1}$, procediamo come nel caso quando $D = Object$; altrimenti, per ipotesi induttiva (la seconda induzione), sappiamo che $body(m, v, D) = \lambda\widetilde{x}.e''$. Se C non contiene nessuna dichiarazione per il metodo m, questo ci basta per concludere; altrimenti, sia $\beta\, m(\alpha\, x)\{$**return** $e'''; \}$ la dichiarazione in questione. Se $\vdash v : \alpha$, allora $body(m, v, D) = \lambda\widetilde{x}.e'''$; altrimenti, $body(m, v, D) = \lambda\widetilde{x}.e''$. Questo ci basta per concludere.

$e =$ **new** $C(\widetilde{e})$**.** Se $\widetilde{e}$ è una sequenza di valori, allora e è un valore. Altrimenti, esiste un i tale che e_i non è un valore; per ipotesi induttiva, abbiamo che $e_i \to e'_i$ con $e' =$ **new** $C(e_1, \ldots, e'_i, \ldots, e_k)$ e quindi concludiamo.

□

CAPITOLO 4

Discussioni sul calcolo

4.1 Sottotipaggio strutturale vs. sottotipaggio nominale

Attualmente, si hanno due paradigmi di sottotipaggio, il sottotipaggio nominale, che risulta più diffuso, e il sottotipaggio strutturale. In un linguaggio orientato agli oggetti con sottotipaggio strutturale, un tipo ρ_1 è sottotipo di un tipo ρ_2 se i suoi campi e metodi sono sottoinsiemi dei campi e metodi di ρ_2, dove con ρ_1 e ρ_2 abbiamo denotato due tipi record. Come conseguenza, l'interfaccia di una classe è semplicemente composta dai campi e metodi (in Java, pubblici); non c'è bisogno di dichiarare un'interfaccia separata. Dall'altra parte, in un linguaggio con sottotipaggio nominale, ρ_1 è sottotipo di ρ_2 se e solo se è dichiarato tale. Quindi, il sottotipaggio strutturale è considerato *intrinseco*, mentre quello nominale è considerato *dichiarativo*. Ognuno di loro ha le sue forze e le sue debolezze. Ma tutt'ora non si ha un modello formale che integra i due paradigmi di sottotipaggio.

Il sottotipaggio strutturale offre una serie di vantaggi. Spesso risulta più espressivo di quello nominale, in quanto non è necessario che le relazioni di sottotipaggio siano dichiarate in anticipo. È intrinseco e quindi solleva il programmatore dal dichiarare esplicitamente le relazioni di sottotipaggio, come richiede quello nominale. Questo fatto ha il vantaggio

di supportare la riusabilità. Illustriamo con un esempio i benefici della riusabilità nel sottotipaggio strutturale: supponiamo che una classe A ha i metodi `foo()`, `a()` e `b()`, e una classe B ha i metodi `foo()`, `x()` e `y()`. In un linguaggio con sottotipaggio strutturale, A e B condividono un'interfaccia comune implicita {`foo`}, e il programmatore può scrivere codice basandosi su questa interfaccia. In un linguaggio con sottotipaggio nominale, invece, siccome il programmatore non ha dichiarato in anticipo un'interfaccia `Ifoo` e non l'ha implementata con le classi A e B, non c'è modo di scrivere codice partendo da un'interfaccia comune a queste due classi. Con il sottotipaggio strutturale, se una classe C viene aggiunta in seguito nella gerarchia delle classi e contiene un metodo `foo()`, allora tale classe condivide l'interfaccia implicita. Se il programmatore aggiunge nuovi metodi alla classe A o B, il tipo dell'interfaccia cambierà automaticamente.

Anche il sottotipaggio nominale ha i suoi vantaggi. Permette al programmatore di esprimere esplicitamente quali sono i suoi obbiettivi di design. Una gerarchia di classi, che induce una gerarchia di sottotipaggio definita dal programmatore, in un certo senso, è una documentazione che specifica come le diverse parti del programma devono lavorare insieme. Inoltre, dare una specifica esplicita ha il vantaggio di prevenire relazioni di sottotipaggio 'accidentali' e magari anche non volute. In più, i messaggi d'errore sono generalmente più comprensibili, in quanto ogni tipo in un errore di tipo è dichiarato esplicitamente dal programmatore, così come sono dichiarate anche le relazioni di sottotipaggio fra tipi.

Il sottotipaggio strutturale è più popolare nella comunità di ricerca ed è usato in linguaggi come Ocaml [LDG+08], PolyTOIL [VGBSF03], Moby [FR99], Strongtalk [BG93] e in diversi calcoli e sistemi di tipi, come in [Car88, AC91]; mentre i linguaggi ad oggetti più diffusi usano il sottotipaggio nominale.

Per quanto detto precedentemente, la definizione di sottotipaggio strutturale come inclusione di sottoinsiemi corrispondenti a campi e metodi di un tipo record, ci rimanda alla definizione della funzione $type(C)$, con $C \in \widetilde{L}$, definita nella sezione 2.1.3. Nello stabilire il tipo di una classe C, si controlla per ogni campo e per ogni metodo in C, che i tipi associati ad essi siano sottotipi dei tipi associati agli stessi campi e metodi nella classe D, dove C estende D in $\widetilde{L}$; questo se in D tali campi e metodi sono dichiarati. Siccome il sottotipaggio nella nostra trattazione è stabilito come inclusione di sottoinsiemi, è molto naturale

adoperare il sottotipaggio strutturale.

Dall'altra parte, è importante notare che il nostro linguaggio ad oggetti è un *core language* di Java, infatti la sintassi è quella di *Featherweight Java*, come si può vedere in [IPW01]. Il costrutto **class** C **extends** D $\{\ldots\}$, per come è definito in [IPW01], induce una relazione di sottotipaggio, ovvero $C \leq D$, e questo viene dato esplicitamente dalla seguente regola:

$$\frac{\textbf{class } C \textbf{ extends } D \{\ldots\}}{C \leq D}$$

Questo è un puro sottotipaggio nominale. Nel nostro calcolo, questo costrutto non viene utilizzato per poi stabilire la relazione di sottotipaggio, anzi è piuttosto il contrario. Ovvero, possiamo scrivere **class** C **extends** D $\{\ldots\}$, se $C \leq D$, come viene determinato da $type(C)$. Facendo così, facilitiamo lo stabilire la relazione di sottotipaggio, in quanto, per due classi C e D, invece di controllare campo per campo e metodo per metodo che il sottotipaggio viene rispettato, ci basta controllare se C estende D. Dall'altra parte, è importante osservare che **extends** ci serve anche per ereditare i campi e i metodi di D non dichiarati in C. Concludendo, nel nostro calcolo adoperiamo un sottotipaggio strutturale, utilizzando però costrutti del sottotipaggio nominale per facilitare il problema dello stabilire quale tipo è sottotipo dell'altro, per quanto riguarda le classi. Abbiamo integrato quindi, i due paradigmi di sottotipaggio.

4.2 Linguaggio ad oggetti di ordine superiore

I linguaggi di ordine superiore (dall'inglese *higher-order language*) sono dei linguaggi che danno la possibilità di usare le funzioni come valori. Ovvero, le funzioni possono essere passate come argomenti ad altre funzioni oppure possono essere il valore di ritorno di un'altra funzione. Questa è una caratteristica più diffusa tra alcuni linguaggi di programmazione funzionali, ma anche i linguaggi orientati agli oggetti la adoperano. Un esempio è il linguaggio Ruby [Rub] che combina tutti i vantaggi della programmazione ad oggetti e della programmazione *higher-order*.

Il nostro linguaggio, presentato nella sezione 2.1.2, non supporta metodi di ordine superiore. Notiamo che un metodo viene tipato con un tipo $\alpha \to \beta$, dove α e β non possono

essere tipi di altri metodi. Infatti, abbiamo presentato i tipi dei metodi con un altra sintassi, μ. Dall'altra parte, anche se non abbiamo metodi di ordine superiore, non risulta difficile dare una codifica da un linguaggio ad oggetti di ordine superiore al nostro linguaggio ad oggetti che è di primo ordine. Il linguaggio di ordine superiore che vogliamo considerare differisce dal nostro solo in questo fatto: è di ordine superiore e i tipi sono introdotti dall'unica grammatica che segue:

$$\tau ::= \mathbf{0} \mid \mathbf{1} \mid \mathbb{B} \mid [\widetilde{l : \tau}] \mid \tau \to \tau \mid \tau \wedge \tau \mid \neg\tau$$

A questo punto, vogliamo presentare una funzione di codifica dal linguaggio che usa la sintassi dei tipi sopra presentata al linguaggio che stiamo studiando noi. Prima di tutto, notiamo che un tipo freccia $\alpha \to (\beta \to \gamma)$, lo possiamo riscrivere come il tipo $(\alpha, \beta) \to \gamma$ che corrisponde al metodo che prende in ingresso due argomenti di tipo α e β e ritorna un valore di tipo γ; quindi ci siamo ridotti al primo ordine. Invece, il tipo $(\alpha \to \beta) \to \gamma$ è di ordine superiore e quello che ci interessa è dare una codifica da questo tipo a un tipo del nostro calcolo. Lo faremmo nel seguente semplice modo:

$$(\alpha \to \beta) \to \gamma \implies [w : \alpha \to \beta] \to \gamma$$

dove $[w : \alpha \to \beta]$ è un tipo record nel nostro calcolo, che corrisponde ad una classe che ha solo un metodo di tipo $\alpha \to \beta$. Ricapitolando, il metodo di ordine superiore di tipo $(\alpha \to \beta) \to \gamma$, corrisponde nel nostro calcolo ad un metodo di primo ordine di tipo $[w : \alpha \to \beta] \to \gamma$ che prende come argomento un oggetto che istanzia una classe di tipo $[w : \alpha \to \beta]$ e restituisce un valore di tipo γ.

4.3 Esempi

In questa sezione, vogliamo dare degli esempi di utilizzo del nostro linguaggio, facendo vedere come possiamo implementare dei costrutti che sono standard per i linguaggi ad oggetti. Questo è molto importante, in quanto noi ci siamo concentrati su un *core language*, e dando questi esempi ci convinciamo che il nostro linguaggio è espressivo e potente. Cominciamo col presentare come possiamo definire delle classi ricorsive.

4.3.1 Definizioni ricorsive di classi

Nel nostro calcolo è possibile dare delle definizioni ricorsive di classi; ad esempio, una lista di interi può essere scritta nel modo seguente:

$$
\begin{array}{l}
\textbf{class}\ \mathit{intList}\ \textbf{extends}\ \mathit{Object}\ \{ \\
\quad \textbf{int}\ \mathit{val}; \\
\quad \alpha\ \mathit{succ}; \\
\quad \mathit{intList}(\textbf{int}\ x, \alpha\ y)\{\textbf{this}.\mathit{val} = x; \textbf{this}.\mathit{succ} = y\} \\
\}
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ \mathit{app}\ \textbf{extends}\ \mathit{Object}\ \{ \\
\quad \alpha\ m()\ \{\textbf{return new}\ \mathit{intList}(0, \textbf{this}.m())\}; \\
\}
\end{array}
$$

dove α è definito come il tipo record $[\mathit{val} : \textbf{int}, \mathit{succ} : \alpha]$. A questo punto, la lista $\langle 1, 2\rangle$ viene scritta come

$$\textbf{new}\ \mathit{intList}(1, \textbf{new}\ \mathit{intList}(2, (\textbf{new}\ \mathit{app}()).m()))$$

Il problema con questo esempio è che il termine sopra presentato, anche se è ben-tipato, non crea nessun nuovo oggetto in quanto il metodo *m* non termina mai. Ricordiamo che abbiamo visto la stessa problematica quando abbiamo dato la definizione di modello ben fondato. Per risolvere questo problema, assumiamo un nuovo valore base **null** e un nuovo tipo base **unit** che ha come suo unico valore, **null**. In Java, si assume che il tipo **unit** è un sottotipo di qualsiasi tipo di classi. Nel nostro sistema, siccome i tipi sono complessi (soprattutto per l'esistenza della negazione), questa assunzione non può essere fatta. A

questo punto, le liste di interi possono essere definite come:

$$
\begin{array}{lll}
L_{intList} & = & \textbf{class}\ intList\ \textbf{extends}\ Object\ \{ \\
 & & \quad \textbf{int}\ val; \\
 & & \quad (\alpha \vee \textbf{unit})\ succ; \\
 & & \quad intList(\textbf{int}\ x, (\alpha \vee \textbf{unit})\ y)\{\textbf{this}.val = x; \textbf{this}.succ = y\} \\
 & & \}
\end{array}
$$

dove α è $[val : \textbf{int}, succ : (\alpha \vee \textbf{unit})]$. Adesso, possiamo creare la lista $\langle 1, 2 \rangle$ come il seguente valore:

$$\textbf{new}\ intList(1, \textbf{new}\ intList(2, \textbf{null}))$$

4.3.2 Costrutti Java-like

In questa sezione vogliamo far vedere come possiamo implementare costrutti classici dei linguaggi di programazzione, come ad esempio *if-then-else*, *instanceof* oppure *exceptions*. Come annunciato precedentemente, questo fa vedere l'espressività del nostro linguaggio.

L'espressione **if** e **then** e_1 **else** e_2 può essere implementata dalla seguente definizione di classe:

$$
\begin{array}{l}
\textbf{class}\ test\ \textbf{extends}\ Object\ \{ \\
\quad \alpha\ m(\{\textbf{true}\}\ x)\{\textbf{return}\ e_1\} \\
\quad \alpha\ m(\{\textbf{false}\}\ x)\{\textbf{return}\ e_2\} \\
\}
\end{array}
$$

dove {**true**} e {**false**} sono tipi *singleton* che tipano solo il valore **true** e **false**, rispettivamente, ed inoltre α è il tipo di e_1 e e_2. A questo punto, **if** e **then** e_1 **else** e_2 può essere simulato da

$$(\textbf{new}\ test()).m(e)$$

Da notare che questo termine può essere tipato, in quanto *test* ha tipo $[m : (\{\textbf{true}\} \to \alpha) \wedge (\{\textbf{false}\} \to \alpha)] = [m : (\{\textbf{true}\} \vee \{\textbf{false}\}) \to \alpha] = [m : \textbf{bool} \to \alpha]$.

Infatti, in [FCB08] è stato dimostrato che $(\alpha_1 \to \alpha) \wedge (\alpha_2 \to \alpha) = (\alpha_1 \vee \alpha_2) \to \alpha$ e quindi, $\{\textbf{true}\} \vee \{\textbf{false}\} = \textbf{bool}$.

Il costrutto e **instanceof** α controlla se e ha tipo α e può essere implementato in modo simile a *if-then-else*:

$$
\begin{array}{l}
\textbf{class } instOf \textbf{ extends } Object \ \{ \\
\quad \textbf{bool } m_{\alpha_1}(\alpha_1\ x)\{\textbf{return true}\} \\
\quad \textbf{bool } m_{\alpha_1}(\neg\alpha_1\ x)\{\textbf{return false}\} \\
\quad \ldots \\
\quad \textbf{bool } m_{\alpha_k}(\alpha_k\ x)\{\textbf{return true}\} \\
\quad \textbf{bool } m_{\alpha_k}(\neg\alpha_k\ x)\{\textbf{return false}\} \\
\}
\end{array}
$$

dove $\alpha_1, \ldots, \alpha_k$ sono i tipi che fanno da argomenti per **instanceof** nel programma. A questo punto, e **instanceof** α può essere simulato da:

$$(\textbf{new } instOf()).m_\alpha(e)$$

Come ultimo, consideriamo il costrutto *try-catch*. L'espressione **try** e **catch**$(\alpha\ x)\ e'$ valuta l'espressione e; se viene sollevata un'eccezzione di tipo α durante la valutazione, allora viene valutata l'espressione e'. Prima di tutto, assumiamo che ogni eccezzione è un oggetto di una sottoclasse della classe $Exception$ che a sua volta estende $Object$. Inoltre, ogni metodo che può sollevare un'eccezione di tipo α deve specificare questo fatto nel suo tipo di ritorno (questo richiama l'utilizzo della parola chiave **throws** in Java); in particolare, se il tipo di m è $\alpha_1 \to \alpha_2$ e può sollevare un'eccezione di tipo α, allora m deve essere dichiarato come:

$$(\alpha \vee \alpha_2)\ \ m(\alpha_1\ x)\{\ldots\}$$

Infatti, ogni espressione della forma **throw** e all'interno di m sarà tradotta nel nostro calcolo come **return** e. A questo punto, possiamo tradurre **try** e **catch**$(\alpha\ x)\ e'$ come:

$$\textbf{let } x = e \textbf{ in } (\textbf{if } (x \textbf{ instanceof } \alpha) \textbf{ then } e' \textbf{ else } x)$$

In questa espressione, abbiamo utilizzato un costrutto standard **let** $y = e_1$ **in** e_2; questo può essere implementato nel nostro calcolo come:

$$\textbf{this}.let(e_1)$$

una volta che abbiamo aggiunto alla classe il metodo:

$$\alpha_2\ let(\alpha_1\ y)\{\textbf{return}\ e_2\}$$

con α_1 e α_2 tipi di e_1 e e_2, rispettivamente.

CAPITOLO 5

Conclusioni e Sviluppi Futuri

In questa tesi è stato affrontato il problema della relazione di sottotipaggio semantico. Come abbiamo già visto, in questo approccio, i tipi del linguaggio di interesse vengono aumentati con i connettivi booleani $\wedge$ (intersezione), $\vee$ (unione) e $\neg$ (complemento); i tipi vengono interpretati come sottoinsiemi di un qualche insieme D e la relazione di sottotipaggio viene definita come inclusione di sottoinsiemi che denotano tipi sintattici. Nel primo capitolo di questa tesi abbiamo introdotto in dettaglio l'approccio semantico al sottotipaggio dando anche i vantaggi che questo approccio offre. In particolare, ci siamo concentrati su due linguaggi: il $\lambda-$calcolo e il $\pi-$calcolo e abbiamo definito il sottotipaggio per questi calcoli.

L'obbiettivo del lavoro svolto in questa tesi è stato quello di definire il sottotipaggio semanticamente per un semplice linguaggio ad oggetti. Nel secondo capitolo abbiamo introdotto un *core-language*, ispirato a *Featherweight Java* [IPW01]. Abbiamo introdotto i tipi di questo linguaggio che abbiamo aumentato opportunamente con i connettivi booleani. Poi abbiamo dato l'interpretazione insiemistica dei tipi come sottoinsiemi di un qualche insieme D, $[\![]\!] : \mathcal{T} \to \mathcal{P}(D)$. Come abbiamo già detto vi sono diversi insiemi dove possiamo

interpretare i nostri tipi. Per gli obbiettivi del nostro lavoro ci basta che ne esista almeno uno. Nella sezione 3.4 abbiamo dimostrato che esiste un modello dei tipi che è strutturale e ben fondato. Abbiamo chiamato questo modello *bootstrap model* e lo abbiamo denotato con $\mathcal{B}$ e su questo e sull'interpretazione $[\![]\!] : \mathcal{T} \to \mathcal{P}(\mathcal{B})$ abbiamo sviluppato la nostra teoria. Questa interpretazione induce una relazione di sottotipaggio definito come segue:

$$\tau_1 \leq_{\mathcal{B}} \tau_2 \stackrel{def}{\iff} [\![\tau_1]\!]_{\mathcal{B}} \subseteq [\![\tau_2]\!]_{\mathcal{B}}$$

Infine, abbiamo definito una nuova interpretazione dei tipi, quella come insiemi di valori. Questa interpretazione induce una relazione di sottotipaggio definita come segue:

$$\tau_1 \leq_{\mathcal{V}} \tau_2 \stackrel{def}{\iff} [\![\tau_1]\!]_{\mathcal{V}} \subseteq [\![\tau_2]\!]_{\mathcal{V}}$$

Questa relazione di sottotipaggio poteva essere diversa da quella da cui siamo partiti, ovvero l'interpretazione in D. Invece, abbiamo dimostrato nel terzo capitolo che il *bootstrap model* e il modello dei valori inducono la stessa relazione di sottotipaggio; così abbiamo chiuso il cerchio di cui abbiamo parlato ampiamente e che abbiamo introdotto nel primo capitolo. Questo è il risultato fondamentale del nostro lavoro, dato dal Teorema 3.5.1. Inoltre, in questo capitolo abbiamo dimostrato diverse proprietà valide per l'interpretazione dei tipi come insiemi di valori; ad esempio che questa interpretazione è un'interpretazione insiemistica. Abbiamo concluso con le dimostrazioni di *type-soundness*, dato dai Lemmi 3.6.3 di *subject reduction* e 3.6.4 di *progress*.

Nell'ultimo capitolo abbiamo introdotto il sottotipaggio strutturale come un nuovo paradigma di sottotipaggio per i linguaggi ad oggetti e abbiamo messo a confronto questo paradigma con quello nominale che è un paradigma standard. Abbiamo dato una variante del linguaggio ad oggetti introdotto, un linguaggio di ordine superiore e per concludere abbiamo presentato una serie di esempi di programmazione che ci fanno capire la potenza e l'espressività del nostro calcolo.

Un possibile sviluppo futuro del lavoro presentato in questo elaborato è quello di implementare un linguaggio prototipo ad oggetti $\mathbb{C}$Obj che utilizza tipi booleani e una relazione di sottotipaggio definita semanticamente. Inoltre, vogliamo ampliare la nostra trattazione

teorica con nuove dimostrazioni di teoremi validi per il nostro calcolo, come ad esempio costruzione di modelli universali, e derivare algoritmi per stabilire il sottotipaggio.

Bibliografia

[AC91] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In *POPL '91: Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 104–118, New York, NY, USA, 1991. ACM.

[AC94] Martìn Abadi and Luca Cardelli. A theory of primitive objects - untyped and first-order systems. In *TACS '94: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 296–320, London, UK, 1994. Springer-Verlag.

[AGH06] Ken Arnold, James Gosling, and David Holmes. *Il linguaggio Java, Manuale ufficiale*. Pearson Education, 2006.

[AW93] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 31–41, New York, NY, USA, 1993. ACM.

[Bar84] Henk P. Barendregt. *The Lambda Calculus: Its Syntax And Semantics*. Northholland, 1984.

[BG93] Gilad Bracha and David Griswold. Strongtalk: typechecking smalltalk in a production environment. *SIGPLAN Not.*, 28(10):215–230, 1993.

[Car88] Luca Cardelli. Structural subtyping and the notion of power type. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–79, New York, NY, USA, 1988. ACM.

[Cas05] Giuseppe Castagna. Semantic subtyping: Challenges, perspectives, and open problems. In *ICTCS*, pages 1–20, 2005.

[CF05] Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 198–199, New York, NY, USA, 2005. ACM.

[CNV08] Giuseppe Castagna, Rocco De Nicola, and Daniele Varacca. Semantic subtyping for the pi-calculus. *Theor. Comput. Sci.*, 398(1-3):217–242, 2008.

[Dam94] Flemming M. Damm. Subtyping with union types, intersection types and recursive types. In *TACS '94: Proceedings of the International Conference on Theoretical Aspects of Computer Software*, pages 687–706, London, UK, 1994. Springer-Verlag.

[FCB08] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. *J. ACM*, 55(4):1–64, 2008.

[FR99] Kathleen Fisher and John Reppy. The design of a class mechanism for moby. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 37–49, New York, NY, USA, 1999. ACM.

[GM08] Joseph Gil and Itay Maman. Whiteoak: introducing structural typing into java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 73–90, New York, NY, USA, 2008. ACM.

[HP01] Haruo Hosoya and Benjamin C. Pierce. Regular expression pattern matching for xml. *SIGPLAN Not.*, 36(3):67–80, 2001.

[HP03] Haruo Hosoya and Benjamin C. Pierce. Xduce: A statically typed xml processing language. *ACM Trans. Internet Technol.*, 3(2):117–148, 2003.

[HVP05] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for xml. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, 2005.

[IPW01] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001.

[KL89] Michael Kifer and Georg Lausen. F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. *SIGMOD Rec.*, 18(2):134–146, 1989.

[LDG+08] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system, release 3.11*, 2008.

[MA08] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 260–284, Berlin, Heidelberg, 2008. Springer-Verlag.

[MA09] Donna Malayeri and Jonathan Aldrich. Is structural subtyping useful? an empirical study. In *ESOP*, pages 95–111, 2009.

[Mil99] Robin Milner. *Communicating and mobile systems: the π−calculus*. Cambridge University Press, New York, NY, USA, 1999.

[Rub] Ruby programming language.

[SW03] Davide Sangiorgi and David Walker. *The Pi-Calculus: A Theory of Mobile Processes*. Cambridge University Press, New York, NY, USA, 2003.

[VGBSF03] Robert Van Gent, Kim B. Bruce, Angela Schuett, and Adrian Fiech. Polytoil: A type-safe polymorphic object-oriented language. *ACM Trans. Program. Lang. Syst.*, 25(2):225–290, 2003.