

UNIVERSITÀ DI PISA
FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI
CORSO DI LAUREA IN INFORMATICA
A. A. 2001/02

TESI DI LAUREA

UNA SEMANTICA GENERALE
PER LA VERIFICA E L'INFERENZA
DI TIPI MONOMORFI

CANDIDATO
Damiano ZANARDINI

RELATORE
Prof. Giorgio LEVI

ai miei genitori

a Carlo

Indice

1	Introduzione	1
2	Definizioni preliminari	3
2.1	Sintassi e semantica	3
2.2	Tipi	4
2.3	Vincoli e sostituzioni	5
2.4	Ambienti di tipo	6
3	Elementi di teoria dei tipi	9
3.1	Relazione di tipaggio	9
3.2	Tipaggio monomorfo	10
3.2.1	λ -calcolo semplicemente tipato	10
3.2.2	Sistema di Church-Curry	10
3.3	Sistema di tipi à la ML	11
3.3.1	Regole	12
3.3.2	Alcune proprietà delle regole di tipaggio	12
3.4	Sicurezza del tipaggio	13
3.4.1	Tipologia degli errori	13
3.4.2	Semantica a riduzioni	13
3.5	Inferenza di tipo	15
3.6	Algoritmo per l'inferenza monomorfa	16
3.6.1	Costruzione del sistema di equazioni	16
3.6.2	Risoluzione del sistema	17
4	Elementi di interpretazione astratta	19
4.1	Introduzione	19
4.2	Esempi di interpretazione astratta	20
4.2.1	Regola dei segni	20
4.2.2	Analisi degli intervalli	21
4.2.3	Calcolo dimensionale	22

4.3	Principi di interpretazione astratta	23
4.3.1	Connessioni di Galois	24
4.3.2	Astrazione tra semantiche	26
4.4	Terminazione dell'analisi astratta	29
4.4.1	Approssimazione del punto fisso	29
4.4.2	Operatori di widening/narrowing	30
4.5	Interpretazione astratta e sistemi di tipi	31
5	Semantica generale	35
5.1	Introduzione	35
5.2	Espressioni annotate	36
5.3	Regole	37
5.4	Esempi di utilizzo	39
5.5	Insieme delle funzioni associate	46
5.6	Inferenza	47
5.6.1	Connessione di Galois	48
5.6.2	Astrazione esatta	49
5.7	Altre semantiche particolari	57
5.7.1	Ordinamento parziale	57
5.8	Ricorsione	59
5.8.1	Operatori di widening	60
5.9	Semantiche di verifica	64
5.10	Compatibilità	64
5.11	G.c. tra l'inferenza e una semantica di verifica	66
5.11.1	$f \in \Phi_c$	66
5.11.2	$f \in \Phi_{ver} \setminus \Phi_c$	67
5.12	G.c. tra l'inferenza e le verifiche	67
5.12.1	Connessione di Galois	68
5.12.2	Soundness	69
6	Conclusioni	71
A	Implementazione della semantica	73
A.1	Definizione degli insiemi di valori	74
A.2	Algoritmo di unificazione	77
A.3	Semantica generale \mathbf{S}_0	79
A.4	Funzione associata all'inferenza pura $f_{S_{inf}}$	83

Capitolo 1

Introduzione

La teoria dei tipi, in particolare lo studio della correttezza (*type soundness*) dell'inferenza di tipo (*type inference*), ha dominato, seguendo il lavoro di Luis Damas e Robin Milner [17, 22], la ricerca degli ultimi decenni sui linguaggi di programmazione, almeno per quanto riguarda i linguaggi funzionali. Nel corso di queste ricerche sono stati proposti diversi *sistemi di tipi*, c'est-à-dire sistemi di regole o algoritmi che si occupano di controllare in fase di compilazione che l'esecuzione di un programma non provochi errori runtime.

In un importante articolo del 1997 [4] Patrick Cousot usa la teoria dell'interpretazione astratta per delineare relazioni tra diversi sistemi di tipi per linguaggi funzionali. Le relazioni cercate sono quelle di maggiore o minore *precisione*, per cui un sistema di tipi più preciso può assegnare un tipo a programmi (espressioni) per cui un sistema meno preciso non è in grado di farlo, oppure può assegnare ad una certa espressione un tipo più generale.

In questo contesto dire che un type-system è meno preciso di un altro significa individuare un rapporto di *astrazione* del primo rispetto al secondo; il sistema più astratto compie un'analisi di tipo meno raffinata che porta a rifiutare come non tipabili anche espressioni per cui non si presenteranno errori a tempo di esecuzione, quindi è in linea di principio possibile l'assegnazione di tipo. I sistemi di tipi più precisi (in generale sono quelli più difficili da implementare, meno efficienti o non effettivi) usano domini di calcolo più articolati (ad esempio insiemi di tipi o tipi con variabili invece che tipi semplici), considerano un maggior numero di informazioni durante l'analisi e sono dotati di meccanismi di calcolo più complessi (come, ad esempio, l'astrazione e/o la ricorsione polimorfa).

Una relazione di astrazione o approssimazione può essere cercata anche tra il concetto di inferenza di tipo e quello di verifica (*type checking*); il primo deduce informazioni di tipo dalla struttura del programma, mentre il secondo si limita a controllare la consistenza di informazioni già esistenti.

La semantica che viene proposta definisce un sistema di tipi monomorfo che compie un'analisi di tipo avente come casi particolari la verifica e l'inferenza; unificare questi concetti in un unico formalismo aiuta ad analizzare differenze ed analogie tra i due approcci. Anche i casi intermedi tra inferenza e verifica (cioè casi in cui parte delle informazioni è specificata a priori e parte è calcolata in itinere dall'algoritmo) possono essere considerati all'interno di questo contesto.

Capitolo 2

Definizioni preliminari

2.1 Sintassi e semantica

Verrà presa in esame una versione eager di λ -calcolo definita dalle seguenti sintassi e semantica [4, 19]:

Definizione 2.1. \mathbf{E} è l'insieme delle espressioni del λ -calcolo:

$$\begin{aligned} n, m \in \mathbf{N} & : \text{ costanti numeriche} \\ X, Y \in \mathbf{X} & : \text{ identificatori} \\ e \in \mathbf{E} & : \text{ espressioni} \\ e ::= n \mid X \mid e + e \mid \text{if } e \text{ then } e \text{ else } e \mid e e \mid \lambda X.e \mid \mu f.\lambda X.e \end{aligned}$$

Solitamente verranno usate espressioni chiuse, cioè espressioni in cui tutti gli identificatori sono introdotti dalla λ -astrazione o dalla μ -ricorsione.

Definizione 2.2. Il dominio semantico Φ è definito come segue:

$$\begin{aligned} \mathbf{W} & \stackrel{def}{=} \{\omega\} & \text{errore} \\ n \in \mathbf{Z} & & \text{interi} \\ u, f, \phi \in \mathbf{U} & \cong \mathbf{W}_\perp \oplus \mathbf{Z}_\perp \oplus [\mathbf{U} \mapsto \mathbf{U}]_\perp & \text{valori} \\ \rho \in \mathbf{R} & \stackrel{def}{=} \mathbf{X} \mapsto \mathbf{U} & \text{ambienti} \\ \phi \in \Phi & \stackrel{def}{=} \mathbf{R} \mapsto \mathbf{U} & \text{dominio semantico} \end{aligned}$$

dove ω è il valore di errore, \perp denota la non-terminazione, D_\perp è il lift dell'insieme D con \perp (con up-iniezione $\uparrow(\bullet) : D \mapsto D_\perp$ e down-iniezione parziale $\downarrow(\bullet) : D_\perp \mapsto D$), $D_1 \oplus D_2$ è la somma di domini (con iniezioni $\bullet :: D_1 : D_1 \mapsto D_1 \oplus D_2$ e $\bullet :: D_2 : D_2 \mapsto D_1 \oplus D_2$), $\Omega \stackrel{def}{=} \uparrow(\omega) :: \mathbf{W}_\perp$ e $[D_1 \mapsto D_2]$ è il dominio delle funzioni continue e strict da D_1 a D_2 .

Definizione 2.3. La semantica denotazionale $\mathbf{S}[\bullet] : \mathbf{E} \mapsto \mathbf{\Phi}$ è definita come segue (solo alcuni casi sono considerati):

$$\begin{aligned} \mathbf{S}[[n]] &\stackrel{def}{=} \lambda\rho. \uparrow(n) :: \mathbf{Z}_\perp \\ \mathbf{S}[[X]] &\stackrel{def}{=} \lambda\rho. \rho(X) \\ \mathbf{S}[[e_1 + e_2]] &\stackrel{def}{=} \lambda\rho. \text{if } (\mathbf{S}[[e_1]]\rho = \perp \vee \mathbf{S}[[e_2]]\rho = \perp) \text{ then } \perp \\ &\quad \text{else if } (\mathbf{S}[[e_1]]\rho = z_1 :: \mathbf{Z}_\perp \wedge \mathbf{S}[[e_2]]\rho = z_2 :: \mathbf{Z}_\perp) \\ &\quad \text{then } \uparrow(\downarrow(z_1) + \downarrow(z_2)) :: \mathbf{Z}_\perp \text{ else } \Omega \end{aligned}$$

2.2 Tipi

Questa versione del λ -calcolo può essere dotata di un sistema di tipi, come verrà mostrato (cap. 3).

Definizione 2.4. \mathbf{V} è l'insieme delle variabili di tipo; esse verranno indicate con il metanome v e i nomi $'a, 'b, \dots$

Definizione 2.5. Verranno chiamate con lo stesso nome, ground_V , alcune funzioni che, dato un termine con variabili $v_i, v_j \in \mathbf{V}$, ne restituiscono l'insieme delle istanze *ground*.

$$\begin{aligned} \text{ground}_V &: \mathbf{M}_V^H \mapsto \wp(\mathbf{M}^H) \\ \text{ground}_V &: \mathbf{H}_V \mapsto \wp(\mathbf{H}) \\ \text{ground}_V &: \mathbf{H}_V \times \mathbf{M}_V^H \mapsto \wp(\mathbf{H} \times \mathbf{M}^H) \end{aligned}$$

Definizione 2.6. \mathbf{M}^H è l'insieme dei tipi *ground* (cioè i tipi senza variabili, o monotipi):

$$\begin{aligned} \tau \in \mathbf{M}^H &: \text{tipi ground} \\ \tau &::= \text{int} \mid \tau \rightarrow \tau \end{aligned}$$

Definizione 2.7. \mathbf{M}_V^H è l'insieme dei tipi con variabili (o schemi di tipi):

$$\begin{aligned} \tau \in \mathbf{M}_V^H &: \text{tipi con variabili} \\ \tau &::= \text{int} \mid 'a \mid \tau \rightarrow \tau \end{aligned}$$

Definizione 2.8. L'ordinamento parziale su \mathbf{M}_V^H è:

$$\begin{aligned} \leq &\subseteq \mathbf{M}_V^H \times \mathbf{M}_V^H \\ \leq &\stackrel{def}{=} \{(\tau_1, \tau_2) \mid \text{ground}_V(\tau_1) \subseteq \text{ground}_V(\tau_2)\} \end{aligned}$$

L'estremo inferiore del reticolo è \perp_M e soddisfa $\text{ground}_V(\perp_M) = \emptyset$.

2.3 Vincoli e sostituzioni

Nelle semantiche astratte per l'inferenza di tipo che saranno usate nel seguito è importante il concetto di vincolo: vincolare una variabile di tipo ad un certo termine significa acquisire informazione sulla struttura dei termini stessi. Generare un vincolo significa in generale rendere più istanziata, e quindi maggiormente precisa, l'informazione sul tipo di uno o più identificatori ed espressioni (quelli nella cui espressione di tipo occorre la variabile di tipo che è stata vincolata). Infatti attribuire ad un identificatore o ad una espressione un tipo meno generale (cioè minore secondo l'ordinamento su \mathbf{M}_V^H) diminuisce il grado di incertezza sul tipo stesso; ad esempio se il tipo di X è $'a \rightarrow 'b$, generando un vincolo che associa ad $'b$ il tipo $int \rightarrow int$ si perviene al tipo $'a \rightarrow (int \rightarrow int)$ che è più preciso.

Definizione 2.9. \mathbf{C}^H è l'insieme delle sostituzioni che definiscono i vincoli prodotti dall'analisi:

$$\mathbf{C}^H \stackrel{def}{=} [\mathbf{V} \mapsto \mathbf{M}_V^H]$$

(ε è la funzione identità, cioè la sostituzione vuota).

Verrà indicata con $\theta(x)$ (oppure $\text{apply}(\theta, x)$) l'applicazione di una sostituzione θ ad un oggetto x appartenente agli insiemi \mathbf{V} (in quest'ultimo caso si tratta dell'applicazione vera e propria della funzione θ a x), \mathbf{M}_V^H oppure \mathbf{H}_V (vedi def. 2.12); questa operazione consiste nell'applicare la funzione θ a tutte le variabili che compaiono in x e sostituire le variabili con il risultato dell'applicazione.

Definizione 2.10. Una sostituzione è ottenuta, a partire da altre sostituzioni e da uguaglianze tra termini in \mathbf{M}_V^H secondo l'algoritmo di unificazione di Montanari e Martelli, denotato dalla funzione $\text{unify} : \wp(\mathbf{M}_V^H \times \mathbf{M}_V^H) \times \wp(\mathbf{C}^H) \mapsto \mathbf{C}^H$; la composizione di sostituzioni, denotata con il simbolo \circ , segue le usuali regole.

Definizione 2.11. L'ordinamento parziale sugli elementi di \mathbf{C}^H è definito come

$$\begin{aligned} \leq & \subseteq \mathbf{C}^H \times \mathbf{C}^H \\ \leq & \stackrel{def}{=} \{(\theta_1, \theta_2) \mid \forall x. \theta_1(x) \geq \theta_2(x)\} \end{aligned}$$

dove gli ordinamenti sui tipi e sugli ambienti sono definiti rispettivamente nelle definizioni 2.8 e 2.13. Ciò significa che la sostituzione maggiore istanzia maggiormente le variabili contenute in x .

Vale il risultato $\theta_1 \leq \theta_2 \iff \exists \theta. \theta_2 = \theta_1 \circ \theta$.

2.4 Ambienti di tipo

Definizione 2.12. \mathbf{H} è l'insieme dei type environments o ambienti di tipo *ground*, cioè delle funzioni

$$\mathbf{X} \mapsto \mathbf{M}^H \times \mathbf{C}^H$$

$\mathbf{H}_V \supseteq \mathbf{H}$ è l'insieme dei type environments con variabili, cioè delle funzioni

$$\mathbf{X} \mapsto \mathbf{M}_V^H \times \mathbf{C}^H$$

tali che

$$\forall H \in \mathbf{H}_V, \forall X \in \mathbf{X}. H(X) = (\tau, \theta) \Rightarrow \theta(\tau) = \tau$$

(cioè la sostituzione contenuta nell'ambiente non ha effetto sul tipo).

Un'assunzione più forte sarebbe che ogni sostituzione contenuta nell'ambiente non deve modificare l'ambiente stesso:

$$\forall H \in \mathbf{H}_V, \forall X \in \mathbf{X}. H(X) = (\tau, \theta) \Rightarrow \theta(H) = H$$

Ciò è verificato in ogni caso (perchè si può assumere che le sostituzioni contenute negli ambienti siano sempre vuote) tranne nella ricorsione con punto fisso (anche se, come si vedrà nella sez. 5.8, l'analisi compie una sola iterazione, quindi il controesempio a questa assunzione non si presenta).

$\tilde{H} \in \mathbf{H}_V$ è l'ambiente più generale, cioè quello che assegna agli identificatori variabili di tipo con nomi sempre diversi e la sostituzione vuota (**fst** e **snd** restituiscono rispettivamente il primo e il secondo elemento di una coppia):

$$\begin{aligned} \forall X, Y \in \mathbf{X}. \quad & \tilde{H}(X), \tilde{H}(Y) \in (\mathbf{V} \times \mathbf{C}^H) \wedge \mathbf{fst}(\tilde{H}(X)) \neq \mathbf{fst}(\tilde{H}(Y)) \\ & \wedge \mathbf{snd}(\tilde{H}(X)) = \mathbf{snd}(\tilde{H}(Y)) = \varepsilon \end{aligned}$$

Fare l'analisi di tipo di un'espressione in un certo type environment $H \in \mathbf{H}$ significa avere informazione sul tipo degli identificatori che occorrono nell'espressione stessa; non è infatti necessario per questa analisi sapere quali siano precisamente i valori attribuiti agli identificatori (in altre parole: non è importante l'ambiente $\rho \in \mathbf{R}$ in cui l'espressione è valutata, a patto che per ogni identificatore X si abbia che $\rho(X)$ ha tipo compatibile con $H(X)$).

Definizione 2.13. L'ordinamento su \mathbf{H}_V è:

$$\begin{aligned} & \leq \subseteq \mathbf{H}_V \times \mathbf{H}_V \\ & \leq \stackrel{def}{=} \{(H_1, H_2) \mid \mathbf{ground}_V(H_1) \subseteq \mathbf{ground}_V(H_2)\} \end{aligned}$$

L'estremo inferiore dell'insieme è l'ambiente \perp_H per cui $\forall X \in \mathbf{X}. H_\perp(X) = \perp_M$.

Type environments di questa forma verranno usati anche con la semantica di Church-Curry \mathbf{T}^C ; poichè quest'ultima non prevede la presenza delle sostituzioni nell'ambiente (cioè usa ambienti di tipo per cui $H(X) \in \mathbf{M}^H$) verrà assunto che queste vengano semplicemente ignorate nell'analisi. Ciò ci consentirà di usare gli insiemi \mathbf{X} e \mathbf{H}_V in modo omogeneo.

Capitolo 3

Elementi di teoria dei tipi

In questo capitolo si parlerà di teoria dei tipi riferendosi a [21] e al λ -calcolo definito nel capitolo 2.

Lo scopo del tipaggio statico è rilevare e rifiutare in fase di compilazione un certo numero di programmi assurdi, come $1\ 2$ o $(\lambda X.X) + 1$. Questo può essere fatto attraverso l'attribuzione di un *tipo* ad ogni sottoespressione (ad esempio *int* per una espressione intera, *int* \rightarrow *int* per una funzione da interi a intero, ...) e la verifica della coerenza di questi tipi. Per essere effettivo un sistema di tipi deve essere decidibile: non si tratta di eseguire completamente il programma a tempo di compilazione.

3.1 Relazione di tipaggio

La relazione di tipaggio è una relazione ternaria tra ambienti di tipi (def. 2.12), espressioni (def. 2.1) e tipi (def. 2.6, 2.7); si scrive $H \vdash e \rightarrow \tau$ quando l'analisi di tipo eseguita sull'espressione e a partire dall'ambiente di tipi H porta al tipo τ . A seconda dei sistemi di tipi si possono usare diverse definizioni di ambienti di tipi (\mathbf{H} , \mathbf{H}_V , ...) e tipi.

Un modo efficace per visualizzare la relazione di tipaggio e la sua derivazione a partire da un'espressione e un ambiente di tipi è l'uso delle *regole di inferenza*; esse si scrivono come

$$\frac{P_1 \quad P_2 \quad \dots \quad P_n}{P}$$

(dove i P_i sono proposizioni)

e corrispondono alla formula logica $P_1 \wedge P_2 \wedge \dots \wedge P_n \implies P$; chiamiamo *premesse* le proposizioni $P_1 \quad P_2 \quad \dots \quad P_n$ e *conclusione* la proposizione P ; alcune regole, dette *assiomi*, non hanno premesse, quindi la loro conclusione è sempre vera.

3.2 Tipaggio monomorfo

3.2.1 λ -calcolo semplicemente tipato

Un sistema di tipi piuttosto semplice è il *sistema monomorfo*, chiamato λ -calcolo *semplicemente tipato* (gli ambienti di tipo H appartengono all'insieme \mathbf{H}_V , vedi def. 2.12):

$$H \vdash n \Rightarrow \text{int} \quad (3.1)$$

$$H \vdash X \Rightarrow H(X) \quad (3.2)$$

$$\frac{H \vdash e_1 \Rightarrow \text{int} \quad H \vdash e_2 \Rightarrow \text{int}}{H \vdash e_1 + e_2 \Rightarrow \text{int}} \quad (3.3)$$

$$\frac{H \vdash e_1 \Rightarrow \text{int} \quad H \vdash e_2 \Rightarrow \tau \quad H \vdash e_3 \Rightarrow \tau}{H \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow \tau} \quad (3.4)$$

$$\frac{H \vdash e_1 \Rightarrow \tau_1 \rightarrow \tau_2 \quad H \vdash e_2 \Rightarrow \tau_1}{H \vdash e_1 e_2 \Rightarrow \tau_2} \quad (3.5)$$

$$\frac{H[X \leftarrow \tau_1] \vdash e \Rightarrow \tau_2}{H \vdash \lambda X.e \Rightarrow \tau_1 \rightarrow \tau_2} \quad (3.6)$$

$$\frac{H[f \leftarrow \tau] \vdash \lambda X.e \Rightarrow \tau}{H \vdash \mu f.\lambda X.e \Rightarrow \tau} \quad (3.7)$$

Con questo sistema di tipi si possono derivare tipaggi come i seguenti (H è un ambiente qualsiasi):

$$\begin{aligned} H \vdash \lambda X.X &\Rightarrow 'a \rightarrow 'a \\ H \vdash \lambda X.X &\Rightarrow \text{int} \rightarrow \text{int} \end{aligned}$$

mentre certe espressioni non possono essere tipate (cioè non esistono H e τ tali che $H \vdash E \Rightarrow \tau$):

$$\begin{aligned} &1\ 2 \\ &\lambda f.f\ f \end{aligned}$$

3.2.2 Sistema di Church-Curry

Un sistema simile a questo, illustrato in [4], è il sistema monomorfo di Church-Curry (graficamente la relazione di tipaggio è indicata con $H \stackrel{c}{\vdash} e \Rightarrow \tau$); esso presenta regole di forma identica al sistema monomorfo appena descritto, ma usa gli ambienti di tipo ground, cioè l'insieme \mathbf{H} . Ciò comporta l'impossibilità di dedurre tipi con variabili (ad esempio non si può derivare $H \stackrel{c}{\vdash} \lambda X.X \Rightarrow 'a \rightarrow 'a$); i risultati sulle espressioni

chiuso (cioè quelle in cui non vengono usate variabili globali) sono però equivalenti: se ho $H \vdash e \Rightarrow \tau$ per una qualche espressione, ambiente di tipi con variabili e tipo con variabili avrò anche $H' \vdash^c e \Rightarrow \tau'$ per ogni $H' \in \mathbf{H}$ (per le espressioni chiuse, cioè quelle che non utilizzano variabili globali, l'ambiente di partenza non conta) e $\tau' \in \mathbf{M}^H$ istanza ground di τ (prop. 3.1).

3.3 Sistema di tipi à la ML

Una debolezza del tipaggio monomorfo è che un identificatore può avere un solo tipo, anche se è legato ad una funzione intrinsecamente polimorfa (come la funzione identità). Per superare questa difficoltà è necessario introdurre la nozione di schema di tipi, una rappresentazione compatta di tutti i tipi che possono essere attribuiti ad un'espressione polimorfa.

Definizione 3.1. *Uno schema di tipi è un'espressione di tipo con zero, una o più variabili di tipo universalmente quantificate:*

$$\begin{aligned}\sigma &\in \Sigma \\ \sigma &= \forall v_1, \dots, v_n. \tau\end{aligned}$$

dove τ è un elemento di \mathbf{M}_V^H che contiene le variabili v_1, \dots, v_n . Se l'insieme di variabili è vuoto lo schema si scrive τ anziché $\forall. \tau$.

Le variabili quantificate universalmente possono essere liberamente rinominate (operazione di α -conversione); l'insieme delle variabili libere di un tipo, di uno schema o di un ambiente è definito nel modo intuitivo.

Uno schema di tipi può essere visto come l'insieme di tipi ottenuti istanziando (specializzando) le sue variabili quantificate con dei tipi particolari. Così $\forall v. v \rightarrow v$ può essere visto come l'insieme di tipi $\{\tau \rightarrow \tau \mid \tau \text{ è un tipo}\}$. Per formalizzare questa nozione si definisce la relazione $\tau \leq \sigma$ ("il tipo τ è una istanza dello schema di tipi σ ") nel seguente modo:

$$\tau \leq \forall v_1, \dots, v_n. \tau' \iff \exists \tau_1, \dots, \tau_n. \tau = \tau'[v_1 \leftarrow \tau_1, \dots, v_n \leftarrow \tau_n]$$

($\tau[v \leftarrow \tau']$ sostituisce in τ tutte le occorrenze di v con il tipo τ')

Ad esempio $int \rightarrow int$ è un'istanza di $\forall v. v \rightarrow v$, così come $(int \rightarrow int) \rightarrow (int \rightarrow int)$, ma $int \rightarrow (int \rightarrow int)$ non lo è.

3.3.1 Regole

Gli ambienti restituiscono schemi di tipi.

$$\frac{}{H \vdash n \Rightarrow \text{int}} \quad \text{ML} \quad (3.8)$$

$$\frac{\tau \leq H(X)}{H \vdash X \Rightarrow \tau} \quad \text{ML} \quad (3.9)$$

$$\frac{H \vdash e_1 \Rightarrow \text{int} \quad H \vdash e_2 \Rightarrow \text{int}}{H \vdash e_1 + e_2 \Rightarrow \text{int}} \quad \text{ML} \quad (3.10)$$

$$\frac{H \vdash e_1 \Rightarrow \text{int} \quad H \vdash e_2 \Rightarrow \tau \quad H \vdash e_3 \Rightarrow \tau}{H \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow \tau} \quad \text{ML} \quad (3.11)$$

$$\frac{H \vdash e_1 \Rightarrow \tau_1 \rightarrow \tau_2 \quad H \vdash e_2 \Rightarrow \tau_1}{H \vdash e_1 e_2 \Rightarrow \tau_2} \quad \text{ML} \quad (3.12)$$

$$\frac{H[X \leftarrow \tau_1] \vdash e \Rightarrow \tau_2}{H \vdash \lambda X.e \Rightarrow \tau_1 \rightarrow \tau_2} \quad \text{ML} \quad (3.13)$$

$$\frac{H[f \leftarrow \tau] \vdash \lambda X.e \Rightarrow \tau}{H \vdash \mu f.\lambda X.e \Rightarrow \tau} \quad \text{ML} \quad (3.14)$$

In (3.13) τ_1 non è uno schema di tipo (cioè non si ha polimorfismo parametrico, chiamato da Cousot in [4] *astrazione polimorfa*), mentre in (3.14) τ può essere uno schema (si ha, cioè, *ricorsione polimorfa*).

3.3.2 Alcune proprietà delle regole di tipaggio

Le regole di tipaggio come sono state viste finora godono di alcune proprietà:

Proposizione 3.1 (stabilità del tipaggio rispetto alla sostituzione). *Sia $\theta \in \mathbf{C}^H$. Se $H \vdash e \Rightarrow \tau$ allora $\theta(H) \vdash e \Rightarrow \theta(\tau)$.*

Proposizione 3.2 (indifferenza del tipaggio rispetto a ipotesi inutili). *Nell'ipotesi che $H_1(X) = H_2(X)$ per tutti gli X liberi nell'espressione e si ha che $H_1 \vdash e \Rightarrow \tau \iff H_2 \vdash e \Rightarrow \tau$.*

Proposizione 3.3 (stabilità del tipaggio rispetto al rafforzamento delle ipotesi). *Supponiamo $\text{dom}(H_1) = \text{dom}(H_2)$ e $H_2(X) \geq H_1(X)$ per tutti gli identificatori. Allora $H_1 \vdash e \Rightarrow \tau \implies H_2 \vdash e \Rightarrow \tau$.*

3.4 Sicurezza del tipaggio

Come già detto, lo scopo del tipaggio è rifiutare staticamente una classe di espressioni scorrette che darebbero errori a tempo di esecuzione. Un tipaggio è *sicuro* rispetto alla valutazione se ogni espressione che viene tipata dal sistema di tipi può essere valutata “senza problemi”, senza cioè che sorgano errori run-time.

3.4.1 Tipologia degli errori

Prima di tutto è necessario definire i “problemi” che si possono presentare durante la valutazione e che un sistema di tipi sicuro dovrebbe impedire. Si tratta di errori a tempo di esecuzione corrispondenti all’applicazione di un’operazione di base su degli argomenti scorretti (ad esempio $e_1 + e_2$ quando le due sottoespressioni non sono compatibili con il tipo intero) o all’applicazione di una funzione ad un argomento di tipo errato.

Nell’interpretazione *in the large* di un linguaggio si preferisce (per ragioni di efficienza e semplicità del codice generato) non controllare esplicitamente a tempo di esecuzione se questi errori si producono. Quindi l’esecuzione di una di queste operazioni erranee può provocare un arresto inaspettato del programma, oppure la prosecuzione della valutazione con risultati assurdi. Il tipaggio statico serve ad evitare che queste situazioni si producano.

La semantica del λ -calcolo non ha regole di valutazione che si applichino a queste espressioni erranee; questo potrebbe far pensare che sia possibile caratterizzare i programmi ben tipati come quelli per cui la valutazione è possibile e calcola un certo valore. Il problema di questo approccio è che esiste un’altra classe di espressioni che non viene valutata in un certo valore, ma non causa errori a tempo di esecuzione: le espressioni che non terminano. Poichè non si può decidere, in un linguaggio Turing-equivalente, se un certo programma termina con un valore, il sistema di tipi non può garantire che i programmi ben tipati terminino.

Attraverso l’uso di una semantica a riduzioni è possibile caratterizzare le espressioni erranee come forme normali (cioè espressioni non ulteriormente riducibili) che non sono dei valori (in altre parole la riduzione di un’espressione non ha prodotto un risultato corretto).

3.4.2 Semantica a riduzioni

In [21] si mostra una semantica operativa strutturale, con regole per la valutazione delle sottoespressioni e regole per la rilevazione e la propagazione di errori: le regole di valutazione modificate per il nostro linguaggio sono del tipo

$$\frac{e_1 \xrightarrow{v} (\lambda X.e) \quad e_2 \xrightarrow{v} v_2 \quad e[X \leftarrow v_2] \xrightarrow{v} v}{e_1 e_2 \xrightarrow{v} v}$$

mentre le regole di errore sono del tipo

$$\frac{e_1 \xrightarrow{v} \mathbf{err}}{e_1 \ e_2 \xrightarrow{v} \mathbf{err}} \quad \frac{e_1 \xrightarrow{v} v_1 \quad e_2 \xrightarrow{v} \mathbf{err}}{e_1 \ e_2 \xrightarrow{v} \mathbf{err}}$$

Questa semantica non è adeguata all'intento di caratterizzare le espressioni che provocano errori a tempo di esecuzione; ciò è dovuto al fatto che non è possibile dimostrare che nel sistema sono state formalizzate tutte le regole di errore necessarie per assicurare che i controlli a tempo di esecuzione sono inutili.

Per questo motivo verrà introdotta una semantica *small-step* (che cioè descrive tutti i passi intermedi della valutazione di un'espressione) detta *semantica a riduzioni*. Essa si presenta come una relazione $e \rightarrow e'$ dove e' si è l'espressione ottenuta da e con un solo passo di riduzione; i passi di riduzione si combinano in derivazioni che terminano con un valore:

$$e \rightarrow e_1 \rightarrow e_2 \rightarrow \dots \rightarrow v$$

Per definire la relazione di riduzione in un passo viene formalizzata la relazione $\xrightarrow{\varepsilon}$ "si riduce in un passo alla testa del termine"; essa si basa sui seguenti assiomi e regole ($e[X \leftarrow e']$ significa che ogni occorrenza di X in e è sostituita da e'):

$$\begin{aligned} n_1 + n_2 &\xrightarrow{\varepsilon} n \quad \text{se } n_1, n_2 \text{ sono interi e } n = n_1 + n_2 \\ \text{if } 0 \text{ then } e_1 \text{ else } e_2 &\xrightarrow{\varepsilon} e_1 \\ \text{if } n \text{ then } e_1 \text{ else } e_2 &\xrightarrow{\varepsilon} e_2 \quad \text{se } n \neq 0 \\ (\lambda X.e) v &\xrightarrow{\varepsilon} e[X \leftarrow v] \\ \mu f.(\lambda X.e) v &\xrightarrow{\varepsilon} e[X \leftarrow v][f \leftarrow (\mu f.(\lambda X.e))] \end{aligned}$$

Naturalmente la riduzione non viene sempre effettuata in testa all'espressione: ad esempio se si ha $(\lambda X.e)e'$ sarà necessario ridurre prima e' , e questo non è possibile con le regole appena descritte. A questo proposito è necessario introdurre la nozione di contesto di valutazione attraverso la regola seguente:

$$\frac{e \xrightarrow{\varepsilon} e'}{\Gamma(e) \rightarrow \Gamma(e')}$$

Definizione 3.2. *I contesti di valutazione sono definiti come*

$\Gamma ::=$	\square	<i>valutazione in testa</i>
	$\Gamma + e$	<i>valutazione a sinistra di una operazione aritmetica</i>
	$v + \Gamma$	<i>valutazione a destra di una operazione aritmetica</i>
	$\text{if } \Gamma \text{ then } e_1 \text{ else } e_2$	<i>valutazione della guardia del condizionale</i>
	Γe	<i>valutazione a sinistra di una applicazione</i>
	$v \Gamma$	<i>valutazione a destra di una applicazione</i>

I contesti sono in sostanza espressioni con un “buco”. L’applicazione di un contesto ad un’espressione $(\Gamma(e))$ significa “riempire” questo buco con l’espressione e .

A questo punto è possibile definire la relazione $\xrightarrow{*}$, chiusura riflessiva e transitiva di \rightarrow , cioè riduzione in zero, uno o più passi.

Definizione 3.3. *L’espressione e è una forma normale se $e \not\rightarrow$, cioè se non esiste alcuna espressione e' tale che $e \rightarrow e'$. Le espressioni erronee sono le forme normali che non sono dei valori (come ad esempio 1 2).*

Nella valutazione di un’espressione e si possono verificare tre tipi di situazioni:

- (i) e si riduce in un numero finito di passi al valore v : $e \rightarrow e_1 \rightarrow \dots \rightarrow v$;
- (ii) e si riduce all’infinito senza provocare errori: $e \rightarrow e_1 \rightarrow e_2 \rightarrow \dots$;
- (iii) e si riduce ad una forma normale che non è un valore: $e \rightarrow e_1 \rightarrow \dots \rightarrow e_n \not\rightarrow$.

La proposizione seguente formalizza la correttezza del ragionamento fin qui adottato, caratterizzando le espressioni mal tipate come quelle la cui valutazione porta ad una forma normale che non è un valore.

Proposizione 3.4 (Sicurezza del tipaggio). *Sia $H \vdash e \Rightarrow \tau$ per un qualche H , e $e \xrightarrow{*} e'$ con e' forma normale. Allora e' è un valore.*

3.5 Inferenza di tipo

Per un linguaggio staticamente tipato un *tipatore* è un algoritmo che prende in entrata un programma e determina se questo è tipabile o no. In generale esso attribuirà anche un tipo a questo programma.

Se diversi tipi sono attribuibili ad un programma, l’algoritmo cercherà di restituire come risultato un *tipo principale*, cioè un tipo più preciso di tutti gli altri tipi possibili. Ad esempio in ML l’espressione $\lambda X.X$ può avere tipo $\tau \rightarrow \tau$ per un qualsiasi tipo τ , ma il tipo $'a \rightarrow 'a$ è il principale perchè tutti gli altri tipi possibili se ne deducono per sostituzione.

La quantità di lavoro svolta dal tipatore è inversamente proporzionale alla quantità di dichiarazioni di tipo già presenti nel programma sorgente, e quindi dalla quantità di informazioni sui tipi fornite dal programmatore.

Verifica pura

Nell’espressione sorgente tutte le sottoespressioni, così come tutti gli identificatori, sono annotati con il loro tipo (lo stile in cui sono specificate le annotazioni è quello usato per le specifiche, vedi def. 5.1):

$$\begin{aligned}
 &(\lambda f : int \rightarrow int. (\lambda X : int. \\
 &\quad (X : int + ((f : int \rightarrow int)(2 : int)) : int) : int) : int \rightarrow int \\
 &\quad : (int \rightarrow int) \rightarrow int \rightarrow int
 \end{aligned}$$

In questo caso il tipaggio è molto semplice, perchè l'informazione sui tipi è già presente e l'algoritmo deve solo verificare che essa sia corretta. Nessun linguaggio realistico utilizza questo approccio perchè obbliga il programmatore a fornire esplicitamente molte informazioni.

Dichiarazione di tipo per gli identificatori e propagazione di tipo

Il programmatore dichiara il tipo dei parametri delle funzioni e delle variabili locali. Il tipatore inferisce il tipo delle sottoespressioni propagando i tipi delle foglie (sottoespressioni di base) verso la radice (l'espressione globale). L'esempio di sopra diventa

$$\lambda f : int \rightarrow int. (\lambda X : int. X + (f \ 2))$$

Il tipatore inferirà il tipo $(int \rightarrow int) \rightarrow int \rightarrow int$ per questa espressione.

Dichiarazione di tipo dei parametri di funzione e propagazione di tipo

La differenza con l'approccio precedente è che le variabili locali (ad esempio quelle introdotte con costrutti come il `let`) non sono annotate con il tipo. Nel nostro caso non c'è differenza.

Inferenza completa dei tipi

L'espressione non contiene alcuna informazione sul tipo e tutto il lavoro è lasciato all'algoritmo; esso determina il tipo dei termini secondo l'uso che ne è fatto all'interno del programma:

$$\lambda f. (\lambda X. X + (f \ 2))$$

Nell'uso comune di ML questa è la soluzione utilizzata; è comunque possibile specificare i tipi durante la dichiarazione delle funzioni.

3.6 Algoritmo per l'inferenza monomorfa

Si coinsidererà il caso dell'inferenza pura, essendo gli altri casi tecnicamente meno interessanti. Il procedimento di inferenza si articola in due passaggi:

- (i) a partire dal programma sorgente si costruisce un sistema di equazioni tra tipi che caratterizza tutti i tipaggi possibili per il programma;
- (ii) si risolve il sistema di equazioni; se non c'è soluzione il programma è mal tipato, altrimenti si determina una soluzione principale al sistema, che sarà il tipo principale del programma.

3.6.1 Costruzione del sistema di equazioni

Si parte da un programma (un'espressione chiusa, senza variabili non legate) e_0 nella quale tutti gli identificatori legati dall'astrazione hanno nomi diversi (altrimenti è

sempre possibile rinominarli). Ad ogni identificatore X in e_0 si associa una variabile di tipo v_X , e ad ogni sottoespressione e si associa la variabile v_e . Il sistema di equazioni $C(e_0)$ è costruito percorrendo l'espressione principale e aggiungendo equazioni per ogni sottoespressione e nel modo seguente:

- e è una costante numerica: $C(e) = \{v_e = int\}$;
- e è un identificatore X : $C(e) = \{v_e = v_X\}$;
- $e \equiv e_1 + e_2$: $C(e) = \{v_e = int; v_{e_1} = int; v_{e_2} = int\} \cup C(e_1) \cup C(e_2)$;
- $e \equiv \text{if } e_1 \text{ then } e_2 \text{ else } e_3$:
 $C(e) = \{v_{e_1} = int; v_{e_2} = v_e; v_{e_3} = v_e\} \cup C(e_1) \cup C(e_2) \cup C(e_3)$;
- $e \equiv e_1 \ e_2$: $C(e) = \{v_{e_1} = v_{e_2} \rightarrow v_e\} \cup C(e_1) \cup C(e_2)$;
- $e \equiv \lambda X. e_1$: $C(e) = \{v_e = v_X \rightarrow v_{e_1}\} \cup C(e_1)$;
- $e \equiv \mu f. e_1$: $C(e) = \{v_f = v_{e_1}\} \cup C(e_1)$.

3.6.2 Risoluzione del sistema

Una volta che il sistema di equazioni è stato costruito l'algoritmo cerca una sua soluzione: essa è una sostituzione θ tale che per tutte le equazioni $\tau_1 = \tau_2$ in $C(e_0)$ si ha $\theta(\tau_1) = \theta(\tau_2)$, cioè un *unificatore* del sistema $C(e_0)$:

Proposizione 3.5 (Correttezza delle soluzioni rispetto al tipaggio). *Se θ è una soluzione di $C(e)$ allora $H \vdash e \Rightarrow \theta(v_e)$ dove H è il sistema di tipi per cui $\forall X$ libero in e . $H(X) = \theta(v_X)$.*

Proposizione 3.6 (Completezza delle soluzioni rispetto al tipaggio). *Sia e un'espressione. Se esistono H e τ tali che $H \vdash e \Rightarrow \tau$ allora il sistema di equazioni $C(e)$ ammette una soluzione.*

La risoluzione del sistema di equazioni è implementata attraverso l'algoritmo di unificazione per il calcolo dell'**mgu**, cioè di una sostituzione che abbia le proprietà descritte sopra.

Per il sistema di tipi di ML esiste un altro algoritmo, l'algoritmo W di Damas-Milner-Tofte.

Capitolo 4

Elementi di interpretazione astratta

4.1 Introduzione

Gli algoritmi di *analisi statica* si propongono di determinare proprietà dinamiche di programmi in modo statico (senza basarsi sull'esecuzione del programma stesso), cioè inferire dalla sintassi o dalla semantica proprietà che questi possiedono durante la loro esecuzione (come ad esempio la correttezza dei tipi o l'assenza di errori a tempo di esecuzione). A causa dei noti risultati di indecidibilità che riguardano i formalismi di calcolo (e i linguaggi di programmazione in particolare) risulta impossibile determinare in modo esatto e automatico certe proprietà che un programma esibisce dinamicamente senza che sia necessaria l'esecuzione del programma stesso (ad esempio la proprietà “il programma p termina per ogni possibile input”).

I manipolatori di programmi (cioè i programmatori che scrivono, verificano, correggono programmi, oppure gli algoritmi che li compilano, interpretano, trasformano o eseguono) prendono in considerazione la sintassi ma soprattutto la semantica, cioè i diversi comportamenti che un programma può avere quando riceve tutti i possibili input. Ad esempio un programma può terminare dando in output uno o più valori (siano essi aderenti o meno alle specifiche), oppure entrare in una computazione divergente (che cioè richiede tempo infinito) o ancora provocare un errore a tempo di esecuzione (ad esempio quando il programma cerca di eseguire una divisione per 0). Per un certo tipo di ragionamento o analisi su dei programmi non tutti gli aspetti della loro esecuzione devono essere necessariamente presi in considerazione: ogni analisi è anzi facilitata dall'avere una semantica adeguata che sia abbastanza complessa da descrivere i comportamenti che si vogliono osservare ma astragga dalle informazioni irrilevanti.

Ad esempio se l'analisi concerne la terminazione di un programma funzionale

informazioni come il modo in cui la memoria viene gestita dalla macchina astratta durante l'esecuzione sono totalmente trascurabili. Seguendo questo ragionamento si può pensare ad una gerarchia di semantiche che, partendo dalla più precisa o dettagliata fino a quella più “ad alto livello”, individuino diversi livelli di raffinamento dell'analisi, ognuno dei quali può essere opportunamente usato per la risoluzione di un dato problema. Avremo quindi semantiche molto dettagliate, come quella che tiene traccia del comportamento delle strutture a tempo di esecuzione che fanno parte della macchina astratta, e semantiche che invece si disinteressano di molti dettagli, come quella che determina la terminazione dei programmi.

L'interpretazione astratta (*abstract interpretation* o *interprétation abstraite*) si propone proprio di individuare i rapporti esistenti tra differenti livelli di analisi di proprietà di programmi, formalizzando la nozione di *astrazione* per cui una semantica è più astratta di un'altra se è meno dettagliata e precisa (questa minore precisione sarà solitamente compensata dalla semplicità o dalla possibilità di implementare algoritmi di analisi più efficienti). Essa è una teoria di approssimazione semantica usata nella costruzione di algoritmi di program analysis basata sulla semantica (a volte chiamata *analisi data flow*), nel confronto tra semantiche formali, nel design di metodi di prova.

4.2 Esempi di interpretazione astratta

In prima approssimazione l'interpretazione astratta può essere vista come una *semantica non standard* [14], cioè una semantica in cui il dominio dei valori è sostituito da un dominio di *descrizioni di valori*, e in cui agli operatori sono attribuiti significati non standard corrispondenti.

4.2.1 Regola dei segni

Ad esempio invece di utilizzare gli interi come valori concreti un'interpretazione astratta potrebbe usare i valori astratti $+1$ e -1 per descrivere rispettivamente interi positivi e negativi, e reinterpretare gli operatori delle operazioni elementari sugli interi secondo le regole dei segni conosciute fin dalla Grecia antica. Questo permetterebbe di stabilire proprietà di programmi come “all'entrata del corpo di un ciclo la variabile x ha valore positivo”.

L'espressione $(x \times x) + (y \times y)$ ha il valore 25 se x vale 3, y vale 4 e $+$ e \times sono i soliti operatori sugli interi, ma applicando la regola dei segni:

$$\begin{aligned} +1 + +1 &= +1, & +1 \times +1 &= +1, & -1 \times +1 &= -1 \\ -1 + -1 &= +1, & +1 \times -1 &= -1, & -1 \times -1 &= +1 \end{aligned}$$

(dove $+1$ e -1 rappresentano rispettivamente un qualsiasi intero positivo e un qualsiasi intero negativo) si vede che il risultato dell'espressione non è mai negativo.

Questo semplice calcolo astratto non riesce però a dimostrare che un'espressione come $x^2 + 2 \times x \times y + y^2$ è sempre non negativa.

Questo esempio mostra, nonostante la sua semplicità, che l'interpretazione astratta può sbagliare: la semplificazione ottenuta tramite l'astrazione può provocare una perdita di informazione che impedisce di pervenire a certi risultati. Per "catturare" questi tipi di errori si può introdurre un nuovo valore astratto, \top , che rappresenta il fatto che niente è conosciuto del risultato:

$$\begin{array}{llll} +1 + -1 = \top & +1 + \top = \top & \top \times +1 = \top & -1 \times \top = \top \\ -1 + +1 = \top & -1 + \top = \top & \top \times -1 = \top & \top \times \top = \top \\ \top + +1 = \top & \top + \top = \top & +1 \times \top = \top & \\ \top + -1 = \top & & & \end{array}$$

In questo modo, considerando x positivo e y negativo il secondo esempio diventa

$$\begin{aligned} & (+1)^2 + (+1) \times (+1) \times (-1) + (-1)^2 \\ = & (+1) + (+1) \times (-1) + (+1) \\ = & (+1) + (-1) + (+1) \\ = & \top + (+1) = \top \end{aligned}$$

Per approssimare un dato valore concreto possono essere utilizzati diversi valori astratti: ad esempio 5 può essere approssimato da +1 o da \top ; una relazione di *ordinamento parziale* \preceq deve essere introdotta tra i valori astratti: $+1 \preceq \top$ perchè è più preciso (individua cioè una proprietà più restrittiva), mentre +1 e -1 non sono confrontabili. Ci sono poi valori concreti, come 0, approssimabili da diversi valori astratti minimali: può essere utile approssimare 0 con +1 o -1 a seconda della particolare espressione da analizzare. Per evitare di dover eseguire di volta in volta delle scelte è comunque possibile aggiungere un nuovo valore astratto, 0, che astrae il valore concreto 0 e introduce nuove regole:

$$0 + +1 = +1 \quad 0 + \top = \top \quad 0 \times +1 = 0 \quad \dots$$

4.2.2 Analisi degli intervalli

La regola dei segni può essere generalizzata, come in [8], all'analisi di intervalli, cioè a proprietà del tipo $l \leq x \leq u$, dove $l, u \in \mathbf{Z} \cup \{-\infty, +\infty\}$, \mathbf{Z} è l'insieme degli interi e $l \leq u$. In questo articolo le maggiori innovazioni erano l'idea di una prova di correttezza (*soundness*) ottenuta mettendo in relazione le semantiche astratte con una semantica concreta e l'uso di domini infiniti, che conduce ad analisi molto potenti.

In questo caso i valori concreti sono insiemi di interi, mentre i valori astratti sono intervalli; dato un valore concreto $X = \{n_1, n_2, \dots, n_k\}$ il valore astratto

corrispondente è $[\min(X), \max(X)]$. Sui valori astratti vengono inoltre definiti un estremo inferiore \perp (corrispondente astratto dell'insieme vuoto), un'operazione di unione \cup_a , astrazione dell'unione \cup tra insiemi di interi, e un ordinamento parziale \leq_a tra valori astratti:

$$\begin{aligned} [l_1, u_1] \cup_a [l_2, u_2] &\stackrel{def}{=} [\min(l_1, l_2), \max(u_1, u_2)] \\ \leq_a &\stackrel{def}{=} \{([l_1, u_1], [l_2, u_2]) \mid l_2 \leq l_1 \wedge u_1 \leq u_2\} \end{aligned}$$

A differenza del esempio dei segni in questo caso il dominio astratto è infinito ed esistono catene infinite strettamente crescenti, come ad esempio $[1, 1], [1, 2], [1, 3], [1, 4], \dots$; ciò comporta la necessità di gestire problemi di terminazione (si veda la sezione 4.4).

Queste definizioni vengono applicate, in [8], all'analisi data flow di programmi imperativi, dove il programma viene eseguito secondo la semantica astratta che usa i valori definiti come intervalli: se, per problemi di indecidibilità, non è possibile calcolare staticamente il valore che una variabile assume in un certo punto di un programma, si può comunque determinare l'intervallo (cioè il valore astratto) in cui la variabile assume valore. Questa perdita di informazione è necessaria se l'analisi è solo statica. Operando in modo simile si possono ottenere risultati del tipo

```
function F(X : integer) : integer;
begin
  if X > 100 then begin
    F := X - 10
  { 101 <= X <= maxint && 91 <= F <= maxint-10 }
  end else begin
    F := F(F(X + 11))
  { minint <= X <= 100 && F = 91 }
  end;
end;
```

dove i commenti sono stati inseriti automaticamente. In generale è possibile determinare proprietà invarianti di programmi in modo migliore che con l'analisi data flow classica.

4.2.3 Calcolo dimensionale

Il calcolo dimensionale, esempio familiare tratto dalla fisica elementare, è per certi versi simile alla verifica dei tipi. Esso utilizza i valori astratti *lunghezza*, *superficie*, *tempo*, *massa*, \dots , *nodimensione*; gli operatori astratti \overline{op} sono ottenuti dai

corrispondenti concreti op con queste ovvie regole:

$$\begin{array}{llll}
 \text{lunghezza} & \overline{\mp} & \text{lunghezza} & = & \text{lunghezza} \\
 \text{lunghezza} & \overline{\times} & \text{lunghezza} & = & \text{superficie} \\
 \text{lunghezza} & \overline{\int} & \text{lunghezza} & = & \text{nodimensione} \\
 \text{lunghezza} & \overline{\int} & \text{tempo} & = & \text{velocità} \\
 \text{velocità} & \overline{\int} & \text{tempo} & = & \text{accelerazione} \\
 \text{massa} & \overline{\times} & \text{accelerazione} & = & \text{forza} \\
 x & \overline{\int} & y^{\overline{n+1}} & = & (x\overline{\int}y^{\overline{n}})\overline{\int}y
 \end{array}$$

I valori astratti *lunghezza*, *massa*, *tempo* sono le astrazioni rispettivamente dei valori concreti *mm*, *Kg*, *μs*; la perdita di informazione si verifica perchè dato un valore astratto non è possibile stabilire il valore concreto da cui questo è stato ricavato (ad esempio a *massa* possono corrispondere *g*, *Kg*, *lbs*,...); ciò comunque è irrilevante nel calcolo dimensionale, quindi l'approssimazione non ha effetti negativi.

4.3 Principi di interpretazione astratta

Il framework dell'interpretazione astratta (*absint*) è stato introdotto da Patrick e Radhia Cousot in [8, 9, 10, 11, 3] per giustificare la specifica di analizzatori di programmi rispetto ad una certa semantica formale. L'idea guida è che questo processo di specifica è l'approssimazione di proprietà individuate dalla semantica concreta con proprietà individuate da una semantica approssimata o astratta; quest'ultima individua esplicitamente una certa struttura particolare implicitamente presente nella più ricca struttura concreta associata all'esecuzione dei programmi (cioè astrae dai dettagli trascurabili per la particolare analisi che si vuole implementare, evidenziando invece le informazioni rilevanti). Quindi l'*absint* ha un aspetto costruttivo, non soltanto una giustificazione a posteriori, perchè la semantica astratta può essere derivata sistematicamente da quella concreta, in un modo che potrebbe tendere ad essere il più possibile automatico.

Per quanto detto lo studio dell'*absint* coinvolge lo studio della semantica dei programmi, dei metodi di prova, della specifica, costruzione e sperimentazione di analizzatori nell'ottica che questi diversi aspetti siano legati tra loro attraverso processi di astrazione/approssimazione. Questo ovviamente necessita di una conoscenza profonda delle strutture matematiche che descrivono l'esecuzione di programmi e dello studio delle relazioni tra di esse.

Il framework presentato in [3] parte da una semantica operativa che descrive l'esecuzione *small-step* di programmi usando un sistema a transizioni (*transition system*). Successivamente viene costruita una semantica *collecting* che è minimale, corretta e relativamente completa per le proprietà di interesse. Il design di analizzatori di programmi è poi basato su semantiche che sono approssimazioni di questa

semantica collecting; qui il compromesso da trovare è tra la difficoltà concettuale, la precisione e il costo dell'analisi.

4.3.1 Connessioni di Galois

Il concetto di *astrazione* o *approssimazione* di una semantica rispetto ad un'altra è formalizzato dalla nozione di connessione di Galois:

Definizione 4.1. *Dati due insiemi parzialmente ordinati $P^b(\preceq^b)$ e $P^\sharp(\preceq^\sharp)$ una connessione di Galois (Galois connection, o G.c.) è una coppia di funzioni α e γ (dette rispettivamente funzione di astrazione e funzione di concretizzazione) tale che:*

$$\begin{aligned} \alpha &: P^b \mapsto P^\sharp \\ \gamma &: P^\sharp \mapsto P^b \\ \forall p^b \in P^b, p^\sharp \in P^\sharp. \quad \alpha(p^b) \preceq^\sharp p^\sharp &\Leftrightarrow p^b \preceq^b \gamma(p^\sharp) \end{aligned}$$

in questo caso si scrive

$$P^b(\preceq^b) \xleftrightarrow[\alpha]{\gamma} P^\sharp(\preceq^\sharp)$$

Tra le proprietà delle G.c. si ha:

Proposizione 4.1. $\alpha \circ \gamma$ è estensiva: $\forall p^b \in P^b. p^b \preceq^b \gamma(\alpha(p^b))$.

Proposizione 4.2. $\gamma \circ \alpha$ è riduttiva: $\forall p^\sharp \in P^\sharp. \alpha(\gamma(p^\sharp)) \preceq^\sharp p^\sharp$.

Proposizione 4.3. α e γ sono monotone.

La congiunzione di queste proprietà è condizione sufficiente perchè α e γ determinino una G.c., quindi si può scrivere la definizione 4.1 in modo equivalente come

Definizione 4.2. α e γ costituiscono una connessione di Galois se e solo se:

(i) sono monotone;

(ii) $\forall p^b \in P^b. p^b \preceq^b \gamma(\alpha(p^b)) \wedge \forall p^\sharp \in P^\sharp. \alpha(\gamma(p^\sharp)) \preceq^\sharp p^\sharp$.

Proposizione 4.4. In una G.c. una funzione determina unicamente l'altra:

$$\begin{aligned} P^b(\preceq^b) \xleftrightarrow[\alpha_1]{\gamma_1} P^\sharp(\preceq^\sharp) \wedge P^b(\preceq^b) \xleftrightarrow[\alpha_2]{\gamma_2} P^\sharp(\preceq^\sharp) \\ \implies \\ (\alpha_1 = \alpha_2) \Leftrightarrow (\gamma_1 = \gamma_2) \end{aligned}$$

Dimostrazione. Assumiamo $\alpha_1 = \alpha_2$; $\forall p^\sharp \in P^\sharp. \alpha_2(\gamma_2(p^\sharp)) \preceq^\sharp p^\sharp$ (prop. 4.2), quindi $\alpha_1(\gamma_2(p^\sharp)) \preceq^\sharp p^\sharp$ per ipotesi, da cui $\gamma_2(p^\sharp) \preceq^b \gamma_1(p^\sharp)$ (def. 4.1). Allo stesso, $\alpha_1(\gamma_1(p^\sharp)) \preceq^\sharp p^\sharp$ (prop. 4.2), quindi $\alpha_2(\gamma_1(p^\sharp)) \preceq^\sharp p^\sharp$ per ipotesi e $\gamma_1(p^\sharp) \preceq^b \gamma_2(p^\sharp)$ (def. 4.1). Per antisimmetria concludiamo che $\gamma_1(p^\sharp) =^b \gamma_2(p^\sharp)$. Il viceversa è analogo. \square

Il concetto di connessione di Galois formalizza una situazione in cui passare dal dominio concreto a quello astratto determina una perdita di informazione (infatti $\alpha \circ \gamma$ è estensiva, cioè astraendo un valore concreto e concretizzando successivamente il valore astratto trovato si può avere un valore concreto meno preciso di quello di partenza).

Ad esempio nell'analisi degli intervalli (sez. 4.2.2) la connessione di Galois è definita nel modo seguente:

$$\begin{aligned} \alpha & : \wp(\mathbf{Z}) \mapsto (\mathbf{Z} \cup \{\pm\infty\}) \times (\mathbf{Z} \cup \{\pm\infty\}) \\ \alpha(X) & \stackrel{def}{=} \langle \min(X), \max(X) \rangle \end{aligned} \quad (4.1)$$

$$\begin{aligned} \gamma & : (\mathbf{Z} \cup \{\pm\infty\}) \times (\mathbf{Z} \cup \{\pm\infty\}) \mapsto \wp(\mathbf{Z}) \\ \gamma(\langle l, u \rangle) & \stackrel{def}{=} \{n \in \mathbf{Z} \mid l \leq n \leq u\} \end{aligned} \quad (4.2)$$

Si vede facilmente che le proposizioni 4.1, 4.2 e 4.3 sono verificate.

In questo caso la perdita di informazione si verifica perchè diversi valori concreti sono astratti dallo stesso valore, quindi l'approssimazione dovuta all'astrazione non permette di distinguerli:

$$\gamma(\alpha(\{1, 3, 5\})) = \gamma(\langle 1, 5 \rangle) = \{1, 2, 3, 4, 5\} = \gamma(\alpha(\{1, 2, 3, 5\})) \quad (4.3)$$

Invece nell'esempio del calcolo dimensionale (sez. 4.2.3) la funzione di astrazione è definita come segue:

$$\begin{aligned} \alpha(m) & = \text{lunghezza} & \alpha(g) & = \text{massa} \\ \alpha(s) & = \text{tempo} & \alpha(N) & = \text{forza} \\ \alpha(Km) & = \text{lunghezza} & \alpha(E_1 \text{ op } E_2) & = \alpha(E_1) \overline{\text{op}} \alpha(E_2) \end{aligned}$$

per cui

$$\begin{aligned} \alpha(Kg \times (m/s^2)) & = \alpha(Kg) \overline{\times} \alpha(m/s^2) \\ & = \text{massa} \overline{\times} (\alpha(m) \overline{/} \alpha(s^2)) \\ & = \text{massa} \overline{\times} (\text{lunghezza} \overline{/} (\alpha(s))^2) \\ & = \text{massa} \overline{\times} (\text{lunghezza} \overline{/} \text{tempo}^2) \end{aligned}$$

Proposizione 4.5. *La perdita di informazione dovuta al processo di astrazione avviene in un solo passo:*

- (i) $\forall p^b \in P^b. \alpha(\gamma(\alpha(p^b))) =^{\#} \alpha(p^b).$
- (ii) $\forall p^{\#} \in P^{\#}. \gamma(\alpha(\gamma(p^{\#}))) =^b \gamma(p^{\#}).$

Dimostrazione. (i) $\alpha(p^b) \succeq^\# \alpha(\gamma(\alpha(p^b)))$ per la prop. 4.2; inoltre $\gamma(\alpha(p^b)) \succeq^b p^b$ (da $\alpha(p^b) =^\# \alpha(p^b)$ e def. 4.1), da cui $\alpha(\gamma(\alpha(p^b))) \succeq^\# \alpha(p^b)$ (prop. 4.3).

(ii) il ragionamento è analogo. \square

Definizione 4.3. Una inserzione di Galois (Galois insertion o G.i.) è una connessione di Galois per cui $\gamma \circ \alpha$ è la funzione identità:

$$\forall p^\# \in P^\#. \alpha(\gamma(p^\#)) =^\# p^\#$$

Definizione 4.4. Una connessione di Galois è un isomorfismo se $\gamma \circ \alpha$ e $\alpha \circ \gamma$ sono la funzione identità; in questo caso non c'è perdita di informazione nel passaggio tra dominio concreto e dominio astratto (cioè possiamo riottenere da un valore astratto il valore concreto che l'aveva generato).

4.3.2 Astrazione tra semantiche

Quanto detto finora su connessioni e astrazioni di Galois si riferisce in generale a insiemi parzialmente ordinati; più in particolare si possono considerare insiemi che siano domini di calcolo di una qualche semantica ([9] per l'analisi data flow, [4] per i linguaggi funzionali). Sia $\mathbf{\Pi}$ l'insieme di programmi definito da una certa sintassi e $\mathbf{\Sigma}$ l'insieme degli *stati* in cui un programma può essere eseguito (ad esempio nei linguaggi funzionali lo stato è costituito dall'*ambiente*, nei linguaggi imperativi è presente anche la *memoria*): la semantica di un programma sarà in generale una funzione da stati in valori (oppure un insieme di coppie (*stato, valore*), secondo la definizione insiemistica di funzione). Il *dominio* è l'insieme cui appartengono le semantiche dei vari programmi.

Definizione 4.5. Per il λ -calcolo (def. 2.1, 2.2, 2.3):

(i) l'insieme dei programmi è l'insieme \mathbf{E} delle possibili espressioni;

(ii) \mathbf{U} è l'insieme dei valori;

(iii) gli stati $\sigma \in \Sigma$ sono gli ambienti, cioè le funzioni $\rho \in \mathbf{R}$ dagli identificatori \mathbf{X} a \mathbf{U} ;

(iv) la semantica ϕ_e di un'espressione $e \in \mathbf{E}$ è un elemento dell'insieme $\mathbf{\Phi}$ delle funzioni $\mathbf{R} \mapsto \mathbf{U}$.

(v) La semantica denotazionale del λ -calcolo è una funzione $\mathbf{S}[\bullet] : \mathbf{E} \mapsto \mathbf{\Phi}$, cioè una funzione che data una certa espressione specifica tutti i possibili valori cui la sua valutazione può portare per ogni possibile stato (ambiente) in cui essa è valutata. Il dominio di \mathbf{S} è $\mathbf{\Phi}$.

Questa semantica attribuisce ad un'espressione e all'ambiente in cui essa è valutata un valore; può darsi però che l'interesse dell'analisi sia meno preciso, cioè si limiti a specificare un insieme di semantiche al cui interno si trova la semantica

dell'espressione da valutare. Ciò impone un cambio di dominio, utilizzando invece di Φ l'insieme delle parti di Φ .

Definizione 4.6. *L'insieme delle proprietà semantiche è $\mathbf{P} = \wp(\Phi)$. Un'espressione e ha una proprietà $P \in \mathbf{P}$ se $\mathbf{S}[e] \in P$.*

Definizione 4.7. *L'ordinamento parziale su \mathbf{P} è definito come*

$$\begin{aligned} \leq_P &\subseteq \mathbf{P} \times \mathbf{P} \\ \leq_P &\stackrel{def}{=} \{(P_1, P_2) \mid P_1 \subseteq P_2\} \end{aligned}$$

Se $P_1 \leq_P P_2$ allora P_1 è più precisa, cioè l'insieme delle espressioni che la verificano è incluso nell'insieme che soddisfa P_2 . È facile vedere che $\mathbf{P}(\leq_P)$ è un reticolo completo, con estremi $\perp \stackrel{def}{=} \emptyset$, che non è verificata da alcuna espressione, e $\top \stackrel{def}{=} \mathbf{P}$.

Definizione 4.8. *La semantica collecting di un'espressione del λ -calcolo è la proprietà semantica più precisa che sia verificata dall'espressione stessa:*

$$\begin{aligned} \mathbf{C}[\bullet] &: \mathbf{E} \mapsto \mathbf{P} \\ \mathbf{C}[e] &\stackrel{def}{=} \{\mathbf{S}[e]\} \end{aligned}$$

Da queste definizioni possiamo formalizzare la nozione di astrazione tra semantiche del λ -calcolo:

Definizione 4.9. *Date due semantiche $\mathbf{S}^b \in \mathbf{E} \mapsto \mathbf{U}^b(\preceq^b)$ e $\mathbf{S}^\sharp \in \mathbf{E} \mapsto \mathbf{U}^\sharp(\preceq^\sharp)$, si dice che \mathbf{S}^\sharp astrae \mathbf{S}^b se esistono due funzioni α e γ che determinano una connessione di Galois e verificano*

$$\forall e \in \mathbf{E}. \mathbf{S}^b[e] \preceq^b \gamma(\mathbf{S}^\sharp[e])$$

cioè data una certa espressione i valori ottenuti dalle due semantiche sono uno un'approssimazione dell'altro (def. 4.1) [4].

Questa definizione formalizza il concetto secondo cui una semantica è meno “precisa” dell'altra, ma non lega nessuna delle due semantiche alla semantica denotazionale del λ -calcolo. È necessaria quindi una nozione che metta in relazione una certa semantica con i valori ottenuti dalla valutazione della espressione, cioè con la semantica collecting:

Definizione 4.10. *Una semantica \mathbf{S}^\sharp è corretta (sound) se è un'astrazione della semantica collecting, cioè se*

$$\forall e \in \mathbf{E}. \mathbf{C}[e] \subseteq \gamma(\mathbf{S}^\sharp[e])$$

Una semantica sound identifica cioè, data un'espressione, una proprietà meno precisa di quella identificata dalla semantica collecting (o, meglio, concretizzando il valore ottenuto con la semantica astratta si ottiene una proprietà meno precisa): l'insieme delle espressioni che verificano questa proprietà include l'insieme che verifica la proprietà calcolata con la semantica collecting.

In generale due semantiche (parleremo sempre di semantiche sound) che siano legate da un rapporto di approssimazione rappresentano l'una il raffinamento dell'altra; può darsi però che la precisione delle due semantiche sia uguale, e che passare dall'una all'altra non comporti una perdita di informazione [4]:

Definizione 4.11. *Sia $\mathbf{S}^b[\bullet] \in \mathbf{E} \mapsto \mathbf{U}^b$ e $\mathbf{U}^b(\preceq^b) \xleftrightarrow[\alpha]{\gamma} \mathbf{U}^\sharp(\preceq^\sharp)$. La semantica astratta $\mathbf{S}^\sharp[\bullet] \in \mathbf{E} \mapsto \mathbf{U}^\sharp$ tale che $\forall e \in \mathbf{E}. \alpha(\mathbf{S}^b[e]) \preceq^\sharp \mathbf{S}^\sharp[e]$ è esatta se e solo se $\forall e \in \mathbf{E}. \mathbf{S}^b[e] =^b \gamma(\mathbf{S}^\sharp[e])$.*

Si noti che questa definizione non presuppone che tra i domini di calcolo esista una relazione di isomorfismo; è possibile che ci sia perdita di informazione su alcuni elementi del dominio concreto (cioè che $\gamma(\alpha(x)) \neq x$), ma ciò non deve succedere sugli elementi del dominio ottenuti tramite la valutazione semantica di una certa espressione del λ -calcolo.

Valgono queste ovvie proposizioni:

Proposizione 4.6. *Se \mathbf{S}^\sharp è un'astrazione di \mathbf{S}^b e \mathbf{S}^\sharp è un'astrazione di \mathbf{S}^\sharp , allora \mathbf{S}^\sharp è un'astrazione di \mathbf{S}^b (vale la composizionalità per l'astrazione).*

Proposizione 4.7. *Se \mathbf{S}^b è una semantica sound e \mathbf{S}^\sharp è una sua astrazione, allora \mathbf{S}^\sharp è una semantica sound.*

Se, come si verifica di solito, le semantiche sono definite composizionalmente è sufficiente, per verificare l'astrazione tra semantiche (def. 4.9), la seguente condizione:

$$\begin{aligned} \forall i = 1, \dots, n. \mathbf{S}^b[e_i] \preceq^b \gamma(\mathbf{S}^\sharp[E_i]) \\ \implies \\ \Psi_e^b(\mathbf{S}^b[e_1], \dots, \mathbf{S}^b[e_n]) \preceq^b \Psi_e^\sharp(\mathbf{S}^\sharp[e_1], \dots, \mathbf{S}^\sharp[e_n]) \end{aligned} \quad (4.4)$$

dove Ψ_e^b e Ψ_e^\sharp sono operatori che calcolano la semantica dell'espressione e date le semantiche delle sottoespressioni.

4.4 Terminazione dell'analisi astratta

Nell'esempio degli intervalli (sez. 4.2.2) si è visto che il dominio astratto può essere anche infinito e avere catene crescenti/decrescenti infinite; ciò pone, per la semantica astratta, il problema della terminazione: nell'analisi data flow trattata in [8] è teoricamente possibile che i valori astratti trovati ad ogni iterazione siano crescenti e sempre diversi, costituiscano cioè una catena crescente infinita. In questo caso l'analisi non è destinata a terminare; poichè si tratta di un'analisi statica questa eventualità è assolutamente da evitare, ed è necessario che il numero di iterazioni sia finito.

Per risolvere questo problema esistono due soluzioni: (i) Usare solo domini astratti finiti o per cui non esistano catene crescenti/decrescenti infinite (si dice in questo caso che il dominio soddisfa la *ascending/descending chain condition*); (ii) Usare operatori di *allargamento/restringimento*.

4.4.1 Approssimazione del punto fisso

Come mostrato in [8, 9, 15] e nelle sezioni precedenti, l'interpretazione astratta di un programma può essere formalizzata come la computazione effettiva di una approssimazione verso l'alto A della semantica collecting del programma.

La semantica collecting può essere specificata come il minimo punto fisso $\text{lfp}_\perp(F)$ di un operatore continuo $F \in \Phi \mapsto \Phi$, dove Φ è il dominio concreto, un ordinamento parziale completo (cpo) con \perp come estremo inferiore. Per il teorema del punto fisso di Kleene si ha che $\text{lfp}_\perp(F) = \bigsqcup_{n \in \mathbb{N}} F^n(\perp)$, cioè l'estremo superiore delle iterate $F^n(\perp)$ definite da $F^0(x) = x$ e $F^{n+1}(x) = F(F^n(x))$.

L'approssimazione A deve essere corretta nel senso che $\text{lfp}_\perp(F) \preceq A$.

Si è già visto come approssimare il dominio concreto L con un dominio astratto \bar{L} e le funzioni di astrazione e concretizzazione; questo sistema può essere esteso anche allo spazio delle funzioni $L \mapsto L$, portando allo spazio astratto $\bar{L} \mapsto \bar{L}$ nel modo seguente:

$$\begin{aligned} \bar{\alpha} : (L \mapsto L) &\mapsto (\bar{L} \mapsto \bar{L}) & \bar{\gamma} : (\bar{L} \mapsto \bar{L}) &\mapsto (L \mapsto L) \\ \bar{\alpha}(\varphi) &\stackrel{def}{=} \gamma \circ \varphi \circ \alpha & \bar{\gamma}(\bar{\varphi}) &\stackrel{def}{=} \alpha \circ \bar{\varphi} \circ \gamma \end{aligned}$$

Un importante teorema (appendice di [15]) stabilisce che se L è un insieme parzialmente ordinato e $\bar{\perp} = \alpha(\perp)$ allora $\text{lfp}_\perp(F) \preceq \gamma(\text{lfp}_{\bar{\perp}}(\bar{\alpha}(F)))$; altrimenti detto, l'operatore di punto fisso preserva la correttezza dell'approssimazione.

Calcolare il punto fisso di questo operatore astratto significa trattare una catena ascendente (come assicurato dalla monotonicità dell'operatore stesso) con elementi nel dominio astratto; se questo dominio non soddisfa l'*ascending chain condition* è possibile che l'analisi astratta non termini, cioè che il punto fisso non sia mai

raggiunto in tempo finito. Per ovviare a questo è possibile utilizzare domini astratti abbastanza semplici che siano finiti o comunque non contengano catene crescenti infinite; come i Cousot argomentano in [15], questa non sembra essere la soluzione migliore.

4.4.2 Operatori di widening/narrowing

Un'altra soluzione (introdotta già in [8, 9]) per indurre la terminazione dell'interpretazione astratta consiste nell'usare un operatore di *allargamento* (*widening*):

Definizione 4.12. Un operatore $\nabla : L \times L \mapsto L$ è detto operatore di widening se

$$(i) \forall x, y \in L. x \preceq x \nabla y$$

$$(ii) \forall x, y \in L. y \preceq x \nabla y$$

(iii) per ogni catena crescente $x^0 \preceq x^1 \preceq x^2 \dots$, la catena crescente definita da $y^0 = x^0, \dots, y^{i+1} = y^i \nabla x^{i+1}$ non è strettamente crescente.

Da questa definizione segue che la sequenza di iterazione con un operatore G e widening definita da

$$\begin{aligned} \hat{X}^0 &= \perp_L \\ \hat{X}^{i+1} &= \hat{X}^i && \text{se } F(\hat{X}^i) \preceq \hat{X}^i \\ &= \hat{X}^i \nabla F(\hat{X}^i) && \text{altrimenti} \end{aligned}$$

è da un certo punto in poi stazionaria e il suo limite \hat{A} è un'approssimazione corretta di $\text{lfp}_\perp(G)$.

Sostanzialmente se la semantica astratta può non terminare perchè il dominio astratto è infinito con catene crescenti finite è possibile, invece di calcolare il minimo punto fisso dell'operatore \overline{F} , usare la sequenza di iterazione definita sopra con \overline{F} e un operatore di widening $\nabla : \overline{\Phi} \times \overline{\Phi} \mapsto \overline{\Phi}$. Ciò assicura che il numero di iterazioni compiuto dalla semantica astratta sia finito.

Ad esempio nel caso degli intervalli di interi [8] l'operatore di widening è definito come

$$\begin{aligned} \perp \nabla X &= X \\ X \nabla \perp &= X \\ [l_1, u_1] \nabla [l_2, u_2] &= [if \ l_2 < l_1 \ \text{then} \ -\infty \ \text{else} \ l_1, \\ &\quad if \ u_2 > u_1 \ \text{then} \ +\infty \ \text{else} \ u_1] \end{aligned} \quad (4.5)$$

ed assicura che la sequenza crescente di intervalli calcolati non sia infinita.

L'approssimazione ottenuta con il widening può poi essere migliorata con un operatore di narrowing:

Definizione 4.13. Un operatore $\Delta : L \times L \mapsto L$ è detto operatore di narrowing se

(i) $\forall x, y \in L. x \preceq y \Rightarrow (x \preceq (x\Delta y) \preceq y$

(iii) per ogni catena decrescente $x^0 \succeq x^1 \succeq x^2 \dots$, la catena decrescente definita da $y^0 = x^0, \dots, y^{i+1} = y^i \Delta x^{i+1}$ non è strettamente decrescente.

Partendo \hat{A} con la sequenza (che per le proprietà del narrowing è decrescente e da un certo punto in poi stazionaria)

$$\begin{aligned}\check{X}^0 &= \hat{A} \\ \check{X}^{i+1} &= \check{X}^i \Delta F(\check{X}^i)\end{aligned}$$

si arriva ad un valore che è un'approssimazione corretta di $\text{lfp}_\perp(F)$ ed è più precisa di \hat{A} .

In [15] si mostra che il metodo del widening/narrowing sembra essere migliore di quello della semplificazione del dominio astratto: esistono domini infiniti e operatori di widening/narrowing per cui

(i) per ogni programma esiste un reticolo finito che può essere utilizzato per ottenere risultati equivalenti a quelli ottenuti con gli operatori di widening/narrowing;

(ii) non esiste un reticolo che sia adeguato per tutti i programmi;

(iii) per ogni programma sono necessari infiniti valori astratti;

(iv) per un particolare programma non è possibile inferire l'insieme di valori astratti di cui si avrà bisogno semplicemente ispezionando il testo del programma.

Ciò fa concludere che non è possibile con domini finiti o senza catene infinite ottenere i risultati del metodo di widening/narrowing; un uso combinato delle due tecniche può essere comunque vantaggioso.

4.5 Interpretazione astratta e sistemi di tipi

In [4] Cousot usa il framework dell'interpretazione astratta per mettere in relazione i sistemi di tipi dei linguaggi funzionali alla semantica del λ -calcolo. L'idea fondamentale è associare ad un certo sistema di tipi una semantica che viene mostrata essere una astrazione della semantica collecting; mettendo in relazione i vari sistemi di tipi tra loro si giunge poi ad una gerarchia tra type systems, per cui i sistemi più complessi sono astrazioni più prossime alla semantica collecting, mentre quelli meno raffinati possono essere astrazioni dei primi. Sostanzialmente i tipi, visti come insiemi di valori, sono descrizioni più o meno accurate dei valori che le espressioni possono assumere (se, ad esempio, una espressione può assumere a seconda degli ambienti valori interi tra 5 e 7, il tipo intero, cioè l'insieme dei numeri interi, è l'approssimazione più vicina che i normali sistemi di tipi offrono per descrivere questo insieme).

Ad esempio al sistema di tipi monomorfo di Church-Curry (sez. 3.2.2) è associata la semantica che segue (sono illustrati solo alcuni casi, si vedano le definizioni 2.12 e 2.6):

$$\begin{aligned}
\mathbf{T} &\stackrel{def}{=} \wp(\mathbf{H} \times \mathbf{M}^H) \quad (\text{program types}) \\
\mathbf{T}^C[\bullet] &: \mathbf{E} \mapsto \mathbf{T} \\
\mathbf{T}^C[X] &\stackrel{def}{=} \{(H, H(X)) \mid H \in \mathbf{H}\} \\
\mathbf{T}^C[e_1 e_2] &\stackrel{def}{=} \{(H, \tau_2) \mid (H, \tau_1 \rightarrow \tau_2) \in \mathbf{T}^C[e_1] \wedge (H, \tau_1) \in \mathbf{T}^C[e_2]\}
\end{aligned}$$

secondo la relazione

$$H \vdash^c e \Rightarrow \tau \iff (H, \tau) \in \mathbf{T}^C[e] \quad (4.6)$$

e la condizione di astrazione rispetto alla semantica collecting è ottenuta esibendo una funzione di concretizzazione γ^C che dato un tipaggio ottenuto su un'espressione con questa semantica calcola una proprietà che astrae la proprietà identificata da $\mathbf{C}[\bullet]$:

$$\begin{aligned}
\gamma_1^C &: \mathbf{M}^H \mapsto \wp(\mathbf{U}) \\
\gamma_1^C(int) &\stackrel{def}{=} \{\uparrow(z) :: \mathbf{Z}_\perp \mid z \in \mathbf{Z}\} \cup \{\perp\} \\
\gamma_1^C(\tau_1 \rightarrow \tau_2) &\stackrel{def}{=} \{\uparrow(\varphi) :: [\mathbf{U} \mapsto \mathbf{U}]_\perp \mid \varphi \in [\mathbf{U} \mapsto \mathbf{U}] \wedge \\
&\quad \forall u \in \gamma_1^C(\tau_1). \varphi(u) \in \gamma_1^C(\tau_2)\} \cup \{\perp\} \\
\gamma_2^C &: \mathbf{H} \mapsto \mathbf{R} \\
\gamma_2^C(H) &\stackrel{def}{=} \{\rho \in \mathbf{R} \mid \forall X \in \mathbf{X}. \rho(X) \in \gamma_1^C(H(X))\} \\
\gamma_3^C &: (\mathbf{H} \times \mathbf{M}^H) \mapsto \mathbf{P} \\
\gamma_3^C((H, \tau)) &\stackrel{def}{=} \{\phi \in \Phi \mid \forall \rho \in \gamma_2^C(H). \phi(\rho) \in \gamma_1^C(\tau)\} \\
\gamma^C &: \mathbf{T} \mapsto \mathbf{P} \\
\gamma^C(T) &\stackrel{def}{=} \bigcap_{\theta \in T} \gamma_3^C(\theta) \quad \gamma^C(\emptyset) \stackrel{def}{=} \mathbf{Phi}
\end{aligned}$$

Un altro sistema di tipi che viene mostrato essere un'astrazione della semantica collecting è il sistema di Hindley, definito dalla semantica \mathbf{T}^H (\tilde{H} , è l'ambiente più generale, si veda def. 2.12):

$$\begin{aligned}
\mathbf{T}^H[\bullet] &: \mathbf{E} \mapsto (\mathbf{H}_V \times \mathbf{M}_V^H) \\
\mathbf{T}^H[X] &\stackrel{def}{=} (\tilde{H}, \tilde{H}(X)) \\
\mathbf{T}^H[e_1 e_2] &\stackrel{def}{=} \text{if } (\mathbf{T}^H[e_2] = (H_2, \tau_2) \\
&\quad \wedge \mathbf{gci}(\{\mathbf{T}^H[e_1], (H_2, \tau_2 \rightarrow 'a)\}) = (H, \tau_2 \rightarrow \tau)) \\
&\quad \text{then } (H, \tau) \text{ else } \emptyset
\end{aligned}$$

Questa semantica è un'astrazione esatta e computer-implementabile di \mathbf{T}^C tramite l'inserzione di Galois

$$\wp(\mathbf{H} \times \mathbf{M}^H) \underset{\text{lcg}_V}{\overset{\text{ground}_V}{\rightleftarrows}} (\mathbf{H}_V \times \mathbf{M}_V^H)$$

con $\forall e. \mathbf{T}^H[e] = \text{lcg}_V(\mathbf{T}^C[e]) \wedge \mathbf{T}^C[e] = \text{ground}_V(\mathbf{T}^H[e])$ ($\text{lcg}_V(x)$ è la *generalizzazione più specifica*, cioè il più piccolo elemento di cui tutti gli elementi di x sono istanze).

Capitolo 5

Semantica generale

5.1 Introduzione

Come già visto nel capitolo 3 ci sono diversi modi per trattare il tipaggio di un'espressione: da una parte lasciare all'algoritmo di tipaggio il compito di inferire il tipo senza che il programmatore fornisca alcuna informazione; dall'altra affidare al programmatore la scrittura delle informazioni di tipo, in modo che l'algoritmo debba solo verificare se queste informazioni sono corrette.

La semantica che verrà illustrata si propone di comprendere questi due approcci e tutti i casi intermedi, cioè quelli in cui solo per alcune delle sottoespressioni sono specificati i tipi, mentre per le altre si cerca di inferirli algebricamente.

Ad una certa sottoespressione può cioè essere attribuita una *specifica* di tipo, cioè un'indicazione su quale tipo il programmatore "si aspetta" per questa sottoespressione: la specifica può essere

(i) più restrittiva del tipo che sarebbe stato inferito: in questo caso l'informazione aggiuntiva viene usata invece di quella inferita nel resto della valutazione (ad esempio se si specifica per la funzione identità il tipo $int \rightarrow int$ non si potrà poi applicarla a un termine di tipo non intero);

(ii) più generale del tipo che sarebbe inferito: in questo caso l'annotazione viene ignorata ma viene conservata l'informazione relativa alle variabili di tipo contenute nella specifica (ad esempio se per il termine $(X + 1)$ si specifica il tipo 'a non si potrà specificare 'a anche per un termine di tipo funzionale);

(iii) non confrontabile con il tipo che sarebbe stato inferito: l'algoritmo calcola un tipo che sia più specifico di entrambi, cioè l'istanza comune più generale, mantenendo anche le informazioni relative ai vincoli sulle variabili usate nella specifica (ad esempio se il tipo inferito sarebbe $int \rightarrow 'a$ e quello specificato è $'b \rightarrow int$ l'algoritmo ricava $int \rightarrow int$ nel caso in cui per 'a e 'b non ci siano già vincoli tra loro inconciliabili, come ad esempio $'a = int \rightarrow int$).

5.2 Espressioni annotate

Si sceglie di rappresentare le specifiche di tipo come etichette assegnate ad ogni sottoespressione; ciò porta alla seguente definizione:

Definizione 5.1. *L'insieme \mathbf{E}_τ delle espressioni annotate con specifiche di tipo è ottenuto da \mathbf{E} nel seguente modo:*

$$\begin{aligned}
n \in \mathbf{E} &\rightsquigarrow n : \tau \in \mathbf{E}_\tau \\
X \in \mathbf{E} &\rightsquigarrow X : \tau \in \mathbf{E}_\tau \\
e_1 + e_2 \in \mathbf{E} &\rightsquigarrow (e'_1 : \tau_1 + e'_2 : \tau_2) : \tau \in \mathbf{E}_\tau \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \in \mathbf{E} &\rightsquigarrow (\text{if } e'_1 : \tau_1 \text{ then } e'_2 : \tau_2 \text{ else } e'_3 : \tau_3) : \tau \in \mathbf{E}_\tau \\
e_1 e_2 \in \mathbf{E} &\rightsquigarrow ((e'_1 : \tau_1)(e'_2 : \tau_2)) : \tau \in \mathbf{E}_\tau \\
\lambda X.e \in \mathbf{E} &\rightsquigarrow (\lambda X : \tau_X.e' : \tau_e) : \tau \in \mathbf{E}_\tau \\
\mu f.e \in \mathbf{E} &\rightsquigarrow (\mu f : \tau_f.e' : \tau_e) : \tau \in \mathbf{E}_\tau
\end{aligned}$$

dove i τ con i vari pedici sono elementi di \mathbf{M}_V^H , e gli apici indicano che le sottoespressioni sono già state annotate al loro interno.

Ad esempio un'espressione annotata come $X : 'a \rightarrow int$ significa che il tipo desiderato per l'identificatore è una funzione da elementi qualunque a interi; se nell'ambiente in cui l'analisi viene eseguita il tipo di X è int il “desiderio” non è soddisfatto e l'analisi viene interrotta; se invece il tipo di X è più generale (ad esempio $'a \rightarrow 'b$) l'analisi considererà successivamente solo il tipo desiderato e le sue istanze.

Naturalmente non si vuole che le specifiche di tipo debbano necessariamente essere assegnate ad ogni sottoespressione; questo modo di procedere consente di aggiungere anche specifiche banali che in sostanza significano l'assenza di specifiche (in particolare nel caso dell'inferenza tutte le specifiche aggiunte sono banali, vedi def. 5.7): le espressioni annotate sono della forma $e' : v$, dove e' è un'espressione già annotata al suo interno e v è una variabile fresca, cioè non ancora usata (se infatti la variabile fosse già stata usata la specifica non sarebbe del tutto banale). I seguenti sono alcuni esempi di specifiche:

$$\begin{aligned}
X \in \mathbf{E} &\rightsquigarrow X : 'a \in \mathbf{E}_\tau \\
e_1 + e_2 \in \mathbf{E} &\rightsquigarrow (e'_1 : 'b + e'_2 : 'b) : 'a \in \mathbf{E}_\tau \\
\lambda X.e \in \mathbf{E} &\rightsquigarrow (\lambda X : int . e' : 'b) : 'a \in \mathbf{E}_\tau
\end{aligned}$$

Nel secondo caso le specifiche non sono banali perchè forzano e_1 ed e_2 ad avere tipi tra loro compatibili; nel terzo caso la specifica non banale è applicata solo alla variabile legata, mentre le altre sottoespressioni non sono forzate ad alcun tipo particolare. In

entrambi questi casi i nomi delle variabili usate nelle specifiche delle sottoespressioni (e'_1, e'_2, e') possono essere gli stessi usati a livello globale (cioè 'a oppure 'b); questo significa avere ulteriori vincoli (ad esempio se in e' ad una costante numerica è assegnata la specifica 'a si avrà un conflitto con la variabile 'a specificata per la λ -espressione: la variabile non può essere contemporaneamente vincolata ad un valore intero e funzionale).

Usare \mathbf{E}_τ consente di unificare i vari casi di specifica di tipo di un'espressione, dall'assenza di specifiche caratteristica dell'inferenza fino a specifiche più o meno restrittive per la verifica. Si vede che \mathbf{E} (cioè l'insieme su cui lavora \mathbf{T}^C , che quindi è sufficiente per l'inferenza) è isomorfo al sottoinsieme di \mathbf{E}_τ che ha specifiche banali per ogni sottoespressione (a meno della ridenominazione delle variabili con altri nomi nuovi). In generale \mathbf{E} è isomorfo ad ogni sottoinsieme di \mathbf{E}_τ ottenuto annotando le espressioni con una certa "politica".

5.3 Regole

A causa di questa "generalità" di \mathbf{E}_τ si può pensare ad una semantica altrettanto generale che si proponga di unificare in un unico schema il caso dell'inferenza e i vari livelli di restrittività della verifica.

Definizione 5.2.

$$\mathbf{S}_0[\bullet] \in \mathbf{E}_\tau \mapsto \wp(\mathbf{H}_V \times (\mathbf{M}_V^H \times \mathbf{C}^H))$$

interi

$$\begin{aligned} \mathbf{S}_0[n : \tau] &= \text{if } (\text{unify}(\{\tau = \text{int}\}) = \theta) \\ &\quad \text{then } \{(H, (\text{int}, \theta)) \mid H \in \mathbf{H}_V\} \\ &\quad \text{else } \emptyset \end{aligned} \tag{5.1}$$

identificatori

$$\mathbf{S}_0[X : \tau] = \{(H, (\theta(\tau), \theta)) \mid \theta = \text{unify}(\{\tau = \text{fst}(H(X)), \text{snd}(H(X))\})\} \tag{5.2}$$

applicazione di funzione

$$\begin{aligned}
\mathbf{S}_0\left[\left((e_1 : \tau_1)(e_2 : \tau_2)\right) : \tau\right] &= \{(H, (\theta(\tau), \theta)) \mid \\
&\quad ((H, (\tau^1, \theta^1)) \in \mathbf{S}_0[e_1 : \tau_1]) \wedge \\
&\quad (H, (\tau^2, \theta^2)) \in \mathbf{S}_0[e_2 : \tau_2]) \wedge \\
&\quad \theta = \text{unify}(\{\tau^1 = \tau^2 \rightarrow \tau, \theta^1, \theta^2\})\} \quad (5.3)
\end{aligned}$$

È possibile esprimere gli stessi concetti in modo equivalente con le regole di inferenza, secondo la relazione

$$H \vdash e : \tau \Rightarrow (\tau', \theta') \iff (H, (\tau', \theta')) \in \mathbf{S}_0[e : \tau]$$

interi

$$\frac{\theta = \text{unify}(\{\tau = \text{int}\})}{H \vdash n : \tau \Rightarrow (\text{apply}(\theta, \tau), \theta)} \quad (5.4)$$

identificatori

$$\frac{H(X) = (\tau_1, \theta_1) \quad \theta = \text{unify}(\{\tau_1 = \tau, \theta_1\})}{H \vdash X : \tau \Rightarrow (\text{apply}(\theta, \tau), \theta)} \quad (5.5)$$

operazioni aritmetiche

$$\frac{H \vdash e_1 : \tau_1 \Rightarrow (\tau^1, \theta^1) \quad H \vdash e_2 : \tau_2 \Rightarrow (\tau^2, \theta^2) \quad \theta = \text{unify}(\{\tau^1 = \text{int}, \tau^2 = \text{int}, \tau = \text{int}, \theta^1, \theta^2\})}{H \vdash (e_1 : \tau_1 + e_2 : \tau_2) : \tau \Rightarrow (\text{apply}(\theta, \tau), \theta)} \quad (5.6)$$

condizionale

$$\frac{H \vdash e_1 : \tau_1 \Rightarrow (\tau^1, \theta^1) \quad H \vdash e_2 : \tau_2 \Rightarrow (\tau^2, \theta^2) \quad H \vdash e_3 : \tau_3 \Rightarrow (\tau^3, \theta^3) \quad \theta = \text{unify}(\{\tau^1 = \text{int}, \tau^2 = \tau^3 = \tau, \theta^1, \theta^2, \theta^3\})}{H \vdash (\text{if } e_1 : \tau_1 \text{ then } e_2 : \tau_2 \text{ else } e_3 : \tau_3) : \tau \Rightarrow (\text{apply}(\theta, \tau), \theta)} \quad (5.7)$$

applicazione di funzione

$$\frac{H \vdash e_1 : \tau_1 \Rightarrow (\tau^1, \theta^1) \quad H \vdash e_2 : \tau_2 \Rightarrow (\tau^2, \theta^2) \quad \theta = \text{unify}(\{\tau^1 = \tau^2 \rightarrow \tau, \theta^1, \theta^2\})}{H \vdash ((e_1 : \tau_1)(e_2 : \tau_2)) : \tau \Rightarrow (\text{apply}(\theta, \tau), \theta)} \quad (5.8)$$

astrazione

$$\frac{H[X \leftarrow (\tau_X, \varepsilon)] \vdash e : \tau_e \Rightarrow (\tau^e, \theta^e) \quad \theta = \text{unify}(\{\tau_X \rightarrow \tau^e = \tau, \theta^e\})}{H \vdash (\lambda X : \tau_X. e : \tau_e) : \tau \Rightarrow (\text{apply}(\theta, \tau), \theta)} \quad (5.9)$$

ricorsione con fix*caso base:*

$$H \vdash (\mu f : \tau_f.e : \tau_e) : \tau \Rightarrow_{T_P}^0 (\tau_f, \varepsilon) \quad (5.10)$$

passo induttivo:

$$\frac{\begin{array}{l} H \vdash (\mu f : \tau_f.e : \tau_e) : \tau \Rightarrow_{T_P}^{n-1} (\tau', \theta') \\ H[f \leftarrow (\tau', \theta')] \vdash e : \tau_e \Rightarrow (\tau'', \theta'') \\ \theta = \text{unify}(\{\tau'' = \tau = \tau_f, \theta''\}) \end{array}}{H \vdash (\mu f : \tau_f.e : \tau_e) : \tau \Rightarrow_{T_P}^n (\text{apply}(\theta, \tau), \theta)} \quad (5.11)$$

punto fisso:

$$\frac{\begin{array}{l} H \vdash (\mu f : \tau_f.e : \tau_e) : \tau \Rightarrow_{T_P}^{n-1} (\tau', \theta') \\ H \vdash (\mu f : \tau_f.e : \tau_e) : \tau \Rightarrow_{T_P}^n (\tau'', \theta'') \\ \tau' = \tau'' \end{array}}{H \vdash (\mu f : \tau_f.e : \tau_e) : \tau \Rightarrow (\tau'', \theta'')} \quad (5.12)$$

ricorsione senza fix

$$\frac{H[f \leftarrow (\tau_f, \varepsilon)] \vdash e : \tau_e \Rightarrow (\tau', \theta') \quad \theta = \text{unify}(\{\tau_f = \tau' = \tau, \theta'\})}{H \vdash (\mu f : \tau_f.e : \tau_e) : \tau \Rightarrow (\text{apply}(\theta, \tau), \theta)} \quad (5.13)$$

*Nel caso della ricorsione e è una λ -espressione***Proposizione 5.1.** $H \vdash e \Rightarrow (\tau, \theta) \implies \theta(\tau) = \tau$ *Dimostrazione.* Tutte le regole applicano la sostituzione al tipo risultato τ' , quindi $\theta(\theta(\tau')) = \theta(\tau')$ per l'idempotenza della sostituzione. \square

Nel caso della ricorsione sono state date due versioni: è immediato vedere che la versione senza punto fisso equivale esattamente al primo passo della versione con punto fisso. Infatti in entrambi i casi si valuta $e : \tau_e$ attribuendo ad f il tipo (τ_f, ε) ; il risultato viene unificato con τ_f e τ .

Si vedrà in seguito che l'approccio senza iterazione raggiunge il punto fisso, quindi anche l'approccio iterativo ottiene questo risultato e lo fa in un solo passo.

5.4 Esempi di utilizzo

In seguito verrà mostrato l'output restituito dall'implementazione della semantica generale (appendice A).

L'analisi di tipo si ottiene valutando l'espressione `s_0 e_tau htilde` dove `e_tau` è un'espressione già annotata (elemento di \mathbf{E}_τ o $\mathbf{E_tau}$), oppure, nel caso dell'inferenza, `s_inf e htilde` dove `e` appartiene a \mathbf{E} e le annotazioni sono aggiunte dalla

semantica secondo le regole di \mathbf{S}_{inf} . \mathbf{htilde} è l'ambiente più generale \tilde{H} ; tutti gli esempi sono espressioni chiuse, quindi possono essere valutati in qualunque ambiente.

Le stringhe $\mathbf{x}n$, dove n è un intero, denotano gli identificatori; le stringhe $\mathbf{a}n$ denotano le variabili di tipo.

In generale un'inferenza come

$$H \vdash (\lambda X_{24} : 'a_0. X_{24} : 'a_1) : 'a_2 \Rightarrow ('a_1 \rightarrow 'a_1, \{ 'a_2 = 'a_1 \rightarrow 'a_1, 'a_0 = 'a_1 \})$$

diventa

$$|- (\backslash x_{24}:a_0.x_{24}:a_1):a_2 ==> ((a_1 \rightarrow a_1) , a_2 ==> (a_1 \rightarrow a_1) | a_0 ==> a_1 | end).$$

L'ambiente non viene rappresentato.

Nelle funzioni ricorsive, ove non espressamente indicato, verrà utilizzato l'approccio iterativo (regole (5.10),(5.11),(5.12)).

Esempio 5.4.1 (Funzione identità, inferenza pura). *Si mostra l'inferenza sulla più tipica funzione polimorfa.*

$$\begin{aligned} e &\equiv \lambda X_{24}. X_{24} \\ f_{S_{inf}}(e) &\equiv (\lambda X_{24} : 'a_0. X_{24} : 'a_1) : 'a_2 \end{aligned}$$

L'espressione valutata è

`s_inf (Lambda 24 (Id 24)) htilde`

L'output della valutazione è il seguente:

`|- x24:a1 ==> (a1 , a0=>a1 | end)`

`|- (\backslash x24:a0.x24:a1):a2 ==> ((a1->a1) , a2=>(a1->a1) | a0=>a1 | end)`

Il tipo inferito è quindi $(a_1 \rightarrow a_1)$.

Esempio 5.4.2 (Funzione identità, vincoli non banali). *In questo esempio si vede che specificando vincoli non banali per certi identificatori o sottoespressioni il tipo di una funzione polimorfa può essere maggiormente istanziato.*

$$\begin{aligned} e &\equiv \lambda X_{24}. X_{24} \\ f(e) &\equiv (\lambda X_{24} : int. X_{24} : 'a_2) : 'a_3 \rightarrow int \end{aligned}$$

L'espressione valutata è

`s_0 (Alambda (24, (Numero))`

`(Aid (24, Variabile 2))`

`(Freccia (Variabile 3) Numero)) htilde`

```
|- x24:a2 ==> ( int , a2=>int | end )

|- (\x24:int.x24:a2):(a3->int) ==>
  ( (int->int) , a3=>int | a2=>int | end )
```

Il tipo calcolato è $int \rightarrow int$ perchè il tipo del primo termine viene posto a int dalla specifica sulla variabile vincolata, il secondo termine è anch'esso posto a int dalla specifica sull'espressione globale. Il tipo calcolato sull'identificatore è intero perchè l'ambiente in cui è eseguita l'analisi è $\tilde{H}[X_{24} \leftarrow int]$.

Esempio 5.4.3 (Espressione con tipo non inferibile). Come è intuitivo la seguente espressione non è tipabile perchè l'applicazione coinvolge tipi incompatibili. Il tipaggio è impossibile indipendentemente dai vincoli.

$$\begin{aligned}
 e &\equiv (\lambda X_{24}.X_{24} + 1)(\lambda X_{25}.X_{25}) \\
 f_{s_{inf}}(e) &\equiv (((\lambda X_{24} : 'a_0.(X_{24} : 'a_1 + 1 : 'a_2) : 'a_3) : 'a_4) \\
 &\quad ((\lambda X_{25} : 'a_5.X_{25} : 'a_6) : 'a_7)) : 'a_8
 \end{aligned}$$

Aggiungendo i vincoli dell'inferenza si ha

```
s_inf (Appl
  (Lambda 24
    (Plus
      (Id 24)
      (Int 1)))
  (Lambda 25
    (Id 25))) htilde
```

e la derivazione è

```
|- x24:a1 ==> ( a1 , a0=>a1 | end )

|- 1:a2 ==> ( int , a2=>int | end )

|- (x24:a1 + 1:a2):a3 ==>
  ( int , a1=>int | a3=>int | a0=>int | a2=>int | end )

|- (\x24:a0.(x24:a1 + 1:a2):a3):a4 ==> ( (int->int) ,
  a4=>(int->int) | a1=>int | a3=>int | a0=>int | a2=>int | end )
```

```

|- x25:a6 ==> ( a6 , a5=>a6 | end )

|- (\x25:a5.x25:a6):a7 ==>
  ( (a6->a6) , a7=>(a6->a6) | a5=>a6 | end )

|- (((\x24:a0.(x24:a1 + 1:a2):a3):a4)((\x25:a5.x25:a6):a7)):a8 ==> (
Program error: applicazione impossibile

```

Esempio 5.4.4 (Vincoli troppo restrittivi). *In questo esempio alla funzione identità sono attribuite specifiche di tipo troppo restrittive che impediscono il tipaggio.*

$$\begin{aligned}
 e &\equiv \lambda X_{24}.X_{24} \\
 f(e) &\equiv (\lambda X_{24}:int \rightarrow 'a_0.X_{24}:int): 'a_1 \rightarrow 'a_2
 \end{aligned}$$

```

s_0 (Alambda (24, (Freccia Numero (Variabile 0)))
      (Aid (24, Numero))
      (Freccia (Variabile 1) (Variabile 2))) htilde

```

```

|- x24:int ==> (
Program error: applicazione impossibile

```

In questo caso X_{24} viene analizzata in un ambiente di tipi in cui all'identificatore stesso è stato assegnato il tipo incompatibile $int \rightarrow 'a_0$, quindi il tipaggio non è possibile.

Esempio 5.4.5 (Funzione “sensata” non tipabile, inferenza pura). *Questo esempio illustra il caso in cui un sistema di tipi monomorfo non riesce a tipare una funzione intuitivamente sensata. Anche il sistema di ML rifiuta questa espressione.*

$$e \equiv \lambda X_6.(X_6 (\lambda X_{24}.X_{24} + 1)) (X_6 1)$$

```

|- x6:a1 ==> ( a1 , a0=>a1 | end )

|- x24:a3 ==> ( a3 , a2=>a3 | end )

|- 1:a4 ==> ( int , a4=>int | end )

```

```

|- (x24:a3 + 1:a4):a5 ==>
  ( int , a3=>int | a5=>int | a2=>int | a4=>int | end )

|- (\x24:a2.(x24:a3 + 1:a4):a5):a6 ==> ( (int->int) ,
  a6=>(int->int) | a3=>int | a5=>int | a2=>int | a4=>int | end )

|- ((x6:a1)((\x24:a2.(x24:a3 + 1:a4):a5):a6)):a7 ==> ( a7 ,
  a1=>((int->int)->a7) | a0=>((int->int)->a7) | a6=>(int->int) |
  a3=>int | a5=>int | a2=>int | a4=>int | end )

```

in questo passaggio la variabile a0, specifica assegnata a λX_6 , viene vincolata al tipo $((int \rightarrow int) \rightarrow a7)$.

```

|- x6:a8 ==> ( a8 , a0=>a8 | end )

|- 1:a9 ==> ( int , a9=>int | end )

|- ((x6:a8)(1:a9)):a10 ==>
  ( a10 , a8=>(int->a10) | a0=>(int->a10) | a9=>int | end )

```

qui il vincolo $a0=>(int \rightarrow a10)$ è in conflitto con il vincolo creato precedentemente.

```

|- (((x6:a1)((\x24:a2.(x24:a3 + 1:a4):a5):a6)):a7)
  (((x6:a8)(1:a9)):a10)):a11 ==> (
Program error: applicazione impossibile

```

L'impossibilità di tipare l'espressione è dovuta al fatto che la funzione X_6 è usata nel corpo con due tipi diversi, anche se essi sono istanze di uno stesso tipo $v \rightarrow v$; il conflitto tra i due vincoli creati per $a0$ impedisce il successo dell'inferenza.

Esempio 5.4.6 (Funzione polimorfa, inferenza). *Questo esempio è proposto da Monsuez in [24] come caso in cui il sistema di tipi monomorfo inferisce per un'espressione un tipo più specifico di quello che intuitivamente verrebbe calcolato.*

$$\begin{aligned}
e &\equiv (\lambda X_6. \lambda X_{24}. X_6(X_6 X_{24}))(\lambda X_{25}. 0) \\
f_{S_{inf}}(e) &\equiv (((\lambda X_6 : 'a_0. (\lambda X_{24} : 'a_1. \\
&\quad ((X_6 : 'a_2)((X_6 : 'a_3)(X_{24} : 'a_4)) : 'a_5)) : 'a_6) : 'a_7) : 'a_8) \\
&\quad ((\lambda X_{25} : 'a_9. 0 : 'a_{10}) : 'a_{11})) : 'a_{12}
\end{aligned}$$

```

s_inf (Appl
  (Lambda 6
    (Lambda 24
      (Appl
        (Id 6)
        (Appl
          (Id 6)
          (Id 24))))))
  (Lambda 25
    (Int 0))) htilde

|- x6:a2 ==> ( a2 , a0=>a2 | end )

|- x6:a3 ==> ( a3 , a0=>a3 | end )

|- x24:a4 ==> ( a4 , a1=>a4 | end )

|- ((x6:a3)(x24:a4)):a5 ==>
  ( a5 , a3=>(a4->a5) | a0=>(a4->a5) | a1=>a4 | end )

|- ((x6:a2)(((x6:a3)(x24:a4)):a5)):a6 ==>
  ( a4 , a2=>(a4->a4) | a0=>(a4->a4) | a3=>(a4->a4) |
    a5=>a4 | a6=>a4 | a1=>a4 | end )

|- (\x24:a1.((x6:a2)(((x6:a3)(x24:a4)):a5)):a6):a7 ==>
  ( (a4->a4) , a7=>(a4->a4) | a2=>(a4->a4) | a0=>(a4->a4) |
    a3=>(a4->a4) | a5=>a4 | a6=>a4 | a1=>a4 | end )

|- (\x6:a0.(\x24:a1.((x6:a2)(((x6:a3)(x24:a4)):a5)):a6):a7):a8 ==>
  ( ((a4->a4)->(a4->a4)) , a8=>((a4->a4)->(a4->a4)) |
    a7=>(a4->a4) | a2=>(a4->a4) | a0=>(a4->a4) | a3=>(a4->a4) |
    a5=>a4 | a6=>a4 | a1=>a4 | end )

|- 0:a10 ==> ( int , a10=>int | end )

|- (\x25:a9.0:a10):a11 ==>
  ( (a9->int) , a11=>(a9->int) | a10=>int | end )

|- (((\x6:a0.(\x24:a1.((x6:a2)(((x6:a3)(x24:a4)):a5)):a6):a7):a8)
  ((\x25:a9.0:a10):a11)):a12 ==> ( (int->int) ,

```

```

a4=>int | a9=>int | a12=>(int->int) |
a8=>((int->int)->(int->int)) | a7=>(int->int) |
a2=>(int->int) | a0=>(int->int) | a3=>(int->int) | a5=>int |
a6=>int | a1=>int | a11=>(int->int) | a10=>int | end )

```

Il tipo inferito per questa espressione è $int \rightarrow int$, mentre il tipo più generale è $v \rightarrow int$; questo perchè l'espressione restituisce 0 indipendentemente dall'argomento.

Esempio 5.4.7 (Funzione polimorfa, verifica). In questo caso i tipi specificati sono sempre ground; l'analisi ha esito positivo perchè le specifiche sono esattamente i tipi che ci si aspetterebbe. Il tipo inferito in assenza di specifiche non banali sarebbe $int \rightarrow v$. Come si vede la sostituzione è sempre la sostituzione vuota ad ogni passaggio; infatti l'espressione è chiusa e nell'analisi degli identificatori locali non entrano mai in gioco variabili.

$$e \equiv \mu X_6. \lambda X_{24}. X_6(X_{24} + 1)$$

$$f(e) \equiv (\mu X_6 : int \rightarrow int. (\lambda X_{24} : int. (X_6 : int \rightarrow int(X_{24} : int + 1 : int) : int) : int) : int \rightarrow int) : int \rightarrow int$$

```
|- x6:(int->int) ==> ( (int->int) , end )
```

```
|- x24:int ==> ( int , end )
```

```
|- 1:int ==> ( int , end )
```

```
|- (x24:int + 1:int):int ==> ( int , end )
```

```
|- ((x6:(int->int))((x24:int + 1:int):int)):int ==> ( int , end )
```

```
|- (\x24:int.((x6:(int->int))((x24:int + 1:int):int))
: int):(int->int) ==> ( (int->int) , end )
```

```
|- (MU 6:(int->int).(\x24:int.((x6:(int->int))((x24:int + 1:int)
: int)):int):(int->int)):int ==> ( (int->int) , end )
```

5.5 Insieme delle funzioni associate

La semantica generale permette di aggiungere specifiche di tipo a piacere alle espressioni dell'insieme \mathbf{E} ; ciò identifica in genere una *politica* di annotazione delle espressioni. Le diverse politiche con cui vengono aggiunte le specifiche ci portano ad identificare un insieme di semantiche che sono casi particolari di \mathbf{S}_0 .

Definizione 5.3. \mathbf{S}_{sp} è l'insieme delle semantiche di specifica dei tipi monomorfi. Esso comprende per esempio le infinite semantiche che formalizzano l'inferenza (che differiscono solo per il nome delle variabili, vedi def. 5.5).

Queste semantiche si differenziano per le specifiche che applicano alle espressioni. Si è detto in precedenza che ad ogni $\mathbf{S} \in \mathbf{S}_{sp}$ corrisponde un sottoinsieme di \mathbf{E}_τ .

Definizione 5.4. Data una semantica $\mathbf{S} \in \mathbf{S}_{sp}$, l'insieme $\mathbf{E}_S \subseteq \mathbf{E}_\tau$ è il dominio su cui \mathbf{S} calcola (cioè $\mathbf{S}[\bullet] \in \mathbf{E}_S \mapsto \wp(\mathbf{H}_V \times (\mathbf{M}_V^H \times \mathbf{C}^H))$). Si ha che \mathbf{E}_S è isomorfo a \mathbf{E} (per passare da un insieme all'altro basta aggiungere o togliere le annotazioni).

Seguendo questo ragionamento si può associare ad ogni semantica una funzione che annota una certa espressione con specifiche di tipo.

Definizione 5.5. Φ_{sp}^0 è l'insieme delle funzioni associate alle semantiche di specifica, cioè data una $\mathbf{S} \in \mathbf{S}_{sp}$ la funzione associata è una $f_S \in \mathbf{E} \mapsto \mathbf{E}_S$ che annota le espressioni con le specifiche caratteristiche di \mathbf{S} .

Definizione 5.6. Φ_{sp} è l'insieme delle classi di equivalenza su Φ_{sp}^0 ottenute tramite la relazione di equivalenza definita come:

$$\begin{aligned} \forall f_1, f_2 \in \Phi_{sp}^0. \quad & f_1 \equiv f_2 \\ & \Leftrightarrow \\ & \forall e \in \mathbf{E} . \exists g_e . f_1(e) = g_e(f_2(e)) \end{aligned} \tag{5.14}$$

dove $g_e : \mathbf{E}_\tau \mapsto \mathbf{E}_\tau$ è una funzione invertibile di trasformazione di espressioni annotate che si limita a cambiare il nome delle variabili di tipo usate nelle specifiche, mantenendo distinti i nomi di due variabili con nomi distinti.

Questa relazione di equivalenza serve a rendere i risultati indipendenti dai nomi delle variabili di tipo (fermo restando che nomi distinti devono continuare ad essere distinti).

Definizione 5.7. Ad esempio $f_{S_{inf}} \in \Phi_{sp}$ (cioè la funzione associata alla semantica \mathbf{S}_{inf} per l'inferenza pura, vedi sez. 5.6 e (5.16)) è definita come segue (i nomi delle

variabili di tipo usate nelle sottoespressioni sono sempre diversi):

$$\begin{aligned}
f_{S_{inf}}(n) &\stackrel{def}{=} n : v_n \\
f_{S_{inf}}(X) &\stackrel{def}{=} X : v_X \\
f_{S_{inf}}(e_1 + e_2) &\stackrel{def}{=} (f_{S_{inf}}(e_1) + f_{S_{inf}}(e_2)) : v \\
f_{S_{inf}}(if\ e_1\ then\ e_2\ else\ e_3) &\stackrel{def}{=} (if\ f_{S_{inf}}(e_1)\ then\ f_{S_{inf}}(e_2)\ else\ f_{S_{inf}}(e_3)) : v \\
f_{S_{inf}}(e_1\ e_2) &\stackrel{def}{=} ((f_{S_{inf}}(e_1))\ (f_{S_{inf}}(e_2))) : v \\
f_{S_{inf}}(\lambda X.e) &\stackrel{def}{=} (\lambda X : v_X.\ f_{S_{inf}}(e)) : v \\
f_{S_{inf}}(\mu f.(\lambda X.e)) &\stackrel{def}{=} (\mu f : v_f.\ (\lambda X : v_X.\ f_{S_{inf}}(e)) : v_\lambda) : v \quad (5.15)
\end{aligned}$$

In generale non è detto che le funzioni associate siano composizionali, perchè pretendere la composizionalità significherebbe porre vincoli troppo forti all'espressività delle specifiche di tipo; tranne alcuni casi particolari è difficile fornirne una definizione analitica. Ad esempio non esiste una funzione f composizionale per cui siano vere entrambe le seguenti verifiche:

$$\begin{aligned}
f(X + 1) &= (X : int + 1 : int) : int \\
f(Y + 2) &= (Y : int + 2 : int) : 'a
\end{aligned}$$

e in questo caso ciò impedisce di specificare due tipi diversi per due sottoespressioni ottenute con la regola delle operazioni aritmetiche.

Vedendo le cose in modo assolutamente formale neppure $f_{S_{inf}}$ è composizionale, perchè la variabile di tipo v specificata per l'espressione principale deve essere diversa da quelle usate nelle sottoespressioni, quindi è necessario avere informazioni sulle variabili già utilizzate nel corso del processo di aggiunta delle specifiche.

Data una semantica \mathbf{S} la funzione associata f_S formalizza la specializzazione che porta da \mathbf{S}_0 a \mathbf{S} :

$$\mathbf{S}[[e]] \stackrel{def}{=} \mathbf{S}_0[[f_S(e)]] \quad (5.16)$$

Quindi possiamo caratterizzare le diverse semantiche di \mathbf{S}_{sp} con la funzioni di “annotazione” $f \in \Phi_{sp}$ ad esse associate.

5.6 Inferenza

Il caso più tipico di analisi di tipi è l'inferenza pura; essa corrisponde, nella semantica generale, all'annotazione di un'espressione secondo la funzione $f_{S_{inf}}$ (oppure, in modo equivalente, all'analisi dell'espressione con la semantica $\mathbf{S}_{inf}[[\bullet]]$) (vedi def. 5.7). In questa sezione verrà mostrata l'equivalenza dei risultati ottenuti con $\mathbf{S}_0[[f_{S_{inf}}(\bullet)]]$ (oppure $\mathbf{S}_{inf}[[\bullet]]$) rispetto a quelli ottenuti con la semantica $\mathbf{T}^C[[\bullet]]$ (vedi sez. 4.5). Ciò sarà illustrato attraverso la nozione di *astrazione esatta*.

5.6.1 Connessione di Galois

Il primo passo è dimostrare che tra i domini delle due semantiche si ha una connessione di Galois (cap. 4), cioè che esistono due funzioni, dette di astrazione e di concretizzazione, che stabiliscono una certa relazione tra i domini. Questa relazione significa che il dominio astratto è un'approssimazione di quello concreto, cioè passando da un elemento concreto x al suo corrispondente astratto y si ha una perdita di informazione che provoca in generale l'impossibilità di risalire ad x partendo da y (si può arrivare ad un elemento concreto meno preciso di x).

Per prima cosa è necessario identificare le funzioni di astrazione e concretizzazione.

Definizione 5.8.

$$\begin{aligned}\alpha_0 & : \wp(\mathbf{H} \times \mathbf{M}^H) \mapsto \wp(\mathbf{H}_V \times (\mathbf{M}_V^H \times \mathbf{C}^H)) \\ \alpha_0(X) & = \{(H, (\tau, \theta)) \mid \text{ground}_V(\theta(H), \tau) \subseteq X\}\end{aligned}$$

Definizione 5.9.

$$\begin{aligned}\gamma_0 & : \wp(\mathbf{H}_V \times (\mathbf{M}_V^H \times \mathbf{C}^H)) \mapsto \wp(\mathbf{H} \times \mathbf{M}^H) \\ \gamma_0(Y) & = \bigcup_{(H, (\tau, \theta)) \in Y} \text{ground}_V(\theta(H), \tau)\end{aligned}$$

Le seguenti proposizioni verificano le condizioni della definizione di Galois connection (def. 4.2); la monotonia delle due funzioni è ovvia.

Proposizione 5.2. $\forall X. \gamma_0(\alpha_0(X)) \subseteq X$

Dimostrazione.

$$\begin{aligned}(H', \tau') \in \gamma_0(\alpha_0(X)) & \implies \{\text{def. 5.9}\} \\ \exists(H, (\tau, \theta)) \in \alpha_0(X) . (H', \tau') \in \text{ground}_V(\theta(H), \tau) & \implies \{\text{def. 5.8}\} \\ (H', \tau') \in X & \end{aligned}$$

□

Proposizione 5.3. $\forall Y. \alpha_0(\gamma_0(Y)) \supseteq Y$

Dimostrazione.

$$\begin{aligned}(H, (\tau, \theta)) \in Y & \implies \{\text{def. 5.9}\} \\ \text{ground}_V(\theta(H), \tau) \subseteq \gamma_0(Y) & \implies \{\text{def. 5.8}\} \\ (H, (\tau, \theta)) \in \alpha_0(\gamma_0(Y)) & \end{aligned}$$

□

Proposizione 5.4. *La seguente è una connessione di Galois:*

$$\langle \wp(\mathbf{H} \times \mathbf{M}^H), \supseteq \rangle \xrightleftharpoons[\alpha_0]{\gamma_0} \langle \wp(\mathbf{H}_V \times (\mathbf{M}_V^H \times \mathbf{C}^H)), \supseteq \rangle$$

Dimostrazione. Segue immediatamente dalle proposizioni 5.2 e 5.3. \square

5.6.2 Astrazione esatta

Il secondo passo è verificare che si tratta di un'astrazione esatta (def. 4.11); ciò si ottiene dimostrando prima due proposizioni:

Proposizione 5.5 (equivalenza tra \mathbf{S}_{inf} e \mathbf{T}^C).

$$\begin{aligned} \forall H, e, \tau, \theta. \quad & H \vdash f_{\mathbf{S}_{inf}}(e) \Rightarrow (\tau, \theta) \\ \implies & \\ \forall (H', \tau') \in \text{ground}_V(\theta(H), \tau). & H' \stackrel{c}{\vdash} e \Rightarrow \tau' \end{aligned}$$

Dimostrazione. In tutti i casi della dimostrazione (tranne i casi base) gli ultimi passaggi (passaggio dalle θ^i a θ , da $\text{ground}_V(H, \tau)$ a $\text{ground}_V(H, int) \dots$) sono resi possibili dal fatto che i passaggi di unificazione che portano a θ sono possibili (cioè la sostituzione θ esiste effettivamente perchè per ipotesi l'analisi con \mathbf{S}_{inf} ha successo). Ad esempio nel caso delle operazioni aritmetiche passare dalla quantificazione su $\text{ground}_V(\theta^1(H), \tau^1)$ e $\text{ground}_V(\theta^2(H), \tau^2)$ a quella su $\text{ground}_V(\theta(H), int)$ è possibile perchè τ^1 e τ^2 sono unificabili con int per ipotesi (cioè int è una loro istanza), mentre θ istanzia maggiormente le variabili rispetto a θ^1 e θ^2 ; di conseguenza il cambiamento della quantificazione è in sostanza il passaggio da due insiemi ad un sottoinsieme della loro intersezione.

interi

$$\begin{aligned} & H \vdash n : v_n \Rightarrow (int, \theta) \\ \implies & \{ \forall H' \in \mathbf{H}. H' \stackrel{c}{\vdash} n \Rightarrow int \} \\ & \forall (H', int) \in \text{ground}_V(\theta(H), int). H' \stackrel{c}{\vdash} n \Rightarrow int \end{aligned}$$

identificatori

$$\begin{aligned}
& H \vdash X : v_X \Rightarrow (\text{fst}(H(X)), \theta) \\
\Rightarrow & \{\forall H' \in \mathbf{H}. H' \overset{c}{\vdash} X \Rightarrow \text{fst}(H'(X))\} \\
& \forall H' \in \text{ground}_V(\theta(H)). H' \overset{c}{\vdash} X \Rightarrow \text{fst}(H'(X)) \\
\Rightarrow & \{H' \in \text{ground}_V(H'') \Rightarrow \exists \theta'. H' = \theta'(H'') \wedge \text{fst}(H'(X)) = \theta'(\text{fst}(H''(X)))\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H), \theta(\text{fst}(H(X)))) \\
& . (H', \tau') = \theta'(\theta(H), \theta(\text{fst}(H(X)))) \Rightarrow H' \overset{c}{\vdash} X \Rightarrow \theta'(\theta(\text{fst}(H(X)))) \\
\Rightarrow & \{\theta(\text{fst}(H(X))) = \text{fst}(H(X)), \text{ vedi prop. 5.1}\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H), \text{fst}(H(X))). H' \overset{c}{\vdash} X \Rightarrow \tau'
\end{aligned}$$

operazioni aritmetiche

$$\begin{aligned}
& H \vdash (e_1 : v_1 + e_2 : v_2) : v \Rightarrow (\text{int}, \theta) \\
\Rightarrow & \{\text{regola 5.6}\} \\
& H \vdash e_1 : v_1 \Rightarrow (\tau^1, \theta^1) \wedge H \vdash e_2 : v_2 \Rightarrow (\tau^2, \theta^2) \wedge \\
& \theta = \text{unify}(\{\tau^1 = \tau^2 = v = \text{int}, \theta^1, \theta^2\}) \\
\Rightarrow & \{\text{ipotesi induttiva}\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta^1(H), \tau^1). H' \overset{c}{\vdash} e_1 \Rightarrow \tau' \wedge \\
& \forall (H', \tau') \in \text{ground}_V(\theta^2(H), \tau^2). H' \overset{c}{\vdash} e_2 \Rightarrow \tau' \wedge \\
& \theta = \text{unify}(\{\tau^1 = \tau^2 = v = \text{int}, \theta^1, \theta^2\}) \\
\Rightarrow & \{\theta \geq \theta_1 \circ \theta^2, \text{ restringimento della quantificazione}\} \\
& \forall (H', \text{int}) \in \text{ground}_V(\theta(H), \text{int}). H' \overset{c}{\vdash} e_1 \Rightarrow \text{int} \wedge H' \overset{c}{\vdash} e_2 \Rightarrow \text{int} \\
\Rightarrow & \{\text{regola 3.10}\} \\
& \forall (H', \text{int}) \in \text{ground}_V(\theta(H), \text{int}). H' \overset{c}{\vdash} e_1 + e_2 \Rightarrow \text{int}
\end{aligned}$$

condizionale

$$\begin{aligned}
& H \vdash (\text{if } e_1 : v_1 \text{ then } e_2 : v_2 \text{ else } e_3 : v_3) : v \Rightarrow (\tau, \theta) \\
\Rightarrow & \{\text{regola 5.7}\} \\
& H \vdash e_1 : v_1 \Rightarrow (\tau^1, \theta^1) \wedge H \vdash e_2 : v_2 \Rightarrow (\tau^2, \theta^2) \wedge H \vdash e_3 : v_3 \Rightarrow (\tau^3, \theta^3) \wedge \\
& \theta = \text{unify}(\{\tau^1 = \text{int}, \tau^2 = \tau^3 = v, \theta^1, \theta^2, \theta^3\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{\text{ipotesi induttiva}\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta^1(H), \tau^1). H' \stackrel{c}{\vdash} e_1 \Rightarrow \tau' \wedge \\
& \forall (H', \tau') \in \text{ground}_V(\theta^2(H), \tau^2). H' \stackrel{c}{\vdash} e_2 \Rightarrow \tau' \wedge \\
& \forall (H', \tau') \in \text{ground}_V(\theta^3(H), \tau^3). H' \stackrel{c}{\vdash} e_3 \Rightarrow \tau' \wedge \\
& \theta = \text{unify}(\{\tau^1 = \text{int}, \tau^2 = \tau^3 = v, \theta^1, \theta^2, \theta^3\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{\theta \geq \theta^1 \circ \theta^2 \circ \theta^3, \text{restringimento della quantificazione}\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H), \text{gci}_V(\tau^2, \tau^3)). (H' \stackrel{c}{\vdash} e_2 \Rightarrow \tau' \wedge H' \stackrel{c}{\vdash} e_3 \Rightarrow \tau') \\
& \wedge \forall (H', \text{int}) \in \text{ground}_V(\theta(H), \text{int}). H' \stackrel{c}{\vdash} e_1 \Rightarrow \text{int} \wedge \\
& \theta = \text{unify}(\{\tau^1 = \text{int}, \tau^2 = \tau^3 = v, \theta^1, \theta^2, \theta^3\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{\text{regola 3.11}\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H), \tau). H' \stackrel{c}{\vdash} \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow \tau'
\end{aligned}$$

applicazione di funzione

$$\begin{aligned}
& H \vdash ((e_1 : v_1)(e_2 : v_2)) : v \Rightarrow (\tau, \theta) \\
\Rightarrow & \{\text{regola 5.8}\} \\
& H \vdash e_1 : v_1 \Rightarrow (\tau^1, \theta^1) \wedge H \vdash e_2 : v_2 \Rightarrow (\tau^2, \theta^2) \wedge \\
& \theta = \text{unify}(\{\tau^1 = \tau^2 \rightarrow v, \theta^1, \theta^2\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{\text{ipotesi induttiva}\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta^1(H), \tau^1). H' \stackrel{c}{\vdash} e_1 \Rightarrow \tau' \wedge \\
& \forall (H', \tau') \in \text{ground}_V(\theta^2(H), \tau^2). H' \stackrel{c}{\vdash} e_2 \Rightarrow \tau' \wedge \\
& \theta = \text{unify}(\{\tau^1 = \tau^2 \rightarrow v, \theta^1, \theta^2\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{\theta \geq \theta^1 \circ \theta^2, \text{restringimento della quantificazione}\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H), \theta(\tau^1)). H' \stackrel{c}{\vdash} e_1 \Rightarrow \tau' \wedge \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H), \theta(\tau^2)). H' \stackrel{c}{\vdash} e_2 \Rightarrow \tau' \wedge \\
& \theta = \text{unify}(\{\tau^1 = \tau^2 \rightarrow v, \theta^1, \theta^2\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{\text{regola 3.12}\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H), \tau). H' \stackrel{c}{\vdash} e_1 e_2 \Rightarrow \tau'
\end{aligned}$$

astrazione

$$\begin{aligned}
& H \vdash (\lambda X : v_X.e : v_e) : v \Rightarrow (\tau, \theta) \\
\Rightarrow & \{regola 5.9\} \\
& H[X \leftarrow (v_X, \varepsilon)] \vdash e : v_e \Rightarrow (\tau^e, \theta^e) \wedge \\
& \theta = \text{unify}(\{v_X \rightarrow \tau^e = v, \theta^e\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{ipotesi induttiva\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta^e(H[X \leftarrow (v_X, \varepsilon)]), \tau^e). H' \stackrel{c}{\vdash} e \Rightarrow \tau' \wedge \\
& \theta = \text{unify}(\{v_X \rightarrow \tau^e = v, \theta^e\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{\theta \geq \theta^e, \text{restringimento della quantificazione}\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H[X \leftarrow (v_X, \varepsilon)]), \theta(\tau^e)). H' \stackrel{c}{\vdash} e \Rightarrow \tau' \wedge \\
& \theta = \text{unify}(\{v_X \rightarrow \tau^e = v, \theta^e\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{logica\} \\
& (\forall (H', \tau') \in \text{ground}_V(\theta(H), \theta(\tau^e)) \text{ con } (H', \tau') = \theta'(\theta(H), \theta(\tau^e))) \\
& . H'[X \leftarrow (\theta'(v_X), \varepsilon)] \stackrel{c}{\vdash} e \Rightarrow \tau' \wedge \\
& \theta = \text{unify}(\{v_X \rightarrow \tau^e = v, \theta^e\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{regola 3.13\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H), \tau). H' \stackrel{c}{\vdash} \lambda X.e \Rightarrow \tau'
\end{aligned}$$

ricorsione con fix

$$\begin{aligned}
& H \vdash (\mu f : v_f.e : v_e) : v \Rightarrow (\tau, \theta) \\
\Rightarrow & \{regola 5.12\} \\
& H \vdash (\mu f : v_f.e : v_e) : v \Rightarrow_{T_P}^{n-1} (\tau, \theta^1) \wedge H \vdash (\mu f : v_f.e : v_e) : v \Rightarrow_{T_P}^n (\tau, \theta) \\
\Rightarrow & \{regola 5.11\} \\
& H[f \leftarrow (\tau, \theta^1)] \vdash e : v_e \Rightarrow (\tau^2, \theta^2) \wedge \\
& \theta = \text{unify}(\{\tau^2 = v = v_f, \theta^2\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{ipotesi induttiva\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta^2(H[f \leftarrow (\tau, \theta^1)]), \tau^2). H' \stackrel{c}{\vdash} e \Rightarrow \tau' \wedge \\
& \theta = \text{unify}(\{\tau^2 = v = v_f, \theta^2\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{\theta \geq \theta^2, \text{restringimento della quantificazione}\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H[f \leftarrow (\tau, \theta^1)]), \theta(\tau^2)). H' \stackrel{c}{\vdash} e \Rightarrow \tau' \wedge \\
& \theta = \text{unify}(\{\tau^2 = v = v_f, \theta^2\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{logica, \theta(\tau) = \tau\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H), \tau) \text{ con } (H', \tau') = \theta'(\theta(H), \tau) \\
& . H'[f \leftarrow (\theta'(\tau), \theta^1)] \stackrel{c}{\vdash} e \Rightarrow \tau' \wedge \\
& \theta = \text{unify}(\{\tau^2 = v = v_f, \theta^2\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{regola 3.14\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H), \tau). H' \stackrel{c}{\vdash} \mu f.e \Rightarrow \tau'
\end{aligned}$$

ricorsione senza fix

$$\begin{aligned}
& H \vdash (\mu f : v_f.e : v_e) : v \Rightarrow (\tau, \theta) \\
\Rightarrow & \{regola 5.13\} \\
& H[f \leftarrow (v_f, \varepsilon)] \vdash e : v_e \Rightarrow (\tau', \theta') \wedge \theta = \text{unify}(\{\tau' = v_f = v, \theta'\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{ipotesi induttiva\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta'(H[f \leftarrow (v_f, \varepsilon)]), \tau'). H' \stackrel{c}{\vdash} e \Rightarrow \tau' \wedge \\
& \theta = \text{unify}(\{\tau' = v_f = v, \theta'\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{\theta \geq \theta', \text{ restringimento della quantificazione}\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H[f \leftarrow (v_f, \varepsilon)]), \theta(\tau')). H' \stackrel{c}{\vdash} e \Rightarrow \tau' \wedge \\
& \theta = \text{unify}(\{\tau' = v_f = v, \theta'\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{\text{logica}, \theta(\tau') = \theta(v) = \tau\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H), \tau) \text{ con } (H', \tau') = \theta''(\theta(H), \tau) \\
& . H'[f \leftarrow (\theta''(\theta(v_f)), \varepsilon)] \stackrel{c}{\vdash} e \Rightarrow \tau' \wedge \\
& \theta = \text{unify}(\{\tau' = v_f = v, \theta'\}) \wedge \tau = \theta(v) \\
\Rightarrow & \{regola 3.14\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H), \tau). H' \stackrel{c}{\vdash} \mu f.e \Rightarrow \tau'
\end{aligned}$$

□

Proposizione 5.6.

$$\forall e. (H', \tau') \in \mathbf{T}^C[[e]] \implies \exists (H', (\tau, \theta)) \in \mathbf{S}_{inf}[[e]]. \tau' \in \text{ground}_V(\tau)$$

Dimostrazione. Nell'ultimo passaggio l'applicazione dell'opportuna regola di \mathbf{S}_{inf} è resa possibile dal fatto che le unificazioni necessarie hanno sempre successo per come sono stati costruiti i sottocasi.

Ad esempio nel caso delle operazioni aritmetiche sappiamo che τ_1 e τ_2 sono unificabili con *int* perché quest'ultimo ne è un'istanza. Inoltre l'unificazione di θ_1 e θ_2 è sempre possibile perché nel caso dell'inferenza le specifiche annotate sono tutte banali e le due sostituzioni non sono mai in conflitto.

interi e identificatori

Va bene $(H', (\tau', \theta))$ per un certo θ .

operazioni aritmetiche

$$\begin{aligned}
& (H', int) \in \mathbf{T}^C[e_1 + e_2] \\
\Rightarrow & \{regola 3.10\} \\
& (H', int) \in \mathbf{T}^C[e_1] \wedge (H', int) \in \mathbf{T}^C[e_2] \\
\Rightarrow & \{ipotesi induttiva\} \\
& \exists(H', (\tau^1, \theta^1)) \in \mathbf{S}_{inf}[e_1].int \in \text{ground}_V(\tau^1) \wedge \\
& \exists(H', (\tau^2, \theta^2)) \in \mathbf{S}_{inf}[e_2].int \in \text{ground}_V(\tau^2) \\
\Rightarrow & \{regola 5.6\} \\
& \exists(H', (int, \theta)) \in \mathbf{S}_{inf}[e_1 + e_2]
\end{aligned}$$

condizionale

$$\begin{aligned}
& (H', \tau') \in \mathbf{T}^C[if\ e_1\ then\ e_2\ else\ e_3] \\
\Rightarrow & \{regola 3.11\} \\
& (H', int) \in \mathbf{T}^C[e_1] \wedge (H', \tau') \in \mathbf{T}^C[e_2] \wedge (H', \tau') \in \mathbf{T}^C[e_3] \\
\Rightarrow & \{ipotesi induttiva\} \\
& \exists(H', (\tau^1, \theta^1)) \in \mathbf{S}_{inf}[e_1].int \in \text{ground}_V(\tau^1) \wedge \\
& \exists(H', (\tau^2, \theta^2)) \in \mathbf{S}_{inf}[e_2].\tau' \in \text{ground}_V(\tau^2) \wedge \\
& \exists(H', (\tau^3, \theta^3)) \in \mathbf{S}_{inf}[e_3].\tau' \in \text{ground}_V(\tau^3) \\
\Rightarrow & \{regola 5.7\} \\
& \exists(H', (\tau, \theta)) \in \mathbf{S}_{inf}[if\ e_1\ then\ e_2\ else\ e_3].\tau' \in \text{ground}_V(\tau)
\end{aligned}$$

applicazione di funzione

$$\begin{aligned}
& (H', \tau') \in \mathbf{T}^C[e_1\ e_2] \\
\Rightarrow & \{regola 3.12\} \\
& (H', \tau'' \rightarrow \tau') \in \mathbf{T}^C[e_1] \wedge (H', \tau'') \in \mathbf{T}^C[e_2] \\
\Rightarrow & \{ipotesi induttiva\} \\
& \exists(H', (\tau^1, \theta^1)) \in \mathbf{S}_{inf}[e_1].\tau'' \rightarrow \tau' \in \text{ground}_V(\tau^1) \wedge \\
& \exists(H', (\tau^2, \theta^2)) \in \mathbf{S}_{inf}[e_2].\tau'' \in \text{ground}_V(\tau^2) \\
\Rightarrow & \{regola 5.8\} \\
& \exists(H', (\tau, \theta)) \in \mathbf{S}_{inf}[e_1\ e_2].\tau' \in \text{ground}_V(\tau)
\end{aligned}$$

astrazione

$$\begin{aligned}
& (H', \tau' \rightarrow \tau'') \in \mathbf{T}^C[\lambda X.e] \\
\Rightarrow & \{\text{regola 3.13}\} \\
& (H'[X \leftarrow (\tau', \varepsilon)], \tau'') \in \mathbf{T}^C[e] \\
\Rightarrow & \{\text{ipotesi induttiva}\} \\
& \exists(H'[X \leftarrow (\tau', \varepsilon)], (\tau^1, \theta^1)) \in \mathbf{S}_{inf}[e]. \tau'' \in \text{ground}_V(\tau^1) \\
\Rightarrow & \{\text{generalizzando l'ambiente il tipaggio resta possibile}\} \\
& \exists(H'[X \leftarrow (v, \varepsilon)], (\tau_1, \theta_1)) \in \mathbf{S}_{inf}[e]. \tau'' \in \text{ground}_V(\tau_1) \\
\Rightarrow & \{\text{regola 5.9}\} \\
& \exists(H', (v \rightarrow \tau_1, \theta)) \in \mathbf{S}_{inf}[\lambda X.e]. \tau' \rightarrow \tau'' \in \text{ground}_V(v \rightarrow \tau_1)
\end{aligned}$$

ricorsione con fix

$$\begin{aligned}
& (H', \tau') \in \mathbf{T}^C[\mu f.e] \\
\Rightarrow & \{\text{regola 3.14}\} \\
& (H'[f \leftarrow (\tau', \varepsilon)], \tau') \in \mathbf{T}^C[e] \\
\Rightarrow & \{\text{ipotesi induttiva}\} \\
& \exists(H'[f \leftarrow (\tau', \varepsilon)], (\tau^1, \theta^1)) \in \mathbf{S}_{inf}[e]. \tau' \in \text{ground}_V(\tau^1) \\
\Rightarrow & \{\text{generalizzando l'ambiente il tipaggio resta possibile}\} \\
& \exists(H'[f \leftarrow (v, \varepsilon)], (\tau_1, \theta_1)) \in \mathbf{S}_{inf}[e]. \tau' \in \text{ground}_V(\tau_1) \\
\Rightarrow & \{\text{regole 5.12, 5.11}\} \\
& \exists(H', (\tau, \theta)) \in \mathbf{S}_{inf}[\mu f.e]. \tau' \in \text{ground}_V(\tau)
\end{aligned}$$

ricorsione senza fix

$$\begin{aligned}
& (H', \tau') \in \mathbf{T}^C[\mu f.e] \\
\Rightarrow & \{\text{regola 3.14}\} \\
& (H'[f \leftarrow (\tau', \varepsilon)], \tau') \in \mathbf{T}^C[e] \\
\Rightarrow & \{\text{ipotesi induttiva}\} \\
& \exists(H'[f \leftarrow (\tau', \varepsilon)], (\tau^1, \theta^1)) \in \mathbf{S}_{inf}[e]. \tau' \in \text{ground}_V(\tau^1) \\
\Rightarrow & \{\text{generalizzando l'ambiente il tipaggio resta possibile}\} \\
& \exists(H'[f \leftarrow (v, \varepsilon)], (\tau_1, \theta_1)) \in \mathbf{S}_{inf}[e]. \tau' \in \text{ground}_V(\tau_1) \\
\Rightarrow & \{\text{regola 5.13}\} \\
& \exists(H', (\tau_1, \theta)) \in \mathbf{S}_{inf}[\mu f.e]. \tau' \in \text{ground}_V(\tau_1)
\end{aligned}$$

□

Ora è possibile dimostrare che \mathbf{S}_{inf} è un'astrazione esatta di \mathbf{T}^C .

Proposizione 5.7. $\forall e . \alpha_0(\mathbf{T}^C[[e]]) \supseteq \mathbf{S}_{inf}[[e]]$

Dimostrazione.

$$\begin{aligned}
& (H, (\tau, \theta)) \in \mathbf{S}_{inf}[[e]] \\
\Rightarrow & \{prop. 5.5\} \\
& \forall (H', \tau') \in \text{ground}_V(\theta(H), \tau). (H', \tau') \in \mathbf{T}^C[[e]] \\
\Rightarrow & \{def. 5.8\} \\
& (H, (\tau, \theta)) \in \alpha_0(\mathbf{T}^C[[e]])
\end{aligned}$$

□

Proposizione 5.8. $\forall e . \gamma_0(\mathbf{S}_{inf}[[e]]) \subseteq \mathbf{T}^C[[e]]$

Dimostrazione.

$$\begin{aligned}
& (H', \tau') \in \gamma_0(\mathbf{S}_{inf}[[e]]) \\
\Rightarrow & \{def. 5.9\} \\
& \exists (H, (\tau, \theta)) \in \mathbf{S}_{inf}[[e]]. (H', \tau') \in \text{ground}_V(\theta(H), \tau) \\
\Rightarrow & \{prop. 5.5\} \\
& (H', \tau') \in \mathbf{T}^C[[e]]
\end{aligned}$$

□

Proposizione 5.9. $\forall e . \gamma_0(\mathbf{S}_{inf}[[e]]) \supseteq \mathbf{T}^C[[e]]$

Dimostrazione.

$$\begin{aligned}
& (H', \tau') \in \mathbf{T}^C[[e]] \\
\Rightarrow & \{prop. 5.6\} \\
& \exists (H', (\tau, \theta)) \in \mathbf{S}_{inf}[[e]]. \tau' \in \text{ground}_V(\tau) \\
\Rightarrow & \{def. 5.9\} \\
& (H', \tau') \in \gamma_0(\mathbf{S}_{inf}[[e]])
\end{aligned}$$

□

Il risultato finale segue direttamente dalla def. 4.11.

5.7 Altre semantiche particolari

Come già detto in precedenza ogni funzione associata (o, meglio, ogni classe di equivalenza delle funzioni associate) identifica un modo di aggiungere specifiche ad un'espressione, quindi in sostanza un modo di fare analisi di tipi.

La semantica \mathbf{S}_{inf} è intuitivamente un estremo dell'insieme \mathbf{S}_{sp} , cioè la semantica che ha specifiche di tipo meno restrittive. All'altro estremo troviamo la semantica che ha specifiche così restrittive da non poter mai essere soddisfatte:

$$\begin{aligned} f_{\perp}(X) &\stackrel{def}{=} X : \perp_M \\ f_{\perp}(e_1 + e_2) &\stackrel{def}{=} (f_{\perp}(e_1) + f_{\perp}(e_2)) : \perp_M \\ f_{\perp}(\lambda X.e) &\stackrel{def}{=} (\lambda X : \perp_M. f_{\perp}(e)) : \perp_M \end{aligned} \quad (5.17)$$

Un'altra semantica interessante è quella che assegna agli identificatori liberi un tipo ground ($\tau \in \mathbf{M}^H$) ed usa specifiche banali sulle altre sottoespressioni. Questo tipo di semantiche è in corrispondenza biunivoca con l'insieme dei type environments \mathbf{H} (oppure, in una versione non ground, con \mathbf{H}_V):

$$\begin{aligned} f_H(X) &\stackrel{def}{=} X : \tau \quad \text{dove } \tau = H(X) \\ f_H(e_1 + e_2) &\stackrel{def}{=} (f_H(e_1) + f_H(e_2)) : v \\ f_H(\lambda X.e) &\stackrel{def}{=} (\lambda X : \tau. f_H(e)) : v \quad \text{dove } \tau = H(X) \end{aligned} \quad (5.18)$$

Perchè il comportamento dell'analisi sia sensato è opportuno che l'insieme degli identificatori liberi sia disgiunto da quello degli identificatori introdotti per astrazione o ricorsione; inoltre tutte le astrazioni e ricorsioni introducono identificatori diversi (ciò non pone limitazioni significative alle espressioni valide, è sufficiente infatti rinominare le variabili legate). In caso contrario si avrebbe il vincolo poco intuitivo che per un identificatore X introdotto ad esempio per astrazione debba essere specificato lo stesso tipo già usato quando lo stesso identificatore occorre come variabile libera (cioè $H(X)$).

5.7.1 Ordinamento parziale

Le semantiche viste finora, tutti casi particolari di \mathbf{S}_0 , hanno intuitivamente un certo grado di "restrittività"; la semantica associata all'inferenza non pone vincoli ai tipi delle sottoespressioni, quindi permette di tipare tutte le espressioni che sono effettivamente tipabili secondo l'inferenza monomorfa. Dall'altro lato la semantica \mathbf{S}_{\perp} , di cui f_{\perp} è la funzione associata, rifiuta qualsiasi espressione ritenendola non tipabile. Tra questi due estremi si trovano tutti gli altri casi particolari.

Si vuole quindi cercare di formalizzare una nozione di *ordinamento parziale* tra semantiche, per cui la semantica “maggiore” è quella che permette di tipare meno facilmente le espressioni. Invece di considerare l’insieme \mathbf{S}_{sp} si userà Φ_{sp} ; infatti \mathbf{S}_{sp} è isomorfo a Φ_{sp}^0 ed un ordinamento ragionevole non dovrebbe fare differenza tra due funzioni appartenenti alla stessa classe di equivalenza (che, cioè, differiscono solo per il nome delle variabili usate).

Definizione 5.10. *L’ordinamento parziale su $\mathbf{H}_V \times (\mathbf{M}_V^H \times \mathbf{C}^H)$ è definito come:*

$$\begin{aligned} \leq_t &\subseteq (\mathbf{H}_V \times (\mathbf{M}_V^H \times \mathbf{C}^H)) \times (\mathbf{H}_V \times (\mathbf{M}_V^H \times \mathbf{C}^H)) \\ \leq_t &\stackrel{def}{=} \{((H_1, (\tau_1, \theta_1)), (H_2, (\tau_2, \theta_2))) \mid \\ &\quad \text{ground}_V(\theta_1(H_1), \tau_1) \subseteq \text{ground}_V(\theta_2(H_2), \tau_2)\} \end{aligned}$$

Definizione 5.11. *L’ordinamento parziale su Φ_{sp} è definito come:*

$$\begin{aligned} \leq_\Phi &\subseteq \Phi_{sp} \times \Phi_{sp} \\ \leq_\Phi &\stackrel{def}{=} \{(f_1, f_2) \mid \forall e, H. \\ &\quad (\exists x.(H, x) \in \mathbf{S}_0[[f_2(e)]] \Rightarrow \exists y.(H, y) \in \mathbf{S}_0[[f_1(e)]] \wedge (H, x) \leq_t (H, y))\} \end{aligned}$$

Ciò significa che se $f_{S_1} \leq_\Phi f_{S_2}$ la semantica meno restrittiva \mathbf{S}_1 riesce a tipare tutte le espressioni tipate da \mathbf{S}_2 e restituisce un tipo meno restrittivo.

Proposizione 5.10. *$f_{S_{inf}}$ è l’estremo inferiore dell’insieme Φ_{sp} parzialmente ordinato secondo \leq_Φ ; ciò significa che la funzione per l’inferenza pura riesce a tipare il maggior numero di espressioni e con il tipo più generale possibile.*

Dimostrazione. È ovvio che \mathbf{S}_{inf} riesca a tipare il maggior numero di espressioni e con il tipo più generale, perchè sottopone le espressioni al minor numero possibile di vincoli. Inoltre si ha che

$$\begin{aligned} \forall f, e, H \vdash f_{S_{inf}}(e) \Rightarrow (\tau_1, \theta_1) \wedge H \vdash f(e) \Rightarrow (\tau_2, \theta_2) \\ \implies \text{ground}_V(\theta_2(H), \tau_2) \subseteq \text{ground}_V(\theta_1(H), \tau_1) \end{aligned}$$

Ciò è vero se e è un intero o un identificatore (supponendo che le sostituzioni all’interno degli ambienti di tipo non modifichino l’ambiente stesso, $\forall X.H(X) = (\tau, \theta) \Rightarrow \theta(H) = H$, vedi def. 2.12; ciò non è vero in generale ma lo è nel caso di inferenza su espressioni chiuse), perchè $\theta_1(H) = H$. Per i casi induttivi la proprietà rimane vera perchè nell’inferenza gli unici vincoli che vengono creati sulle variabili dell’ambiente sono quelli presenti implicitamente nelle regole (ad esempio in (5.6) si ha il vincolo $\tau^1 = \tau^2 = int$); questi vincoli comunque sono creati con tutte le possibili f , quindi nel complesso i vincoli creati dall’inferenza su H tramite la sostituzione θ_1 sono il minimo sull’insieme delle funzioni associate. \square

5.8 Ricorsione

Come si è già visto nell'enunciazione delle regole della semantica generale ci sono due diverse regole per l'analisi di tipo delle funzioni ricorsive: solo una delle due usa il metodo iterativo nella ricerca del punto fisso. I due approcci sono assolutamente equivalenti.

Come si può vedere facilmente il metodo non iterativo (5.13) è equivalente al primo passo del metodo iterativo (5.10, 5.11, 5.12). Per dimostrare che i due approcci sono equivalenti basta quindi mostrare che il metodo iterativo converge dopo un solo passo.

Proposizione 5.11 (corollario della proposizione 5.5).

$$\begin{aligned} \forall f, e, H. \quad H \vdash f(e) \Rightarrow (\tau, \theta) \\ \implies \\ \forall (H', \tau') \in \text{ground}_V(\theta(H), \tau). H' \vdash^c e \Rightarrow \tau' \end{aligned}$$

Dimostrazione.

$$\begin{aligned} & H \vdash f(e) \Rightarrow (\tau, \theta) \\ \implies & \{prop. 5.10\} \\ & H \vdash f_{S_{in,f}}(e) \Rightarrow (\tau'', \theta'') \wedge \text{ground}_V(\theta(H), \tau) \subseteq \text{ground}_V(\theta''(H), \tau'') \\ \implies & \{prop. 5.5\} \\ & \forall (H', \tau') \in \text{ground}_V(\theta''(H), \tau''). H' \vdash^c e \Rightarrow \tau' \\ & \wedge \text{ground}_V(\theta(H), \tau) \subseteq \text{ground}_V(\theta''(H), \tau'') \\ \implies & \{logica\} \\ & \forall (H', \tau') \in \text{ground}_V(\theta(H), \tau). H' \vdash^c e \Rightarrow \tau' \end{aligned}$$

□

Come conseguenza si ha che tutte le istanze ground del tipo calcolato per una funzione ricorsiva sono punti fissi secondo \mathbf{T}^C :

$$\begin{aligned} H \vdash f(\mu f.e) \Rightarrow (\tau^0, \theta^0) \\ \implies \forall (H', \tau') \in \text{ground}_V(\theta^0(H), \tau^0). H' \vdash^c \mu f.e \Rightarrow \tau' \end{aligned} \tag{5.19}$$

Ora si vuole dimostrare che il tipo (τ^0, θ^0) calcolato con l'approccio non iterativo (o, equivalentemente, con il primo passo dell'approccio iterativo) è un punto fisso. La seguente proposizione è una condizione sufficiente.

Proposizione 5.12. $(H[f \leftarrow (\tau^0, \theta^0)] \vdash e : \tau_e \Rightarrow (\tau^1, \theta^1)) \implies \tau^0 = \tau^1.$

Dimostrazione. Sia per assurdo $\tau^0 \neq \tau^1$; in particolare ci saranno istanze ground di τ^0 che non sono istanze di τ^1 o viceversa.

Per la prop. 5.11 si ha $\forall(H', \tau') \in \text{ground}_V(\theta^1(H), \tau^1). H'[f \leftarrow (\tau'', \theta'')] \vdash^c e \Rightarrow \tau'$ con $(\tau', \tau'') = \theta_0(\tau^1, \tau^0)$ per ognuna delle sostituzioni grounding θ_0 .

Se $\tau^0 \neq \tau^1$ esiste una sostituzione grounding θ_0 per cui $\tau' \neq \tau''$. Di conseguenza ci sarebbero τ'' che non sono punti fissi secondo \mathbf{T}^C , ma ciò è assurdo in quanto per (5.19) tutte le istanze ground di τ^0 sono punti fissi (infatti τ^0 è il tipo ottenuto con (5.13), nonchè il tipo ottenuto con il primo passo iterativo, quindi si può applicare (5.19)). \square

Al secondo passaggio iterativo viene calcolato il tipo τ^1 che è uguale a τ^0 per la proposizione appena dimostrata. Esso verrà poi unificato con le specifiche attribuite all'espressione (τ_f e τ se $f(e) \equiv (\mu f : \tau_f.e : \tau_e) : \tau$), ma poichè τ^0 è già di per sè il prodotto di questa unificazione il tipo risultante rimane $\tau^1 = \tau^0$. Di conseguenza il punto fisso è raggiunto dopo una sola iterazione, e il metodo iterativo è equivalente a quello non iterativo.

Il punto fisso raggiunto non è sempre il più generale (si veda l'esempio 5.8.1).

5.8.1 Operatori di widening

In [24] Bruno Monsuez mostra che il sistema di tipi di ML è ottenuto da ML+ con l'uso di un operatore di widening. Per fare ciò Monsuez astrae il tipo di una costante definita ricorsivamente con il termine (τ, θ, I) dove $I \in \wp(\mathbf{M}_V^H \times \mathbf{C}^H)$ è l'insieme delle istanze di τ usate durante l'esecuzione del programma. In questa sezione il concetto di widening sarà duale rispetto a quello delineato nella def. 4.12: l'ordinamento è opposto, le catene sono discendenti e il punto di partenza è l'estremo superiore.

Nel caso di \mathbf{S}_0 se f è una funzione ricorsiva le varie occorrenze di f nel corpo della funzione hanno tutte una specifica di tipo; i vincoli creati per le variabili contenute nelle specifiche indicano le istanze usate durante l'esecuzione.

Esempio 5.8.1. *Questa funzione ha intuitivamente tipo $v \rightarrow \text{int}$, ma la semantica generale e ML calcolano $\text{int} \rightarrow \text{int}$; infatti nonostante il termine $X_6 0$ non venga mai considerato l'applicazione della funzione ricorsiva a un intero ne forza comunque il tipo inferito.*

$$\begin{aligned}
e &\equiv \mu X_6. (\lambda X_{24}. ((\lambda X_{25}. \lambda X_{26}. X_{25}) 0) (X_6 0)) \\
f_{s_{in_f}}(e) &\equiv (\mu X_6 : 'a_0. (\lambda X_{24} : 'a_1. \\
&\quad (((((\lambda X_{25} : 'a_2. (\lambda X_{26} : 'a_3. X_{25} : 'a_4) : 'a_5) : 'a_6)(0 : 'a_7)) : 'a_8) \\
&\quad (((X_6 : 'a_9)(0 : 'a_{10})) : 'a_{11})) : 'a_{12}) : 'a_{13}) : 'a_{14}
\end{aligned}$$

In questo caso I è formato dalla sola coppia (τ_9, θ_9) , dove θ_9 è la sostituzione calcolata al momento in cui l'occorrenza di X_6 è applicata (cioè quando si valuta X_6 0) e $\tau_9 = \theta_9('a_9)$ (infatti $'a_9$ è la specifica attribuita all'unica occorrenza di X_6 nel corpo della definizione ricorsiva).

Viene usata la regola per la ricorsione senza punto fisso, implementata dalla funzione `s_inf_nofix`.

```
s_inf_nofix (Mu 6
             (Lambda 24
              (Appl
               (Appl
                (Lambda 25
                 (Lambda 26
                  (Id 25))))
                (Int 0))
               (Appl
                (Id 6)
                (Int 0)))))) htilde

|- x25:a4 ==> ( a4 , a2=>a4 | end )

|- (\x26:a3.x25:a4):a5 ==> ( (a3->a4) , a5=>(a3->a4) | a2=>a4 | end )

|- (\x25:a2.(\x26:a3.x25:a4):a5):a6 ==> ( (a4->(a3->a4)) ,
    a6=>(a4->(a3->a4)) | a5=>(a3->a4) | a2=>a4 | end )

|- 0:a7 ==> ( int , a7=>int | end )

|- (((\x25:a2.(\x26:a3.x25:a4):a5):a6)(0:a7)):a8 ==> ( (a3->int) ,
    a4=>int | a8=>(a3->int) | a6=>(int->(a3->int)) | a5=>(a3->int) |
    a2=>int | a7=>int | end )

|- x6:a9 ==> ( a9 , a0=>a9 | end )

|- 0:a10 ==> ( int , a10=>int | end )

|- ((x6:a9)(0:a10)):a11 ==>
    ( a11 , a9=>(int->a11) | a0=>(int->a11) | a10=>int | end )
```

θ_9 è la sostituzione calcolata a questo passaggio; τ_9 è l'applicazione di questa a a_9 , cioè `int->a11`.

```

|- (((((\x25:a2.(\x26:a3.x25:a4):a5):a6)(0:a7)):a8)
  (((x6:a9)(0:a10)):a11)):a12 ==> ( int ,
  a3=>a11 | a12=>int | a4=>int | a8=>(a11->int) |
  a6=>(int->(a11->int)) | a5=>(a11->int) | a2=>int | a7=>int |
  a9=>(int->a11) | a0=>(int->a11) | a10=>int | end )

|- (\x24:a1.((((\x25:a2.(\x26:a3.x25:a4):a5):a6)(0:a7)):a8)
  (((x6:a9)(0:a10)):a11)):a12):a13 ==> ( (a1->int) ,
  a13=>(a1->int) | a3=>a11 | a12=>int | a4=>int | a8=>(a11->int) |
  a6=>(int->(a11->int)) | a5=>(a11->int) | a2=>int | a7=>int |
  a9=>(int->a11) | a0=>(int->a11) | a10=>int | end )

```

Sia θ' sostituzione ottenuta dopo questo passaggio.

```

|- (MU 6:a0.(\x24:a1.((((\x25:a2.(\x26:a3.x25:a4):a5):a6)(0:a7)):a8)
  (((x6:a9)(0:a10)):a11)):a12):a13):a14 ==> ( (int->int) ,
  a0=>(int->int) | a14=>(int->int) | a13=>(int->int) | a3=>int |
  a12=>int | a4=>int | a8=>(int->int) | a6=>(int->(int->int)) |
  a5=>(int->int) | a2=>int | a7=>int | a9=>(int->int) | a1=>int |
  a11=>int | a10=>int | end )

```

La regola 5.13 viene modificata con

$$\frac{H[f \leftarrow (\tau_f, \varepsilon, \emptyset)] \vdash e : \tau_e \Rightarrow (\tau', \theta', I') \quad (\tau'', \theta'', I'') = (\tau_f, \varepsilon, \emptyset) \nabla (\tau', \theta', I')}{H \vdash (\mu f : \tau_f. e : \tau_e) : \tau \Rightarrow (\text{apply}(\theta, \tau), \theta')} \quad (5.20)$$

La regola 5.20 applica ai tipi $(\tau_f, \varepsilon, \emptyset)$ e (τ', θ', I') un operatore ∇ definito come

$$\begin{aligned}
L &\stackrel{def}{=} (\mathbf{M}_V^H \times \mathbf{H}_V \times \wp(\mathbf{M}_V^H \times \mathbf{C}^H)) \\
\sqcap &: L \times L \mapsto L \\
(\tau_1, \theta_1, I_1) \sqcap (\tau_2, \theta_2, I_2) &\stackrel{def}{=} (\theta(\tau_1), \theta, I_1 \cup I_2) \\
&\text{con } \theta = \text{unify}(\tau_1 = \tau_2, \theta_1, \theta_2) \\
\nabla_0 &: L \mapsto L \\
\nabla_0(\tau, \theta, \emptyset) &\stackrel{def}{=} (\tau, \theta, \emptyset) \\
\nabla_0(\tau, \theta, I) &\stackrel{def}{=} (\tau', \theta', I') \\
&\text{con } (\tau', \theta') = (\tau, \theta) \sqcap (\sqcap_{(\tau_0, \theta_0) \in I} (\tau_0, \theta_0)) \\
&e I' = \bigcup_{(\tau_0, \theta_0) \in I} (\tau_0, \theta_0) \sqcap (\tau', \theta') \\
\nabla &: L \times L \mapsto L \\
\nabla((\tau_1, \theta_1, I_1), (\tau_2, \theta_2, I_2)) &\stackrel{def}{=} \nabla_0((\tau_1, \theta_1, I_1) \sqcap (\tau_2, \theta_2, I_2))
\end{aligned}$$

Si può vedere che le regole 5.13 e 5.20 sono equivalenti; inoltre l'operatore ∇ è lo stesso usato in [24], quindi si tratta di un operatore di widening. Ciò mostra che il calcolo del tipo di una funzione ricorsiva è può essere visto come l'uso di un operatore di allargamento.

Esempio 5.8.2 (seguito dell'esempio 5.8.1). *Applicando la regola (5.20) si ha (si veda quanto detto in precedenza su I)*

$$H[X_6 \leftarrow ('a_0, \varepsilon, \emptyset)] \vdash e : 'a_{13} \Rightarrow ('a_1 \rightarrow \text{int}, \theta', \{\text{int} \rightarrow 'a_{11}, \theta_9\})$$

A questo punto applicando a $('a_0, \varepsilon, \emptyset)$ e $('a_1 \rightarrow \text{int}, \theta', \{\text{int} \rightarrow 'a_{11}, \theta_9\})$ l'operatore ∇ si ottiene

$$\begin{aligned}
&('a_0, \varepsilon, \emptyset) \nabla ('a_1 \rightarrow \text{int}, \theta', \{\text{int} \rightarrow 'a_{11}, \theta_9\}) \\
&= \nabla_0(('a_0, \varepsilon, \emptyset) \sqcap ('a_1 \rightarrow \text{int}, \theta', \{\text{int} \rightarrow 'a_{11}, \theta_9\})) \\
&= \nabla_0('a_1 \rightarrow \text{int}, \text{unify}(\{ 'a_0 = 'a_1 \rightarrow \text{int}, \theta' \}), \{\text{int} \rightarrow 'a_{11}, \theta_9\}) \\
&= (\text{int} \rightarrow \text{int}, \text{unify}(\{ 'a_1 = 'a_{11} = \text{int}, 'a_0 = 'a_1 \rightarrow \text{int}, \theta', \theta_9 \}), \{\text{int} \rightarrow \text{int}, \theta''\})
\end{aligned}$$

È facile vedere che la sostituzione ottenuta con quest'ultimo passaggio di unificazione, con l'aggiunta del vincolo $'a_{14} = \text{int} \rightarrow \text{int}$ creato in accordo con (5.20) unificando τ'' e τ , è esattamente la sostituzione finale ottenuta con la (5.13). Quindi i due calcoli risultano essere equivalenti.

5.9 Semantiche di verifica

All'interno dell'insieme Φ_{sp} si trovano le funzioni che attribuiscono, con varie modalità, dei tipi ground ad ogni sottoespressione. Un esempio è il seguente:

$$\lambda f. \lambda g. (gf)1 \rightsquigarrow$$

$$\begin{aligned} & (\lambda f : int \rightarrow int. \\ & \quad (\lambda g : (int \rightarrow int) \rightarrow (int \rightarrow int). \\ & \quad \quad (((g : (int \rightarrow int) \rightarrow (int \rightarrow int))(f : int \rightarrow int)) : int \rightarrow int)(1 : int)) : int) \\ & \quad : (int \rightarrow int) \rightarrow (int \rightarrow int) \rightarrow int \\ & : (int \rightarrow int) \rightarrow ((int \rightarrow int) \rightarrow (int \rightarrow int)) \rightarrow int \end{aligned}$$

I vincoli sui tipi sono già interamente specificati prima dell'analisi; a questa non resta che verificare la consistenza. Durante l'analisi verranno dedotti solo tipi ground, e i vincoli creati saranno solo quelli relativi agli identificatori liberi (perchè se l'ambiente di tipi contiene delle variabili queste dovranno essere vincolate nell'unificazione con il tipo ground specificato; ad esempio l'analisi di $(X : int + 4 : int) : int$ nell'ambiente H porterà al vincolo $'a = int$ se $H(X) = 'a$).

Definizione 5.12. *L'insieme delle funzioni associate alle semantiche di verifica è*

$$\Phi_{ver} \stackrel{def}{=} \{f \in \Phi_{sp} \mid \forall e \in \mathbf{E}. \text{gr}_V(f(e))\} \subseteq \Phi_{sp}$$

dove $\text{gr}_V \in \mathbf{E}_\tau \mapsto Bool$ è un predicato definito per induzione sulla struttura delle espressioni e significa che tutte le annotazioni di tipo aggiunte a e sono ground.

Definizione 5.13. *L'insieme delle semantiche di verifica di tipo è l'insieme delle semantiche a cui sono associate le funzioni di Φ_{ver} :*

$$\mathbf{S}_{ver} \subseteq \mathbf{S}_{sp}$$

5.10 Compatibilità

Il predicato comp formalizza una relazione di compatibilità tra una funzione $f \in \Phi_{sp}$ e un $H \in \mathbf{H}_V$ per cui è possibile partendo da H tipare un'espressione con tipo τ (secondo \mathbf{T}^C) ogni volta che la verifica del tipo di e secondo f restituisce il tipo τ .

Definizione 5.14.

$$\begin{aligned} & \text{comp} : (\Phi_{sp} \times \mathbf{H}_V) \mapsto Bool \\ & \text{comp}(f, H) \stackrel{def}{=} \forall e \in \mathbf{E}, \tau \in \mathbf{M}^H. \\ & \exists \theta'. (\tilde{H}, (\tau, \theta')) \in \mathbf{S}_0[f(e)] \Rightarrow \exists \theta''. (H, (\tau, \theta'')) \in \mathbf{S}_{inf}[e] \end{aligned} \quad (5.21)$$

Il significato di questo predicato è stabilire una relazione tra funzioni associate e ambienti per cui un certo ambiente H può tipare per inferenza almeno tutte le espressioni per cui le specifiche indicate da f permettono il tipaggio. Si nota che sulle espressioni chiuse l'implicazione è sempre vera (perchè l'analisi non dipende dall'ambiente).

Si specializza ora il predicato comp alle sole semantiche di verifica:

Definizione 5.15.

$$\begin{aligned} \text{comp} &: (\Phi_{ver} \times \mathbf{H}) \mapsto \text{Bool} \\ \text{comp}(f, H) &\stackrel{def}{=} \forall e \in \mathbf{E}, \tau \in \mathbf{M}^H, \theta \in \mathbf{C}^H. \\ &(\tilde{H}, (\tau, \theta)) \in \mathbf{S}_0[[f(e)]] \Rightarrow (H, \tau) \in \mathbf{T}^C[[e]] \end{aligned}$$

Proposizione 5.13. *Data una certa $f \in \Phi_{ver}$ esiste al più un H compatibile (cioè $\forall f, H_1, H_2. \text{comp}(f, H_1) \wedge \text{comp}(f, H_2) \Rightarrow H_1 = H_2$).*

Dimostrazione. Una data $f \in \Phi_{ver}$ deve poter tipare almeno gli identificatori $X \in \mathbf{X}$.

$$\begin{aligned} \text{comp}(f, H) \wedge (\forall X \in \mathbf{X}. f(X) = X : \tau_X) &\Rightarrow \{\text{def comp}\} \\ \forall X \in \mathbf{X}. (H, \tau_X) \in \mathbf{T}^C[[X]] &\Rightarrow \{\text{def } \mathbf{T}^C\} \\ \forall X \in \mathbf{X}. H(X) = \tau_X & \end{aligned}$$

Esiste un solo type environment che soddisfa questa condizione; inoltre non è detto che questo type environment soddisfi la condizione di compatibilità per tutte le altre espressioni. Un controesempio è il seguente:

$$\begin{aligned} f(X \ 1) &= ((X : \text{int} \rightarrow \text{int})(1 : \text{int})) : \text{int} \\ f(X \ 2) &= ((X : \text{int} \rightarrow \text{int} \rightarrow \text{int})(1 : \text{int})) : \text{int} \rightarrow \text{int} \end{aligned}$$

dove nessun H verifica

$$(H, \text{int}) \in \mathbf{T}^C[[X \ 1]] \wedge (H, \text{int} \rightarrow \text{int}) \in \mathbf{T}^C[[X \ 2]]$$

□

Definizione 5.16. *Le funzioni associate che hanno un type environment compatibile costituiscono l'insieme*

$$\Phi_c \stackrel{def}{=} \{f \in \Phi_{ver} \mid \exists H \in \mathbf{H}. \text{comp}(f, H)\} \subseteq \Phi_{ver}$$

Proposizione 5.14. *Sia $\Phi_{compose}$ l'insieme delle $f \in \Phi_{ver}$ composizionali. Allora*

$$\Phi_{compose} \subseteq \Phi_c$$

Dimostrazione. Sia $f \in \Phi_{compose}$; se esiste un $H \in \mathbf{H}$ compatibile con f allora sarà sicuramente quello che tipa gli identificatori in modo corrispondente alle annotazioni che f aggiunge agli identificatori (prop. 5.13). Inoltre se un'espressione annotata da f risulta tipabile significa che assegnando questi particolari tipi ground agli identificatori si può arrivare allo stesso tipo (formalmente: $(\tilde{H}, (\tau, \theta)) \in \mathbf{S}_0[[f(e)]] \Rightarrow \exists H. (H, \tau) \in \mathbf{T}^C[[e]]$). Poichè la composizionalità assicura che f specifica per gli identificatori sempre gli stessi tipi abbiamo che il type environment è lo stesso per tutte le espressioni, quindi $\text{comp}(f, H)$. \square

5.11 Connessione di Galois tra l'inferenza e una particolare semantica di verifica

Si è visto (sez. 5.6) che la semantica \mathbf{S}_{inf} è equivalente al sistema di tipi monomorfo à la Church-Curry (a sua volta equivalente al sistema à la Hindley \mathbf{T}^H).

In questa sezione si cercherà di stabilire il rapporto di approssimazione che esiste tra il concetto di inferenza di tipo e quello di verifica secondo una delle semantiche a cui sono associate le funzioni elementi di Φ_{ver} . È necessario distinguere due casi.

5.11.1 $f \in \Phi_c$

Sia $f \in \Phi_c$; chiamiamo H_f l'unico type environment compatibile con f . Si ha

$$\langle \mathbf{T}, \supseteq \rangle \xrightarrow[\alpha_f]{\gamma_f} \langle \wp(\mathbf{M}^H), \supseteq \rangle$$

$$\alpha_f(X) \stackrel{def}{=} \{\tau \mid (H_f, \tau) \in X\} \quad (5.22)$$

$$\gamma_f(Y) \stackrel{def}{=} \{(H_f, \tau) \mid \tau \in Y\} \quad (5.23)$$

Sostanzialmente l'astrazione si disinteressa di ciò che non riguarda l'unico ambiente compatibile con f . È facile verificare che si ha un'inserzione di Galois:

$$\forall Y . \alpha_f(\gamma_f(Y)) = Y \quad (5.24)$$

$$\forall X . \gamma_f(\alpha_f(X)) \subseteq X \quad (5.25)$$

Definizione 5.17. *La semantica associata a questa astrazione è*

$$\mathbf{T}_f[[e]] \stackrel{def}{=} \text{if } (\exists \theta. (\tilde{H}, (\tau, \theta)) \in \mathbf{S}_0[[f(e)]] \text{ then } \{\tau\} \text{ else } \emptyset$$

Con la seguente proposizione viene mostrato che \mathbf{T}_f è una semantica sound; infatti stabilendo che essa è un'astrazione di \mathbf{S}_{inf} , a sua volta astrazione esatta (l'astrazione è componibile, vedi prop. 4.6) della semantica sound \mathbf{T}^C , si mostra che essa astrae la semantica collecting (def. 4.10).

Proposizione 5.15 (soundness). $\forall e. \alpha_f(\mathbf{T}^C[[e]]) \supseteq \mathbf{T}_f[[e]]$

Dimostrazione.

$$\begin{aligned} \tau \in \mathbf{T}_f[[e]] &\Rightarrow \{\text{def. 5.17}\} \\ \mathbf{T}_f[[e]] = \{\tau\} &\Rightarrow \{\text{def. 5.17}\} \\ \exists \theta. (\tilde{H}, (\tau, \theta)) \in \mathbf{S}_0[[f(e)]] &\Rightarrow \{\text{def. 5.15}\} \\ (H_f, \tau) \in \mathbf{T}^C[[e]] &\Rightarrow \{(5.22)\} \\ \tau \in \alpha_f(\mathbf{T}^C[[e]]) & \end{aligned}$$

□

5.11.2 $f \in \Phi_{ver} \setminus \Phi_c$

In questo caso le cose diventano per forza banali:

$$\alpha_f(X) \stackrel{def}{=} \mathbf{M}^H \quad (5.26)$$

$$\gamma_f(Y) \stackrel{def}{=} \emptyset \quad (5.27)$$

Si ha una Galois connection e non una Galois insertion. La perdita di informazione è maggiore.

5.12 Connessione di Galois tra l'inferenza e le verifiche

Si cerca ora una connessione di Galois tra la “solita” semantica per l'inferenza di tipo monomorfa e un framework generale che in qualche modo comprenda tutte le semantiche di verifica.

$$\langle \mathbf{T}^C, \supseteq \rangle \stackrel{\gamma_f}{\underset{\alpha_f}{\rightleftarrows}} \langle (\Phi_{ver} \mapsto \wp(\mathbf{M}^H)), \leq_{\Phi} \rangle \quad (5.28)$$

con

$$\leq_{\Phi} \stackrel{def}{=} \{(\phi, \psi) \mid \forall f \in \Phi_{ver}. \phi(f) \supseteq \psi(f)\} \quad (5.29)$$

Le funzioni di astrazione e concretizzazione sono:

$$\alpha_f(X) \stackrel{def}{=} \lambda f. \alpha_f(X) \quad (5.30)$$

$$\gamma_f(\phi) \stackrel{def}{=} \bigcup_{f \in \Phi_{ver}} \gamma_f(\phi(f)) \quad (5.31)$$

dove α_f e γ_f sono definite rispettivamente in (5.22, 5.26) e (5.23, 5.27).

Sostanzialmente si tratta di un'astrazione (in senso funzionale) rispetto a f dei risultati precedentemente ottenuti con una sola semantica di verifica.

5.12.1 Connessione di Galois

Per dimostrare che si tratta di una connessione di Galois dobbiamo provare che:

Proposizione 5.16. $\forall X . \gamma_?(\alpha_?(X)) \subseteq X$

Dimostrazione.

$$\begin{aligned}
 (H, \tau) \in \gamma_?(\alpha_?(X)) &\Leftrightarrow \{(5.31)\} \\
 \exists f . (H, \tau) \in \gamma_f((\alpha_?(X))(f)) &\Leftrightarrow \{(5.30)\} \\
 \exists f . (H, \tau) \in \gamma_f(\alpha_f(X)) &\Rightarrow \{(5.25)\} \\
 \exists f . (H, \tau) \in X &\Leftrightarrow \{logica\} \\
 (H, \tau) \in X &
 \end{aligned}$$

□

Proposizione 5.17. $\forall \phi . \alpha_?(\gamma_?(\phi)) \leq_{\Phi} \phi$

Dimostrazione. Si vuole dimostrare che $\forall \phi . \forall f, \tau . \tau \in \phi(f) \Rightarrow \tau \in (\alpha_?(\gamma_?(\phi)))(f)$:

$$\begin{aligned}
 f \in \Phi_c : \quad \tau \in \phi(f) &\Leftrightarrow \{(5.23)\} \\
 (H_f, \tau) \in \gamma_f(\phi(f)) &\Rightarrow \{(5.31)\} \\
 (H_f, \tau) \in \gamma_?(\phi) &\Leftrightarrow \{(5.22)\} \\
 \tau \in \alpha_f(\gamma_?(\phi)) &\Leftrightarrow \{(5.30)\} \\
 \tau \in (\alpha_?(\gamma_?(\phi)))(f) &
 \end{aligned}$$

$$f \notin \Phi_c : \alpha_?(\gamma_?(\phi))(f) = \alpha_f(\gamma_?(\phi)) = \mathbf{M}^H \supseteq \phi(f)$$

da cui la tesi per definizione di \leq_{Φ} .

□

Proposizione 5.18. *La relazione espressa da (5.28) è una connessione di Galois.*

Dimostrazione. Segue banalmente dalle proposizioni 5.16 e 5.17.

□

5.12.2 Soundness

Vale quanto detto in precedenza per α_f e γ_f (vedi l'introduzione alla prop. 5.15).

Definizione 5.18.

$$\begin{aligned} \mathbf{T}_{\rightarrow}[[e]] &\stackrel{def}{=} \lambda f . \text{if } (\exists \theta. (\tilde{H}, (\tau, \theta)) \in \mathbf{S}_0[[f(e)]]) \text{ then } \{\tau\} \text{ else } \emptyset \\ &= \lambda f . \mathbf{T}_f[[e]] \end{aligned}$$

Proposizione 5.19 (soundness). $\forall e . \alpha_?(\mathbf{T}^C[[e]]) \leq_{\Phi} \mathbf{T}_{\rightarrow}[[e]]$

Dimostrazione. Si deve dimostrare che

$\forall e . \forall f, \tau. (\tau \in (\mathbf{T}_{\rightarrow}[[e]])(f) \implies \tau \in (\alpha_?(\mathbf{T}^C[[e]]))(f))$. Infatti

$$\begin{aligned} f \in \Phi_c : \quad & \tau \in (\mathbf{T}_{\rightarrow}[[e]])(f) && \Leftrightarrow \{def\ 5.18\} \\ & \mathbf{S}_0[[f(e)]] = (\tau, \varepsilon) && \Rightarrow \{def\ 5.15\} \\ & (H_f, \tau) \in \mathbf{T}^C[[e]] && \Leftrightarrow \{(5.22)\} \\ & \tau \in \alpha_f(\mathbf{T}^C[[e]]) && \Leftrightarrow \{(5.30)\} \\ & \tau \in (\alpha_?(\mathbf{T}^C[[e]]))(f) && \\ \\ f \notin \Phi_c : \quad & (\alpha_?(\mathbf{T}^C[[e]]))(f) && = \{(5.30)\} \\ & \alpha_f(\mathbf{T}^C[[e]]) && = \{(5.26)\} \\ & \mathbf{M}^H && \supseteq (\mathbf{T}_{\rightarrow}[[e]])(f) \end{aligned}$$

e la tesi segue dalla definizione di \leq_{Φ} . □

Capitolo 6

Conclusioni

In questa trattazione si è cercato di includere in un formalismo omogeneo diversi livelli di restrittività nell'analisi di tipo monomorfa sui linguaggi funzionali.

Un simile approccio all'analisi di tipo può essere pensato anche per sistemi di tipi più complessi, ad esempio sistemi polimorfi (con astrazione o ricorsione polimorfa). In questo caso si avrebbero semantiche generali $\mathbf{S}'_0[\bullet]$ ottenute da \mathbf{S}_0 modificando opportunamente le regole di inferenza.

Ci si aspetta che anche in questo caso aggiungere specifiche banali alle espressioni rimandi al caso dell'inferenza pura, cioè esattamente alla semantica da cui si parte per la generalizzazione (nel caso di \mathbf{S}_0 la semantica di partenza è \mathbf{T}^C).

Inoltre l'ambito di applicazione può essere esteso alla strictness analysis, ad altri linguaggi di programmazione funzionali (con meccanismi aggiuntivi come il `let`) con filosofie *eager* o *lazy*, e anche ad altri paradigmi di programmazione (imperativo, object-oriented ...).

Appendice A

Implementazione della semantica

La semantica generale S_0 è stata implementata nel linguaggio Haskell e testata con l'interprete Hugs98:

```
--      -- --      --      -----      -----  
||      || ||      || ||      || ||__ Hugs 98: Based on the Haskell 98 standard  
||__||      ||__||      ||__||      __|| Copyright (c) 1994-2001  
||---||              ---||              World Wide Web: http://haskell.org/hugs  
||      ||              Report bugs to: hugs-bugs@haskell.org  
||      || Version: Dec 2001      -----
```

Di seguito è riportato il codice sorgente.

A.1 Definizione degli insiemi di valori

```
module Domini where
```

```
-----
-- Insieme dei tipi ground
```

```
data M_h = Numero_g | Freccia_g M_h M_h
```

```
instance Show M_h where
```

```
    show (Numero_g) = "int"
```

```
    show (Freccia_g t1 t2) = "(" ++ (show t1) ++ "->" ++ (show t2) ++ ")"
```

```
-----
-- Insieme dei tipi con variabili
```

```
-- Variabile di tipo
```

```
type Var = Int
```

```
data M_hv = Numero | Variabile Var | Freccia M_hv M_hv
```

```
instance Show M_hv where
```

```
    show (Numero) = "int"
```

```
    show (Variabile v) = "a" ++ show v
```

```
    show (Freccia t1 t2) = "(" ++ (show t1) ++ "->" ++ (show t2) ++ ")"
```

```
instance Eq M_hv where
```

```
    Numero == Numero = True
```

```
    Variabile a == Variabile b = a == b
```

```
    Freccia t1 t2 == Freccia t3 t4 = (t1 == t3 && t2 == t4)
```

```
    _ == _ = False
```

```
-----
-- Insieme E delle espressioni
```

```
-- Identificatori
```

```
type Id = Int
```

```

data E = Int Int
      | Id Id
      | Plus E E
      | If E E E
      | Appl E E
      | Lambda Id E
      | Mu Id E

-----

-- Insieme delle espressioni annotate

-- Espressioni di base con annotazione di tipo
type Intero = (Int, M_hv)
type Identificatore = (Id, M_hv)

data E_tau = Aint Intero
          | Aid Identificatore
          | Aplus E_tau E_tau M_hv
          | Aif E_tau E_tau E_tau M_hv
          | Aappl E_tau E_tau M_hv
          | Alambda Identificatore E_tau M_hv
          | Amu Identificatore E_tau M_hv

instance Show E_tau where
  show (Aint(n,tau)) = show n ++ ":" ++ show tau
  show (Aid(i,tau)) = "x" ++ show i ++ ":" ++ show tau
  show (Aplus e1 e2 tau) = "(" ++ show e1 ++ " + " ++ show e2 ++ "):"
    ++ show tau
  show (Aif e1 e2 e3 tau) = "(if " ++ show e1 ++ " then " ++ show e2 ++
    " else " ++ show e3 ++ "):" ++ show tau
  show (Aappl e1 e2 tau) = "(" ++ show e1 ++ ")( " ++ show e2 ++ "):"
    ++ show tau
  show (Alambda(i,t) e tau) = "(\\x" ++ show i ++ ":" ++ show t ++ "."
    ++ show e ++ "):" ++ show tau
  show (Amu(i,t) e tau) = "(MU " ++ show i ++ ":" ++ show t ++ "."
    ++ show e ++ "):" ++ show tau

```

```

-----
-- Ambienti di tipo con variabili

type H_v = Id -> (M_hv, C_h)

-- Costruisce un ambiente da una lista di coppie
makenv :: [ (Id, (M_hv, C_h)) ] -> H_v
makenv l = \x -> trovaenv x l

trovaenv :: Id -> [ (Id, (M_hv, C_h)) ] -> (M_hv, C_h)
trovaenv x [] = (Variabile x, C [])
trovaenv x ((y, (t,c)):l) = if x == y then (t,c) else trovaenv x l

-- Aggiorna un ambiente con un nuovo assegnamento
aggiorna :: H_v -> (Id, (M_hv, C_h)) -> H_v
aggiorna h (x, (t, c)) = \y -> if x == y then (t,c) else h y

-----
-- Sostituzioni di tipo

data C_h = C [ (Var, M_hv) ] | Errore String

instance Eq C_h where
  (C []) == (C []) = True
  (Errore s1) == (Errore s2) = True
  (C (c1:cs1)) == (C cs2) = (C cs1) == (estrai_termine c1 cs2)
  _ == _ = False

estrai_termine :: (Var, M_hv) -> [ (Var,M_hv) ] -> C_h
estrai_termine _ [] = Errore "termine non trovato"
estrai_termine c1 (c:cs) = if (c == c1) then C cs
  else let (C cs1) = estrai_termine c1 cs in
  C (c:cs1)

instance Show C_h where
show (C []) = "end"
show (C ((v,t):cs)) = "a" ++ (show v) ++ "=>" ++ (show t) ++ " | "
  ++ (show (C cs))
show (Errore s) = s
--

```

A.2 Algoritmo di unificazione

```

module Unificazione where

import Domini

-----

-- Funzione principale

unifica :: [ (M_hv, M_hv) ] -> C_h
unifica [] = (C [])
unifica ((Numero,Numero):vs) = unifica vs
unifica ((Variabile v,t):vs)
  | t == Variabile v = unifica vs
  | occorre v t = Errore "unificazione impossibile"
  | otherwise = componi (C [ (v,t) ])
  (unifica (sostituisci (v,t) vs))
unifica ((t,Variabile v):vs) = unifica ((Variabile v,t):vs)
unifica ((Freccia t1 t2,Freccia t3 t4):vs) =
  unifica ([(t1,t3),(t2,t4)] ++ vs)
unifica _ = Errore "unificazione impossibile"

sostituisci :: (Var,M_hv) -> [ (M_hv,M_hv) ] -> [ (M_hv,M_hv) ]
sostituisci (v,t) cs =
  map ( \ (c1,c2) -> (applica (C [ (v,t) ]) c1,
                           applica (C [ (v,t) ]) c2)) cs

-- Occur check
occorre :: Var -> M_hv -> Bool
occorre v (Numero) = False
occorre v (Variabile w) = v == w
occorre v (Freccia t1 t2) = (occorre v t1) || (occorre v t2)

```

```

-- Composizione di sostituzioni
componi :: C_h -> C_h -> C_h
componi (Errore s) _ = Errore s
componi _ (Errore s) = Errore s
componi (C c1) (C c2) =
  (C ((map ( \ (x,y) -> (x,(applica (C c2) y))) c1)
    ++ (let (C temp) = (ripulisci (C c2) (C c1))
        in temp)))

-- Funzione ausiliaria della composizione
ripulisci :: C_h -> C_h -> C_h
ripulisci (Errore s) _ = Errore s
ripulisci _ (Errore s) = Errore s
ripulisci (C []) _ = C []
ripulisci (C ((v,t):cs)) c1 =
  if (t == (Variabile v) || (cerca v c1))
  then ripulisci (C cs) c1
  else let (C temp) = (ripulisci (C cs) c1) in
        C ((v,t):temp)

-- Trovano il tipo assegnato ad una variabile
cerca :: Var -> C_h -> Bool
cerca _ (C []) = False
cerca v1 (C ((v2, m2):cs)) = if v1 == v2 then True
  else cerca v1 (C cs)

trova :: Var -> C_h -> M_hv
trova v1 (C ((v2, m2):cs)) = if v1==v2 then m2
  else trova v1 (C cs)

-- Applicazione di una sostituzione ad un tipo

applica :: C_h -> M_hv -> M_hv
applica (Errore s) _ = error "applicazione impossibile"
applica _ Numero = Numero
applica c (Variabile v) = if (cerca v c)
  then trova v c else (Variabile v)
applica c (Freccia t1 t2) = Freccia (applica c t1) (applica c t2)
--

```

A.3 Semantica generale S_0

```

module Semantica where

import Domini
import Costanti
import Unificazione
import Funzioni_associate

-----
-- Compone un insieme di sostituzioni in una lista di coppie di tipi

inlista :: [ C_h ] -> [ (M_hv, M_hv) ]
inlista [] = []
inlista ((C c):cs) = (map ( \ (a,b) -> (Variabile a, b)) c)
                    ++ inlista cs

-----
-- Tipo monadico utilizzato dalla semantica

data N a = Trace ([String] -> (a,[String]))

instance Monad N where
    return a = Trace (\s -> (a, s))
    m >>= n = Trace (\x -> let (Trace f) = m in
                            let (a, y) = f x in
                            let (Trace g) = n a in g y)

-----
-- Gestione della traccia

costruiscitraccia :: H_v -> (E_tau) -> (M_hv,C_h) -> N (M_hv,C_h)
costruiscitraccia h e (tau,theta) =
    Trace (\s -> ((tau,theta),
                  s ++ ["|- " ++ show e ++ " ==> ( "
                      ++ show tau ++ " , " ++ show theta ++ " )" ]))

```

```

scrivitraccia :: Num a => [String] -> IO a
scrivitraccia [] = return 0
scrivitraccia (s:ss) = do putStr s
                          putChar '\n'
                          putChar '\n'
                          scrivitraccia ss

```

```

-- Semantica vera e propria

```

```

s_0_aux :: E_tau -> H_v -> N (M_hv, C_h)

```

```

s_0_aux (Aint (n, tau)) h =
  let theta = unifica [(tau, Numero)] in
      let ris = (applica theta tau, theta) in
          costruiscitraccia h (Aint (n, tau)) ris

```

```

s_0_aux (Aid (i, tau)) h =
  let (tau1, theta1) = h i in
      let theta = unifica ((tau1, tau):(inlista [theta1])) in
          costruiscitraccia h (Aid (i, tau))(applica theta tau, theta)

```

```

s_0_aux (Aplus e1 e2 tau) h =
  s_0_aux e1 h >>= \(tau1, theta1) -> s_0_aux e2 h >>= \(tau2, theta2)
  -> let theta = unifica([(tau1, Numero), (tau2, Numero), (tau, Numero)]
                        ++ inlista [theta1, theta2]) in
      costruiscitraccia h (Aplus e1 e2 tau)
      (applica theta tau, theta)

```

```

s_0_aux (Aif e1 e2 e3 tau) h =
  s_0_aux e1 h >>= \(tau1, theta1) -> s_0_aux e2 h >>=
  \(tau2, theta2) -> s_0_aux e3 h >>= \(tau3, theta3) ->
  let theta = unifica([(tau1, Numero), (tau2, tau), (tau3, tau)]
                    ++ inlista [theta1, theta2, theta3]) in
      costruiscitraccia h (Aif e1 e2 e3 tau)
      (applica theta tau, theta)

```

```

s_0_aux (Aappl e1 e2 tau) h =
  s_0_aux e1 h >>= \(tau1,theta1) -> s_0_aux e2 h >>= \(tau2,theta2)
    -> let theta = unifica ((tau1, Freccia tau2 tau)
      :(inlista [theta1,theta2])) in
      costruiscitraccia h (Aappl e1 e2 tau)
        (applica theta tau, theta)

s_0_aux (Alambda (x,tau_x) e tau) h =
  s_0_aux e (aggiorna h (x,(tau_x,C []))) >>= \(tau_e,theta_e) ->
    let theta = unifica ((tau, Freccia tau_x tau_e)
      :(inlista [theta_e])) in
      costruiscitraccia h (Alambda (x,tau_x) e tau)
        (applica theta tau, theta)

s_0_aux (Amu (f,tau_f) e tau) h =
  let passo_zero = (tau_f, C []) in
    ricorsione (Amu (f,tau_f) e tau) h passo_zero >>=
      \(tau_finale,theta_finale) ->
        costruiscitraccia h (Amu (f,tau_f) e tau)
          (tau_finale,theta_finale)

ricorsione :: E_tau -> H_v -> (M_hv,C_h) -> N (M_hv,C_h)
ricorsione e h (tau1,theta1) =
  (passo_induttivo e h (tau1,theta1)) >>= \(tau2,theta2) ->
    if tau1 == tau2 then return (tau2,theta2)
      else (ricorsione e h (tau2,theta2))

passo_induttivo :: E_tau -> H_v -> (M_hv,C_h) -> N (M_hv,C_h)
passo_induttivo (Amu (f,tau_f) e tau) h (tau1,theta1) =
  s_0_aux e (aggiorna h (f,(tau1,theta1))) >>= \(tau2,theta2) ->
    let theta = unifica
      ([(tau_f,tau2),(tau,tau2)]
      ++ inlista [theta2]) in
      return (applica theta tau, theta)

```

```

-----
-- Semantica con traccia

s_0 :: Num a => E_tau -> H_v -> IO a
s_0 e h = let (Trace temp) = s_0_aux e h in
           let ris = temp [] in scrivitraccia (snd ris)

-----
-- Semantica dell'inferenza

s_inf :: Num a => E -> H_v -> IO a
s_inf e h = let (D temp) = f_s_inf e in
            let e_tau = ( \ (x,y)->x) (temp 0) in s_0 e_tau h

-----
-- Stesse semantiche ma senza il punto fisso

s_0_aux_nofix :: E_tau -> H_v -> N (M_hv,C_h)
s_0_aux_nofix (Amu (f,tau_f) e tau) h =
  s_0_aux e (aggiorna h (f,(tau_f,C []))) >>= \ (tau1,theta1) ->
    let theta = unifica ([(tau_f,tau),(tau1,tau)]
                        ++ inlista [theta1]) in
      costruiscitraccia h (Amu (f,tau_f) e tau)
                        (applica theta tau, theta)
s_0_aux_nofix e h = s_0_aux e h

s_0_nofix :: Num a => E_tau -> H_v -> IO a
s_0_nofix e h = let (Trace temp) = s_0_aux_nofix e h in
                let ris = temp [] in scrivitraccia (snd ris)

s_inf_nofix :: Num a => E -> H_v -> IO a
s_inf_nofix e h = let (D temp) = f_s_inf e in
                  let e_tau = ( \ (x,y)->x) (temp 0) in
                    s_0_nofix e_tau h
--

```

A.4 Funzione associata all'inferenza pura f_{Sinf}

```

module Funzioni_associate where

import Domini
-----
-- Tipo monadico usato dalla funzione di annotazione
data M a = D (Var -> (a, Var))

instance Monad M where
    return a = D (\var -> (a, var))
    m >>= n = D (\x -> let (D f) = m in
                        let (a, y) = f x in
                        let (D g) = n a in g y)
-----
-- Funzione di annotazione

f_s_inf :: E -> M E_tau
f_s_inf (Int n) = D (\var -> (Aint (n, Variabile var), var + 1))
f_s_inf (Id i) = D (\var -> (Aid (i, Variabile var), var + 1))
f_s_inf (Plus e1 e2) =
    f_s_inf e1 >>= \e1_tau -> f_s_inf e2 >>= \e2_tau ->
    D (\var -> ((Aplus e1_tau e2_tau (Variabile var)), var + 1))
f_s_inf (If e1 e2 e3) =
    f_s_inf e1 >>= \e1_tau -> f_s_inf e2 >>= \e2_tau ->
    f_s_inf e3 >>= \e3_tau -> D (\var -> ((Aif e1_tau e2_tau e3_tau
                                           (Variabile var)), var+1))
f_s_inf (Appl e1 e2) =
    f_s_inf e1 >>= \e1_tau -> f_s_inf e2 >>= \e2_tau ->
    D (\var -> ((Aappl e1_tau e2_tau (Variabile var)), var + 1))
f_s_inf (Lambda i e) =
    f_s_inf (Id i) >>= \e1_tau ->
    f_s_inf e >>= \e_tau ->
    D (\var -> ((Alambda i_tau e_tau (Variabile var)), var + 1))
f_s_inf (Mu i e) =
    f_s_inf (Id i) >>= \e1_tau -> f_s_inf e >>= \e_tau ->
    D (\var -> ((Amu i_tau e_tau (Variabile var)), var + 1))

```


Ringraziamenti

Cantate al Signore un canto di grazie, intonate sulla cetra inni al nostro Dio.

Salmo 147

A Dio, in primis. Per tutto.

Ai familiari: è lì che tutta la mia vita ha trovato ogni tipo di sostegno e comprensione. Grazie per aver sopportato i miei difetti.

Agli amici di *Borgo*, immancabile punto di approdo tra le tante incertezze; tra i molti meriti che hanno c'è sicuramente quello di avermi riportato molte volte con i piedi per terra.

A tutti quanti hanno avuto un ruolo nel mio cammino spirituale e personale, la parrocchia, l'Azione Cattolica e le persone che vi ho incontrato; anche se il *prodotto finito* non è granché, senza di voi sarebbe stato molto peggiore.

Alle persone incontrate nell'ambiente universitario: la loro cultura e preparazione ha permesso che questa esperienza fosse per me molto proficua. Un doveroso e particolare ringraziamento va a Roberto Zunino per l'incessante e paziente opera di educazione informatica svolta nei miei confronti; il fatto che i giorni delle nostre lauree coincidano non rende giustizia alla sua bravura.

Infine, al Prof. Giorgio Levi per la pazienza e la disponibilità con cui mi ha aiutato in questo lavoro.

DZ

Bibliografia

- [1] H. Barendregt. Lambda calculi with types. Vol. 2 di *Handbook of Logic in Computer Science*, pagg. 117-309. Clarendon Press, 1992.
- [2] L. Cardelli. Type systems. *ACM Comput. Surv.*, 28(1), pagg. 262-267, 1996.
- [3] P. Cousot. Semantic foundations of program analysis. In S.S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 10, pagg. 303-342. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, USA, 1981.
- [4] P. Cousot. Types as abstract interpretations. In *Conference Record of the Twentyfourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pagg. 316-331, Paris, France, January 1997. ACM Press, New York, NY.
- [5] P. Cousot. Design of Semantics by Abstract Interpretation, Invited Address. In *Mathematical Foundations of Programming Semantics, Thirteenth Annual Conference (MFPS XIII)*, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA, March 23-26, 1997.
- [6] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. In *Electronic Notes in Theoretical Computer Science*, 6 (1997), 25 pagine, URL: <http://www.elsevier.nl/locate/entcs/volume6.html>.
- [7] P. Cousot. Abstract Interpretation Based Formal Methods and Future Challenges. In *Informatics — 10 Years Back, 10 Years Ahead*, R. Wilhelm (Ed.). Lecture Notes in Computer Science Vol. 2000, pagg. 138-156. Springer-Verlag, 2001.
- [8] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pagg. 106-130. Dunod, Paris, France, 1976.

- [9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pagg. 238-252, Los Angeles, California, 1977. ACM Press, New York, NY, USA.
- [10] P. Cousot and R. Cousot. Static determination of dynamic properties of recursive procedures. In *IFIP Conf. on Formal Description of Programming Concepts, St-Andrews, N.B., CA*, E.J. Neuhold (Ed.), pagg. 237-277, St-Andrews, N.B., Canada, 1977. North-Holland Publishing Company (1978).
- [11] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pagg. 269-282, San Antonio, Texas, 1979. ACM Press, New York, NY, U.S.A.
- [12] P. Cousot and R. Cousot. Relational abstract interpretation of higher-order functional programs (abstract). *Actes JTASPEFL'91, Bordeaux, 9-11 octobre 1991*, in BIGRE 74, pagg. 33-36, IRISA, Rennes, France, 1991.
- [13] P. Cousot and R. Cousot. Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation (abstract). *Actes JTA-SPEFL'91, Bordeaux, 9-11 octobre 1991*, in BIGRE 74, pagg. 107-110, IRISA, Rennes, France, 1991.
- [14] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2-3), pagg. 103-179, 1992.
- [15] P. Cousot and R. Cousot. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation, invited paper. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming, Proceedings of the Fourth International Symposium, PLILP'92*, Leuven, Belgium, 13-17 August 1992, LNCS 631, pagg. 269-295. Springer-Verlag, Berlin, Germany, 1992.
- [16] P. Cousot and R. Cousot. Higher-Order Abstract Interpretation (and Application to Compartment Analysis Generalizing Strictness, Termination, Projection and PER Analysis of Functional Languages), invited paper. In *Proceedings of the 1994 International Conference on Computer Languages*, Toulouse, France, pagg. 95-112. IEEE Computer Society Press, Los Alamitos, California, U.S.A., May 16-19, 1994.
- [17] L. Damas and R. Milner. Principal type-schemes for functional programs. *9th ACM POPL*, pagg. 207-212, 1982.

- [18] C. Gunter. The semantics of types in programming languages. Vol. 3 di *Handbook of Logic in Computer Science*, pagg. 395-475. Clarendon Press, 1994.
- [19] C. Gunter and D. Scott. Semantic Domains. Vol. B of *Handbook of Theoretical Computer Science*, pagg. 633-674. Elsevier, 1990.
- [20] R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. Amer. Math. Soc.*, pagg. 633-674. Elsevier, 1969.
- [21] X. Leroy. Notes du Cours de DEA "Typage et programmation". Materiale del corso di DEA coabilitato dalle Università Paris VI, Paris VII, Paris Sud (XI), Ecole Normale Supérieure de Paris (Ulm), Ecole Normale Supérieure de Cachan , Ecole Polytechnique, CNAM, in convenzione con INRIA, 2001. URL: <http://pauillac.inria.fr/~xleroy/dea/>.
- [22] R. Milner. A theory of polymorphism in programming. *J. Comput. Sys. Sci.*, 17(3), pagg. 348-375, 1978.
- [23] B. Monsuez. Polymorphic typing by abstract interpretation. LNCS 652, pagg. 127-138. Springer, 1992.
- [24] B. Monsuez. Polymorphic types and widening operators. LNCS 724, pagg. 267-281. Springer, 1993.
- [25] B. Monsuez. System F and abstract interpretation. LNCS 983, pagg. 279-295. Springer, 1995.