Note per il Corso di

Informatica Teorica

Elena Zucca

26 Settembre 2006

Indice

Introduzione

Obiettivo del corso. In questo corso studieremo i fondamenti dei linguaggi di programmazione (assumiamo come prerequisiti i contenuti del corso di Linguaggi di Programmazione, L. P. nel seguito). In particolare, studieremo i seguenti *elementi* di un linguaggio di programmazione (si veda l'ultima parte dell'Introduzione delle dispense di L. P.):

- *sintassi* (abbiamo già visto in L. P. come descrivere formalmente un linguaggio attraverso una grammatica libera da contesto, faremo un ripasso);

- *semantica statica* (ossia, la descrizione dei *vincoli contestuali* che caratterizzano i programmi staticamente corretti); si usa anche il termine *sistema di tipi* in quanto i vincoli contestuali tipicamente sono relativi a requisiti di correttezza di tipo;
- *semantica dinamica* (o semplicemente *semantica* (ossia, la descrizione del significato/comportamento dei programmi).

Inoltre, vedremo alcuni esempi di formalismi per esprimere e provare la *correttezza* di un programma rispetto a una specifica. Questi formalismi corrispondono a una visione più astratta di un programma in cui non ci interessa la sua semantica (comportamento) in dettaglio ma ci basta sapere che tale comportamento soddisfa certi requisiti.

Stili di semantica dinamica. Esistono diversi tipi di formalismi per descrivere la semantica (significato, comportamento) dei programmi di un linguaggio. Per esempio:

- la semantica *denotazionale:* associa a ogni programma (espressione del linguaggio) un valore; per esempio il programma che prende in input un numero reale e un intero e restituisce il primo input elevato al secondo, in qualunque modo sia calcolato il risultato è rappresentato dalla funzione $f(x, n) = x^n$;
- la semantica *operazionale small-step* descrive il comportamento dei programmi attraverso *regole di riduzione*, come per esempio si fa nella matematica della scuola secondaria quando si semplificano le espressioni aritmetiche letterali.

In questo corso siamo interessati soprattutto a descrizioni formali della semantica in stile small-step, perché, come vedremo, queste descrizioni si prestano meglio a esprimere e provare la *soundness* di un sistema di tipi, ossia, la proprietà che programmi staticamente corretti non possono dar luogo a (una certa classe di) errori a run-time; questo utilizzo dei tipi per prevenire errori a run-time è infatti uno dei temi principali del corso. Illustreremo tuttavia sommariamente anche lo stile denotazionale sia per mostrare un approccio alternativo molto importante culturalmente, sia perché la semantica denotazionale di un linguaggio imperativo serve come base per il successivo studio della correttezza di un programma rispetto a una specifica.
In particolare vedremo la semantica formale (oltre che il sistema di tipi) dei seguenti linguaggi:

- semplice linguaggio di espressioni su cui illustreremo i due stili;
- nucleo di linguaggio funzionale, arricchito poi con altri costrutti tra cui costrutti imperativi ed eccezioni (in stile operazionale small-step, a partire dal *λ-calcolo*); per queste prime due parti il testo per approfondimenti è [1];
- nucleo di linguaggio imperativo tipo C (in stile denotazionale);
- nucleo di linguaggio object-oriented tipo Java (anche per questa parte il testo per approfondimenti è [1]);

Correttezza. Relativamente alla correttezza dei programmi rispetto a una specifica data, vedremo due esempi:

- correttezza di definizioni di funzioni ricorsive;
- correttezza di programmi imperativi espressa tramite *asserzioni alla Hoare*.

1 Definizioni induttive e prove per induzione

In matematica e informatica si usano molto spesso definizioni ricorsive, cioè definizioni in cui l'espressione che definisce l'oggetto fa riferimento all'oggetto stesso. Si considerino i due seguenti esempi.

Esempio 1.1 L'insieme dei numeri pari è definito nel modo seguente:

- 0 è un numero pari,
- se n è un numero pari, allora $n + 2$ è un numero pari.

Esempio 1.2 Si consideri la seguente definizione ricorsiva di funzione `f: int * int -> int` (scritta con la sintassi Caml).

```
let rec f = function
  (x,y) -> if x = y then 0 else f(x-1,y)
```

Qual è il significato di una definizione come le precedenti? Dato che un oggetto viene definito in termini di se stesso non è chiaro a priori se abbiamo effettivamente dato una "buona definizione" o no.
Un tipo di definizioni ricorsive alle quali è possibile attribuire in modo semplice un significato preciso sono le definizioni cosiddette *induttive*, illustrate nel seguito. L'idea base è quella di definire un insieme come il più piccolo tra tutti gli insiemi X che verificano delle condizioni di "chiusura" del tipo "se certi elementi appartengono a X, allora anche un certo altro elemento deve appartenere a X".

1.1 Sistemi induttivi e definizioni induttive

Sia nel seguito U un insieme fissato detto *universo*.

Def. 1.3 [Regole e sistemi induttivi] Una *regola* è una coppia $<Pr, c>$, dove $Pr \subseteq U$ e $c \in U$; si dice che Pr è l'insieme delle *premesse* e c la *conseguenza* della regola.
Un insieme $X \subseteq U$ è *chiuso* rispetto a una regola $<Pr, c>$ sse $Pr \subseteq X$ implica $c \in X$; se R è un insieme di regole, X si dice R*-chiuso*, o chiuso rispetto a R, sse è chiuso rispetto a ogni regola di R.
Un *sistema induttivo* è un insieme di regole. Se R è un sistema induttivo, *l'insieme $I(R)$ definito (induttivamente) da R* è l'intersezione di tutti gli insiemi R-chiusi (si vede facilmente che è a sua volta un insieme R-chiuso, quindi si può dire equivalentemente *il più piccolo insieme R-chiuso*).

È facile vedere che U è R-chiuso e quindi la famiglia degli insiemi R-chiusi non è vuota, perciò la precedente è una buona definizione. Si noti inoltre che, dato un sistema induttivo R, è sempre possibile considerare come universo l'insieme dagli elementi delle coppie di R.[1] Quindi in realtà è possibile evitare di fissare un universo a priori.
Una *definizione induttiva* è una descrizione finita, data utilizzando un qualche metalinguaggio (per esempio l'usuale metalinguaggio matematico), di un sistema induttivo R, e quindi dell'insieme $I(R)$ da esso definito.
In genere una definizione induttiva consiste di un insieme di *metaregole*, cioè terne della forma $(pre, ce, cond)$, dove pre, ce, e $cond$ sono espressioni del metalinguaggio rappresentanti rispettivamente un sottoinsieme dell'universo U, un elemento di U e un valore booleano. Ogni metaregola "rappresenta" un insieme che può essere infinito di regole, tutte quelle ottenute istanziando le metavariabili che compaiono in pre e ce con elementi dell'insieme universo in un modo che verifica le condizioni in $cond$.
In generale si dice che le metaregole con un insieme vuoto di premesse costituiscono la *base* mentre le altre regole costituiscono il *passo induttivo* della definizione induttiva.
Per esempio, le due definizioni date sopra possono essere viste come definizioni induttive. Questo si può vedere più chiaramente riscrivendole nel modo seguente.

> L'insieme dei numeri pari è il più piccolo insieme tale che (oppure: è l'insieme definito induttivamente da):
>
> - 0 è un numero pari,
> - se n è un numero pari, allora $n + 2$ è un numero pari.

Si usa anche scrivere la definizione nello stile a *regole di inferenza*, in cui le premesse vengono scritte sopra la linea di frazione, la conseguenza sotto e le eventuali condizioni a lato:

$$\frac{}{0} \qquad \frac{n}{n+2}$$

In questo caso, l'insieme universo è $\mathbb{N}$, e le regole del sistema induttivo sono $(\emptyset, 0) \cup \{(n, n+2) \mid n \in \mathbb{N}\}$.

> La funzione f è la più piccola (ricordiamo che una funzione è un insieme di coppie, vedi Def.A.1) funzione da interi a interi tale che (oppure: è la funzione da interi a interi definita induttivamente da):
>
> - $f(x, y) = 0$ (cioè $((x, y), 0) \in f$), se $x = y$,
> - $f(x, y) = f(x - 1, y)$ (cioè se $((x - 1, y), r) \in f$ allora $((x, y), r) \in f$), se $x \neq y$.

Nella forma a regole di inferenza:

$$\frac{}{((x,y),0)} \quad x = y \qquad \frac{((x-1,y),r)}{((x,y),r)} \quad x \neq y$$

In questo caso, l'insieme universo è $\mathbb{Z} \times \mathbb{Z}$, e le regole del sistema induttivo sono

$$\{(\emptyset, ((x, y), 0)) \mid x, y \in \mathbb{Z}, x = y\} \cup \{(\{((x - 1, y), r)\}, ((x, y), r)) \mid x, y \in \mathbb{Z}, x \neq y\}.$$

Nell'esempio, f risulta essere la funzione che vale 0 sulle coppie $(x, y) \in \mathbb{Z} \times \mathbb{Z}$ per cui $x \geq y$, è indefinita altrimenti. Per provare formalmente tale fatto si procede secondo lo schema seguente.
Sia $\bar{f} = \{((x, y), 0) \mid x \geq y\}$. La tesi è $f = \bar{f}$.

[1] Si potrebbero anche considerare solo gli elementi conseguenza in quanto le regole con premesse che non appaiono mai come conseguenza di un'altra regola sono "inutili".

Prova di $f \subseteq \bar{f}$ Poiché f è per definizione il più piccolo insieme R_f-chiuso, dove R_f è il sistema induttivo descritto dalle due metaregole sopra, basta far vedere che $\bar{f}$ è un insieme R_f-chiuso, cioè che è chiuso rispetto a ogni possibile istanziazione delle due metaregole. Infatti:

- $((x, x), 0) \in \bar{f}$, per ogni $x \in \mathbb{Z}$;
- se $((x-1, y), r) \in \bar{f}$, significa che $x - 1 \geq y$ e $r = 0$; ma allora $((x, y), r) \in \bar{f}$.

Prova di $\bar{f} \subseteq f$ Occorre far vedere che ogni coppia $((x, y), 0)$ in $\bar{f}$ è un elemento di f, cioè può essere ottenuta applicando le regole in R_f. Non c'è un modo "canonico" di procedere per provare questa inclusione. Si può ragionare, in questo esempio, per induzione aritmetica (vedi dopo) sulla differenza tra x e y.

Base. Ogni coppia del tipo $((x, x), 0)$ può essere ottenuta da un'istanziazione della prima metaregola.

Passo induttivo. Assumiamo per ipotesi che ogni coppia del tipo $((x, y), 0)$ con $x - y = k$, $k \in \mathbb{N}$, sia un elemento di f. Allora ogni coppia del tipo $((x, y), 0)$ con $x - y = k + 1$ può essere ottenuta dalla seguente istanziazione della seconda metaregola:

$$\frac{((x-1, y), 0)}{((x, y), 0)}$$

(infatti si ha $x \neq y$ e $(x - 1) - y = k$).

Si noti che nel caso $f, \bar{f}$ siano funzioni e $\bar{f}$ sia *totale* lo schema di prova precedente si limita alla seconda parte. Infatti, essendo $f, \bar{f}$ funzioni (cioè insiemi di coppie che soddisfano la proprietà di univocità, vedi Def.A.1), e avendo provato $\bar{f} \subseteq f$, f non potrebbe contenere altre coppie, quindi $f = \bar{f}$. In altri termini in questo caso è sufficiente far vedere che f e $\bar{f}$ coincidono per ogni elemento del dominio.
Quando si richiede di *definire induttivamente* una funzione il dominio della funzione stessa deve essere a sua volta definito induttivamente e le (meta)regole del sistema che definisce il dominio costituiscono una guida per la scelta delle (meta)regole per definire la funzione (vedi esempi nel seguito).

1.2 Principio di induzione (generale)

Prop. 1.4 [Principio di induzione] Sia R un sistema induttivo, U un insieme tale che $I(R) \subseteq U$, e P un predicato su U, cioè una funzione $P\colon U \to \{T, F\}$.
Assumiamo che per ogni $<Pr, c> \in R$

$$(\star) \qquad (P(d) = T \text{ per ogni } d \in Pr) \text{ implica } P(c) = T.$$

Allora $P(d) = T$ per ogni $d \in I(R)$.

Dimostrazione. Poniamo $C = \{d | P(d) = T\}$ e osserviamo che la condizione $(\star)$ può essere scritta nella forma equivalente:

$$Pr \subseteq C \text{ implica } c \in C.$$

Allora è chiaro che C risulta R-chiuso e dunque $I(R) \subseteq C$, da cui si ha che $P(d) = T$ per ogni $d \in I(R)$. □
Si noti che nel caso $Pr = \emptyset$ la condizione $(\star)$ equivale a $P(c) = T$ (dal momento che la premessa dell'implicazione è verificata) e dunque $(\star)$ implica che P sia vera su tutti gli elementi della base. Ciò suggerisce una formulazione equivalente ma lievemente diversa di $(\star)$:

$(\star\star)$ (Base)
$P(c) = T$ per ogni c t.c. $<\emptyset, c> \in R$
(Passo induttivo)
$(P(d) = T$ per ogni $d \in Pr)$ implica $P(c) = T$ per ogni $<Pr, c> \in R, Pr \neq \emptyset$

Un caso particolare del principio di induzione dato qui in forma generale è il principio di induzione aritmetica, nelle due varianti date sotto.

Prop. 1.5 [Principio di induzione aritmetica] Sia P un predicato sui numeri naturali t.c.

1. $P(0) = T$;

2. per ogni $n \in \mathbb{N}$, $P(n) = T$ implica $P(n+1) = T$.

Allora $P(n) = T$ per ogni $n \in \mathbb{N}$.

Dimostrazione. $\mathbb{N}$ può essere visto come il sottoinsieme dei naturali definito induttivamente da

- $0 \in \mathbb{N}$;
- se $n \in \mathbb{N}$ allora $n+1 \in \mathbb{N}$.

Allora la proposizione è un caso particolare del principio di induzione. □
Vediamo un esempio di applicazione.

Esempio 1.6 Teorema: $2^0 + 2^1 + ... + 2^n = 2^{n+1} - 1$, per tutti gli $n \in \mathbb{N}$. **Dimostrazione.**

- Sia, per ogni $i \in \mathbb{N}$, $P(i)$ il predicato "$2^0 + 2^1 + ... + 2^i = 2^{i+1} - 1$."
- Consideriamo il *caso base*: $P(0)$:
$$2^0 = 1 = 2^1 - 1$$
- Consideriamo il *caso induttivo* : assumiamo $P(i)$ e dimostriamo $P(i+1)$:
$$\begin{array}{rcll} 2^0 + 2^1 + ... + 2^{i+1} & = & (2^0 + 2^1 + ... + 2^i) + 2^{i+1} & \\ & = & (2^{i+1} - 1) + 2^{i+1} & \text{(per ipotesi induttiva)} \\ & = & 2 \cdot (2^{i+1}) - 1 & \\ & = & 2^{i+2} - 1 & \end{array}$$
- Dal principio di induzione aritmetica, deriviamo che $P(n)$ vale per tutti gli $n \in \mathbb{N}$.

Prop. 1.7 [Principio di induzione aritmetica completa] Sia P un predicato sui numeri naturali t.c.

Per ogni $n \in \mathbb{N}$, ($P(m) = T$ per ogni $m < n$) implica $P(n) = T$.

Allora $P(n) = T$ per ogni $n \in \mathbb{N}$.

Dimostrazione. $\mathbb{N}$ può essere visto come il sottoinsieme dei naturali definito induttivamente da

- $0 \in \mathbb{N}$
- se $\{m \mid m < n\} \subseteq \mathbb{N}$ allora $n \in \mathbb{N}$.

Allora la proposizione è un caso particolare del principio di induzione. □

1.3 Definizioni induttive multiple

Nella pratica sono particolarmente utili in informatica le definizioni induttive di famiglie di insiemi (Def.A.4), dette anche definizioni induttive *multiple*, con l'associato principio di induzione multipla. Si tratta di una facile generalizzazione delle definizioni e dei risultati già dati.

Def. 1.8 Sia nel seguito S un insieme i cui elementi sono detti *tipi* (o *sort* o *indici*). Sia U una S-famiglia fissata detta *universo*.
Una *regola indiciata* su S è una coppia $<\{Pr_s\}_{s\in S}, c : \overline{s}>$, dove Pr è una S-famiglia di insiemi (gli insiemi *premesse*), con $Pr \subseteq U$ (cioè $Pr_s \subseteq U_s$ per ogni $s \in S$), c è un elemento (la *conseguenza*), $\overline{s} \in S$ è un tipo, con $c \in U_{\overline{s}}$.
Una famiglia $X = \{X_s\}_{s\in S}$ di insiemi è *R-chiusa* rispetto a una regola $<\{Pr_s\}_{s\in S}, c : \overline{s}>$ sse $Pr_s \subseteq X_s$ per ogni $s \in S$ implica $c \in X_{\overline{s}}$; se R è un insieme di regole indiciate su S, X si dice *R-chiusa* (o *chiusa rispetto a R*) sse è chiusa rispetto a ogni regola di R. Un *sistema induttivo multiplo* su S è un insieme di regole indiciate su S.
Se R è un sistema induttivo multiplo su S, *la famiglia $I(R) = \{I(R)_s\}_{s\in S}$ definita induttivamente da R* è l'intersezione di tutte le famiglie R-chiuse.

Analogamente a quanto si è visto per le definizioni induttive non multiple, i sistemi induttivi sono in generale presentati dando un insieme di regole di inferenza o metaregole.
Lo schema di risoluzione degli esercizi per il caso dell'induzione multipla ricalca esattamente quello del caso omogeneo.
Come nel caso omogeneo, si possono definire induttivamente famiglie di funzioni.
Analogamente si estende il principio di induzione.

Prop. 1.9 [Principio di induzione multipla] Sia R un sistema induttivo multiplo su S, U una S-famiglia di insiemi tale che $I(R) \subseteq U$, e P una S-famiglia di predicati su U t.c.

per ogni $<\{Pr_s\}_{s\in S}, c : \overline{s}> \in R$

$$(P_s(d) = T \text{ per ogni } d \in Pr_s, \text{ per ogni } s \in S) \text{ implica } P_{\overline{s}}(c) = T.$$

Allora $P_s(d) = T$ per ogni $d \in I_s(R)$ e per ogni $s \in S$.

Dimostrazione. Analoga a quella del caso non multiplo. □

Anche qui l'ipotesi del principio di induzione può essere utilmente spezzata in "base" e "passo induttivo" in corrispondenza dell'analoga suddivisione delle regole.
In conclusione nel caso multiplo si ragiona e procede come nel caso non multiplo, eccetto per il fatto che si ha a che fare con una famiglia di insiemi e che occorre quindi tener conto dell'appartenenza di un elemento a un particolare insieme della famiglia.

Esempio 1.10 Diamo una definizione mutuamente induttiva di tre sottoinsiemi delle stringhe su $\{\mathtt{a}, \mathtt{b}\}$: l'insieme delle stringhe con un numero uguale di `a` e di `b`, l'insieme delle stringhe con una `a` in più, e l'insieme delle stringhe con una `b` in più (questa definizione induttiva può anche essere espressa come grammatica libera da contesto, vedi Sez.1.4; si provi a farlo per esercizio).

- Λ è una stringa con numero di `a` uguale al numero di `b`,
- se b è una stringa con una `b` in più, allora $\mathtt{a}b$ è una stringa con numero di `a` uguale al numero di `b`; analogamente se a è una stringa con una `a` in più, allora $\mathtt{b}a$ è una stringa con numero di `a` uguale al numero di `b`;
- se s è una stringa con numero di `a` uguale al numero di `b`, allora $\mathtt{a}s$ è una stringa con una `a` in più, e analogamente $\mathtt{b}s$ è una stringa con una `b` in più,
- se a_1, a_2 sono due stringhe con una `a` in più, allora $\mathtt{b}a_1a_2$ è una stringa con una `a` in più; analogamente, se b_1, b_2 sono due stringhe con una `b` in più, allora $\mathtt{a}b_1b_2$ è una stringa con una `b` in più.

Anche in questo caso, ci si può convincere con il ragionamento (e si può provare) che abbiamo effettivamente definito tutte e sole le stringhe del tipo richiesto. Nella forma a regole di inferenza, indicando con S, A, B i tre tipi (corrispondenti alla stringhe con numero di `a` uguale al numero di `b`, a quelle con una `a` in più e a quelle con una `b` in più), la definizione sopra può essere scritta nel modo seguente.

$$\frac{}{\Lambda : S} \qquad \frac{b : B}{\mathtt{a}b : S} \qquad \frac{a : A}{\mathtt{b}a : S}$$

$$\frac{s : S}{\mathtt{a}s : A} \qquad \frac{a_1 : A, a_2 : A}{\mathtt{b}a_1a_2 : A}$$

$$\frac{s : S}{\mathtt{b}s : B} \qquad \frac{b_1 : B, b_2 : B}{\mathtt{a}b_1b_2 : B}$$

Esempio 1.11 Si consideri la seguente definizione mutuamente ricorsiva di due funzioni `f: int -> int` (scritta con la sintassi Caml).

```
let rec f = function
  x -> if x = 0 then 0 else if x > 0 then f(x-1)+1 else g(x)
  and g = function
  x -> if x = 0 then f(x) else g(x+1)
```

La funzione f risulta essere l'identità sugli argomenti positivi, la costante 0 sugli altri; la funzione g risulta essere definita (e dare 0) solo sugli argomenti minori o uguali a zero.
Nella forma a regole di inferenza, la definizione sopra può essere scritta nel modo seguente.

$$\frac{}{f(0) = 0} \qquad \frac{f(x-1) = r}{f(x) = r+1}\ x > 0 \qquad \frac{g(x) = r}{f(x) = r}\ x < 0$$

$$\frac{f(0) = r}{g(0) = r} \qquad \frac{g(x+1) = r}{g(x) = r}\ x \neq 0$$

1.4 Sintassi e induzione strutturale

Come visto nel corso di L. P., un linguaggio (formalmente, una famiglia di insiemi di stringhe) può essere definito per mezzo di una grammatica. Si noti che ogni produzione

$$A ::= \alpha_0 B_1 \alpha_1 \dots B_n \alpha_n$$

di una grammatica può essere vista come una metaregola di una definizione induttiva. Il linguaggio generato dalla grammatica può allora essere visto come l'insieme (la famiglia di insiemi) definito induttivamente dalle metaregole.

$$\frac{x_1 : B_1 \dots x_n : B_n}{\alpha_0 x_1 \alpha_1 \dots x_n \alpha_n : A}$$

Si noti che le definizioni induttive sono molto più generali; per esempio, nelle definizioni induttive è possibile utilizzare più volte la stessa metavariabile nelle premesse, cioè esprimere il fatto che due sottocomponenti della stringa devono essere uguali. Questo infatti è un esempio di vincolo contestuale.
Per esempio la seguente grammatica per un semplice linguaggio di espressioni $\mathcal{E}$:

$$\mathit{Exp} \quad ::= \quad \texttt{true} \mid \texttt{false} \mid \texttt{succ}\ \mathit{Exp} \mid \texttt{pred}\ \mathit{Exp} \mid \texttt{if}\ \mathit{Exp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \mid \texttt{0} \mid \texttt{iszero}\ \mathit{Exp}$$

corrisponde a definire il linguaggio come il più piccolo insieme $\mathcal{E}$ tale che:

1. $\{\texttt{true}, \texttt{false}, 0\} \subseteq \mathcal{E}$;
2. se $t \in \mathcal{E}$, allora $\{\texttt{succ}\ t, \texttt{pred}\ t, \texttt{iszero}\ t\} \subseteq \mathcal{E}$;
3. se $t, t_1, t_2 \in \mathcal{E}$, allora $\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \in \mathcal{E}$.

oppure, nella versione a regole di inferenza:

$$\frac{}{\texttt{true}} \quad \frac{}{\texttt{false}} \quad \frac{}{0} \quad \frac{t}{\texttt{succ}\ t} \quad \frac{t}{\texttt{pred}\ t} \quad \frac{t}{\texttt{iszero}\ t} \quad \frac{t \quad t_1 \quad t_2}{\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2}$$

Si usa anche talvolta (noi lo faremo) lo stile BNF (però utilizzando le metavariabili invece dei non terminali):

$$t \quad ::= \quad \texttt{true} \mid \texttt{false} \mid \texttt{succ}\ t \mid \texttt{pred}\ t \mid \texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \mid \texttt{0} \mid \texttt{iszero}\ t$$

Un modo equivalente di definire formalmente un linguaggio è come famiglia dei *termini* (o *espressioni*[2]) su una segnatura.
Data una segnatura (Def.A.5) $\Sigma = (S, O)$, la famiglia $\mathcal{T}$ dei termini su Σ è la S-famiglia definita induttivamente da:

per ogni operatore $op\colon s_1 \dots s_n \to s$ in O, se $t_1 \in \mathcal{T}_{s_1}, \dots, t_n \in \mathcal{T}_{s_n}$, allora $op(t_1, \dots, t_n) \in \mathcal{T}_s$.

Si noti che la base della definizione è data dalle costanti (operatori zerari).
Quando definiamo un linguaggio come famiglia dei termini su una segnatura ne mettiamo in evidenza la struttura algebrica, cioè il fatto che abbiamo implicitamente anche degli *operatori* su stringhe. Per esempio, un'espressione della forma $\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2$ viene vista come il risultato dell'applicazione dell'operatore ternario infisso $\texttt{if}\_\texttt{then}\_\texttt{else}\_$ alle sottoespressioni t, t_1, t_2. Questo può essere formalizzato dicendo che i termini su una segnatura non sono semplicemente una famiglia di insiemi ma un'*algebra* (Def.A.6) in cui l'interpretazione di ogni operatore $op\colon s_1 \dots s_n \to s$ è la funzione (totale) che dati i termini $t_1, \dots, t_n$ restituisce il termine $op(t_1, \dots, t_n)$.
La definizione sopra definisce i termini nella notazione *funzionale*, tuttavia chiaramente la notazione utilizzata non è rilevante e in modo analogo si possono definire i termini in notazione prefissa, postfissa, distribuita infissa etc., o anche scegliere una diversa notazione per l'applicazione di ogni operatore.
Come visto nel corso di L. P., in realtà anche quando diamo una grammatica definiamo implicitamente una segnatura associata e quindi un'algebra di termini su tale segnatura.

Def. 1.12 [Segnatura associata a una grammatica] La *segnatura associata* alla grammatica $G = (T, N, P)$ è la segnatura $\Sigma = (S, O)$ dove:

- $S = N$, cioè le sort sono esattamente i non terminali;

[2] In questo caso la parola "espressioni" viene usata in modo assolutamente generale: per esempio nel caso del linguaggio C espressioni del linguaggio saranno le dichiarazioni, i comandi, anche le espressioni propriamente dette, etc.

- per ogni produzione $A ::= \alpha_0 B_1 \alpha_1 \dots B_n \alpha_n$, con $B_1, \dots, B_n \in N$, $\alpha_0, \dots, \alpha_n \in T^\star$, vi è in O un operatore distribuito infisso $\alpha_0 \_ \alpha_1 \dots \_ \alpha_n \colon B_1 \dots B_n \to A$, e non vi sono altri operatori in O.

La segnatura associata a una grammatica è anche detta *sintassi astratta*, perché descrive la struttura algebrica del linguaggio (i tipi e gli operatori) astraendo rispetto alle particolari rappresentazioni concrete possibili (cioè le diverse algebre di termini).
Volendo astrarre ulteriormente, si può osservare che anche i simboli di sort e di operazione effettivamente utilizzati sono irrilevanti; quindi la definizione precedente può anche essere espressa in modo più generale dicendo che la sintassi astratta è determinata da una qualunque segnatura in cui le sort sono in corrispondenza biunivoca con i non terminali e gli operatori sono in corrispondenza biunivoca con le produzioni nel senso precisato sopra.
Quando si ragiona su proprietà (tipicamente la semantica statica o dinamica) che sono indipendenti dalla particolare rappresentazione concreta scelta per i termini, parlando di "linguaggio" si intende fare riferimento alla sintassi astratta. In particolare quindi, non ci si preoccupa di eventuali problemi di ambiguità della rappresentazione usata, perché fissata la sintassi astratta è sempre possibile trovare una rappresentazione concreta non ambigua. Si utilizza quindi una rappresentazione concreta comoda, e in caso di ambiguità semanticamente rilevante si aggiungono delle parentesi o si introducono delle convenzioni di precedenza.

Prop. 1.13 [Principio di induzione strutturale]
Sia $\mathcal{T}$ un linguaggio definito come famiglia dei termini su una segnatura (S, O).
Sia P una S-famiglia di predicati su U, $\mathcal{T} \subseteq U$ e, per ogni $op \colon s_1 \dots s_n \to s$ in O,

$$(P_{s_i}(t_i) = T \text{ per ogni } i \in 1..n) \text{ implica } P_s(op(t_1, \dots, t_n)) = T.$$

Allora $P_s(t) = T$ per ogni $t \in \mathcal{T}_s$ e per ogni $s \in S$.

Dimostrazione. Come visto sopra, $\mathcal{T}$ è la sottofamiglia di U definita induttivamente da

per ogni $op \colon s_1 \dots s_n \to s$, $t_i \in \mathcal{T}_{s_i}$, per $i \in 1..n$ implica $op(t_1, \dots, t_n) \in \mathcal{T}_s$.

Allora la proposizione è un caso particolare del principio di induzione multipla. □
Nel caso del semplice linguaggio di espressioni visto prima il principio di induzione strutturale prende la forma seguente. Dato un predicato P su (un soprainsieme di) $\mathcal{E}$,

- per prima cosa dobbiamo dimostrare che valgono $P(\texttt{true})$, $P(\texttt{false})$ e $P(\texttt{0})$;
- poi
  - con l'ipotesi che $P(t)$ valga dobbiamo dimostrare che $P(\texttt{succ}\ t)$, $P(\texttt{pred}\ t)$, $P(\texttt{iszero}\ t)$ valgono, e
  - con l'ipotesi che $P(t)$, $P(t_1)$, e $P(t_2)$ valgano dobbiamo dimostrare che $P(\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2)$ vale.
- Da questo deriviamo che $P(t)$ vale per tutte le espressioni $t \in \mathcal{E}$.

Vediamo qualche esempio di applicazione del principio di induzione strutturale nel caso di $\mathcal{E}$. Abbiamo già osservato che quando il dominio di una funzione è un insieme definito induttivamente risulta naturale e comodo definire la funzione in modo diretto da tale definizione induttiva. Per esempio, definiamo induttivamente l'insieme delle costanti (`true`, `false`, `0`) che compaiono in un'espressione $t \in \mathcal{E}$, e chiamiamolo $Consts(t)$.

$$\begin{array}{lcl}
Consts(\texttt{true}) & = & \{\texttt{true}\} \\
Consts(\texttt{false}) & = & \{\texttt{false}\} \\
Consts(\texttt{0}) & = & \{\texttt{0}\} \\
Consts(\texttt{succ}\ t) & = & Consts(t) \\
Consts(\texttt{pred}\ t) & = & Consts(t) \\
Consts(\texttt{iszero}\ t) & = & Consts(t) \\
Consts(\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2) & = & Consts(t) \cup Consts(t_1) \cup Consts(t_2)
\end{array}$$

In questo modo è facile provare proprietà della definizione data, anzitutto che abbiamo effettivamente definito una funzione.
Un'altro esempio di definizione di funzione per induzione sui termini è la seguente (dimensione di un termine, cioè numero di sottotermini):

$$\begin{array}{lcl}
dim(\texttt{true}) & = & 1 \\
dim(\texttt{false}) & = & 1 \\
dim(\texttt{0}) & = & 1 \\
dim(\texttt{succ}\ t) & = & dim(t) + 1 \\
dim(\texttt{pred}\ t) & = & dim(t) + 1 \\
dim(\texttt{iszero}\ t) & = & dim(t) + 1 \\
dim(\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2) & = & dim(t) + dim(t_1) + dim(t_2) + 1
\end{array}$$

Vediamo un esempio di proprietà che può essere facilmente provata per induzione strutturale.

Prop. 1.14 Il numero delle costanti distinte in un termine è al più la dimensione del termine, cioè[3] $|Consts(\mathtt{t})| \leq \mathtt{dim(t)}$.

2 Semantica operazionale

2.1 Principi

Vediamo come formalizzare la semantica di un linguaggio nello stile *operazionale small step* (piccoli passi). L'idea è quella di dare un modello astratto dei passi di esecuzione di un programma nel linguaggio. Si dice anche che si definisce una *macchina astratta* che modella l'esecuzione del linguaggio.
Una *macchina astratta* consiste di:

- un insieme di *stati* $s \in S$
- una relazione di *riduzione* (o *transizione*) tra stati, che indicheremo con $\rightarrow$.

Una stato rappresenta *tutte* le informazioni sufficienti a descrivere la configurazione della macchina ad un certo momento. Per esempio, una descrizione stile macchina astratta di un microprocessore convenzionale includerebbe: program counter, contenuto dei registri, contenuto della memoria principale, e il codice macchina che si sta eseguendo.
In molti casi, la relazione di riduzione è una funzione parziale: cioè, dato uno stato, c'è al più uno stato possibile in cui transire.
Per i primi linguaggi che considereremo lo stato della macchina astratta è costituito semplicemente dal termine che stiamo valutando, e un valore finale è semplicemente un termine non ulteriormente riducibile di forma particolare. In questo caso il linguaggio con la relazione di riduzione si chiama anche *calcolo*. Un calcolo consiste quindi di:

- un insieme di termini: $t \in \mathcal{T}$
- una relazione di riduzione: $t \rightarrow t'$
- inoltre, un insieme di valori, che modellano i "risultati finali": $v \in \mathit{Val} \subseteq \mathcal{T}$, tale che $\forall v \in \mathit{Val}.\ \nexists t. v \rightarrow t$ (scriveremo anche semplicemente $v \not\rightarrow$).

2.2 Espressioni booleane

Diamo ora, come primo esempio, la semantica operazionale small-step per le espressioni booleane. La sintassi dei termini e dei valori è la seguente:

$$\begin{array}{lcl} t & ::= & \mathtt{true} \mid \mathtt{false} \mid \mathtt{if}\ t\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \\ v & ::= & \mathtt{true} \mid \mathtt{false} \end{array}$$

La relazione di riduzione $\rightarrow$ è definita induttivamente in Fig.1.

$$\text{(IfTrue)}\ \frac{}{\mathtt{if\ true\ then}\ t_1\ \mathtt{else}\ t_2 \rightarrow t_1} \qquad \text{(IfFalse)}\ \frac{}{\mathtt{if\ false\ then}\ t_1\ \mathtt{else}\ t_2 \rightarrow t_2}$$

$$\text{(If)}\ \frac{t \rightarrow t'}{\mathtt{if}\ t\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 \rightarrow \mathtt{if}\ t'\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2}$$

Figura 1: Regole di riduzione per le espressioni booleane

Regole come (IfTrue) e (IfFalse) sono dette regole di *computazione*, in quanto descrivono i passi base di valutazione (modifiche del termine), mentre regole come (If) sono dette regole di *propagazione* (o *congruenza*), in quanto determinano in che ordine le regole di computazione vengono applicate ai sottotermini (cioè, la *strategia* di valutazione).

[3] $|A|$ denota la cardinalità di A.

Esercizio 2.1 Supponiamo di voler cambiare la strategia di valutazione in modo tale da forzare la valutazione del ramo `then` e del ramo `else` prima di valutare la condizione dell'`if`. Come si devono cambiare le regole?
Supponiamo che nel caso in cui il valore del ramo `then` e del ramo `else` siano uguali si voglia produrre questo valore (senza valutare la condizione dell' `if`). Come si devono cambiare le regole?
Delle regole date quali sono computazionali e quali di congruenza?

Possiamo provare che una certa coppia di termini è nella relazione di riduzione con un *albero di prova*, come illustrato dal seguente esempio.

$$\text{(If)}\ \frac{\text{(IfTrue)}\ \dfrac{}{\texttt{if true then true else false} \to \texttt{true}}}{\texttt{if (if true then true else false) then true else false} \to \texttt{if true then true else false}}$$

Dato che la relazione di riduzione $\to$ è definita induttivamente dalle metaregole viste prima, è possibile provarne delle proprietà applicando il principio di induzione; ossia, basta far vedere che, per ogni metaregola, se la proprietà vale per le premesse allora vale per la conseguenza. Per esempio si può provare facilmente che per le espressioni booleane vale la seguente proprietà.

Prop. 2.2 Se $t \to t'$, allora $\texttt{dim}(t) > \texttt{dim}(t')$.

2.3 Espressioni naturali

Aggiungiamo ora qualche forma sintattica

$$\begin{array}{lcl} t & ::= & \ldots \mid \texttt{succ}\ t \mid \texttt{pred}\ t \mid \texttt{0} \mid \texttt{iszero}\ t \\ v & ::= & \ldots \mid n \\ n & ::= & \texttt{0} \mid \texttt{succ}\ n \end{array}$$

Le nuove regole di riduzione sono date in Fig.2. Si noti che abbiamo aggiunto la nuova categoria sintattica n perché nelle regole di riduzione (in particolare, in (PredSucc) e (IsZeroSucc)) è necessario distinguere i valori di tipo naturale da quelli di tipo booleano.

$$\text{(Succ)}\ \frac{t \to t'}{\texttt{succ}\ t \to \texttt{succ}\ t'}$$

$$\text{(Pred)}\ \frac{t \to t'}{\texttt{pred}\ t \to \texttt{pred}\ t'} \qquad \text{(PredSucc)}\ \frac{}{\texttt{pred succ}\ n \to n} \qquad \text{(PredZero)}\ \frac{}{\texttt{pred 0} \to \texttt{0}}$$

$$\text{(IsZeroZero)}\ \frac{}{\texttt{iszero 0} \to \texttt{true}} \qquad \text{(IsZeroSucc)}\ \frac{}{\texttt{iszero succ}\ n \to \texttt{false}} \qquad \text{(IsZero)}\ \frac{t \to t'}{\texttt{iszero}\ t \to \texttt{iszero}\ t'}$$

Figura 2: Regole di riduzione per le espressioni naturali

Le regole di riduzione sono date assumendo, per semplicità, che il predecessore di zero sia zero. Una modellazione più "realistica" dovrebbe includere un termine particolare che rappresenti la situazione "errore: tentativo di applicare l'operatore predecessore a zero" (si veda la Sez.5.3).

2.4 Risultati

Def. 2.3 Un termine t è una *forma normale* (o è *in forma normale*) se non può essere ulteriormente ridotto, cioè $t \not\to$.

È immediato vedere che per il linguaggio $\mathcal{E}$ vale la seguente proprietà, che ci aspettiamo in genere essere verificata in un calcolo.

Fatto 2.4 Ogni valore è una forma normale.

Nel linguaggio limitato alle espressioni booleane vale anche il viceversa:

Prop. 2.5 Ogni forma normale è un valore.

Dimostrazione. Per induzione strutturale (farlo per esercizio). □
Tuttavia, il viceversa non vale in genere; per esempio, non vale per tutto il linguaggio $\mathcal{E}$. Infatti, vi sono termini in forma normale che non sono valori, per esempio `succ true`. Questi termini si dicono *stuck* (*bloccati*) e modellano una situazione di errore a run-time (ossia, il programma ha raggiunto uno stato inconsistente). Lo scopo di introdurre un sistema di tipi (vedi Sez.4) sarà proprio quello di scartare i termini (che possono ridursi in termini) stuck, ossia dar luogo a (una certa classe di) errori a run-time.

Prop. 2.6 [Determinismo] La relazione di riduzione $\rightarrow$ in $\mathcal{E}$ è *deterministica*, cioè per ogni $t \in \mathcal{T}$ esiste *al più* un t' tale che $t \rightarrow t'$.

Dimostrazione. Per induzione strutturale (farlo per esercizio).

Def. 2.7 La relazione di *riduzione in più passi*, scritta $\rightarrow^\star$, è la chiusura riflessiva e transitiva della relazione $\rightarrow$, cioè la relazione definita induttivamente nel modo seguente:

- se $t \rightarrow t'$ allora $t \rightarrow^\star t'$,
- $t \rightarrow^\star t$ per tutti i t,
- se $t \rightarrow^\star t'$ e $t' \rightarrow^\star t''$, allora $t \rightarrow^\star t''$.

È possibile provare che la relazione di riduzione per il linguaggio $\mathcal{E}$ è *terminante*, cioè non esistono sequenze di riduzione infinite.

Teorema 2.8 [Terminazione] Ogni sequenza di riduzione $\rightarrow$ in $\mathcal{E}$ è finita.

Dimostrazione. È facile provare che anche per tutto $\mathcal{E}$ vale la Prop.2.2. Allora, se assumiamo per assurdo che esista una sequenza di riduzione infinita

$$t_0 \rightarrow t_1,\ t_1 \rightarrow t_2,\ t_2 \rightarrow t_3,\ \ldots,$$

la seguente sarebbe una una sequenza infinita, strettamente decrescente di numeri naturali (positivi):

$$dim(t_0),\ dim(t_1),\ dim(t_2),\ dim(t_3),\ \ldots$$

Ma tale sequenza non può esistere. □
Nella dimostrazione precedente abbiamo utilizzato la seguente tecnica standard per provare che una relazione $R \subseteq X \times X$ è terminante.

- Si scelgono
  - un insieme parzialmente ordinato $(P, <)$ ben fondato (vedi Appendice A.2),
  - una funzione $f\colon X \rightarrow P$.
- Si dimostra che $f(x) > f(y)$ per tutti gli $(x, y) \in R$.
- Si conclude che non ci sono sequenze infinite $(x_i, x_{i+1}) \in R$ per ogni i, poiché, se ci fossero, potremmo costruire una catena discendente infinita in P.

La Prop.2.6 e il Teorema 2.8 insieme implicano immediatamente il seguente teorema di *esistenza e unicità della forma normale*.

Teorema 2.9 [Esistenza e unicità della forma normale] Per ogni $t \in \mathcal{T}$, esiste un unico t' in forma normale tale che $t \rightarrow^\star t'$.

Esercizio 2.10

- Supponiamo di aggiungere al linguaggio delle espressioni booleane la seguente nuova regola:

  (IfTrueBis) $$\frac{}{\texttt{if true then } t_1 \texttt{ else } t_2 \rightarrow t_2}$$

  Quali tra le seguenti proprietà continuano a valere?
  - ogni valore è una forma normale (Fatto 2.4),
  - ogni forma normale è un valore (Prop.2.5),
  - determinismo (Prop.2.6),

- terminazione (Teorema 2.8),
- esistenza e unicità della forma normale (Teorema 2.9).

- Lo stesso se invece aggiungiamo la regola:

$$\text{(IfBis)}\quad \frac{t_2 \to t_2'}{\texttt{if } t \texttt{ then } t_1 \texttt{ else } t_2 \to \texttt{if } t \texttt{ then } t_1 \texttt{ else } t_2'}$$

Esercizio 2.11 Si consideri la semantica alternativa in Fig.3 per il linguaggio delle espressioni booleane data nello stile *big step* (*grandi passi*), basato sull'idea di associare direttamente a ogni termine il suo risultato finale, attraverso una relazione tra termini e valori $t \Downarrow v$. (Si noti che la semantica big-step è più astratta di quella small-step in quanto prescinde dall'ordine di valutazione.)

$$\text{(Big-True)}\quad \frac{}{\texttt{true} \Downarrow \texttt{true}} \qquad \text{(Big-False)}\quad \frac{}{\texttt{false} \Downarrow \texttt{false}}$$

$$\text{(Big-IfTrue)}\quad \frac{t \Downarrow \texttt{true} \quad t_1 \Downarrow v}{\texttt{if } t \texttt{ then } t_1 \texttt{ else } t_2 \Downarrow v} \qquad \text{(Big-IfFalse)}\quad \frac{t \Downarrow \texttt{false} \quad t_2 \Downarrow v}{\texttt{if } t \texttt{ then } t_1 \texttt{ else } t_2 \Downarrow v}$$

Figura 3: Regole di riduzione big-step per le espressioni booleane

Si provi che le due semantiche coincidono, cioè $t \Downarrow v \Leftrightarrow t \to^\star v$.

Soluzione. Proviamo separatamente le due implicazioni.

$\Rightarrow$ Per induzione sulla definizione di $\Downarrow$, ossia, per ogni (meta)regola che definisce $\Downarrow$, proviamo che se la proprietà vale per le premesse allora vale anche per la conseguenza.

(Big-True) Si ha banalmente che $\texttt{true} \to^\star \texttt{true}$ (in zero passi).

(Big-False) Analogo al caso precedente.

(Big-IfTrue) Dobbiamo provare che $\texttt{if } t \texttt{ then } t_1 \texttt{ else } t_2 \to^\star v$. Per ipotesi induttiva sappiamo che $t \to^\star \texttt{true}$. Allora, applicando (If) un numero di volte pari al numero di passi in $t \to^\star \texttt{true}$, otteniamo:

$$\texttt{if } t \texttt{ then } t_1 \texttt{ else } t_2 \to^\star \texttt{if true then } t_1 \texttt{ else } t_2$$

A questo punto, applicando (IfTrue), otteniamo

$$\texttt{if true then } t_1 \texttt{ else } t_2 \to^\star t_1$$

e possiamo concludere poiché per ipotesi induttiva vale $t_1 \to^\star v$.

$\Leftarrow$ Per induzione aritmetica sulla lunghezza della derivazione.

$t \to^0 v$ Allora t coincide con v, e deve essere $\texttt{true}$ oppure $\texttt{false}$; in entrambi i casi si ha banalmente la tesi.

$t \to^{n+1} v$ Allora $t \to t' \to^n v$. Per ipotesi induttiva sappiamo che $t' \Downarrow v$. Proviamo, per induzione sulla definizione di $\to$, che se $t \to t'$ e $t' \Downarrow v$, allora $t \Downarrow v$.

(IfTrue) Dobbiamo provare che se $t_1 \Downarrow v$, allora $\texttt{if true then } t_1 \texttt{ else } t_2 \Downarrow v$. Applicando (Big-True) e (Big-IfTrue) otteniamo la tesi. Analogamente per la regola (IfFalse).

(If) Dobbiamo provare che se $\texttt{if } t' \texttt{ then } t_1 \texttt{ else } t_2 \Downarrow v$, allora $\texttt{if } t \texttt{ then } t_1 \texttt{ else } t_2 \Downarrow v$. Abbiamo derivato $\texttt{if } t' \texttt{ then } t_1 \texttt{ else } t_2 \Downarrow v$ applicando (Big-IfTrue) oppure (Big-IfFalse). Consideriamo per esempio il primo caso (l'altro è analogo). Allora, sappiamo che valgono le premesse $t' \Downarrow \texttt{true}$ e $t_1 \Downarrow v$. Dalla prima di queste e dalla premessa $t \to t'$ della regola (If) applicando l'ipotesi induttiva possiamo concludere che vale $t \Downarrow \texttt{true}$. A questo punto, applicando (Big-IfTrue) con premesse $t \Downarrow \texttt{true}$ e $t_1 \Downarrow v$ otteniamo la tesi.

3 Lambda-calcolo

3.1 Introduzione

Nella sezione precedente abbiamo visto un semplice esempio di linguaggio non banale. In questa sezione presenteremo il lambda-calcolo, introdotto negli anni '30 da Alonso Church e Stephen Cole Kleene, che può essere considerato nonostante la sua estrema semplicità un vero e proprio linguaggio di programmazione; infatti, è un formalismo Turing completo (ossia, permette di esprimere "tutti gli algoritmi"), quindi ogni linguaggio di programmazione reale può essere codificato in esso. Per questa combinazione di semplicità e potenza espressiva il lambda-calcolo è stato considerato come *il* modello privilegiato di tutti i linguaggi di programmazione. In particolare, si basano più direttamente sul lambda-calcolo i linguaggi *funzionali* come ML.
L'idea fondamentale alla base del lambda-calcolo (poi ereditata dai linguaggi funzionali) è stata quella di definire un calcolo per le *funzioni*; ossia, dove fosse possibile scrivere termini corrispondenti alle funzioni "matematiche" e avere regole di riduzione che modellano il comportamento dell'applicazione di una funzione ai suoi argomenti.
I costrutti linguistici base sono quindi: definizione di funzioni e applicazione. Per quel che riguarda la prima, un'importante ossevazione è quella che nel definire una funzione il nome è irrilevante: $f(x) = x + 3$ e $g(x) = x + 3$ definiscono la stessa funzione. Infatti talvolta in matematica si usa la notazione $x \mapsto x + 3$. Nel lambda-calcolo si scrive $\lambda \mathtt{x}.\mathtt{x} + 3$. Supponendo di utilizzare gli operatori del linguaggio $\mathcal{E}$:

$\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x}$

Un termine come il precedente, che rappresenta una funzione, si chiama λ-*astrazione*. Per comodità, negli esempi spesso utilizzeremo nomi per abbreviare delle lambda-astrazioni, per esempio:

sia $add3$ il termine $\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x}$

Tuttavia, occorre sempre ricordare che tali nomi sono a livello di *metalinguaggio*, mentre nel lambda-calcolo descritto in questo capitolo le lambda-astrazioni sono *anonime* e non vi è alcun costrutto che permetta di associare a esse un nome. Vedremo più avanti come estendere il calcolo con un meccanismo di *dichiarazioni* (Sez.5.1).
L'applicazione di una funzione f a un argomento e è scritta semplicemente $f\ e$. Per esempio:

$(\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x})\,(\mathtt{succ\ 0})$

o utilizzando l'abbreviazione $add3\ (\mathtt{succ\ 0})$.
L'effetto di un'applicazione è modellato da una regola di riduzione (detta *beta-regola*), formalizzata nel seguito, che la trasforma nel termine ottenuto sostituendo l'argomento al posto del parametro nel corpo dell'applicazione stessa, per esempio:

$(\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x})\,(\mathtt{succ\ 0}) \rightarrow \mathtt{succ\ succ\ succ\ succ\ 0}$

Come nell'uso matematico, è possibile nel lambda-calcolo descrivere funzioni *di ordine superiore* (*higher-order*), che cioè prendono come argomento e/o restituiscono come risultato ancora una funzione. Per esempio nella lambda-astrazione

$g = \lambda \mathtt{f}.\mathtt{f}\ (\mathtt{f}\ (\mathtt{succ\ 0}))$

il parametro `f` è utilizzato come *funzione* nel corpo di g. Se applichiamo g a un argomento come $add3$, la beta-regola produce la seguente computazione:

$$\begin{array}{rcl} g\ add3 & = & (\lambda \mathtt{f}.\mathtt{f}\ (\mathtt{f}\ (\mathtt{succ\ 0})))(\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x}) \\ & \rightarrow & (\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x})((\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x})\ (\mathtt{succ\ 0})) \\ & \rightarrow & (\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x})(\mathtt{succ\ succ\ succ\ succ\ 0}) \\ & \rightarrow & \mathtt{succ\ succ\ succ\ succ\ succ\ succ\ succ\ 0} \end{array}$$

Analogamente, la seguente variante di g:

$double = \lambda \mathtt{f}.\lambda \mathtt{y}.\mathtt{f}\ (\mathtt{f\ y})$

è una funzione che, applicata ad una funzione `f`, produce una funzione che, quando applicata a un argomento y, produce `f` (`f` y).

$$\begin{array}{rcl} double\ add3\ \mathtt{0} & = & (\lambda \mathtt{f}.\lambda \mathtt{y}.\mathtt{f}\ (\mathtt{f\ y}))(\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x})\mathtt{0} \\ & \rightarrow & (\lambda \mathtt{y}.(\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x})\ ((\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x})\ \mathtt{y}))\mathtt{0} \\ & \rightarrow & (\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x})\ ((\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x})\ \mathtt{0}) \\ & \rightarrow & (\lambda \mathtt{x}.\mathtt{succ\ succ\ succ\ x})\ (\mathtt{succ\ succ\ succ\ 0}) \\ & \rightarrow & \mathtt{succ\ succ\ succ\ succ\ succ\ succ\ 0} \end{array}$$

3.2 Sintassi

La seguente è la sintassi del *lambda-calcolo puro*, dove gli unici costrutti sintattici sono quelli relativi alle funzioni, quindi variabili, lambda-astrazione e appllcazione.

$$\begin{array}{lcl} t & ::= & x \mid \lambda x.t \mid t_1\ t_2 \\ x & ::= & \mathtt{x} \mid \mathtt{y} \mid \mathtt{f} \mid \ldots \end{array}$$

Si noti che la differenza tra x, che è la metavariabile usata per le variabili del lambda-calcolo, e `x`, `y`, `f`, ... che sono esempi concreti di variabili (anche se a volte potremo confondere le due cose).
Si usano le seguenti convenzioni per disambiguare i termini.

- L'applicazione associa a sinistra, cioè $t_1\ t_2\ t_3 = (t_1\ t_2)\ t_3$, che è diverso da $t_1\ (t_2\ t_3)$.
- Il corpo di una λ-astrazione estende quanto possibile a destra, cioè $\lambda x.t_1 t_2 = \lambda x.(t_1\ t_2)$, che è diverso da $(\lambda x.t_1)\ t_2$.

Nella λ-astrazione $\lambda x.t$ l'occorrenza della variabile x è detta *legante*. Lo *scope* di questa (occorrenza di) variabile legante è il *corpo* (*body*) t. Le occorrenze (non già legate a livello più interno) di x dentro t sono dette *legate* (all'occorrenza nell'astrazione). Le occorrenze di x in un termine che non sono nello scope di una astrazione sono dette essere *libere*. Quali sono le occorrenze di variabili libere nei due termini seguenti?

$\lambda\mathtt{x}.\lambda\mathtt{y}.\mathtt{x}\ \mathtt{y}\ \mathtt{z}$

$\lambda\mathtt{x}.(\lambda\mathtt{y}.\mathtt{z}\ \mathtt{y})\ \mathtt{y}$

Per esercizio, definire formalmente l'insieme $FV(t)$ delle variabili libere di un termine t e la dimensione $dim(t)$ di un termine t (analogamente a quanto visto per $\mathcal{E}$), e provare che per ogni t si ha $\mid FV(t) \mid \leq dim(t)$.

3.3 Regole di riduzione

Definiamo i valori nel modo seguente:

$$v \quad ::= \quad \lambda x.t$$

Le regole di riduzione sono date in Fig.4. L'unica regola di computazione è (APPABS^v), mentre (APP1) e (APP2) sono regole di congruenza.

$$\text{(APPABS}^v\text{)}\ \frac{}{(\lambda x.t)\ v \rightarrow t[v/x]} \qquad \text{(APP1)}\ \frac{t_1 \rightarrow t_1'}{t_1\ t_2 \rightarrow t_1'\ t_2} \qquad \text{(APP2)}\ \frac{t_2 \rightarrow t_2'}{v\ t_2 \rightarrow v\ t_2'}$$

Figura 4: Regole di riduzione per il lambda-calcolo *call-by-value*

Queste definizioni formalizzano la seguente strategia di valutazione, che corrisponde a quanto avviene in genere nei linguaggi di programmazione: le definizioni di funzione non vengono mai semplificate; in un'applicazione (chiamata di funzione), si valuta prima il termine che denota la funzione, poi il termine che denota l'argomento, e solo quando entrambi sono stati valutati si può effettivamente eseguire la chiamata. In particolare, la regola (APPABS^v) è una versione ristretta (in cui si richiede che l'argomento della lambda-astrazione sia un valore) della seguente regola, che come già detto si chiama anche *β-regola*:

$$\text{(APPABS)}\ \frac{}{(\lambda x.t_1)\ t_2 \rightarrow t_1[t_2/x]}$$

dove $t_1[t_2/x]$ è il termine che si ottiene dalla sostituzione delle occorrenze (libere) di x in t_1 con t_2.
Definiamo formalmente $t_1[t_2/x]$ (per induzione strutturale su t_1):

- $x[t/x] = t$
- $y[t/x] = y$ se $x \neq y$

- $(\lambda y.t_1)[t_2/x] = \lambda y.(t_1[t_2/x])$ se $x \neq y$, $y \notin FV(t_2)$
- $(t_1\ t_2)[t/x] = t_1[t/x]\ t_2[t/x]$

Si noti che nella terza regola la condizione $x \neq y$ è necessaria perché solo le occorrenze libere di x devono essere sostituite, altrimenti si avrebbe per esempio la seguente derivazione (intuivamente errata):

$$(\lambda \mathtt{x}.\lambda \mathtt{x}.\mathtt{x})\ \mathtt{z} \not\to \lambda \mathtt{x}.\mathtt{z}$$

Inoltre, la condizione $y \notin FV(t_2)$ è necessaria per non catturare, effettuando la sostituzione, delle occorrenze libere, altrimenti si avrebbe per esempio la seguente derivazione (intuivamente errata):

$$(\lambda \mathtt{x}.\lambda \mathtt{z}.\mathtt{x})\ \mathtt{z} \not\to \lambda \mathtt{z}.\mathtt{z}$$

Queste condizioni non sono restrittive, perché possono sempre essere garantite effettuando un'opportuna ridenominazione di una variabile legante e tutte le occorrenze legate a essa in un termine, cioè applicando la cosiddetta α*-regola.* Nei due esempi sopra si ottengono le seguenti derivazioni (intuitivamente corrette):

$$(\lambda \mathtt{x}.\lambda \mathtt{x}.\mathtt{x})\ \mathtt{z} \stackrel{\alpha}{=} (\lambda \mathtt{x}.\lambda \mathtt{x}'.\mathtt{x}')\ \mathtt{z} \to \lambda \mathtt{x}'.\mathtt{x}'$$
$$(\lambda \mathtt{x}.\lambda \mathtt{z}.\mathtt{x})\ \mathtt{z} \stackrel{\alpha}{=} (\lambda \mathtt{x}.\lambda \mathtt{z}'.\mathtt{x})\ \mathtt{z} \to \lambda \mathtt{z}'.\mathtt{z}$$

Un termine della forma $(\lambda x.t_1)\ t_2$, cioè una λ-astrazione applicata a un termine, è detto *redex* ("espressione riducibile").
La strategia di valutazione che abbiamo scelto, detta *chiamata per valore* (*call-by-value*), riflette come già detto la convenzione della maggior parte dei linguaggi di programmazione (una funzione è un valore, quindi non se ne valuta il corpo, e prima di effettuare l'applicazione si valutano gli argomenti). Vi sono tuttavia altre strategie di valutazione interessanti, per esempio:

- *beta-riduzione piena* (si può ridurre qualunque redex in modo non deterministico),
- *ordine normale* (si riduce prima il redex più a sinistra e più esterno),
- *chiamata per nome* (*call-by-name*) (come l'ordine normale ma non si riduce dentro una λ-astrazione).

Vediamo un esempio che illustra le differenze tra le diverse strategie: $id\ (id\ \lambda \mathtt{z}.id\ \mathtt{z})$ dove $id = \lambda \mathtt{x}.\mathtt{x}$. In questo termine sono presenti i tre redex sottolineati:

1. $\underline{id\ (id\ \lambda \mathtt{z}.id\ \mathtt{z})}$
2. $id\ \underline{(id\ \lambda \mathtt{z}.id\ \mathtt{z})}$
3. $id\ id\ \lambda \mathtt{z}.\underline{id\ \mathtt{z}}$

Nella stategia per valore si ha, riducendo a ogni passo il redex sottolineato:

$$\begin{array}{l} \underline{id\ (id\ \lambda \mathtt{z}.id\ \mathtt{z})} \to \\ \underline{id\ \lambda \mathtt{z}.id\ \mathtt{z}} \to \\ \lambda \mathtt{z}.id\ \mathtt{z} \end{array}$$

Si noti che a ogni passo vi è un'unica scelta possibile. Per esempio, al primo passo non è possibile ridurre il redex (1) in quanto il suo argomento non è ancora stato valutato, nè il redex (3) in quanto si trova dentro una lambda-astrazione.
La sequenza di riduzione sopra è ovviamente anche una possibile sequenza di riduzione nella strategia di beta-riduzione piena, nella quale però è anche possibile per esempio la seguente, che corrisponde all'ordine normale:

$$\begin{array}{l} \underline{id\ (id\ \lambda \mathtt{z}.id\ \mathtt{z})} \to \\ \underline{id\ \lambda \mathtt{z}.id\ \mathtt{z}} \to \\ \lambda \mathtt{z}.\underline{id\ \mathtt{z}} \\ \lambda \mathtt{z}.\mathtt{z} \end{array}$$

Infine, nella strategia *call-by-name* si hanno i primi tre passi di riduzione sopra, ma non l'ultimo.

Esercizio 3.1 Quali delle proprietà descritte nell'esercizio 2.10 valgono nel lambda-calcolo?

Esercizio 3.2 Si dia una semantica big-step per il lambda-calcolo, analogamente a quanto visto nell'esercizio 2.11.

3.4 Programmare nel lambda-calcolo

Currying. Come visto nel corso di L. P. per il linguaggio Caml, una funzione $\lambda x.\lambda y.t$ "ha lo stesso comportamento" di una funzione che prende come argomento una coppia. La trasformazione che, data una funzione che prende come argomento una coppia, restituisce una funzione che dato un argomento restituisce un'altra funzione di un argomento è chiamata *currying*. Poiché il λ-calcolo (puro) ha solo funzioni di un argomento, tutte le funzioni di più di un argomento saranno scritte in forma curried.
Per esempio (assumendo di aggiungere costanti e operatori sugli interi):

$sum = \lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{x} + \mathtt{y}$
$sum\ 2 \rightarrow \lambda \mathtt{y}.2 + \mathtt{y}$
$sum\ 2\ 3 \rightarrow (\lambda \mathtt{y}.2 + \mathtt{y})\ 3 \rightarrow 2 + 3$

Booleani di Church. I seguenti due termini del lambda-calcolo puro "simulano" i valori booleani.

$c - true = \lambda \mathtt{t}.\lambda \mathtt{f}.\mathtt{t}$
$c - false = \lambda \mathtt{t}.\lambda \mathtt{f}.\mathtt{f}$

Assumendo che v_1 e v_2 siano due valori arbitrari, si hanno le seguenti sequenze di riduzione:

	$c - true\ v_1\ v_2$	
$=$	$\underline{(\lambda t.\lambda f.t)\ v_1}\ v_2$	per definizione
$\rightarrow$	$\underline{(\lambda f.v_1)\ v_2}$	riducendo il redex sottolineato
$\rightarrow$	v_1	riducendo il redex sottolineato
	$c - false\ v_1\ v_2$	
$=$	$\underline{(\lambda t.\lambda f.f)\ v_1}\ v_2$	per definizione
$\rightarrow$	$\underline{(\lambda f.f)\ v_2}$	riducendo il redex sottolineato
$\rightarrow$	v_2	riducendo il redex sottolineato

Il seguente termine funziona da condizionale, cioè, applicato a tre valori b, v_1 e v_2, si riduce a v_1 se b è $c - true$ e a v_2 se b è $c - false$.

c-if $= \lambda \mathtt{b}.\lambda \mathtt{x}.\lambda \mathtt{y}.\mathtt{b}\ \mathtt{x}\ \mathtt{y}$

Vediamo altri esempi di funzioni sui booleani di Church:

c-not $= \lambda \mathtt{b}.\mathtt{b}\ c - false\ c - true$

Cioè *c-not* è una funzione che, dato un booleano di Church *b*, restituisce $c - false$ se *b* è $c - true$ e $c - true$ se *b* è $c - false$.

c-and $= \lambda \mathtt{b}_1.\lambda \mathtt{b}_2.\mathtt{b}_1\ \mathtt{b}_2\ c - false$

Cioè *c-and* è una funzione che, dati due booleani di Church b_1 e b_2, restituisce b_2 se b_1 è $c - true$ e $c - false$ se b_1 è $c - false$.

Numerali di Church. Analogamente, i seguenti termini "simulano" i numeri naturali. L'idea base è quella di rappresentare il numero naturale n con una funzione che "ripete una certa azione n volte."

$c_0 = \lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{z}$
$c_1 = \lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{s}\ \mathtt{z}$
$c_2 = \lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{s}\ (\mathtt{s}\ \mathtt{z})$
$c_3 = \lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{s}\ (\mathtt{s}\ (\mathtt{s}\ \mathtt{z}))$
...

Cioè, ogni numero n è reppresentato dal termine c_n che prende due argomenti `s` e `z` (per "successore" e "zero"), e applica n volte `s` a `z`.
Vediamo alcuni esempi di funzioni sui numerali di Church.

Successore *c-succ* $= \lambda \mathtt{n}.\lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$ oppure $\lambda \mathtt{n}.\lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{n}\ \mathtt{s}\ (\mathtt{s}\ \mathtt{z})$

Somma *c-sum* $= \lambda \mathtt{m}.\lambda \mathtt{n}.\lambda \mathtt{s}.\lambda \mathtt{z}.\mathtt{m}\ \mathtt{s}\ (\mathtt{n}\ \mathtt{s}\ \mathtt{z})$

Si noti che *c-sum* n è la funzione che aggiunge n all'argomento. Sfruttando questo fatto possiamo dare la prima delle due definizioni sotto di prodotto.

Prodotto *c-times* $= \lambda\mathtt{m}.\lambda\mathtt{n}.\mathtt{m}\ (c\text{-}sum\ \mathtt{n})\mathtt{c}_0$ oppure $\lambda\mathtt{m}.\lambda\mathtt{n}.\lambda\mathtt{s}.\lambda\mathtt{z}.\mathtt{m}(\mathtt{n}\ \mathtt{s})\ \mathtt{z}$

Test su zero *c-iszero* $= \lambda\mathtt{m}.\mathtt{m}\ (\lambda\mathtt{x}.c - false)\ c - true$

Conversioni Nel lambda-calcolo esteso con le espressioni booleane e naturali di $\mathcal{E}$ possiamo definire i seguenti operatori di conversione tra booleani e naturali e le loro rappresentazioni di Church:

$\mathtt{church2bool} = \lambda\mathtt{b}.\mathtt{b}\ \mathtt{true}\ \mathtt{false}$
$\mathtt{bool2church} = \lambda\mathtt{b}.\mathtt{if}\ \mathtt{b}\ \mathtt{then}\ c - true\ \mathtt{else}\ c - false$
$\mathtt{church2nat} = \lambda\mathtt{n}.\mathtt{n}\ (\lambda\mathtt{x}.\mathtt{succ}\ x)\ \mathtt{0}$

Non è invece banale dare un operatore di conversione `nat2church` che trasformi un naturale nel corrispondente numerale di Church; infatti per definirlo è necessario utilizzare la ricorsione, vedi nel seguito.

Si noti che gli operatori sui booleani e numerali di Church dati sopra funzionano correttamente "modulo equivalenza osservazionale", nel senso che in genere non si ottiene *esattamente* il termine voluto, ma un termine che si comporta allo stesso modo: per esempio il termine *c-succ* c_0 non si riduce a c_1, ma al seguente termine:

$\lambda\mathtt{s}.\lambda\mathtt{z}.\mathtt{s}((\lambda\mathtt{s}'.\lambda\mathtt{z}'.\mathtt{z}')\mathtt{s}\ \mathtt{z})$

(si noti che questo termine si ridurrebbe a c_1 se fosse ammesso ridurre dentro una lambda-astrazione).

Divergenza e operatore di punto fisso. La relazione di riduzione definita per il lambda-calcolo *non* è terminante. Per esempio il termine

$\omega = (\lambda\mathtt{x}.\mathtt{x}\ \mathtt{x})\ (\lambda\mathtt{x}.\mathtt{x}\ \mathtt{x})$

si riduce a se stesso in un passo, quindi la valutazione di ω non raggiunge mai una forma normale: si dice che *diverge*.
Il termine

$\mathtt{fix} = \lambda\mathtt{f}.(\lambda\mathtt{x}.\mathtt{f}\ (\lambda\mathtt{y}.\mathtt{x}\ \mathtt{x}\ \mathtt{y}))\ (\lambda\mathtt{x}.\mathtt{f}\ (\lambda\mathtt{y}.\mathtt{x}\ \mathtt{x}\ \mathtt{y}))$

è detto *operatore di punto fisso* e può essere utilizzato per simulare la ricorsione. Infatti, nel lambda-calcolo puro non vi sono costrutti sintattici per associare un nome a una lambda-astrazione, a maggior ragione non in modo ricorsivo. Quindi, non è chiaro a priori come poter definire funzioni ricorsive. Per esempio la seguente definizione di una funzione (nel lambda-cacolo esteso con $\mathcal{E}$) che controlla se un numero naturale è pari:

```
even =
    λn.
        if iszero n then true
        else if iszero pred n then false
        else even (pred pred n)
```

non è una buona definizione, in quanto nel corpo della funzione si usa il nome $even$ della funzione che si sta definendo. Più precisamente, dato che attribuire un nome a un termine è qualcosa che facciamo solo al metalivello, nel termine a destra $even$ risulta essere una variabile libera, quindi non si ha un termine chiuso.[4]
Tuttavia, è possibile ottenere un lambda termine che si comporta come la funzione voluta nel modo seguente. Definiamo una funzione che "astrae" rispetto alla chiamata interna a $even$, prendendo un argomento aggiuntivo.

```
Even =
    λeven.
        λn.
            if iszero n then true
            else if iszero pred n then false
            else even (pred pred n)
```

In questo modo, il termine a destra risulta correttamente chiuso (non ci sono più variabili libere). La funzione desiderata si ottiene allora applicando l'operatore `fix` a $Even$. Possiamo verificare che il comportamento della funzione è quello voluto applicando il risultato a un argomento, come illustrato sotto,

[4] Si noti che, interpretando quella sopra come definizione induttiva, la funzione desiderata $even$ è la più piccola funzione che *rende vera* l'uguaglianza (ossia, che è un *punto fisso* di tale uguaglianza).

$\texttt{fix}\ \mathit{Even}\,(\texttt{succ succ 0}) =$
$\lambda\texttt{f}.(\lambda\texttt{x}.\texttt{f}\,(\lambda\texttt{y}.\texttt{x x y}))\,(\lambda\texttt{x}.\texttt{f}\,(\lambda\texttt{y}.\texttt{x x y}))\,\mathit{Even}\,(\texttt{succ succ 0}) \to$
$(\lambda\texttt{x}.\mathit{Even}\,(\lambda\texttt{y}.\texttt{x x y}))\,(\lambda\texttt{x}.\mathit{Even}\,(\lambda\texttt{y}.\texttt{x x y}))\,(\texttt{succ succ 0}) =$ (abbreviando $h = \lambda\texttt{x}.\mathit{Even}\,(\lambda\texttt{y}.\texttt{x x y})$)
$h\,h\,(\texttt{succ succ 0}) \to$
$\mathit{Even}\,(\lambda\texttt{y}.h\,h\,\texttt{y})\,(\texttt{succ succ 0}) \to$
$(\lambda\texttt{n}.\texttt{if iszero n then true else if iszero pred n then false else}\,(\lambda\texttt{y}.h\,h\,\texttt{y})\,(\texttt{pred pred n}))\,(\texttt{succ succ 0}) \to^\star$
$(\lambda\texttt{y}.h\,h\,\texttt{y})\,\texttt{0} \to$
$h\,h\,\texttt{0}$

4 Sistemi di tipi

In questo capitolo vedremo come definire un *type system* (sistema di tipi) per un calcolo, ossia definire formalmente un sottoinsieme dei termini del linguaggio, detti *ben formati* o *ben tipati*, in modo tale che i termini ben tipati non possano dar luogo a errori a run-time (si dice in questo caso che il type system è *sound*). Tipicamente questo si ottiene per mezzo di una classificazione dei termini in diversi *tipi*, in modo tale che sia possibile controllare che gli operatori del linguaggio siano sempre applicati in modo coerente con tali tipi.

4.1 Sistema di tipi per $\mathcal{E}$

Nel linguaggio $\mathcal{E}$, possiamo classificare i termini in quelli di tipo booleano e quelli di tipo naturale.

$$T \quad ::= \quad \texttt{Bool} \mid \texttt{Nat}$$

Le regole di tipo sono date in Fig.5.

$$\text{(T-True)}\ \frac{}{\texttt{true} : \texttt{Bool}} \qquad \text{(T-False)}\ \frac{}{\texttt{false} : \texttt{Bool}} \qquad \text{(T-If)}\ \frac{t : \texttt{Bool} \quad t_1 : T \quad t_2 : T}{\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 : T}$$

$$\text{(T-Zero)}\ \frac{}{\texttt{0} : \texttt{Nat}} \qquad \text{(T-Succ)}\ \frac{t : \texttt{Nat}}{\texttt{succ}\ t : \texttt{Nat}}$$

$$\text{(T-Pred)}\ \frac{t : \texttt{Nat}}{\texttt{pred}\ t : \texttt{Nat}} \qquad \text{(T-IsZero)}\ \frac{t : \texttt{Nat}}{\texttt{iszero}\ t : \texttt{Bool}}$$

Figura 5: Regole di tipo per $\mathcal{E}$

Queste (meta)regole definiscono induttivamente una relazione $t : T$ tra termini e tipi. È possibile provare (per induzione strutturale) che questa relazione è in realtà in questo caso una funzione, cioè ogni termine ha al più un tipo (questa proprietà non vale sempre, per esempio non vale in linguaggi con una nozione di sottotipo).
Il sistema di tipi fornisce una stima "pessimistica" dell'esecuzione, nel senso che tutti i termini ben tipati non causano errori a run-time, ma non vale in genere il viceversa. Per esempio, usando le regole sopra non possiamo assegnare un tipo al termine

$$\texttt{if true then 0 else false}$$

anche se questo si riduce a un valore naturale.
La *correttezza* (o *soundness*) del type system garantisce che la valutazione di un termine ben tipato non possa portare a un termine bloccato, e può essere espressa formalmente come segue.

Teorema 4.1 [Soundness] Assumiamo che t sia un termine ben tipato, cioè $t : T$ per qualche T. Per ogni t' tale che $t \to^\star t'$, o t' è un valore oppure $t' \to t''$ per qualche t'' (scriveremo anche semplicemente $t' \to$).

In genere, la soundness viene dimostrata provando le due seguenti proprietà:

Progress un termine ben tipato non si riduce a un termine bloccato, formalmente:

se $t : T$ allora o t è un valore o $t \to$.

Conservazione dei tipi (anche detta *subject reduction*) la relazione di riduzione trasforma termini ben tipati in termini ben tipati, formalmente:

se $t : T$ e $t \to t'$ allora $t' : T$.

In generale, non è detto che il tipo si conservi *esattamente*, per esempio nel caso di linguaggi con relazione di sottotipo si ha tipicamente che un termine di tipo T può ridursi in un termine di un sottotipo di T.
È facile vedere che le proprietà di progress e conservazione dei tipi implicano la soundness, per induzione aritmetica sulla lunghezza della riduzione.

$t \to^0 t'$ Allora t coincide con t', e la tesi segue dal progress.

$t \to^{n+1} t'$ Allora $t \to t'' \to^n t'$. Dalla conservazione dei tipi sappiamo che $t'' : T$, e quindi per ipotesi induttiva segue la tesi.

Analogamente a quanto visto per la relazione di riduzione, possiamo provare che un *judgment* (*giudizio di tipo*) $t : T$ è valido con un *albero di prova*, come illustrato dal seguente esempio.

$$\text{(T-If)}\ \frac{\text{(T-IsZero)}\ \dfrac{\text{(T-Zero)}\ \dfrac{}{\texttt{0} : \texttt{Nat}}}{\texttt{iszero 0} : \texttt{Bool}} \quad \text{(T-Zero)}\ \dfrac{}{\texttt{0} : \texttt{Nat}} \quad \text{(T-Pred)}\ \dfrac{\text{(T-Zero)}\ \dfrac{}{\texttt{0} : \texttt{Nat}}}{\texttt{pred 0} : \texttt{Nat}}}{\texttt{if iszero 0 then 0 else pred 0} : \texttt{Nat}}$$

Per provare le proprietà di progress e conservazione dei tipi per $\mathcal{E}$, diamo prima alcuni lemmi.
Il lemma di *inversione* o di *generazione* dice, per ogni termine ben tipato, cosa sappiamo sul suo tipo e sui tipi dei suoi sottotermini.

Lemma 4.2 [Inversione]

1. Se $\texttt{true} : T$ allora $T = \texttt{Bool}$.
2. Se $\texttt{false} : T$ allora $T = \texttt{Bool}$.
3. Se $\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 : T$ allora $t : \texttt{Bool}$, $t_1 : T$ e $t_2 : T$.
4. Se $\texttt{0} : T$ allora $T = \texttt{Nat}$.
5. Se $\texttt{succ}\ t : T$ allora $T = \texttt{Nat}$, $t : \texttt{Nat}$.
6. Se $\texttt{pred}\ t : T$ allora $T = \texttt{Nat}$, $t : \texttt{Nat}$.
7. Se $\texttt{iszero}\ t : T$ allora $T = \texttt{Bool}$, $t : \texttt{Nat}$.

Dimostrazione. Immediata. □
Il lemma di inversione fornisce direttamente un algoritmo ricorsivo per il calcolo del tipo di un termine (se esiste).

```
typeof t  =  if t = true then Bool
          else if t = false then Bool
          else if t = (if t1 then t2 else t3) then
             let T1 = typeof(t1) in
             let T2 = typeof(t2) in
             let T3 = typeof(t3) in
                if T1 = Bool and T2 = T3 then T2
                else "Non tipabile"
          else if t = 0 then Nat
          else if t = succ t1 then
             let T1 = typeof(t1) in
                if T1 = Nat then Nat else "Non tipabile"
          else if t = pred t1 then
             let T1 = typeof(t1) in
                if T1 = Nat then Nat else "Non tipabile"
          else if t = iszero t1 then
             let T1 = typeof(t1) in
                if T1 = Nat then Bool else "Non tipabile"
```

Il lemma *delle forme canoniche* dice, per ogni tipo, quali sono i valori di quel tipo.

Lemma 4.3 [Forme canoniche]

1. Se v è un valore di tipo `Bool` allora $v = \texttt{true}$ oppure $v = \texttt{false}$.
2. Se v è un valore di tipo `Nat` allora v è della forma n, con

$$n \quad ::= \quad 0 \mid \texttt{succ}\ n$$

Dimostrazione. Facile verifica. □

Teorema 4.4 [Progress] Assumiamo che t sia un termine ben tipato (cioè $t : T$ per qualche T). Allora o t è un valore oppure $t \rightarrow$.

Dimostrazione. Per induzione sulla relazione di tipaggio[5]. Vediamo alcuni casi.

(T-TRUE), (T-FALSE), (T-ZERO) Immediati poiché in questi casi t è un valore.

(T-IF) Si ha $\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 : T$, $t : \texttt{Bool}$, $t_1 : T$ e $t_2 : T$.

Per ipotesi induttiva, o t è un valore oppure esiste un t' tale che $t \rightarrow t'$.

Se t è un valore allora il lemma sulle forme canoniche ci dice che $t = \texttt{true}$ oppure $t = \texttt{false}$, quindi si applicano o la regola (IFTRUE) o la regola (IFFALSE).

D'altra parte, se $t \rightarrow t'$, dalla regola (IF) deriviamo che $\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \rightarrow \texttt{if}\ t'\ \texttt{then}\ t_1\ \texttt{else}\ t_2$. □

Teorema 4.5 [Conservazione dei tipi] Assumiamo che t sia un termine ben tipato (cioè $t : T$ per qualche T). Allora $t \rightarrow t'$ implica $t' : T$.

Dimostrazione. Per induzione sulla relazione di riduzione[6]. Vediamo alcuni casi.

(IF-TRUE) Si ha $\texttt{if true then}\ t_1\ \texttt{else}\ t_2 \rightarrow t_1$. Dall'ipotesi che $\texttt{if true then}\ t_1\ \texttt{else}\ t_2 : T$ e dal lemma di inversione punto 3 abbiamo che $t_1 : T$, quindi si ha la tesi. Analogamente se la regola applicata è (IF-FALSE).

(IF) Si ha $\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \rightarrow \texttt{if}\ t'\ \texttt{then}\ t_1\ \texttt{else}\ t_2$ e $t \rightarrow t'$. Dall'ipotesi che $\texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 : T$ e dal lemma di inversione punto 3 abbiamo che $t : \texttt{Bool}$, $t_1 : T$ e $t_2 : T$. Applicando l'ipotesi induttiva a $t \rightarrow t'$ e $t : \texttt{Bool}$ deriviamo che $t' : \texttt{Bool}$, per cui applicando (T-IF) deriviamo $\texttt{if}\ t'\ \texttt{then}\ t_1\ \texttt{else}\ t_2 : T$.

(SUCC) Si ha $\texttt{succ}\ t \rightarrow \texttt{succ}\ t'$ e $t \rightarrow t'$. Dall'ipotesi che $\texttt{succ}\ t : T$ e dal lemma di inversione punto 5 abbiamo che $\texttt{succ}\ t : \texttt{Nat}$ e $t : \texttt{Nat}$. Applicando l'ipotesi induttiva a $t \rightarrow t'$ e $t : \texttt{Nat}$ deriviamo che $t' : \texttt{Nat}$, per cui dalla regola (T-SUCC) deriviamo che $\texttt{succ}\ t' : \texttt{Nat}$. Analogamente se la regola applicata è (PRED). □

4.2 Lambda calcolo con tipi semplici

Consideriamo ora il problema di definire un sistema di tipi per il lambda calcolo, assumendo per concretezza di estenderlo con dei costrutti sintattici relativi a dei tipi base, per esempio (il sottoinsieme booleano di) $\mathcal{E}$. Vi sono due possibili approcci. Nel primo, più semplice, che seguiremo in questo corso, si definisce una sintassi leggermente diversa in cui le variabili nelle λ-astrazioni sono annotate esplicitamente con il loro tipo, come illustrato sotto.

$$\begin{array}{lll} t & ::= & x \mid \lambda x : T.t \mid t_1\ t_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \mid \ldots \\ v & ::= & \lambda x.t \mid \texttt{true} \mid \texttt{false} \mid \ldots \end{array}$$

Questa presentazione del lambda-calcolo tipato è detta *alla Church* o *esplicitamente tipata*. I tipi sono i *tipi funzionali* costruiti a partire dai tipi base considerati, per esempio solo `Bool`.

$$T \quad ::= \quad \texttt{Bool} \mid \ldots \mid T_1 \rightarrow T_2$$

[5] Questo significa che, per ogni regola di tipo, si prova che se la proprietà vale per le premesse allora vale per la conseguenza.

[6] Questo significa che, per ogni regola di riduzione, si prova che se la proprietà vale per le premesse allora vale per la conseguenza. È possibile alternativamente dare una prova per induzione sulla relazione di tipaggio.

In questo approccio, non vi sono tipi *polimorfi*, come avviene invece in linguaggi funzionali come Caml, quindi vi è, per esempio, una funzione identità per ogni tipo: $\lambda \mathtt{x} : \mathtt{Bool}.\mathtt{x}$ è diversa da $\lambda \mathtt{x} : \mathtt{Bool} \to \mathtt{Bool}.\mathtt{x}$. Il vantaggio è che i tipi hanno una semplice struttura induttiva e, come vedremo, è possibile dare un semplice algoritmo che trova, se c'è, il tipo di un termine in un dato contesto.
Nell'approccio alternativo, che corrisponde a ciò che accade nei linguaggi con polimorfismo come Caml, si mantiene invece la sintassi senza annotazioni di tipo. Questa presentazione del lambda-calcolo tipato è detta *alla Curry* o implicitamente tipata. In questo caso, per esempio, vi è un'unica funzione identità $\lambda \mathtt{x}.\mathtt{x}$ che ha come *tipo più generale* il tipo polimorfo $\alpha \to \alpha$. Ovviamente questa versione del calcolo tipato è più potente (come vedremo meglio nel seguito), ma occorre introdurre dei tipi più complessi (in particolare, *variabili di tipo* come α) e soprattutto l'algoritmo per ricavare il tipo (più generale) di un termine in un dato contesto non è banale. In questo corso ci limiteremo quindi al primo approccio.
Le regole di tipo, date in Fig.6, definiscono una relazione di tipaggio della forma $\Gamma \vdash t : T$ dove Γ è un *contesto di tipo* o *ambiente (statico)*. Il contesto di tipo è necessario (a differenza di quanto visto per $\mathcal{E}$) in quanto un termine contenente variabili libere non è ben formato se considerato come "programma" (cioè, a livello top), ma è ben formato se inserito come sottotermine in un contesto che fornisca definizioni del tipo appropriato per tali variabili.
Un contesto di tipo Γ è una sequenza di coppie $x : T$ tale che, per ogni variabile x, c'è al più un T per cui $x : T \in \Gamma$ (quindi Γ rappresenta una funzione parziale da variabili a tipi). Indichiamo con $\emptyset$ il contesto vuoto, e scriviamo anche $t : T$ invece di $\emptyset \vdash t : T$. Ricordiamo che $\Gamma[T/x]$ indica la funzione ottenuta da Γ aggiungendo l'associazione tra x e T, eventualmente cancellando precedenti associazioni per x (vedi Def.A.2 nell'Appendice).

$$\text{(T-TRUE)}\ \frac{}{\Gamma \vdash \mathtt{true} : \mathtt{Bool}} \qquad \text{(T-FALSE)}\ \frac{}{\Gamma \vdash \mathtt{false} : \mathtt{Bool}}$$

$$\text{(T-IF)}\ \frac{\Gamma \vdash t : \mathtt{Bool} \quad \Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \mathtt{if}\ t\ \mathtt{then}\ t_1\ \mathtt{else}\ t_2 : T} \qquad \text{(T-VAR)}\ \frac{}{\Gamma \vdash x : T}\ \Gamma(x) = T$$

$$\text{(T-ABS)}\ \frac{\Gamma[T_1/x] \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2} \qquad \text{(T-APP)}\ \frac{\Gamma \vdash t_1 : T_2 \to T \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1\ t_2 : T}$$

Figura 6: Regole di tipo per il lambda-calcolo con tipi semplici

Come al solito, possiamo provare formalmente che un judgment (giudizio di tipo) $\Gamma \vdash t : T$ è valido dando un albero di prova.

Esercizio 4.6 Provare formalmente la validità dei seguenti judgment:

- $\Gamma \vdash (\lambda \mathtt{x} : \mathtt{Bool}.\mathtt{x})\,\mathtt{true} : \mathtt{Bool}$
- $\Gamma \vdash \mathtt{f\ if\ false\ then\ true\ else\ true} : \mathtt{Bool}$, con $\Gamma = \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}$
- $\Gamma \vdash \lambda \mathtt{x} : \mathtt{Bool}.\mathtt{f\ if\ false\ then\ true\ else\ true} : \mathtt{Bool} \to \mathtt{Bool}$, con $\Gamma = \mathtt{f} : \mathtt{Bool} \to \mathtt{Bool}$.

Esercizio 4.7 Trovare un contesto di tipo Γ per cui al termine `f x y` sia assegnato il tipo `Bool`, cioè tale che $\Gamma \vdash \mathtt{f\ x\ y} : \mathtt{Bool}$. Possiamo caratterizzare tutti questi contesti di tipo?
È possibile trovare un contesto di tipo Γ e un tipo T tali che $\Gamma \vdash \mathtt{x\ x} : T$?

Come in precedenza, le proprietà fondamentali del type system che giustificano la sua correttezza (soundness) rispetto alla semantica operazionale sono:

Progress se $t : T$, allora o t è un valore oppure esiste un t' tale che $t \to t'$.

Conservazione dei tipi se $\Gamma \vdash t : T$ e $t \to t'$ allora $\Gamma \vdash t' : T$.

Si noti che in questo caso la proprietà di progress (e quindi la soundness) vale solo per i termini *chiusi* (infatti i termini aperti possono ovviamente restare bloccati). Tuttavia, la proprietà di conservazione dei tipi va formulata su tutti i termini per poterla provarla induttivamente, in quanto i sottotermini di un termine chiuso possono essere aperti.
Come in precedenza, la prova è strutturata nel modo seguente:

- prova del lemma di inversione
- prova del lemma delle forme canoniche

- prova dei teoremi di progress e conservazione dei tipi.

Lemma 4.8 [Inversione]

1. Se $\Gamma \vdash \texttt{true} : T$ allora $T = \texttt{Bool}$.
2. Se $\Gamma \vdash \texttt{false} : T$ allora $T = \texttt{Bool}$.
3. Se $\Gamma \vdash \texttt{if}\ t\ \texttt{then}\ t_1\ \texttt{else}\ t_2 : T$ allora $\Gamma \vdash t : \texttt{Bool}$, $\Gamma \vdash t_1 : T$ e $\Gamma \vdash t_2 : T$.
4. Se $\Gamma \vdash x : T$ allora $x : T \in \Gamma$.
5. Se $\Gamma \vdash \lambda x : T_1.t : T$ allora $T = T_1 \to T_2$ e $\Gamma[T_1/x] \vdash t : T_2$ per qualche T_2.
6. Se $\Gamma \vdash t_1\ t_2 : T$ allora $\Gamma \vdash t_1 : T_2 \to T$ e $\Gamma \vdash t_2 : T_2$ per qualche T_2.

Lemma 4.9 [Forme canoniche]

1. Se v è un valore di tipo $\texttt{Bool}$, allora v è $\texttt{true}$ oppure v è $\texttt{false}$.
2. Se v è un valore di tipo $T_1 \to T_2$, allora v è della forma $\lambda x : T_1.t$.

Teorema 4.10 [Progress] Assumiamo che t sia un termine chiuso e ben tipato, cioè $t : T$ per qualche T. Allora o t è un valore oppure $t \to t'$ per qualche t'.

Dimostrazione. Per induzione sulla relazione di tipaggio.
I casi delle espressioni booleane sono come prima, il caso delle variabili è banale (perché non sono termini chiusi), il caso delle astrazioni è banale (perché sono valori).
Consideriamo il caso dell'applicazione. Si ha $t_1\ t_2 : T$, $t_1 : T_2 \to T$ e $t_2 : T_2$. Per ipotesi induttiva o t_1 è un valore oppure può fare un passo di riduzione, analogamente per t_2. Se t_1 può fare un passo di riduzione allora si può applicare la regola (APP1). Se t_1 è un valore e t_2 può fare un passo di riduzione si può applicare la regola (APP2). Se sia t_1 che t_2 sono valori, dal lemma delle forme canoniche punto 2, $t_1 = \lambda x : T_2.t$ e quindi è applicabile la regola (APPABS^v). □

Teorema 4.11 [Conservazione dei tipi] Se $\Gamma \vdash t : T$ e $t \to t'$ allora $\Gamma \vdash t' : T$.

Dimostrazione. Per induzione sulla relazione di riduzione.
I casi delle espressioni booleane sono come prima. Consideriamo le regole del lambda-calcolo, cioè (APPABS^v), (APP1) e (APP2), applicate a $t_1\ t_2$. Dall'ipotesi che $\Gamma \vdash t_1\ t_2 : T$ e dal lemma di inversione punto 6 abbiamo che $\Gamma \vdash t_1 : T_2 \to T$ e $\Gamma \vdash t_2 : T_2$ per qualche T_2. Se le regole applicate sono (APP1) o (APP2) applichiamo semplicemente l'ipotesi induttiva (una volta a $t_1 \to t_1'$ e l'altra a $t_2 \to t_2'$). Resta il caso che la regola applicata sia (APPABS^v). Allora

- $t_1 = \lambda x : T_2.t$,
- t_2 è un valore v,
- $t_1\ t_2 \to t[v/x]$

Dall'ipotesi che $\Gamma \vdash \lambda x : T_2.t : T_2 \to T$ e dal lemma di inversione punto 5 sappiamo che $\Gamma[T_2/x] \vdash t : T$. Per concludere, dobbiamo applicare il seguente lemma (che non proveremo). □

Lemma 4.12 [Sostituzione] I tipi sono conservati per sostituzione, formalmente:

se $\Gamma[T_1/x] \vdash t : T$ e $\Gamma \vdash t' : T_1$, allora $\Gamma \vdash t[t'/x] : T$.

Analogamente a quanto visto per $\mathcal{E}$, il lemma di inversione fornisce direttamente un algoritmo che calcola il tipo, se esiste, di un termine in un dato contesto.

```
typeof (ctx , t) =
          if t = true oppure t = false then return Bool
          else if t = (if t1 then t2 else t3) then
             let T1 = typeof (ctx, t1) in
             let T2 = typeof(ctx, t2) in
             let T3 = typeof(ctx, t3) in
```

```
        if T1 = Bool and T2 = T3 then return T2
        else "Non tipabile"
else if t = x then if apply(ctx,x) = T
   then return T
   else "Non tipabile"
else if t = (lambda x:T1.t2) then
   let T2 = typeof(update(ctx,x,T1),t2) in
      return T1->T2
else if t = (t1 t2) then
   let T1 = typeof(ctx, t1) in
   let T2 = typeof(ctx, t2) in
      if T1 = T2->T then return T
      else "Non tipabile"
```

Definiamo la seguente funzione da termini del λ-calcolo tipato (con le annotazioni) a termini del λ-calcolo (senza tipi).

- $erase(x) = x$,
- $erase(\lambda x : T.t) = \lambda x.erase(t)$,
- $erase(t_1\ t_2) = erase(t_1)\ erase(t_2)$.

Un termine del λ-calcolo (senza tipi) t è detto *tipabile* se c'è un termine del λ-calcolo tipato t' tale che $erase(t') = t$ e $\Gamma \vdash t' : T$ per qualche Γ e T. Passare da t a t' è il processo di *ricostruzione di tipo*.

Esercizio 4.13 Quale è il termine o quali sono i termini t' tali che $erase(t') = t$ per i seguenti t?

- $t = \lambda$x.λy.`if y then x else true`,
- $t = \lambda$x.x,
- $t = \lambda$f.λx.f (`if true then x else f x`),
- $t = \lambda$f.λg.`if` (f (g `true`)) `then` f (λx.`true`) `else` f(λx.x),

È possibile trovare un termine del lambda-calcolo tipato t tale che $erase(t) = \lambda$x.x x x?

5 Estensioni del lambda calcolo con tipi semplici

In questo capitolo studieremo alcune estensioni del lambda-calcolo con tipi semplici che vanno nella direzione dei linguaggi di programmazione "reali".

5.1 Estensioni semplici

Tipi base. Nel lambda calcolo con tipi semplici visto precedentemente abbiamo considerato come unico tipo base `Bool`. Abbiamo di conseguenza esteso la sintassi dei termini con gli operatori `true`, `false`, `if _ then _ else _`, e aggiunto le appropriate regole di tipaggio e di riduzione. Analogamente potremmo aggiungere il tipo base `Nat` e gli operatori `0`, `succ`, `pred` e `iszero`.
Da un punto di vista più teorico possiamo ignorare i termini dei tipi base e semplicemente trattare questi tipi come costanti senza fissarne una particolare interpretazione. Si noti che questo consente comunque di introdurre variabili dei tipi base, per esempio se B è un tipo base arbitrario $\lambda x : B.x$ è la funzione identità su elementi di questo tipo.

Il tipo `Unit`. È il tipo banale, il cui insieme di valori consiste di un solo elemento. Diventa di uso più interessante in un contesto imperativo, in cui le funzioni hanno anche side effect (in questo senso corrisponde al tipo `void` di C o Java).
Nuove forme sintattiche:

$$\begin{array}{lll} T & ::= & \ldots \mid \texttt{Unit} \\ t & ::= & \ldots \mid \texttt{unit} \\ v & ::= & \ldots \mid \texttt{unit} \end{array}$$

Nuova regola di tipo:

$$\text{(T-UNIT)}\ \frac{}{\Gamma \vdash \texttt{unit} : \texttt{Unit}}$$

Sequenza. Anche la sequenza è più interessante in un contesto imperativo in cui l'esecuzione della prima espressione ha un side effect.
Nuova forma sintattica:

$$t \quad ::= \quad \ldots \mid t_1 \texttt{ ; } t_2$$

Nuove regole di riduzione:

$$\text{(SEQ)} \quad \frac{t_1 \to t_1'}{t_1 \texttt{ ; } t_2 \to t_1' \texttt{ ; } t_2} \qquad \text{(SEQNEXT)} \quad \frac{}{\texttt{unit ; } t_2 \to t_2}$$

Nuova regola di tipo:

$$\text{(T-SEQ)} \quad \frac{\Gamma \vdash t_1 : \texttt{Unit} \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash t_1 \texttt{ ; } t_2 : T_2}$$

Alcune estensioni al linguaggio possono essere viste come *forme derivate*, cioè "zucchero sintattico". In altri termini, si distingue il *linguaggio interno* (per l'interprete) contrapposto al *linguaggio esterno* (per il programmatore). Per esempio la sequenza può essere espressa come forma derivata nel modo seguente:

$$t_1 \texttt{ ; } t_2 \triangleq (\lambda x : \texttt{Unit}.t_2)\, t_1 \text{ con } x \text{ non libera in } t_2$$

La definizione della sequenza come forma derivata è equivalente a quella data precedentemente. Più precisamente possiamo definire:

- il *linguaggio esterno* $\lambda^{\texttt{E}}$, che è il lambda-calcolo tipato semplice con tipo base `Unit`, sequenza, regole di riduzione: (APP1), (APP2), (APPABS^v), (SEQ), (SEQNEXT), e regole di tipo: (T-VAR), (T-APP), (T-ABS), (T-SEQ), (T-UNIT);
- il *linguaggio interno* $\lambda^{\texttt{I}}$, che è il lambda-calcolo tipato semplice con tipo base `Unit`, regole di riduzione: (APP1), (APP2), (APPABS^v), e regole di tipo: (T-VAR), (T-APP), (T-ABS), (T-UNIT);
- la funzione $e\colon \lambda^E \to \lambda^I$, che traduce da linguaggio esterno a interno, sostituendo le occorrenze di t_1 `;` t_2 con $(\lambda x : \texttt{Unit}.t_2)\, t_1$ (con x nuova ogni volta).

Teorema 5.1 Per ogni termine t di $\lambda^{\texttt{E}}$:

- $t \to t'$ se e solo se $e(t) \to e(t')$,
- $\Gamma \vdash t : T$ se e solo se $\Gamma \vdash e(t) : T$.

Attribuzione (ascription). Anche questo costrutto, che consiste nell'annotare un termine con un tipo, non è molto interessante nel semplice linguaggio che stiamo considerando, all'interno del quale ha unicamente un valore di documentazione o controllo. Diventa più significativo per esempio in linguaggi con nozione di sottotipo (si penso al cast in Java).
Nuova forma sintattica:

$$t \quad ::= \quad \ldots \mid t \texttt{ as } T$$

Nuove regole di riduzione:

$$\text{(ASCR)} \quad \frac{}{v \texttt{ as } T \to v} \qquad \text{(ASCR1)} \quad \frac{t \to t'}{t \texttt{ as } T \to t' \texttt{ as } T}$$

Nuova regola di tipo:

$$\text{(T-ASCR)} \quad \frac{\Gamma \vdash t : T}{\Gamma \vdash t \texttt{ as } T : T}$$

L'attribuzione può essere vista come forma derivata nel modo seguente:

$$t \texttt{ as } T \triangleq (\lambda x : T.x)\, t$$

Let-binding. Questo costrutto permette di introdurre un nome locale per un termine, come avviene nelle dichiarazioni dei linguaggi di programmazione.
Nuova forma sintattica:

$$t \quad ::= \quad \ldots \mid \texttt{let}\ x = t_1\ \texttt{in}\ t_2$$

Nuove regole di riduzione:

$$\text{(LetV)}\ \frac{}{\texttt{let}\ x = v\ \texttt{in}\ t \to t[v/x]} \qquad \text{(Let)}\ \frac{t_1 \to t_1'}{\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \to \texttt{let}\ x = t_1'\ \texttt{in}\ t_2}$$

Nuova regola di tipo:

$$\text{(T-Let)}\ \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma[T_1/x] \vdash t_2 : T_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \colon T_2}$$

Il let-in può essere visto come forma derivata nel modo seguente:

$$\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \triangleq (\lambda x : T_1.t_2)\, t_1$$

Si noti però che la parte destra contiene l'annotazione di tipo T_1 che non appare a sinistra (ma può essere ottenuta calcolando il tipo di t_1). Quindi il let-in può essere visto come forma derivata ma la codifica richiede di effettuare il type-checking del termine di partenza.

Coppie. Nuove forme sintattiche:

$$\begin{array}{lcl} t & ::= & \ldots \mid (t_1, t_2) \mid \texttt{fst}\ t \mid \texttt{snd}\ t \\ v & ::= & \ldots \mid (v_1, v_2) \\ T & ::= & \ldots \mid T_1 * T_2 \end{array}$$

Nuove regole di riduzione:

$$\text{(PairBeta1)}\ \frac{}{\texttt{fst}\ (v_1, v_2) \to v_1} \qquad \text{(PairBeta2)}\ \frac{}{\texttt{snd}\ (v_1, v_2) \to v_2}$$

$$\text{(Proj1)}\ \frac{t \to t'}{\texttt{fst}\ t \to \texttt{fst}\ t'} \qquad \text{(Proj2)}\ \frac{t \to t'}{\texttt{snd}\ t \to \texttt{snd}\ t'}$$

$$\text{(Pair1)}\ \frac{t_1 \to t_1'}{(t_1, t_2) \to (t_1', t_2)} \qquad \text{(Pair2)}\ \frac{t_2 \to t_2'}{(v_1, t_2) \to (v_1, t_2')}$$

Nuove regole di tipo:

$$\text{(T-Pair)}\ \frac{\Gamma \vdash t_1 : T_1 \quad \Gamma \vdash t_2 : T_2}{\Gamma \vdash (t_1, t_2) : T_1 * T_2}$$

$$\text{(T-Proj1)}\ \frac{\Gamma \vdash t : T_1 * T_2}{\Gamma \vdash \texttt{fst}\ t : T_1} \qquad \text{(T-Proj2)}\ \frac{\Gamma \vdash t : T_1 * T_2}{\Gamma \vdash \texttt{snd}\ t : T_2}$$

Ricorsione. È possibile provare che nel lambda-calcolo tipato la relazione di riduzione è *terminante*, quindi termini come ω e `fix` non sono tipabili. Possiamo però *estendere* il calcolo con un operatore di punto fisso tipato.
Per esempio, il termine seguente non è tipabile, in quanto la funzione `even: Nat → Bool` che controlla se un numero è pari è definita ricorsivamente:

```
let even =
    λn : Nat.
        if iszero n then true
        else if iszero pred n then false
        else even (pred pred n)
in even (succ succ 0)
```

Possiamo però definire questa funzione come *punto fisso* di una funzione `Even: (Nat → Bool) → (Nat → Bool)`

```
let Even =
    λev : Nat → Bool.
        λn : Nat.
            if iszero n then true
            else if iszero pred n then false
            else ev (pred pred n)
in fix Even (succ succ 0)
```

La funzione `Even`, applicata a una funzione `ev` che fornisce un'approssimazione di `even` (una funzione che controlla correttamente i primi n numeri naturali), restituisce una funzione `Even(ev)` che è una *migliore* approssimazione di `even` (ossia, che controlla correttamente i primi $n+2$ numeri naturali). Per esercizio si provino a ridurre, fissata $\texttt{ev} = \lambda \texttt{n} : \texttt{Nat}.\texttt{false}$, i termini `Even(ev)`, `Even(Even(ev))`, e così via. È intuitivamente chiaro che "al limite" si ottiene esattamente il comportamento di `even`. In altri termini, se diamo come argomento a `Even` proprio la funzione `even`, otteniamo come risultato ancora `even`, cioè `even` è un *punto fisso* di `Even` (più precisamente, il minimo punto fisso cioè la più piccola funzione che sia punto fisso, ossia la funzione definita da `F` visto come definizione induttiva).
Nuova forma sintattica:

$$t \;::=\; \ldots \mid \texttt{fix}\; t$$

Nuove regole di riduzione:

$$\text{(FixRid)}\quad \frac{}{\texttt{fix}\,(\lambda x : T.t) \to t[\texttt{fix}\,(\lambda x : T.t)/x]}$$

$$\text{(Fix)}\quad \frac{t \to t'}{\texttt{fix}\; t \to \texttt{fix}\; t'}$$

Per esercizio, provare a verificare che il termine `let Even = ... in fix Even (succ succ 0)` si riduce effettivamente a `true`.
Nuova regola di tipo:

$$\text{(T-Fix)}\quad \frac{\Gamma \vdash t : T \to T}{\Gamma \vdash \texttt{fix}\; t : T}$$

Una forma più conveniente è la seguente:

$$\texttt{letrec}\; x : T_1 = t_1\; \texttt{in}\; t_2 \stackrel{\Delta}{=} \texttt{let}\; x = \texttt{fix}\; \lambda x : T_1.t_1\; \texttt{in}\; t_2$$

Utilizzando questa sintassi possiamo riscrivere correttamente il termine precedente in questo modo:

```
let rec even : Nat → Bool =
    λn : Nat.
        if iszero n then true
        else if iszero pred n then false
        else even (pred pred n)
in even (succ succ 0)
```

5.2 Lambda calcolo imperativo

Vediamo ora come estendere il lambda calcolo tipato semplice con costrutti imperativi. L'estensione che descriviamo segue lo stile di ML, in cui, per ogni tipo T, si introduce un tipo `Ref` T delle *locazioni* (indirizzi di memoria) che possono contenere valori di tipo T. Si introducono tre nuove forme sintattiche: `ref` t alloca una nuova locazione inizializzandola con il valore di t; $!t$ restituisce il contenuto del valore di t (che dovrà quindi essere una locazione); t_1 `:=` t_2 assegna al valore di t_1 (che dovrà quindi essere una locazione) il valore di t_2.
Per esempio:

```
let x = ref succ succ 0 in
!x
```

alloca una nuova locazione inizializzandola con 2, introduce una variabile locale `x` che denota questa locazione, e ne restituisce il contenuto, quindi 2.

```
let x = ref succ succ 0 in
x := succ(!x); !x
```

alloca una nuova locazione inizializzandola con 2, introduce una variabile locale `x` che denota questa locazione, assegna alla locazione il suo contenuto corrente più uno, e ne restituisce il contenuto, quindi 3.

```
let x = ref succ succ 0 in
let y = x in
y := 0; !x
```

alloca una nuova locazione inizializzandola con 2, introduce una variabile locale `x` che denota questa locazione, introduce un'altra variabile locale `y` che denota il valore di `x` (quindi la stessa locazione), assegna alla locazione il valore 0, e restituisce il contenuto della locazione denotata da `x` (e `y`), quindi 0.
L'ultimo esempio illustra la differenza tra:

- allocazione e assegnazione (cambiano la memoria, nel primo caso aggiungendo una nuova locazione, nel secondo caso cambiando il contenuto di una locazione esistente)
- introduzione (tramite costrutto `let-in`) di una variabile locale, che nel caso denoti un indirizzo può dar luogo ad *aliasing* (più variabili che denotano la stessa locazione).

Sintassi.

$$\begin{array}{lcl} t & ::= & x \mid \lambda x : T.t \mid t_1\ t_2 \mid \texttt{unit} \mid \texttt{ref}\ t \mid !t \mid t_1 := t_2 \mid l \mid \ldots \\ v & ::= & \lambda x : T.t \mid \texttt{unit} \mid l \mid \ldots \end{array}$$

Oltre ai costrutti sintattici di allocazione, dereferenziazione e assegnazione introdotti prima informalmente, aggiungiamo come termini (e valori) le locazioni. Non specificheremo la natura precisa delle locazioni: per i nostri fini, ci basta assumere che l vari su un insieme infinito Loc.
Il fatto che abbiamo aggiunto le locazioni nei termini *non* significa che i programmatori potranno scrivere locazioni esplicite nei loro programmi. In questo caso (come in molti altri) l'idea è quella di distinguere tra *espressioni del linguaggio utente* (quelle effettivamente utilizzabili dai programmatori) ed *espressioni a run-time* (quelle che possono essere ottenute come passi intermedi della riduzione). Il secondo sarà in genere un soprainsieme del primo.

Regole di riduzione. Per modellare l'esecuzione di un programma imperativo, introduciamo un modello della memoria. Concretamente, la memoria è, per esempio, un array di byte, indicizzati su interi di 32-bit. Più astrattamente, possiamo vederla come una funzione parziale da locazioni a valori $\mu \in Loc \to Val$.
La relazione di riduzione quindi non sarà più semplicemente una relazione tra *termini*, come negli esempi visti finora (che si chiamano *calcoli*), ma una relazione tra coppie (termine, memoria):

$$t \mid \mu \to t' \mid \mu'$$

Vediamo ora le regole di riduzione per i costrutti imperativi.
Un termine $!t$ viene valutato riducendo prima t finché non è un valore:

$$\text{(DEREF)}\quad \frac{t \mid \mu \to t' \mid \mu'}{!t \mid \mu \to !t' \mid \mu'}$$

e poi restituendo il contenuto del suo valore (che deve essere una locazione) nella memoria corrente:

$$\text{(DEREFLOC)}\quad \frac{}{!l \mid \mu \to v \mid \mu}\ \ \mu(l) = v$$

Un assegnazione $t_1 := t_2$ è valutata riducendo prima t_1 e t_2 fino a che non diventano valori:

$$\text{(ASSIGN1)}\quad \frac{t_1 \mid \mu \to t_1' \mid \mu'}{t_1 := t_2 \mid \mu \to t_1' := t_2 \mid \mu'}$$

$$\text{(ASSIGN2)}\quad \frac{t_2 \mid \mu \to t_2' \mid \mu'}{v := t_2 \mid \mu \to v := t_2' \mid \mu'}$$

e poi restituendo `unit` e la memoria aggiornata:

(ASSIGN) $$\frac{}{l := v \mid \mu \to \texttt{unit} \mid \mu[v/l]}$$

Si noti che nel caso l'espressione a sinistra non si valuti a una locazione si valuta comunque l'espressione a destra (si potrebbe anche bloccare subito l'esecuzione, come?)
Un termine della forma $\texttt{ref}\ t$ si valuta riducendo prima t fino a che non diventa un valore:

(REF) $$\frac{t \mid \mu \to t' \mid \mu'}{\texttt{ref}\ t \mid \mu \to \texttt{ref}\ t' \mid \mu'}$$

quindi si alloca una *nuova* locazione l, e si aggiorna la memoria con un'associazione da l a questo valore. Si restituisce infine l come risultato.

(REFV) $$\frac{}{\texttt{ref}\ v \mid \mu \to l \mid \mu[v/l]} \quad l \notin dom(\mu)$$

Le regole di riduzione per l'applicazione sono modificate per tener conto della memoria:

(APP1) $$\frac{t_1 \mid \mu \to t_1' \mid \mu'}{t_1\ t_2 \mid \mu \to t_1'\ t_2 \mid \mu'}$$ (APP2) $$\frac{t_2 \mid \mu \to t_2' \mid \mu'}{v\ t_2 \mid \mu \to v\ t_2' \mid \mu'}$$

(APPABS) $$\frac{}{(\lambda x : T.t)\ v \mid \mu \to t[v/x] \mid \mu}$$

Si noti che non viene modellata la deallocazione della memoria non utilizzata.

Sistema di tipi. Vogliamo esprimere il fatto che una memoria è *ben tipata*, ossia, a ogni locazione è associato un valore del tipo corrispondente. Una prima idea potrebbe essere quella di ricavare il tipo di una locazione dal valore associato in memoria:

(T-LOC) $$\frac{\Gamma \mid \mu \vdash \mu(l) : T}{\Gamma \mid \mu \vdash l : \texttt{Ref}\ T}$$

Si noti che in questo modo la relazione di tipaggio avrebbe quindi quattro componenti: contesti, *memorie*, termini e tipi.
Tuttavia questa regola non è soddisfacente, perché, dato che un valore in memoria può a sua volta far riferimento a delle locazioni, può dar luogo ad alberi di prova molto grandi: si consideri per esempio l'albero di prova per $!l_5$ nella memoria

$$\mu = \begin{array}{l} \texttt{l}_1 \mapsto \lambda \texttt{x} : \texttt{Nat}.\texttt{0}, \\ \texttt{l}_2 \mapsto \lambda \texttt{x} : \texttt{Nat}.(!\texttt{l}_1)\ \texttt{x}, \\ \texttt{l}_3 \mapsto \lambda \texttt{x} : \texttt{Nat}.(!\texttt{l}_2)\ \texttt{x}, \\ \texttt{l}_4 \mapsto \lambda \texttt{x} : \texttt{Nat}.(!\texttt{l}_3)\ \texttt{x}, \\ \texttt{l}_5 \mapsto \lambda \texttt{x} : \texttt{Nat}.(!\texttt{l}_4)\ \texttt{x}, \end{array}$$

o, peggio, può capitare che non si riesca a costruire un albero di prova finito: si consideri per esempio l'albero di prova per $!l_2$ nella memoria

$$\mu = \begin{array}{l} \texttt{l}_1 \mapsto \lambda \texttt{x} : \texttt{Nat}.(!\texttt{l}_2)\ \texttt{x}, \\ \texttt{l}_2 \mapsto \lambda \texttt{x} : \texttt{Nat}.(!\texttt{l}_1)\ \texttt{x}, \end{array}$$

Quindi, si utilizza piuttosto un'*assegnazione di tipo* Σ, cioè una funzione parziale da locazioni a tipi.
Per esempio, per la prima memoria considerata un'assegnazione di tipo possibile è

$$\Sigma = \begin{array}{l} \texttt{l}_1 \mapsto \texttt{Nat} \to \texttt{Nat}, \\ \texttt{l}_2 \mapsto \texttt{Nat} \to \texttt{Nat}, \\ \texttt{l}_3 \mapsto \texttt{Nat} \to \texttt{Nat}, \\ \texttt{l}_4 \mapsto \texttt{Nat} \to \texttt{Nat}, \\ \texttt{l}_5 \mapsto \texttt{Nat} \to \texttt{Nat} \end{array}$$

La relazione di tipaggio avrà quindi quattro componenti: contesto, assegnazione di tipo alla memoria, termini e tipi.
Le regole di tipo sono le seguenti:

(T-LOC) $$\frac{}{\Gamma \mid \Sigma \vdash l : \texttt{Ref}\ T} \quad \Sigma(l) = T$$ (T-REF) $$\frac{\Gamma \mid \Sigma \vdash t : T}{\Gamma \mid \Sigma \vdash \texttt{ref}\ t : \texttt{Ref}\ T}$$

(T-DEREF) $$\frac{\Gamma \mid \Sigma \vdash t : \texttt{Ref}\ T}{\Gamma \mid \Sigma \vdash !t : T}$$ (T-ASSIGN) $$\frac{\Gamma \mid \Sigma \vdash t_1 : \texttt{Ref}\ T \quad \Gamma \mid \Sigma \vdash t_2 : T}{\Gamma \mid \Sigma \vdash t_1\ := t_2 : \texttt{Unit}}$$

La conservazione dei tipi e il progress vanno formulati opportunamente. Anzitutto, dato che ora la relazione di riduzione è su coppie (termine, memoria), occorre definire quand'è che una memoria è ben formata rispetto a un contesto e un'assegnazione di tipo:

$\Gamma \mid \Sigma \vdash \mu$ sse $dom(\mu) = dom(\Sigma)$, e per ogni $l \in dom(\mu)$ si ha $\Gamma \mid \Sigma \vdash \mu(l) : \Sigma(l)$.

Teorema 5.2 [Progress] Se $\emptyset \mid \Sigma \vdash t : T$, $\emptyset \mid \Sigma \vdash \mu$, allora t è un valore oppure $t \mid \mu \to t' \mid \mu'$, per qualche $t' \mid \mu'$.

Teorema 5.3 [Conservazione dei tipi] Se $\Gamma \mid \Sigma \vdash t : T$, $\Gamma \mid \Sigma \vdash \mu$, e $t \mid \mu \to t' \mid \mu'$, allora $\Gamma \mid \Sigma' \vdash t' : T$, $\Gamma \mid \Sigma' \vdash \mu'$, per $\Sigma \subseteq \Sigma'$.

Si noti che un passo di riduzione potrebbe allocare una nuova locazione, quindi la memoria risultante sarà ben formata rispetto a un'assegnazione di tipo più grande che tiene conto di tale nuova locazione.

5.3 Eccezioni

In molti linguaggi di programmazione vengono forniti dei meccanismi per interrompere il flusso normale di esecuzione di un programma in caso si verifichino delle condizioni di "errore". Altrimenti ogni chiamata di funzione dovrebbe essere seguita da un test che ci dice se è andata a buon fine o no.
Vediamo per prima cosa un meccanismo di errore molto semplice che interrompe in ogni caso l'esecuzione del programma.

$$t \quad ::= \quad \ldots \mid \texttt{error}$$

Nuove regole di riduzione:

$$\text{(AppErr1)} \quad \frac{}{\texttt{error}\; t \to \texttt{error}}$$

$$\text{(AppErr2)} \quad \frac{}{v\; \texttt{error} \to \texttt{error}}$$

Nuova regola di tipo:

$$\text{(T-Err)} \quad \frac{}{\Gamma \vdash \texttt{error} : T}$$

Si noti che questa regola di tipo fa sì che, per esempio, sia

```
if x > 0 then 5 else error
```

che

```
if x > 0 then true else error
```

siano termini ben tipati.
Si noti che questa regola non ci permette di dire quale è il tipo di `error` indipendentemente dal contesto in cui è usato. In altri termini, non è più vero che ogni termine ha un unico tipo.
La soluzione alternativa in cui si utilizza una versione decorata del termine `error`:

$$\text{(T-Err)} \quad \frac{}{\Gamma \vdash \texttt{error as}\; T : T}$$

non funziona; infatti per esempio (assumendo di avere nel linguaggio espressioni booleane e naturali) il termine

`succ (if (error as Bool) then 3 else 5)`

è ben tipato e ha tipo `Nat`, ma si riduce al termine

`succ (error as Bool)`

che non è ben tipato (quindi verrebbe meno la proprietà di conservazione dei tipi).
Con la soluzione data sopra invece, vale il teorema di conservazione dei tipi nella forma usuale, perché, se un termine di tipo T si riduce a `error`, questo va bene in quanto `error` ha ogni tipo T.
Il teorema di progress invece va formulato diversamente. Infatti, per prima cosa notiamo che non sarebbe corretto estendere l'insieme dei valori a includere `error`, perché questo renderebbe la nuova regola di propagazione dell'errore attraverso l'applicazione in competizione con la regola esistente di applicazione di una astrazione:

$$\text{(AppAbs}^v\text{)}\quad \frac{}{(\lambda x.t)\ v \to t[v/x]}$$

Per esempio il termine $(\lambda \mathtt{x} : \mathtt{Nat}.\mathtt{0})\ \mathtt{error}$ potrebbe ridursi sia a `0` (erroneamente) che a `error` (che è il risultato che ci aspettiamo).
Si considera quindi `error` come un termine in forma normale che non è un valore e si modifica la formulazione del teorema di progress prevedendo esplicitamente la possibilità che un termine possa ridursi a `error` invece che a un valore.

Teorema 5.4 [Progress] Sia t un termine ben tipato e chiuso. Allora t è un valore, oppure $t \to$, oppure $t = \mathtt{error}$.

Vediamo ora un meccanismo di cattura degli errori (come in ML e Java).
Nuova forma sintattica:

$$t \quad ::= \quad \ldots \mid \mathtt{try}\ t_1\ \mathtt{with}\ t_2$$

Il secondo argomento si chiama *gestore* (*handler*) dell'errore. Nuove regole di riduzione:

$$\text{(TryV)}\quad \frac{}{\mathtt{try}\ v\ \mathtt{with}\ t \to v} \qquad \text{(TryErr)}\quad \frac{}{\mathtt{try}\ \mathtt{error}\ \mathtt{with}\ t \to t}$$

$$\text{(Try)}\quad \frac{t_1 \to t_1'}{\mathtt{try}\ t_1\ \mathtt{with}\ t_2 \to \mathtt{try}\ t_1'\ \mathtt{with}\ t_2}$$

Nuova regola di tipo:

$$\text{(T-Try)}\quad \frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T}{\Gamma \vdash \mathtt{try}\ t_1\ \mathtt{with}\ t_2 : T}$$

Vediamo infine un meccanismo in cui, anziché avere un unico termine che corrisponde a una situazione di errore generico, possiamo codificare diversi tipi di errore con diverse *eccezioni* (come in ML e Java). L'handler diventa quindi una *funzione* che prende come argomento un'eccezione. Qui non ci interessa dettagliare la natura delle eccezioni, quindi le modelleremo con un tipo $T_{\mathtt{exn}}$ non specificato.
Nuova forma sintattica:

$$t \quad ::= \quad \ldots \mid \mathtt{raise}\ t$$

Nuove regole di riduzione:

$$\text{(Raise1)}\quad \frac{}{(\mathtt{raise}\ v)\ t \to \mathtt{raise}\ v} \qquad \text{(Raise2)}\quad \frac{}{v_1\ (\mathtt{raise}\ v_2) \to \mathtt{raise}\ v_2}$$

$$\text{(Raise)}\quad \frac{t \to t'}{\mathtt{raise}\ t \to \mathtt{raise}\ t'} \qquad \text{(RaiseRaise)}\quad \frac{}{\mathtt{raise}\ (\mathtt{raise}\ v) \to \mathtt{raise}\ v}$$

$$\text{(TryV)}\quad \frac{}{\mathtt{try}\ v\ \mathtt{with}\ t \to v} \qquad \text{(TryRaise)}\quad \frac{}{\mathtt{try}\ \mathtt{raise}\ v\ \mathtt{with}\ t \to t\ v}$$

$$\text{(Try)}\quad \frac{t_1 \to t_1'}{\mathtt{try}\ t_1\ \mathtt{with}\ t_2 \to \mathtt{try}\ t_1'\ \mathtt{with}\ t_2}$$

Nuove regole di tipo:

$$\text{(T-Exn)}\quad \frac{\Gamma \vdash t : T_{\mathtt{exn}}}{\Gamma \vdash \mathtt{raise}\ t : T}$$

$$\text{(T-Try)}\quad \frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : T_{\mathtt{exn}} \to T}{\Gamma \vdash \mathtt{try}\ t_1\ \mathtt{with}\ t_2 : T}$$

La regola (RAISERAISE) gestisce la situazione in cui la valutazione dell'espressione argomento del `raise` solleva a sua volta un'eccezione.
Vi sono varie scelte possibili per il tipo $T_{\texttt{exn}}$ delle eccezioni, per esempio:

- numeri naturali, corrispondenti a codici di errore (come per esempio in Unix);
- stringhe (più informative per l'utente, ma scomode da analizzare per l'eventuale handler dell'eccezione);
- tipo somma, o tipo somma estendibile, come in ML o Java. Il meccanismo di eccezioni di Java offre in più, rispetto al modello presentato qui, due ingredienti importanti: la relazione di sottotipazione tra tipi eccezione che permette di gestire la cattura in modo più flessibile; il controllo statico delle eccezioni che possono essere sollevate da un metodo attraverso la clausola `throws`.

6 Featherweight Java

6.1 Introduzione

In questo capitolo presentiamo un calcolo molto semplice che modella gli ingredienti base dei linguaggi object-oriented class-based e con tipi nominali (cioè, dove i nomi delle classi sono tipi) come Java. Questo calcolo si chiama Featherweight Java (Java "peso piuma"), FJ da qui in poi, ed è stato proposto nel 1999 da Igarashi, Pierce e Wadler. FJ non include molti costrutti di Java come interfacce e classi astratte, costruttori definiti dall'utente, overloading, hiding di campi, super, tipi primitivi e `void`, modificatori di accesso, eccezioni; inoltre, una semplificazione ancora maggiore è data dal fatto che si considera un linguaggio *funzionale*, quindi non vi è nozione di stato (universo di oggetti esistenti) nè di identità degli oggetti. Tutti questi costrutti possono essere aggiunti al linguaggio analogamente a quello che abbiamo visto con le estensioni del lambda-calcolo tipato semplice.
La sintassi è data in Fig.7 dove C, f e *m* indicano un generico nome rispettivamente di classe, campo e metodo.

p	::=	$cd_1 \ldots\ cd_n$
cd	::=	`class` C `extends` C' { fds mds }
fds	::=	$fd_1 \ldots\ fd_n$
fd	::=	$C\ f$ `;`
mds	::=	$md_1 \ldots md_n$
md	::=	$C_0\ m(C_1\ x_1, \ldots, C_n\ x_n)$ {`return` e`;`}
e	::=	$x \mid e.f \mid e_0.m(e_1, \ldots, e_n) \mid$ `new` $C(e_1, \ldots, e_n) \mid (C)e$

Figura 7: Sintassi di Featherweight Java

Un programma in FJ è una sequenza di dichiarazioni di classi. Una dichiarazione di classe consiste del nome della classe, il nome della classe parent (sempre indicato esplicitamente), una sequenza di dichiarazioni di campo e una sequenza di dichiarazioni di metodo. Assumiamo che esista un nome di classe speciale `Object` che non può essere dichiarato. Una dichiarazione di campo consiste del nome e del tipo del campo, mentre una dichiarazione di metodo consiste del tipo di ritorno, il nome del metodo, la lista dei parametri formali (nome e tipo) e il corpo del metodo, che è un'espressione. Vi sono solo cinque tipi di espressioni: variabili, accesso a campo, invocazione di metodo, creazione di un oggetto e cast. Dato che non vi sono tipi primitivi, i tipi coincidono con i nomi delle classi. La parola chiave `this` (che non si può omettere) è vista come una variabile speciale.
Non vi sono costruttori: infatti, si assume che per ogni classe esista implicitamente un unico costruttore, che ha tanti parametri quanti i campi della classe, con gli stessi tipi, nell'ordine da quelli ereditati a quelli dichiarati direttamente (vedi esempi sotto).
Vediamo un esempio di programma in FJ.

```
class A extends Object {}

class B extends Object {}

class Pair extends Object {
  Object first;
  Object second;
  Pair setfirst (Object newfirst) { return new Pair(newfirst, this.second); }
}
```

```
class Triple extends Pair {
  Object third;
}
```

Le classi `A` e `B` non hanno campi, quindi il loro costruttore implicito sarà senza parametri; ossia, per creare un oggetto di classe `A` scriveremo `new A()`, e analogamente per `B`. Invece, la classe `Pair` possiede due campi di tipo `Object`, quindi il suo costruttore implicito avrà due parametri di tipo `Object`; ossia, per creare un oggetto di classe `Pair` scriveremo `new Pair`(e_1, e_2) con e_1, e_2 di tipo `Object`. Analogamente la classe `Triple` possiede tre campi di tipo `Object`, quindi il suo costruttore implicito avrà tre parametri di tipo `Object`; ossia, per creare un oggetto di classe `Triple` scriveremo `new Triple`(e_1, e_2, e_3) con e_1, e_2, e_3 di tipo `Object`.
Il comportamento del costruttore implicito consiste nell'inizializzare i campi con gli argomenti, nell'ordine. Nell'esempio, il comportamento dei costruttori impliciti corrisponde alle seguenti dichiarazioni in Java:

```
class A extends Object {
  A () {super();}
}

class B extends Object {
  B() {super();}
}

class Pair extends Object {
  Object first;
  Object second;
  Pair (Object first, Object second) {super(); this.first = first; this.second = second;}
  ...
}

class Triple extends Pair {
  Object third;
  Triple (Object first, Object second, Object third) {super(first,second); this.third = third;}
}
```

Si noti che parliamo di "creazione di un oggetto" per mantenere la terminologia dei linguaggi object-oriented, ma trattandosi di un linguaggio funzionale non vi è in realtà alcuna "creazione" nel senso di modifica di uno stato corrente; l'espressione `new A()` indica semplicemente "l'unico oggetto di classe `A`" (o in altri termini l'unico valore di tipo `A`), mentre le espressioni `new Pair(new Object(),new Object())` e `new Pair(new A(), new A())` sono due diversi oggetti di classe `Pair` (valori di tipo `Pair`).
Per lo stesso motivo, il metodo `setfirst`, anziché modificare il campo `second` dell'oggetto ricevitore come suggerito dal nome, restituisce un nuovo oggetto (o meglio: *l'oggetto*) uguale al ricevitore tranne che per il campo `first` che è uguale all'argomento.
La riduzione di un'espressione FJ avviene nel contesto di un programma (necessario in quanto tiene traccia dei campi e metodi di ogni classe) e produce come risultato finale un oggetto, ossia un'espressione formata solo da `new`. Si noti che il fatto di considerare un linguaggio funzionale permette di evitare di dover modellare strutture ausiliarie come la memoria, quindi si ha un *calcolo* (vedi Sez.2.1). Formalmente si ha la seguente definizione dei valori:

$$v \ ::= \ \texttt{new}\ C(v_1, \ldots, v_n)$$

Per esempio, il termine

```
new Pair(new A(), new B()).setfirst(new B())
```

nel contesto del programma precedente, si riduce a

```
new Pair(new B(), new B())
```

La semantica operazionale di FJ si basa su tre regole computazionali, una per l'accesso a campo, una per l'invocazione di metodo e una per il cast. Il seguente è un esempio di applicazione della regola per l'accesso a campo:

```
new Pair(new A(), new B()).second → new B()
```

Si controlla che la classe `Pair` abbia un campo `second`. In caso contrario si avrebbe un errore a run-time ("campo non trovato") che sarà evitato dal sistema di tipi. Poiché il controllo ha esito positivo, si restituisce il valore corrispondente.
Il seguente è un esempio di applicazione della regola per l'invocazione di metodo:

```
new Pair(new A(), new B()).setfirst(new B()) →
new Pair(new B(), new Pair(new A(), new B()).second)
```

Si controlla che la classe `Pair` abbia un metodo `setFirst` e che questo abbia tanti parametri quanti gli argomenti (uno in questo caso). In caso contrario si avrebbe un errore a run-time ("metodo non trovato") che sarà evitato dal sistema di tipi. Poiché il controllo ha esito positivo, si restituisce il corpo del metodo dove i parametri formali sono sostituiti con gli argomenti corrispondenti (nell'esempio, `newfirst` è sostituito con `new B()`) e `this` è sostituito con il ricevitore (nell'esempio, con `new Pair(new A(), new B())`). Si noti che questa semantica è simile alla beta-regola del lambda-calcolo. Tuttavia, vi è in più il *binding dinamico*, cioè il fatto che (in caso di ridefinizione) la versione del metodo da invocare dipende dal tipo (dinamico) del ricevitore. Questo è illustrato meglio dal seguente esempio.

```
class C extends Object () {
  Object m() { return new C();}
}
class D extends C {
  Object m() {return new D();}
}
class CPair extends Object {
  C first; C second;
}
```

Il termine

```
new CPair(new D(),new D()).first.m()
```

si riduce a `new D()`. Staticamente l'invocazione di metodo è corretta perché il tipo statico del termine `new CPair(new D(),new D()).first` è `C`, la classe `C` possiede un metodo `m` e questo non ha parametri. A run-time, questo termine si riduce a `new D()`, quindi viene invocata la versione di `m` nella classe `D`. Questo esempio illustra anche il fatto che il tipo dinamico può essere diverso dal tipo statico, e ne è sempre un sottotipo.
Vediamo infine un esempio di applicazione della regola per il cast.

```
(Pair)new Pair(new A(), new B()) → new Pair(new A(), new B())
```

Si controlla che il tipo dinamico (la classe) dell'oggetto a cui è applicato il cast sia un sottotipo del tipo nel cast. In caso positivo si restituisce l'oggetto stesso (l'unico effetto a run-time di un cast è il test). In caso negativo si ha un errore a run-time che *non* viene evitato dal type system (per semplicità infatti qui modelliamo anche questo tipo di errore con una riduzione stuck, ma sarebbe più appropriato introdurre un errore dinamico corrispondente a `ClassCastException` in Java).
Il seguente ulteriore esempio illustra l'utilità del downcast. Il termine

```
((Pair)new Pair  (   new Pair(new A(), new B()),
                     new A()
                 ).first
).second
```

si riduce a `new B()`. Si noti che il cast è necessario per poter accedere al campo `second` dell'oggetto denotato da `new Pair(new Pair(new A(), new B()), new A()).first`; infatti questo termine ha staticamente tipo `Object`, poiché il campo `first` ha tipo `Object`. Dinamicamente, questo termine si riduce a `new Pair(new A(), new B())`, che ha tipo `Pair`, quindi il cast ha successo.

(FIELDNEW) $\dfrac{}{\texttt{new } C(v_1,\ldots,v_m).f \to v_i} \quad \begin{array}{l} \mathit{fields}(C) = C_1\ f_1\texttt{;}\ \ldots\ C_n\ f_n\texttt{;} \\ f = f_i \end{array}$

(INVKNEW) $\dfrac{}{\texttt{new } C(v_1,\ldots,v_m).m(u_1,\ldots,u_n) \to e[u_1/x_1\ldots u_n/x_n][\texttt{new } C(v_1,\ldots,v_m)/\texttt{this}]} \quad \mathit{mbody}(C,m) = (x_1\ldots x_n, e)$

(CASTNEW) $\dfrac{}{(C')\texttt{new } C(v_1,\ldots,v_m) \to \texttt{new } C(v_1,\ldots,v_m)} \quad C \le C'$

(FIELD) $\dfrac{e \to e'}{e.f \to e'.f}$ (INVKRCV) $\dfrac{e_0 \to e_0'}{e_0.m(e_1,\ldots,e_n) \to e_0'.m(e_1,\ldots,e_n)}$

(INVKARG) $\dfrac{e_i \to e_i'}{v_0.m(v_1\ldots,v_{i-1},e_i,\ldots,e_n) \to v_0.m(v_1\ldots,v_{i-1},e_i',\ldots,e_n)}$ (CAST) $\dfrac{e \to e'}{(C)e \to (C)e'}$

(NEW) $\dfrac{e_i \to e_i'}{\texttt{new } C(v_1\ldots,v_{i-1},e_i,\ldots,e_n) \to \texttt{new } C(v_1\ldots,v_{i-1},e_i',\ldots,e_n)}$

Figura 8: Regole di riduzione per Featherweight Java

6.2 Formalizzazione

La semantica formale di FJ è data in Fig.8. Per semplicità si assume un programma fissato.
Le regole utilizzano alcune funzioni ausiliarie che sono definite formalmente in Fig.9: $\mathit{fields}(C)$ restituisce la lista dei campi (tipo e nome) della classe C nel programma, nell'ordine da quelli ereditati a quelli dichiarati direttamente; $\mathit{mbody}(C,m)$ restituisce lista dei parametri e corpo del metodo *m* della classe C nel programma, se questo metodo esiste; infine, $C \le C'$ vale vero se la classe C è sottotipo della classe C' nel programma (ossia, $\le$ è la chiusura riflessiva e transitiva della relazione `extends`). In Fig.9 è anche definita la funzione mtype, analoga a mbody, utilizzata nelle regole di tipo: $\mathit{mtype}(C,m)$ restituisce la sequenza dei tipi dei parametri e il tipo di ritorno del metodo *m* della classe C nel programma, se questo metodo esiste.

(≤-REFL) $\dfrac{}{C \le C}$ (≤-TRANS) $\dfrac{C \le C' \quad C' \le C''}{C \le C''}$ (≤-EXTENDS) $\dfrac{}{C \le C'} \quad p(C) = \texttt{class } C \texttt{ extends } C' \{\ldots\}$

(FIELDS-1) $\dfrac{}{\mathit{fields}(\texttt{Object}) = \Lambda}$ (FIELDS-2) $\dfrac{\mathit{fields}(C') = \mathit{fds}'}{\mathit{fields}(C) = \mathit{fds}'\ \mathit{fds}} \quad p(C) = \texttt{class } C \texttt{ extends } C' \{\ \mathit{fds}\ \ldots\}$

(MBODY-1) $\dfrac{}{\mathit{mbody}(C,m) = (x_1\ldots x_n, e)} \quad \begin{array}{l} p(C) = \texttt{class } C \texttt{ extends } C' \{\ \mathit{fds}\ \mathit{mds}\ \} \\ C_0\ m(C_1\ x_1,\ldots,C_n\ x_n)\{\texttt{return } e\texttt{;}\} \in \mathit{mds} \end{array}$

(MBODY-2) $\dfrac{\mathit{mbody}(C',m) = (x_1\ldots x_n, e)}{\mathit{mbody}(C,m) = (x_1\ldots x_n, e)} \quad \begin{array}{l} p(C) = \texttt{class } C \texttt{ extends } C' \{\ \mathit{fds}\ \mathit{mds}\ \} \\ m \notin \mathit{mds} \end{array}$

(MTYPE-1) $\dfrac{}{\mathit{mtype}(C,m) = C_1\ldots C_n \to C_0} \quad \begin{array}{l} p(C) = \texttt{class } C \texttt{ extends } C' \{\ \mathit{fds}\ \mathit{mds}\ \} \\ C_0\ m(C_1\ x_1,\ldots,C_n\ x_n)\{\texttt{return } e\texttt{;}\} \in \mathit{mds} \end{array}$

(MTYPE-2) $\dfrac{\mathit{mtype}(C',m) = C_1\ldots C_n \to C}{\mathit{mtype}(C,m) = C_1\ldots C_n \to C} \quad \begin{array}{l} p(C) = \texttt{class } C \texttt{ extends } C' \{\ \mathit{fds}\ \mathit{mds}\ \} \\ m \notin \mathit{mds} \end{array}$

Figura 9: Funzioni ausiliarie per FJ

Il sistema di tipo per FJ definisce quando un programma è ben formato, attraverso la definizione di judgment (giudizi di tipo) per le dichiarazioni di classe, le dichiarazioni di metodo e le espressioni. In queste note diamo solo la formalizzazione del giudizio

di tipo per le espressioni, che ha forma $\Gamma \vdash e : C$ con significato "l'espressione *e* ha tipo C nel contesto Γ". Un contesto è, come al solito, una funzione parziale da variabili (nomi di parametri oppure `this`) in tipi (in questo caso nomi di classe). In realtà il tipo di un'espressione dipende anche dal programma in cui essa appare, ma per semplicità, come abbiamo fatto per la semantica operazionale, assumiamo questo programma fissato.
Le regole di tipo per le espressioni sono date in Fig.10.

(T-VAR) $\dfrac{}{\Gamma \vdash x : C} \quad \Gamma(x) = C$

(T-FIELD) $\dfrac{\Gamma \vdash e : C_0}{\Gamma \vdash e.f : C_i} \quad \begin{array}{l} \mathit{fields}(C_0) = C_1\ f_1\texttt{;}\ \ldots\ C_n\ f_n\texttt{;} \\ f = f_i \end{array}$

(T-INVK) $\dfrac{\Gamma \vdash e_i : C_i\ \forall i \in 0..n}{\Gamma \vdash e_0.m(e_1, \ldots, e_n) : C} \quad \begin{array}{l} \mathit{mtype}(C_0, m) = C'_1 \ldots C'_n \to C \\ C_i \leq C'_i\ \forall i \in 1..n \end{array}$

(T-NEW) $\dfrac{\Gamma \vdash e_i : C_i\ \forall i \in 1..n}{\Gamma \vdash \texttt{new}\ C(e_1, \ldots, e_n) : C} \quad \begin{array}{l} \mathit{fields}(C) = C'_1\ f_1\texttt{;}\ \ldots\ C'_n\ f_n\texttt{;} \\ C_i \leq C'_i\ \forall i \in 1..n \end{array}$

(T-UPCAST) $\dfrac{\Gamma \vdash e : C}{\Gamma \vdash (C')e : C'} \quad C \leq C'$

(T-DOWNCAST) $\dfrac{\Gamma \vdash e : C}{\Gamma \vdash (C')e : C'} \quad C' \leq C$

Figura 10: Regole di tipo per le espressioni di FJ

Per quanto riguarda programmi, dichiarazioni di classe e di metodo diamo solo una descrizione informale dei vincoli di tipo. Un programma *p* è ben formato se verifica le seguenti condizioni:

- non possono apparire due dichiarazioni per la stessa classe, quindi *p* può esser visto come una funzione parziale da nomi di classe a dichiarazioni di classe (questo giustifica la notazione $p(C)$ usata nelle regole di tipo),
- per ogni classe $C \in \mathit{dom}(p)$, $p(C) = \texttt{class}\ C\ \ldots$,
- $\texttt{Object} \notin \mathit{dom}(p)$,
- per ogni C che appare in *p*, si ha che $C \in \mathit{dom}(p)$,
- la relazione $C \leq C''$ è aciclica (antisimmetrica).
- ogni dichiarazione di classe in *p* è ben formata (nel senso precisato sotto).

Una dichiarazione di classe $\texttt{class}\ C\ \texttt{extends}\ C'\ \{\ \mathit{fds}\ \mathit{mds}\ \}$ è ben formata in *p* se verifica le seguenti condizioni:

- non possono apparire due dichiarazioni per lo stesso campo, e un campo f dichiarato in *fds* non può essere un campo di C' (ossia, non è ammesso l'hiding di campi come invece accade in Java);
- non possono apparire due dichiarazioni per lo stesso metodo, e se un metodo *m* è dichiarato in *mds* ed è anche un metodo di C' il tipo di ritorno e quello dei parametri devono coincidere (ossia, valgono le regole per l'overriding di Java[7] ma non è ammesso l'overloading);
- ogni dichiarazione di metodo è ben formata nel senso che l'espressione che costituisce il corpo del metodo è ben tipata nel contesto dove i parametri hanno i tipi indicati e `this` ha tipo C; inoltre, il tipo dell'espressione è un sottotipo del tipo di ritorno del metodo.

È possibile dare un risultato di soundness per il sistema di tipi così definito per FJ. Non ne daremo qui una formulazione precisa, perché questa richiede alcuni aggiustamenti rispetto a quanto visto in precedenza. In particolare, non vale per FJ la conservazione dei tipi, come illustrato dal seguente esempio. Si consideri il programma

```
class A extends Object {}
class B extends Object {}
```

Allora, il termine `(A)(Object)new B()` è staticamente corretto (e di tipo `A`), in quanto si ha prima un casting up da `B` a `Object` e poi un casting down da `Object` ad `A`. Tuttavia, a run-time questo termina si riduce nel termine `(A)new B()` che non è ben tipato, in quanto le classi `A` e `B` sono scorrelate.

[7] In Java 5 queste regole sono state rese più liberali, nel senso che il tipo di ritorno del metodo ridefinito può essere più specifico.

7 Semantica denotazionale

7.1 Principi

Sia $\mathcal{T}$ un linguaggio definito come algebra dei termini su una segnatura (S, O) (vedi Sez.1.4). Informalmente definire una semantica del linguaggio significa attribuire un significato (interpretazione, valore) ad ogni termine.
I principi su cui si basa la semantica denotazionale sono i seguenti.

Unicità del valore Vogliamo associare ad ogni termine (di un certo tipo) del linguaggio $\mathcal{T}$ un unico valore del tipo corrispondente. Questo principio può essere espresso formalmente dicendo che la semantica denotazionale è una *famiglia di funzioni* $\{[\![\_]\!]_s\}_{s\in S}$ (una funzione per ogni categoria sintattica) da $\mathcal{T}$ in una certa famiglia di valori A:

$$[\![\_]\!]_s : \mathcal{T}_s \to A_s$$

Il termine *denotazionale* fa proprio riferimento al fatto che ogni termine (oggetto sintattico) denota uno ed un solo valore. Tuttavia, storicamente si è usato il termine "denotazionale" per le semantiche che rispettano non solo questo principio, ma anche il successivo[8].

Composizionalità Questa associazione deve essere descritta in modo composizionale, cioè il valore corrispondente ad un termine deve essere ottenuto componendo i valori dei suoi sottotermini in modo determinato dall'operatore più esterno. Questo principio può essere espresso formalmente nel modo seguente:

per ogni simbolo di operazione $op : s_1 \dots s_n \to s$ in O,
per ogni $t_1 \in \mathcal{T}_{s_1}, \dots, t_n \in \mathcal{T}_{s_n}$,
$[\![op(t_1, \dots, t_n)]\!]_s = \widetilde{op}([\![t_1]\!]_{s_1}, \dots, [\![t_n]\!]_{s_n})$

dove $\widetilde{op}$ è una funzione fissata associata all'operatore op. In altri termini, la semantica è data per induzione strutturale.

I due principi possono essere riassunti dicendo che la semantica denotazionale è un *omomorfismo* dall'algebra sintattica $\mathcal{T}$ in un'algebra semantica A (vedi Def.A.7 nell'Appendice); in questo modo $\widetilde{op}$ è l'*interpretazione* op^A dell'operatore op in A .
Quindi, per dare la semantica del linguaggio è sufficiente definire l'algebra semantica A, cioè fissare le interpretazioni delle sort e dei simboli di operazione: infatti, dato che deve valere l'uguaglianza sopra, la semantica è determinata univocamente[9] per ogni termine. Ricordiamo infatti (vedi Sez.1.4) che la definizione della semantica è data in modo indipendente dalla particolare rappresentazione scelta per i termini. In particolare quindi assumiamo di risolvere eventuali ambiguità semanticamente rilevanti, in quanto fissata la sintassi astratta è sempre possibile trovare una rappresentazione concreta non ambigua, quindi l'uguaglianza sopra definisce effettivamente una funzione. Si noti che in caso contrario la definizione per induzione strutturale potrebbe attribuire più valori allo stesso oggetto sintattico, in quanto questo potrebbe essere decomposto in più modi.
Tuttavia, in genere si preferisce presentare la semantica dando il codominio delle funzioni semantiche (una per tipo) e, per ognuna di queste, dando delle *clausole semantiche*, una per ogni operatore che costruisce termini di quel tipo.
Consideriamo ad esempio il semplice linguaggio di espressioni naturali e booleane $\mathcal{E}$ introdotto in Sez.2.2 e 2.3, in una presentazione (a scopo illustrativo) leggermente diversa in cui distinguiamo tra espressioni naturali e booleane direttamente a livello sintattico:

$$\begin{array}{lcl} \mathit{BExp} & ::= & \texttt{true} \mid \texttt{false} \mid \texttt{iszero}\ \mathit{Exp} \mid \texttt{if}\ \mathit{BExp}\ \texttt{then}\ \mathit{BExp}\ \texttt{else}\ \mathit{BExp} \\ \mathit{Exp} & ::= & \texttt{if}\ \mathit{BExp}\ \texttt{then}\ \mathit{Exp}\ \texttt{else}\ \mathit{Exp} \mid \texttt{succ}\ \mathit{Exp} \mid \texttt{pred}\ \mathit{Exp} \mid \texttt{0} \end{array}$$

Una possibile semantica denotazionale per tale linguaggio è la seguente. Sia $\Sigma_{\mathcal{E}}$ la segnatura che corrisponde alla sintassi astratta del linguaggio, quindi con sort BExp, Exp e simboli di operazione:
`true`: $\to \mathit{BExp}$
`false`: $\to \mathit{BExp}$
`iszero` _: $\mathit{Exp} \to \mathit{BExp}$
`if` _ `then` _ `else` _: $\mathit{BExp}\ \mathit{BExp}\ \mathit{BExp} \to \mathit{BExp}$
`if` _ `then` _ `else` _: $\mathit{BExp}\ \mathit{Exp}\ \mathit{Exp} \to \mathit{Exp}$
`succ` _: $\mathit{Exp} \to \mathit{Exp}$
`pred` _: $\mathit{Exp} \to \mathit{Exp}$
`0`: $\to \mathit{Exp}$
Sia A l'algebra su $\Sigma_{\mathcal{E}}$ definita come segue:

[8] Lo stile denotazionale è stato introdotto a metà degli anni sessanta da Christopher Strachey e i suoi principi matematici sono stati studiati approfonditamente da Dana Scott sul finire degli anni sessanta.
[9] Nel caso totale, vedi dopo.

- $A_{BExp} = \{T, F\}$
- $A_{Exp} = \mathbb{N}$
- l'interpretazione degli operatori è quella usuale[10], cioè:

  $\texttt{true}^A = T$
  $\texttt{false}^A = F$
  $(\texttt{iszero}\ \_)^A(n) = T$ se $n = 0$, F altrimenti
  $(\texttt{if}\ \_\ \texttt{then}\ \_\ \texttt{else}\ \_)^A(T, bv_1, bv_2) = bv_1$, $(\texttt{if}\ \_\ \texttt{then}\ \_\ \texttt{else}\ \_)^A(F, bv_1, bv_2) = bv_2$, per $bv_1, bv_2 \in \{T, F\}$
  $(\texttt{if}\ \_\ \texttt{then}\ \_\ \texttt{else}\ \_)^A(T, n_1, n_2) = n_1$, $(\texttt{if}\ \_\ \texttt{then}\ \_\ \texttt{else}\ \_)^A(F, n_1, n_2) = n_2$, per $n_1, n_2 \in \mathbb{N}$
  $(\texttt{succ}\ \_)^A(n) = n + 1$
  $(\texttt{pred}\ \_)^A(n) = n - 1$ se $n > 0$, $(\texttt{pred}\ \_)^A(0) = 0$
  $0^A = 0$

Fissare quest'algebra corrisponde a definire la semantica del linguaggio $\mathcal{E}$ con le due funzioni semantiche

$[\![\_]\!]_{BExp} : \mathcal{E}_{BExp} \to \{T, F\}$
$[\![\_]\!]_{Exp} : \mathcal{E}_{Exp} \to \mathbb{N}$

definite dalle seguenti clausole semantiche (utilizziamo *b* come metavariabile per le espressioni booleane ed *e* come metavariabile per le espressioni naturali):

$[\![\texttt{true}]\!]_{BExp} = T$
$[\![\texttt{false}]\!]_{BExp} = F$
$[\![\texttt{iszero}\ e]\!]_{BExp} = T$ se $[\![e]\!]_{Exp} = 0$, F altrimenti
$[\![\texttt{if}\ b\ \texttt{then}\ b_1\ \texttt{else}\ b_2]\!]_{BExp} = [\![b_1]\!]_{BExp}$ se $[\![b]\!]_{BExp} = T$, $[\![b_2]\!]_{BExp}$ se $[\![b]\!]_{BExp} = F$
$[\![\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]_{Exp} = [\![e_1]\!]_{Exp}$ se $[\![b]\!]_{BExp} = T$, $[\![e_2]\!]_{Exp}$ se $[\![b]\!]_{BExp} = F$
$[\![\texttt{succ}\ e]\!]_{Exp} = [\![e]\!]_{Exp} + 1$
$[\![\texttt{pred}\ e]\!]_{Exp} = [\![e]\!]_{Exp} - 1$ se $[\![e]\!]_{Exp} > 0$, 0 altrimenti
$[\![\texttt{0}]\!]_{Exp} = 0$

Si può provare che questa semantica denotazionale è equivalente alla semantica operazionale a piccoli passi data in Sez.2.2 e 2.3, cioè:

- per ogni $b \in \mathcal{E}_{BExp}$, $[\![b]\!]_{BExp} = T$ sse $b \to^\star \texttt{true}$, $[\![b]\!]_{BExp} = F$ sse $b \to^\star \texttt{false}$
- per ogni $e \in \mathcal{E}_{Exp}$, $[\![e]\!]_{Exp} = n$ sse $e \to^\star \texttt{succ}^n 0$.

Si noti che lo stile denotazionale è più astratto, nel senso che siamo interessati solo al valore denotato da un termine, a prescindere dalle diverse strategie di riduzione.
Naturalmente ci sono infinite altre possibili semantiche per il linguaggio $\mathcal{E}$ (una per ogni algebra sulla segnatura corrispondente alla sintassi astratta del linguaggio), per esempio:

- i termini di tipo *Exp* sono interpretati come numeri interi e il simbolo di operazione `pred` è interpretato come il predecessore sugli interi;
- le interpretazioni di `succ` e `pred` sono scambiate, e l'interpretazione dell'if-then-else restituisce sempre il secondo argomento;
- anche i numeri naturali sono interpretati come $\{T, F\}$ e la semantica è quella di parità (ossia tutti i naturali pari sono interpretati come T e tutti i dispari come F);
- tutti i termini sono interpretati come un'unico valore e tutte le funzioni restituiscono questo valore (in questa interpretazione tutti i termini vengono identificati);
- ogni termine è interpretato come se stesso (ossia, si sceglie come interpretazione la sintassi e non si fanno identificazioni tra termini).

[10] Però per semplicità, come già fatto in Sez.2.3, assumiamo che l'operatore predecessore applicato a zero dia zero.

Si diano per esercizio i dettagli delle semantiche illustrate sopra.
Quanto visto finora, cioè il fatto che la semantica denotazionale è l'unico omomorfismo dal linguaggio (dato come algebra sintattica non ambigua) in un'algebra semantica, vale nel caso semplice in cui l'algebra semantica è *totale* (cioè, non ci sono operazioni non definite su qualche argomento, come sarebbe nell'esempio precedente considerando l'interpretazione di `pred` non definita su zero).
Nel caso parziale i due principi continuano a valere (per il primo principio si noti però che vi possono essere termini a cui non viene attribuito alcun valore). Tuttavia, la controparte formale è meno semplice e non ne parleremo in questo corso.

7.2 Semantica denotazionale di $\mathcal{W}$

Come esempio più elaborato, presenteremo la semantica denotazionale di un semplice linguaggio imperativo $\mathcal{W}$. La grammatica di $\mathcal{W}$ è data in Figura 11.

Prog	::=	$\{Decs\ Coms\}$
Decs	::=	$\Lambda \mid Dec\ Decs$
Dec	::=	*Type* ID $=$ *Exp*;
Coms	::=	$\Lambda \mid Com\ Coms$
Com	::=	$\{Decs\ Coms\} \mid$ `if` $(Exp)\ Com$ `else` $Com \mid$
		`while` $(Exp)\ Com \mid$ `print`$(Exp); \mid$ ID $=$ *Exp*;
Exp	::=	NUM $\mid$ `true` $\mid$ `false` $\mid$ ID $\mid (Exp + Exp) \mid \ldots$
Type	::=	`bool` $\mid$ `int`

Figura 11: Grammatica di $\mathcal{W}$

I non terminali NUM e ID corrispondono a due insiemi non specificati rispettivamente dei *numerali* (rappesentazioni dei numeri interi) e degli identificatori.
Nella figura e nel seguito vengono usate le seguenti metavariabili[11]:

- p per indicare un programma;
- ds e d per indicare rispettivamente una sequenza di dichiarazioni e una dichiarazione;
- cs per indicare una sequenza di comandi;
- c per indicare un comando;
- e per indicare un'espressione;
- T per indicare uno dei due tipi `bool` e `int`.

La semantica (dinamica) in stile denotazionale di $\mathcal{W}$ è definita in Figura 12.
Il linguaggio $\mathcal{W}$ è imperativo, cioè il suo modello di esecuzione è basato sulla nozione di *stato*; in questo caso uno stato $s \in State$ è composto da due componenti: la *memoria* e l'*output*. La memoria è modellata da una funzione dall'insieme degli indirizzi (locazioni) Loc all'insieme dei valori Val, mentre l'output è una stringa (anche vuota) di valori corrispondente alla sequenza stampata durante l'esecuzione di un programma. L'insieme Loc degli indirizzi può essere qualsiasi insieme numerabile; l'insieme dei valori è dato dall'unione dell'insieme $\mathbb{Z}$ dei numeri interi e dell'insieme dei valori di verità T e F.
Un ambiente (dinamico) $r \in Env$ è una funzione da identificatori a locazioni; gli unici costrutti che modificano l'ambiente dinamico sono le dichiarazioni. Lo stato, invece, può essere modificato sia dai comandi (in particolare, l'assegnazione modifica la memoria, mentre l'istruzione `print` modifica l'output) che dalle dichiarazioni (infatti l'inizializzazione delle variabili dichiarate ha lo stesso effetto di un'assegnazione). Infine, la valutazione di un'espressione non modifica né l'ambiente, né lo stato, ma restituisce semplicemente un valore.
La dichiarazione di una variabile implica due aggiornamenti: l'ambiente r corrente va modificato in modo che all'identificatore id della variabile sia associata una nuova locazione l, mentre la memoria corrente μ va aggiornata in modo che la locazione l contenga il valore v di inizializzazione della variabile. Il valore v viene ottenuto calcolando l'espressione e nell'ambiente r e nello stato (μ, o). Infine, l'output o rimane immutato.
Il comando `print` permette di modificare l'output; il valore v ottenuto valutando l'espressione e viene aggiunto (tramite l'operatore $\cdot$ di concatenazione tra stringhe) in fondo all'output corrente o.
L'assegnazione permette di modificare la memoria: il contenuto della memoria all'indirizzo $r(id)$ associato all'identificatore id nell'ambiente corrente r viene aggiornato col valore v ottenuto valutando e. L'output non viene modificato. Si noti che

[11] Si noti la differenza tra metavariabili e variabili di programma (denotate da identificatori id).

$[\_]_{Prog}$:$Prog \to Output$
$[\_]_{Decs}$:$Decs \to Env \to State \to (Env \times State)$
$[\_]_{Dec}$:$Dec \to Env \to State \to (Env \times State)$
$[\_]_{Coms}$:$Coms \to Env \to State \to State$
$[\_]_{Com}$:$Com \to Env \to State \to State$
$[\_]_{Exp}$:$Exp \to Env \to State \to Val$

$Output = Val^*$
$Env = \textsc{Id} \to Loc$
$State = Mem \times Output$
$Mem = Loc \to Val$
$Val = \mathbb{Z} \cup \{T, F\}$

$[\{ds\ cs\}]_{Prog} = o'$ se $[\{ds\ cs\}]_{Com}\emptyset\ (\emptyset, \Lambda) = (\mu', o')$

$[\Lambda]_{Decs}\ r\ s = (r, s)$
$[d\ ds]_{Decs}\ r\ s = [ds]_{Decs}\ r'\ s'$ se $[d]_{Dec}\ r\ s = (r', s')$

$[T\ id = e;]_{Dec}\ r\ (\mu, o) = (r[l/id], (\mu[v/l], o))$ se $[e]_{Exp}\ r\ (\mu, o) = v,\ \mu(l)$ indefinito

$[\Lambda]_{Coms}\ r\ s = s$
$[c\ cs]_{Coms}\ r\ s = [cs]_{Coms}\ r\ s'$ se $[c]_{Com}\ r\ s = s'$

$[\{ds\ cs\}]_{Com}\ r\ s = [cs]_{Coms}\ r'\ s'$ se $[ds]_{Decs}\ r\ s = (r', s')$
$[\texttt{if}\ (e)\ c_1\ \texttt{else}\ c_2]_{Com}\ r\ s = \begin{array}{l} [c_1]_{Com}\ r\ s \text{ se } [e]_{Exp}\ r\ s = T, \\ [c_2]_{Com}\ r\ s \text{ se } [e]_{Exp}\ r\ s = F \end{array}$
$[\texttt{while}\ (e)\ c]_{Com}\ r\ s = \begin{cases} [\texttt{while}\ (e)\ c]_{Com}\ r\ s' & \text{se } [e]_{Exp}\ r\ s = T, [c]_{Com}\ r\ s = s' \\ s & \text{se } [e]_{Exp}\ r\ s = F \end{cases}$
$[\texttt{print}(e);]_{Com}\ r\ (\mu, o) = (\mu, o \cdot v)$ se $[e]_{Exp}\ r\ (\mu, o) = v$
$[id = e;]_{Com}\ r\ (\mu, o) = (\mu[v/l], o)$ se $[e]_{Exp}\ r\ (\mu, o) = v, r(id) = l$

$[n]_{Exp}\ r\ s = n^{\mathbb{Z}}$
$[id]_{Exp}\ r\ (\mu, o) = \mu(r(id))$
$[e_1 + e_2]_{Exp}\ r\ s = [e_1]_{Exp}\ r\ s +^{\mathbb{Z}} [e_2]_{Exp}\ r\ s$
$[\texttt{true}]_{Exp}\ r\ s = T$
$[\texttt{false}]_{Exp}\ r\ s = F$
...

Figura 12: Semantica denotazionale (dinamica) di $\mathcal{W}$

in un'assegnazione un identificatore id è valutato in modo diverso a seconda se appare a sinistra o a destra del simbolo di assegnazione: nel primo caso denota una locazione $r(id)$, mentre nel secondo caso denota un valore $\mu(r(id))$. Si parla di *valore sinistro* e *valore destro* dell'identificatore. Questo accade perché nel minilinguaggio imperativo che abbiamo considerato, come per esempio in C e Java, l'operazione di dereferenziazione (ossia, l'estrazione del contenuto di una locazione) è implicita. Nell'estensione imperativa del lambda-calcolo in Sez.5.2, invece, che segue lo stile di ML, tale operazione deve essere indicata esplicitamente.
Si noti che la clausola relativa al while corrisponde alla seguente definizione induttiva di una funzione $f_{e,c,r} = [\![\texttt{while}\ (e)\ c]\!]_{Com}\ r \colon State \to State$:

$$\frac{}{f_{e,c,r}(s) = s}[\![e]\!]_{Exp}s = F$$

$$\frac{f_{e,c,r}([\![c]\!]_{Com}s) = s' \qquad [\![e]\!]_{Exp}s = T}{f_{e,c,r}(s) = s'}$$

Nella clausola di valutazione per un numerale n, $n^{\mathbb{Z}}$ indica il numero intero che esso rappresenta.
Questo modello permette di descrivere in modo naturale altri meccanismi tipici dei linguaggi imperativi (per esempio, espressioni strutturate a sinistra di un'assegnazione, puntatori, passaggio per riferimento).

8 Correttezza di algoritmi imperativi

8.1 Introduzione

Una *specifica* è una descrizione di *cosa* deve fare un sistema software. Una particolare implementazione descriverà il *come*. Una specifica rappresenta quindi un contratto ideale tra sviluppatore e utente.
Una specifica informale è ovviamente spesso ambigua e imprecisa. Una specifica formale è data invece attraverso delle formule logiche (o *asserzioni*) che hanno una semantica precisa.
Alcuni stili di specifica:

- algebrico (tipi di dato in contesto funzionale),
- asserzioni alla Hoare (paradigma imperativo e object-oriented),
- logica temporale (paradigma concorrente).

L'uso di asserzioni è relativo alla *correttezza* dei programmi. Si noti che "correttezza" è un concetto relativo. Non ha senso per esempio chiedersi se il comando

```
x = y + 1
```

è corretto; lo è rispetto alla specifica "rendere `x`, `y` di valore diverso", mentre non lo è rispetto alla specifica "rendere `x` negativo".
Quindi la nozione di correttezza si applica a coppie ⟨specifica, codice⟩. Di conseguenza, per poter definire formalmente cosa significa che un programma è corretto rispetto a una specifica, entrambi devono essere valutati nello stesso dominio semantico.
Semplicemente *scrivere* la specifica è già un passo in avanti verso l'assicurare la correttezza perché:

- chiarisce la differenza tra ciò che si vuol fare e come,
- guida la documentazione del software (formalizzazione dei commenti),
- fornisce una base per testing/debugging.

In queste note descriveremo un tipo particolare di asserzioni, le *asserzioni alla Hoare*, che permettono di esprimere formalmente la correttezza di algoritmi descritti tramite un linguaggio imperativo. Un'asserzione alla Hoare è una tripla $\{P\}\ c\ \{R\}$ dove c è un comando (quindi, semanticamente, una trasformazione di stato) e P, R sono predicati[12] sullo stato manipolato da c, detti rispettivamente *precondizione* e *postcondizione*.
Distingueremo asserzioni di *correttezza parziale*, scritte nella forma detta sopra, e asserzioni di *correttezza totale*, scritte $\{P\}\ c\ \{\Downarrow R\}$. La differenza è che per parlare di correttezza totale è necessario che il comando termini, mentre ciò non è richiesto per la correttezza parziale. Ovviamente la differenza è significativa solo nel caso di comandi che possono non terminare, come avviene nel caso di cicli (while, repeat, etc.), di funzioni ricorsive o di errori a run-time. Nel semplice linguaggio che considereremo nel seguito l'unica fonte di non terminazione sarà data dal comando while.

[12] Nell'approccio *estensionale*, vedi dopo.

La correttezza parziale potrebbe apparire poco utile in pratica (di qualunque algoritmo che risolva un problema sembra un requisito minimo richiedere che termini!), ma viene studiata perché spesso risulta conveniente provare separatamente la correttezza parziale e la terminazione.
Per esempio, la seguente asserzione (in questo caso possiamo indifferentemente usare la correttezza parziale o totale)

$\{\mathtt{x} \geq 9\}$ `x=x+5` $\{\Downarrow \mathtt{x} \geq 13\}$

corrisponde a richiedere che, partendo da uno stato in cui vale $\mathtt{x} \geq 9$, l'esecuzione del comando `x=x+5` termini in uno stato in cui vale $\mathtt{x} \geq 13$. Questa asserzione è intuitivamente valida.

8.2 Linguaggio di riferimento

Utilizzeremo, per semplicità espositiva, un linguaggio analogo a quello su cui è stata data la versione originale delle asserzioni alla Hoare, cioè un linguaggio molto semplice MINI$\mathcal{W}$ che corrisponde a un sottoinsieme del linguaggio $\mathcal{W}$ descritto precedentemente (Sez.7.2) dove si considerano solo comandi ed espressioni, le espressioni non hanno side-effect, l'ambiente è fissato (cioè non vi è un costrutto di blocco) e non si considera l'output.[13]
La sintassi di MINI$\mathcal{W}$ e la seguente:

$$\begin{array}{lll} \mathit{Com} & ::= & \mathtt{skip} \mid \textsc{Id} = \mathit{Exp} \mid \mathit{Com}\,\mathtt{;}\,\mathit{Com} \mid \mathtt{while\ (}\mathit{BExp}\mathtt{)\ \{}\ \mathit{Com}\ \mathtt{\}} \mid \\ & & \mathtt{if\ (}\mathit{BExp}\mathtt{)\{}\mathit{Com}\mathtt{\}\ else\ \{}\mathit{Com}\mathtt{\}} \\ \mathit{Exp} & ::= & \textsc{Num} \mid \textsc{Id} \mid (\mathit{Exp} + \mathit{Exp}) \mid \ldots \\ \mathit{BExp} & ::= & \mathtt{true} \mid \mathtt{false} \mid \ldots \end{array}$$

Si noti che, grazie alle assunzioni dette sopra, la semantica formale di MINI$\mathcal{W}$ risulta molto più semplice di quella di $\mathcal{W}$ (si provi a darla tutta per esercizio); in particolare, dato che si considera un ambiente fissato (cioè un insieme fissato di identificatori di variabile), le nozioni di ambiente e stato possono "collassare", quindi le funzioni semantiche per comandi ed espressioni hanno il seguente tipo:

$[\![-]\!]_{\mathit{Com}} : \mathit{Com} \to \mathit{State} \to \mathit{State}$
$[\![-]\!]_{\mathit{Exp}} : \mathit{Exp} \to \mathit{State} \to \mathit{Val}$
$[\![-]\!]_{\mathit{BExp}} : \mathit{BExp} \to \mathit{State} \to \{T, F\}$

Il problema di trovare un comando che soddisfi una specifica data si può quindi formulare come segue:

dati P, R predicati su State, trovare c tale che:
$\forall s \in \mathit{State}.P(s) \Rightarrow R([\![C]\!]\, s)$.

8.3 Asserzioni alla Hoare di correttezza parziale

Specificheremo proprietà dei programmi in MINI$\mathcal{W}$ attraverso delle triple dette *asserzioni alla Hoare*, della forma $\{P\}\ c\ \{R\}$, dove P e R sono dei predicati sullo stato, detti rispettivamente *precondizione* e *postcondizione*, e c è un comando. Intuitivamente, il significato di tali asserzioni è il seguente:

> se P vale nello stato iniziale (cioè prima di eseguire c) e l'esecuzione di c termina (producendo quindi uno stato finale), allora R deve valere su tale stato.

Si noti che perché l'asserzione sia valida *non* si richiede che l'esecuzione del comando termini; si parla in questo caso di asserzioni di *correttezza parziale*.

Esempio 8.1 Un esempio di asserzione alla Hoare di correttezza parziale è il seguente.

$\{\mathtt{x} = x_0 \wedge \mathtt{y} = y_0\}$ `while (x` $\neq$ `0) { x=x-1; y=y+1 }` $\{\mathtt{x} = 0 \wedge \mathtt{y} = x_0 + y_0\}$

Qui x_0, y_0 denotano due numeri interi arbitrari. Questa asserzione è intuitivamente valida. Infatti, se $x_0 \geq 0$ è facile vedere che l'esecuzione del comando termina e alla fine `x` ha sicuramente assunto il valore 0 e `y` ha incrementato il suo valore iniziale di una quantità uguale al valore iniziale di `x`; se invece $x_0 < 0$ l'esecuzione del comando non termina e quindi, in base alla definizione data sopra di correttezza parziale, l'asserzione risulta banalmente valida.

Vedremo tra poco come definire e provare formalmente la validità di un'asserzione. Prima però sono necessarie alcune precisazioni.

[13] Consideriamo inoltre per comodità una sintassi leggermente diversa in cui distinguiamo le espressioni booleane a livello sintattico, il `;` è un operatore binario, utilizziamo sempre le parentesi {, } per evitare ambiguità e il comando nullo `skip` sostituisce la sequenza di comandi vuota.

Approccio intensionale ed estensionale. In generale, si parla di descrizione *estensionale* di una funzione intendendo il grafico, cioè l'insieme delle coppie messe in corrispondenza (che può essere infinito). Una descrizione *intensionale* è invece una descrizione finita data utilizzando un qualche formalismo sintattico. Ovviamente diverse descrizioni intensionali possono corrispondere alla stessa descrizione estensionale.
Nel caso delle asserzioni alla Hoare, per non essere vincolati a un particolare linguaggio, adotteremo l'approccio estensionale, cioè considereremo triple $\{P\}\ c\ \{R\}$ dove P e R non sono formule (oggetti sintattici) ma oggetti semantici, cioè predicati sullo stato (funzioni totali da stati in valori booleani). In altre parole non ci preoccuperemo di fissare una particolare rappresentazione sintattica per le asserzioni sullo stato; negli esempi utilizzeremo il metalinguaggio matematico e logico usuale.
Ad un'espressione booleana *b* del linguaggio MINI$\mathcal{W}$ corrisponderà ovviamente un predicato $[\![b]\!]$ sullo stato (la sua semantica). Assumiamo infatti l'ipotesi semplificativa che la valutazione delle espressioni di MINI$\mathcal{W}$ non possa dar luogo a non terminazione.

Variabili di programma e variabili logiche. Generalizzando quanto detto sopra per le espressioni booleane, a un'espressione *e* del linguaggio MINI$\mathcal{W}$ corrisponderà una funzione $[\![e]\!]$ da stati a valori (la sua semantica). Tali funzioni potranno essere utilizzate per descrivere predicati sullo stato. Per esempio, a un identificatore di variabile id è associata la funzione $[\![id]\!]$ che per ogni stato s restituisce $s(id)$. L'esempio sopra dovrebbe quindi essere più correttamente scritto nel modo seguente
$\{\ [\![\texttt{x}]\!] = x_0 \wedge [\![\texttt{y}]\!] = y_0\}$ `while (x ≠ 0) { x=x-1; y=y+1 }` $\{\ [\![\texttt{x}]\!] = 0 \wedge [\![\texttt{y}]\!] = x_0 + y_0\}$
Tuttavia, quando non ci sia ambiguità ometteremo le parentesi semantiche per alleggerire la notazione. Bisogna però sempre tenere presente la differenza tra le *variabili di programma* (come `x`, `y` nell'esempio) che possono essere utilizzate nella descrizione di un predicato sullo stato, e denotano una funzione da stati in valori, e le *variabili logiche* che possono essere utilizzate nel metalinguaggio, come x_0, y_0 nell'esempio per indicare due arbitrari valori interi. Per sottolineare tale differenza utilizzeremo in queste note il font `typewriter` per gli elementi del linguaggio MINI$\mathcal{W}$, mentre utilizzeremo il font *italico* per il metalinguaggio.

Operatori su predicati. Data un'asserzione P su $State$, indicheremo con $\{P\}$ l'insieme degli stati che soddisfano P e talvolta potremo confondere per comodità le due nozioni.
Diremo che un'asserzione P è *più forte* di un'asserzione Q se l'insieme degli stati che verificano P è contenuto nell'insieme degli stati che verificano Q.
Indicheremo con `false` e `true` le asserzioni rispettivamente sempre falsa e sempre vera, che risultano quindi essere la più forte e la più debole:

$\{\texttt{false}\} = \emptyset$,
$\{\texttt{true}\} = State$.

Inoltre, se P, Q sono asserzioni sullo stato, allora $\neg P$, $P \vee Q$, $P \wedge Q$, $P \Rightarrow Q$ sono le asserzioni sullo stato definite da:

$(\neg P)(s) = \neg P(s)$,
$(P \vee Q)(s) = P(s) \vee Q(s)$,
$(P \wedge Q)(s) = P(s) \wedge Q(s)$,
$(P \Rightarrow Q)(s) = P(s) \Rightarrow Q(s)$.

In altri termini questi operatori sono semplicemente gli usuali operatori booleani "trasportati" a livello dei predicati sullo stato.
Vale il fatto seguente:

$P \Rightarrow Q$ sse $\{P\} \subseteq \{Q\}$.

Infine, se P è un predicato sullo stato, *e* e id sono rispettivamente un'espressione e un identificatore di MINI$\mathcal{W}$, indicheremo con $P[[\![e]\!]/id]$ il predicato sullo stato definito nel modo seguente:
$P[[\![e]\!]/id](s) = P(s[[\![e]\!]\, s/id])$
dove a destra $-[-/-]$ indica l'usuale operazione di sostituzione (aggiornamento) su funzioni parziali (vedi Def.A.2 nell'Appendice). Si noti che questa operazione corrisponde, per ogni rappresentazione sintattica fissata dei predicati, alla sostituzione testuale dell'identificatore con l'espressione; tuttavia non possiamo dare una definizione come sostituzione testuale in quanto abbiamo adottato l'approccio estensionale, quindi abbiamo predicati (oggetti semantici) e non formule (oggetti sintattici).

Validità. Diremo che un'asserzione alla Hoare di correttezza parziale $\{P\}\ c\ \{R\}$ è *valida*, e scriveremo $\models \{P\}\ c\ \{R\}$, se e solo se:

per ogni $s \in State$, se $P(s) = T$ e $[\![c]\!]\, s = s'$, allora $R(s') = T$.

Si noti che se $[\![c]\!]\, s$ è indefinito, cioè il comando c non termina a partire dallo stato iniziale s, l'implicazione risulta vera (poiché la premessa è sempre falsa). Ciò corrisponde al fatto che stiamo considerando la correttezza parziale. Quindi la definizione potrebbe equivalentemente essere data nel modo seguente:

per ogni $s \in State$, se $P(s) = T$, allora $[\![c]\!]\, s$ è indefinito oppure $R([\![c]\!]\, s) = T$.

Se un'asserzione $\{P\}\ c\ \{R\}$ è valida, diremo che P è una *precondizione* per c rispetto a R (e viceversa che R è una *postcondizione* per c rispetto a P).
È possibile provare la validità di un'asserzione alla Hoare direttamente dalla definizione.

Esempio 8.2 Si consideri per esempio la seguente asserzione:

$\{\mathtt{x} > 0 \vee \mathtt{y} > 0\}\ \mathtt{if\ (x>0)\{z=x\}\ else\ \{z=y\}}\ \{\mathtt{z} > 0\}$

Per provare che è valida, consideriamo uno stato s su cui vale la precondizione e l'esecuzione del comando termina, cioè

$s(\mathtt{x}) > 0 \vee s(\mathtt{y}) > 0$,
$[\![\mathtt{if\ (x>0)\{z=x\}\ else\ \{z=y\}}]\!]\, s = s'$.

Dobbiamo far vedere che vale la postcondizione sullo stato finale, cioè $s'(\mathtt{z}) > 0$. Infatti, nel caso $s(\mathtt{x}) > 0$ sia vera, $s' = [\![\mathtt{if\ (x>0)\{z=x\}\ else\ \{z=y\}}]\!]\, s = [\![\mathtt{z=x}]\!]\, s = s[s(\mathtt{x})/\mathtt{z}]$, quindi $s'(\mathtt{z}) = s(\mathtt{x}) > 0$; nel caso $s(\mathtt{x}) > 0$ sia falsa, $s(\mathtt{y}) > 0$ e $s' = [\![\mathtt{if\ (x>0)\{z=x\}\ else\ \{z=y\}}]\!]\, s = [\![\mathtt{z=y}]\!]\, s = s[s(\mathtt{y})/\mathtt{z}]$, quindi $s'(\mathtt{z}) = s(\mathtt{y}) > 0$.

Weakest (liberal) precondition. Tra tutte le precondizioni P che rendono valida un'asserzione alla Hoare $\{P\}\ c\ \{R\}$, è di particolare interesse *la più debole*, cioè quella implicata da tutte le altre. Infatti questa precondizione può essere vista come un insieme minimo di requisiti da richiedere per ottenere che R valga se il comando termina. Formalmente, definiremo, per ogni c comando e R postcondizione, la *weakest (liberal) precondition*[14] (abbreviato wlp) di c rispetto a R, denotata $wlp(c, R)$, nel modo seguente:

$wlp(c, R)(s) = T$ sse $[\![c]\!]\, s = s' \Rightarrow R(s')$.

Si noti che, se $[\![c]\!]\, s$ è indefinito, cioè il comando c non termina a partire dallo stato iniziale s, l'implicazione risulta vera (poiché la premessa è sempre falsa) quindi $wlp(c, R)(s)$ è vera. Ciò corrisponde al fatto che stiamo considerando la correttezza parziale. Quindi la definizione potrebbe equivalentemente essere data nel modo seguente:

$wlp(c, R)(s) = T$ sse $[\![c]\!]\, s$ indefinito oppure $R([\![c]\!]\, s)$.

La definizione può anche essere data in maniera equivalente in termini di insiemi di stati:

$\{wlp(c, R)\} = \{s \mid [\![c]\!]\, s = s' \Rightarrow R(s')\} = \{s \mid [\![c]\!]\, s$ indefinito oppure $R([\![c]\!]\, s)\}$.

Ossia, la weakest (liberal) precondition di c rispetto a R è data da tutti gli stati iniziali per cui eseguendo c si ottiene o non terminazione o uno stato finale su cui vale R.
Proviamo ora che $wlp(c, R)$ è effettivamente la precondizione più debole per c rispetto a R, ossia che: è davvero una precondizione per c rispetto a R; ogni altra precondizione per c rispetto a R la implica. Formalmente:

1. $\models \{wlp(c, R)\}\ c\ \{R\}$
2. se $\models \{P\}\ c\ \{R\}$, allora $P \Rightarrow wlp(c, R)$.

Prova di (1) Dobbiamo provare che per ogni $s \in State$, se $wlp(c, R)(s) = T$ e $[\![c]\!]\, s = s'$, allora $R(s') = T$. Ma questo segue direttamente dalla definizione di $wlp(c, R)$.

Prova di (2) Sia $\models \{P\}\ c\ \{R\}$. Dobbiamo provare che, per ogni $s \in State$, se $P(s) = T$ allora $wlp(c, R)(s) = T$. Ma dalla validità di $\{P\}\ c\ \{R\}$ sappiamo che $[\![C]\!](s) = s' \Rightarrow R(s')$, quindi $wlp(c, R)(s) = T$.

Esempio 8.3 Ricaviamo, a titolo di esempio, la wlp del comando $c = \mathtt{if\ (x>0)\{z=x\}\ else\ \{z=y\}}$ rispetto alla postcondizione $\mathtt{z} > 0$. Dalla definizione, $wlp(c, \mathtt{z} > 0)(s) = T$ sse $[\![c]\!]\, s = s' \Rightarrow \mathtt{z}(s') > 0$. Poiché $[\![c]\!]\, s = [\![\mathtt{z=x}]\!](s) = s[s(\mathtt{x})/\mathtt{z}]$ se $s(\mathtt{x}) > 0$, e $[\![c]\!]\, s = [\![\mathtt{z=y}]\!](s) = s[s(\mathtt{y})/\mathtt{z}]$ se $s(\mathtt{x}) \leq 0$, si ha che $wlp(c, \mathtt{x} > 0)(s) = tt$ sse vale $s(\mathtt{x}) > 0$ oppure $s(\mathtt{y}) > 0$.

[14] L'appellativo "liberal" distingue questa nozione da quella analoga nel caso di correttezza totale.

$$\text{(SKIP)} \qquad \frac{}{\{R\}\ \texttt{skip}\ \{R\}}$$

$$\text{(ASSIGN)} \qquad \frac{}{\{R[[\![e]\!]\,/id]\}\ id = e\ \{R\}}$$

$$\text{(SEQ)} \qquad \frac{\{P\}\ c_1\ \{Q\} \quad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1\,;c_2\ \{R\}}$$

$$\text{(IF)} \qquad \frac{\{P \wedge [\![b]\!]\}\ c_1\ \{R\} \quad \{P \wedge \neg\,[\![b]\!]\}\ c_2\ \{R\}}{\{P\}\ \texttt{if (}b\texttt{)}\ \{c_1\}\ \texttt{else}\ \{c_2\}\ \{R\}}$$

$$\text{(WHILE)} \qquad \frac{\{INV \wedge [\![b]\!]\}\ c\ \{INV\}}{\{INV\}\ \texttt{while (}b\texttt{)}\ \{\ c\ \}\ \{INV \wedge \neg [\![b]\!]\}}$$

$$\text{(CONS)} \qquad \frac{\{P'\}\ c\ \{R'\} \quad P \Rightarrow P' \quad R' \Rightarrow R}{\{P\}\ c\ \{R\}}$$

Figura 13: Sistema di prova di Hoare per la correttezza parziale

Diamo ora una caratterizzazione esplicita della weakest (liberal) precondition per i comandi di MINI$\mathcal{W}$ (omettiamo la caratterizzazione per il comando while).

Skip Per definizione di wlp $wlp(\texttt{skip}, R)(s) = T$ sse $[\![\texttt{skip}]\!]\, s = s' \Rightarrow R(s')$, quindi, per definizione di $[\![\texttt{skip}]\!]$, sse $R(s)$. Quindi $wlp(\texttt{skip}, R) = R$.

Assegnazione Per definizione di wlp $wlp(X = e, R)(s) = T$ sse $[\![X = e]\!]\, s = s' \Rightarrow R(s')$, quindi, per definizione di $[\![X = e]\!]$, sse $R(s[[\![e]\!]\, s/id])$. Quindi $wlp(X = e, R) = R[[\![E]\!]\,/id]$.

Sequenza Per definizione di wlp $wlp(c_1\,;c_2, R)(s) = T$ sse $[\![c_1\,;c_2]\!]\, s = s' \Rightarrow R(s')$, quindi, per definizione di $[\![c_1\,;c_2]\!]$, sse $[\![c_2]\!]([\![c_1]\!]\, s) = s' \Rightarrow R(s')$ sse $[\![c_1]\!]\, s = s'' \Rightarrow wlp(c_2, R)(s'')$ sse $wlp(C_1, wlp(c_2, R))(s)$. Quindi $wlp(c_1\,;c_2, R) = wlp(C_1, wlp(c_2, R))$.

If Per definizione di wlp $wlp(\texttt{if (}b\texttt{)}\ \{c_1\}\ \texttt{else}\ \{c_2\}, R)(s) = T$ sse $[\![\texttt{if (}b\texttt{)}\ \{c_1\}\ \texttt{else}\ \{c_2\}]\!]\, s = s' \Rightarrow R(s')$, quindi, per definizione di $[\![\texttt{if (}b\texttt{)}\ \{c_1\}\ \texttt{else}\ \{c_2\}]\!]$, sse $[\![b]\!]\, s = T$ e $[\![c_1]\!]\, s = s' \Rightarrow R(s')$ oppure $[\![b]\!]\, s = F$ e $[\![c_2]\!]\, s = s' \Rightarrow R(s')$. Quindi $wlp(\texttt{if (}b\texttt{)}\ \{c_1\}\ \texttt{else}\ \{c_2\}, R) = ([\![b]\!] \wedge wlp(C_1, R)) \vee (\neg\,[\![b]\!] \wedge wlp(C_2, R))$.

Un modo alternativo di provare che un'asserzione alla Hoare $\{P\}\ c\ \{R\}$ è valida è quindi di provare che $P \Rightarrow wlp(c, R)$.

Sistema di prova. Definiamo ora un *sistema di prova* che ci fornirà un metodo molto più pratico per provare la validità di un'asserzione alla Hoare.
Nel contesto dei metodi di specifica formale, fissato un certo insieme di asserzioni e definitane formalmente la validità, si intende in genere per *sistema di prova* una definizione induttiva, data tramite metaregole, di un sottoinsieme delle asserzioni.
Il sistema di prova di Hoare per la correttezza parziale è dato in Fig.13. È formato da sei regole di inferenza (o metaregole), una per ogni tipo di comando di MINI$\mathcal{W}$ più la regola (CONS) che permette di rafforzare la precondizione e indebolire la postcondizione in un'asserzione alla Hoare.
Il termine metaregole sta a significare che ognuna di esse definisce uno schema di regola, nel senso che le metavariabili (per esempio c, P, R, e, ...) che vi appaiono possono essere istanziate con arbitrari comandi, predicati, espressioni, etc. Per ogni istanziazione si ottiene una *regola* (nel senso dei sistemi induttivi, vedi Sez.1.1), cioè una coppia $\langle$ insieme di premesse, conseguenza$\rangle$ in cui premesse e conseguenza sono asserzioni alla Hoare. Il sistema di prova di Hoare definisce quindi induttivamente un sottoinsieme $\mathcal{H}$ delle asserzioni alla Hoare; scriveremo $\vdash \{P\}\ c\ \{R\}$ per indicare che l'asserzione $\{P\}\ c\ \{R\}$ appartiene all'insieme $\mathcal{H}$, ossia è deducibile (ha un albero di prova). Infatti, dato un sistema di prova, l'uso del sistema per provare la validità di un'asserzione consiste nel costruire un albero (etichettato con asserzioni), detto albero di prova, che abbia l'asserzione da provare come radice e dove, per ogni nodo, l'etichetta del nodo e le etichette dei figli corrispondano rispettivamente alla conseguenza e alle premesse di un'istanziazione di una metaregola. In particolare quindi le foglie devono corrispondere a istanziazioni di assiomi (regole senza premesse).

Si noti che il sistema *non* fornisce un algoritmo per controllare se un'asserzione $\{P\}\ c\ \{R\}$ è valida o no. Infatti, se la regola corrispondente al tipo di comando c non è direttamente istanziabile, occorrerà trovare un'istanziazione opportuna della regola (CONS), e inoltre essere in grado di stabilire se valgono le due implicazioni nelle side condition.
Nella regola (WHILE), un'asserzione tale che la premessa $\{INV \wedge [\![b]\!]\}\ c\ \{INV\}$ sia verificata si dice *invariante* del ciclo while; infatti, assumendo che essa valga prima di eseguire il ciclo while, dalla premessa si ha che essa vale ancora ogni volta che si esegue il body del ciclo while, e quindi è una asserzione la cui verità non *varia* qualunque sia il numero di iterazioni eseguite.
Proviamo ora che il sistema di Hoare permette di dedurre tutte e sole le asserzioni valide. Ciò è espresso formalmente dal seguente teorema.

Teorema 8.4 Il sistema di prova di Hoare per la correttezza parziale dato in Fig.13 è sound e completo rispetto alla validità, cioè

$$\models \{P\}\ c\ \{R\} \text{ sse } \vdash \{P\}\ c\ \{R\}.$$

Dato un sistema di prova, dire che il sistema è *sound* significa dire che non è possibile dedurre cose sbagliate, cioè non è possibile provare asserzioni non valide. Questa è una proprietà "indispensabile" che ci aspettiamo valere per ogni sistema di prova. Invece, dire che il sistema è *completo* significa dire che è possibile dedurre *tutte* le cose giuste, cioè è possibile provare tutte le asserzioni valide. Questa è una proprietà che non sempre vale per un sistema di prova, perché in genere il sistema potrebbe non essere sufficientemente potente per riuscire a provare tutte le asserzioni valide. Nel nostro caso, il risultato di completezza vale nell'approccio estensionale, cioè assumendo che la precondizione e la postcondizione siano predicati arbitrari sullo stato. In un approccio intensionale dove la precondizione e la postcondizione sono espresse in un linguaggio di asserzioni fissato, il fatto che la completezza valga o meno dipende da quanto è potente tale linguaggio.

Prova di soundness. Dobbiamo provare che $\vdash \{P\}\ c\ \{R\} \Rightarrow \models \{P\}\ c\ \{R\}$. La prova è per induzione sulla definizione di $\mathcal{H}$, cioè dobbiamo provare che ogni (istanziazione di) metaregola è sound nel senso che se le premesse sono valide allora la conseguenza è valida. Nella terminologia dei sistemi induttivi questo corrisponde a provare che l'insieme delle asserzioni valide è chiuso rispetto alla definizione induttiva di $\mathcal{H}$.
Per ogni metaregola relativa a un tipo di comando daremo una prova intuitiva informale, una prova formale diretta (cioè dalla definizione di validità) che corrisponde a una formalizzazione della prova intuitiva, e una prova formale alternativa che utilizza la caratterizzazione della wlp per quel comando (non daremo quindi questa prova formale alternativa per il comando while).

Skip **Prova informale** Se R vale prima di eseguire `skip`, dato che il comando non fa nulla è chiaro che R vale anche dopo.

Prova diretta Dobbiamo provare che per ogni $s \in State$, se $P(s) = T$, allora $[\![\texttt{skip}]\!]\, s = s' \Rightarrow R(s') = T$. Poiché $[\![\texttt{skip}]\!]\, s = s$ la tesi vale banalmente.

Prova con wlp Dobbiamo provare che $R \Rightarrow wlp(\texttt{skip}, R)$. Poiché $wlp(\texttt{skip}, R) = R$ la tesi vale banalmente.

Assegnazione **Prova informale** Se $R[[\![e]\!]/id]$ vale prima di eseguire $X = e$, dato che il comando aggiorna lo stato associando $[\![e]\!]$ a id è chiaro che vale R dopo.

Prova diretta Dobbiamo provare che per ogni $s \in State$, se $P(s) = T$, allora $[\![X = e]\!]\, s = s' \Rightarrow R(s') = T$. Poiché $[\![X = e]\!]\, s = s[[\![e]\!]/id]$, dobbiamo provare che $R(s[[\![e]\!]/id]) = T$, e la tesi vale per definizione dell'operatore di aggiornamento.

Prova con wlp Dobbiamo provare che $R[[\![e]\!]/id] \Rightarrow wlp(X = e, R)$. Poiché $wlp(X = e, R) = R[[\![e]\!]/id]$ la tesi vale banalmente.

Sequenza **Prova informale** Dobbiamo provare che se vale P prima di eseguire $c_1 ; c_2$ e il comando termina, allora vale R. Se il comando termina deve ovviamente terminare anche c_1, e dalla prima premessa sappiamo allora che vale Q, quindi vale Q prima di eseguire c_2. Anche c_2 deve ovviamente terminare, quindi dalla seconda premessa sappiamo che vale R.

Prova diretta Dobbiamo provare che per ogni $s \in State$, se $P(s) = T$, allora $[\![c_1 ; c_2]\!]\, s = s' \Rightarrow R(s') = T$, cioè $[\![C_2]\!]([\![C_1]\!]\, s) = s' \Rightarrow R(s') = T$. Se $[\![C_2]\!]([\![C_1]\!]\, s) = s'$, deve esistere uno stato s'' tale che $[\![C_1]\!]\, s = s''$ e $[\![C_2]\!]\, s'' = s'$. Allora dal fatto che la prima premessa è valida si ha che $Q(s'') = T$, e da questo più la validità della seconda premessa si ha che $R(s') = T$.

Prova con wlp Dobbiamo provare che $P \Rightarrow wlp(c_1, wlp(c_2, R))$. Dalla prima premessa sappiamo che $P \Rightarrow wlp(c_1, Q)$; dalla seconda premessa sappiamo che $Q \Rightarrow wlp(c_2, R)$ e quindi, dato che l'operatore $wlp(c, -)$ è monotono[15], che $wlp(c_1, Q) \Rightarrow wlp(c_1, wlp(c_2, R))$. Dai due fatti segue la tesi per transitività dell'implicazione.

If **Prova informale** Dobbiamo provare che se vale P prima di eseguire `if (`b`) {`c_1`} else {`c_2`}` e il comando termina, allora vale R. Eseguire il comando significa anzitutto valutare l'espressione b; poiché abbiamo assunto che la valutazione delle espressioni non possa dar luogo a non terminazione, otterremo T o F. Nel primo caso si esegue c_1, l'esecuzione termina per l'ipotesi quindi dalla prima premessa sappiamo che vale R dopo. Analogamente nel secondo caso.

Prova diretta Dobbiamo provare che per ogni $s \in State$, se $P(s) = T$, allora $[\![\texttt{if (}b\texttt{) \{}c_1\texttt{\} else \{}c_2\texttt{\}}]\!]\, s = s' \Rightarrow R(s') = T$. Sia $[\![b]\!]\, s = T$. Allora $[\![\texttt{if (}b\texttt{) \{}c_1\texttt{\} else \{}c_2\texttt{\}}]\!]\, s = [\![c_1]\!]\, s$, e la tesi segue dal fatto che la prima premessa è valida. Analogamente se $[\![b]\!]\, s = F$.

Prova con wlp Dobbiamo provare che $P \Rightarrow ([\![b]\!] \wedge wlp(C_1, R)) \vee (\neg\, [\![b]\!] \wedge wlp(C_1, R))$. Dalla prima premessa sappiamo che $P \wedge [\![b]\!] \Rightarrow wlp(c_1, R)$; dalla seconda premessa sappiamo che $P \wedge \neg\, [\![b]\!] \Rightarrow wlp(c_2, R)$. Dai due fatti segue facilmente la tesi.

While **Prova informale** Dobbiamo provare che se vale INV prima di eseguire `while (`b`) {` c `}` e il comando termina, allora vale $INV \wedge \neg\, [\![b]\!]$. Se il comando termina vuol dire che si è eseguito un numero finito di volte il body, sia $n \geq 0$ questo numero. È semplice allora provare per induzione aritmetica su n che ogni volta all'inizio dell'esecuzione del body vale INV: infatti questo vale la prima volta e ogni volta continua a valere in base alla premessa. All'ultimo passo quindi vale ancora INV, vale $\neg b$ perché altrimenti si effettuerebbe ancora un passo e quindi non si fa nulla e vale INV alla fine.

[15] Cioè se $P \Rightarrow Q$, allora per ogni comando c si ha che $wlp(C, P) \Rightarrow wlp(c, Q)$; lo si provi per esercizio.

Prova diretta Dobbiamo provare che per ogni $s \in State$, se $INV(s) = T$, allora $[\![\texttt{while (}b\texttt{) \{ }c\texttt{ \}}]\!]\, s = s' \Rightarrow (INV \wedge \neg [\![b]\!])(s') = T$. Proviamo questa tesi per induzione sulla definizione della funzione $[\![\texttt{while (}b\texttt{) \{ }c\texttt{ \}}]\!]$.

- Sia $[\![b]\!]\, s = F$, quindi $[\![\texttt{while (}b\texttt{) \{ }c\texttt{ \}}]\!]\, s = s$. Allora la tesi vale banalmente.
- Sia $[\![b]\!]\, s = T$, quindi $[\![\texttt{while (}b\texttt{) \{ }c\texttt{ \}}]\!]\, s = s' = [\![\texttt{while (}b\texttt{) \{ }c\texttt{ \}}]\!]([\![c]\!]\, s)$; allora esiste s'' tale che $s'' = [\![c]\!]\, s$, quindi dalla premessa sappiamo che vale $INV(s'')$ e per ipotesi induttiva, dato che $[\![\texttt{while (}b\texttt{) \{ }c\texttt{ \}}]\!]\, s'' = s'$, si ha la tesi.

Cons Dobbiamo provare che se $\{P'\}\ c\ \{R'\}$ è valida, allora lo è anche $\{P\}\ c\ \{R\}$ se $P \Rightarrow P'$ e $R' \Rightarrow R$; questo segue banalmente dalla definizione di validità.

Prova di completezza. Dobbiamo provare che $\models \{P\}\ c\ \{R\} \Rightarrow \vdash \{P\}\ c\ \{R\}$. A tale scopo, proveremo che

(*) dato un comando c e una postcondizione R si ha $\vdash \{wlp(c, R)\}\ c\ \{R\}$.

Infatti a questo punto $\vdash \{P\}\ c\ \{R\}$ si può provare istanziando la regola (CONS); le condizioni a lato sono verificate perché se $\models \{P\}\ c\ \{R\}$ sappiamo che $P \Rightarrow wlp(c, R)$.
La prova di (*) è per induzione sulla struttura dei comandi di MINI$\mathcal{W}$, cioè dobbiamo provare che l'implicazione sopra vale per un comando c (con R arbitraria) se vale per i sottocomandi che lo compongono.

Skip Sappiamo che $wlp(\texttt{skip}, R) = R$, e $\vdash \{R\}\ \texttt{skip}\ \{R\}$ si ottiene istanziando la regola (SKIP).

Assegnazione Sappiamo che $wlp(X = e, R) = R[[\![e]\!]/id]$, e $\vdash \{R[[\![e]\!]/id]\}\ \ id = e\ \ \{R\}$ si ottiene istanziando la regola (ASSIGN).

Sequenza Sappiamo che $wlp(c_1\texttt{;}\, c_2, R) = wlp(c_1, wlp(c_2, R))$, e $\vdash \{wlp(c_1, wlp(c_2, R))\}\ c_1\texttt{;}\, c_2\ \{R\}$ si ottiene istanziando la regola (SEQ) nel modo seguente:

$$\text{(SEQ)}\ \ \frac{\{wlp(c_1, wlp(c_2, R))\}\ C_1\ \{wlp(c_2, R)\}\quad \{wlp(c_2, R)\}\ C_2\ \{R\}}{\{wlp(c_1, wlp(c_2, R))\}\ C_1\texttt{;}\, C_2\ \{R\}}$$

e osservando che le premesse sono deducibili per ipotesi induttiva.

If Sia $P_{if} = wlp(\texttt{if (}b\texttt{) \{}c_1\texttt{\}\ else \{}c_2\texttt{\}}, R) = ([\![b]\!] \wedge wlp(C_1, R)) \vee (\neg [\![b]\!] \wedge wlp(C_1, R))$. Allora $\vdash \{P_{if}\}\ \texttt{if (}b\texttt{) \{}c_1\texttt{\}\ else \{}c_2\texttt{\}}\ \{R\}$ si ottiene con il seguente albero di prova:

$$\text{(IF)}\ \ \frac{\text{(CONS)}\ \ \dfrac{\{wlp(c_1, R)\}\ c_1\ \{R\}}{\{P_{if} \wedge [\![b]\!]\}\ c_1\ \{R\}}\qquad \text{(CONS)}\ \ \dfrac{\{wlp(c_2, R)\}\ c_2\ \{R\}}{\{P_{if} \wedge \neg [\![b]\!]\}\ c_2\ \{R\}}}{\{P_{if}\}\ \texttt{if (}b\texttt{) \{}c_1\texttt{\}\ else \{}c_2\texttt{\}}\ \{R\}}$$

osservando che le implicazioni necessarie per le due istanziazioni di (CONS) valgono e che le premesse sono deducibili per ipotesi induttiva.

While Sia $INV = wlp(\texttt{while (}b\texttt{) \{ }c\texttt{ \}}, R)$. Allora $\vdash \{INV\}\ \texttt{while (}b\texttt{) \{ }c\texttt{ \}}\ \{R\}$ si ottiene con il seguente albero di prova:

$$\text{(CONS)}\ \ \frac{\text{(WHILE)}\ \ \dfrac{\text{(CONS)}\ \ \dfrac{\{wlp(c, INV)\}\ c\ \{INV\}}{\{[\![B]\!] \wedge INV\}\ c\ \{INV\}}}{\{INV\}\ \texttt{while (}b\texttt{) \{ }c\texttt{ \}}\ \{\neg [\![B]\!] \wedge INV\}}}{\{INV\}\ \texttt{while (}b\texttt{) \{ }c\texttt{ \}}\ \{R\}}$$

osservando che la premessa $\{wlp(c, INV)\}\ c\ \{INV\}$ è deducibile per ipotesi induttiva e che le istanziazioni della regola (CONS) sono lecite perché valgono le due seguenti implicazioni:

1. $\neg [\![b]\!] \wedge INV \Rightarrow R$,
2. $[\![b]\!] \wedge INV \Rightarrow wlp(c, INV)$.

Prova di (1) Sia s tale che $[\![b]\!]\, s = F$ e $INV(s) = T$. Dobbiamo far vedere che $R(s) = tt$. Poiché $[\![b]\!]\, s = F$, $[\![\texttt{while (}b\texttt{) \{ }c\texttt{ \}}]\!]\, s = s$, e dalla definizione di INV si ha la tesi.

Prova di (2) Sia s tale che $[\![b]\!]\, s = T$ e $INV(s) = T$. Dobbiamo far vedere che $wlp(c, INV)(s) = tt$, cioè che se $[\![c]\!]\, s = s'$ allora $INV(s') = T$, cioè che se $[\![\texttt{while (}b\texttt{) \{ }c\texttt{ \}}]\!]\, s' = s''$ allora $R(s'') = T$. Sia effettivamente $[\![c]\!]\, s = s'$ e $[\![\texttt{while (}b\texttt{) \{ }c\texttt{ \}}]\!]\, s' = s''$; poiché $[\![b]\!]\, s = T$, $[\![\texttt{while (}b\texttt{) \{ }c\texttt{ \}}]\!]\, s = [\![\texttt{while (}b\texttt{) \{ }c\texttt{ \}}]\!]\, s' = s''$, e poiché sappiamo che $INV(s) = T$ si ha la tesi.

Regole derivate. Nel seguito useremo per comodità le regole derivate in Fig.14 ottenute combinando una regola relativa a un tipo di comando con la regola (CONS).

$$\text{(SKIP+CONS)} \qquad \frac{}{\{P\}\ \texttt{skip}\ \{R\}} \quad P \Rightarrow R$$

$$\text{(ASSIGN+CONS)} \qquad \frac{}{\{P\}\ id = e\ \{R\}} \quad P \Rightarrow R[[\![e]\!]\,/id]$$

$$\text{(WHILE+CONS)} \qquad \frac{\{INV \wedge [\![B]\!]\}\ C\ \{INV\}}{\{P\}\ \texttt{while (}B\texttt{) \{ }C\texttt{ \}}\ \{R\}} \quad \begin{array}{l} P \Rightarrow INV \\ INV \wedge \neg [\![b]\!] \Rightarrow R \end{array}$$

Figura 14: Regole derivate

Mostriamo ora un esempio di prova di validità di un'asserzione di correttezza parziale utilizzando il sistema di Hoare.

Esempio 8.5 Consideriamo l'asserzione

$$\{\mathtt{x} = x_0 \wedge \mathtt{y} = y_0\}\ c\ \{\mathtt{x} = 0 \wedge \mathtt{y} = x_0 + y_0\}$$

dove c indica il comando `while (x ≠ 0) { x=x-1; y=y+1 }`.
Un albero di prova per quest'asserzione è il seguente, dove come invariante del ciclo abbiamo scelto $INV = \mathtt{x} + \mathtt{y} = x_0 + y_0$.

$$\text{(WHILE+CONS)}\ \frac{\text{(SEQ)}\ \dfrac{\text{(ASSIGN+CONS)}\ \dfrac{}{\{INV \wedge \mathtt{x} \neq 0\}\ \mathtt{x=x-1}\ \{INV[\mathtt{y}+1/\mathtt{y}]\}}\quad \text{(ASSIGN)}\ \dfrac{}{\{INV[\mathtt{y}+1/\mathtt{y}]\}\ \mathtt{y=y+1}\ \{INV\}}}{\{INV \wedge \mathtt{x} \neq 0\}\ \mathtt{x=x-1; y=y+1}\ \{INV\}}}{\{\mathtt{x} = x_0 \wedge \mathtt{y} = y_0\}\ c\ \{\mathtt{x} = 0 \wedge \mathtt{y} = x_0 + y_0\}}$$

L'istanziazione della regola (WHILE+CONS) è corretta perché le side condition sono banalmente verificate:

- $\mathtt{x} = x_0 \wedge \mathtt{y} = y_0 \Rightarrow INV$,
- $(INV \wedge \mathtt{x} = 0) \Rightarrow (\mathtt{x} = 0 \wedge \mathtt{y} = x_0 + y_0)$.

Inoltre, l'istanziazione della regola (ASSIGN+CONS) è corretta perché $INV \wedge \mathtt{x} \neq 0$ implica $INV[\mathtt{y}+1/\mathtt{y}][\mathtt{x}-1/\mathtt{x}] = INV$.

Mostriamo un altro esempio che illustra come possiamo esprimere e provare la non terminazione di un comando.

Esempio 8.6 Consideriamo l'asserzione

$$\{\mathtt{x} < 0\}\ c\ \{\mathtt{false}\}$$

dove c indica, come prima, il comando `while (x ≠ 0) { x=x-1; y=y+1 }`.
Si noti che, in base alla definizione, richiedere che questa asserzione sia valida corrisponde a richiedere che il comando c non termini a partire da stati che verificano la precondizione.
Un albero di prova per quest'asserzione è il seguente, dove come invariante del ciclo abbiamo scelto $\mathtt{x} < 0$.

$$\text{(WHILE+CONS)}\ \frac{\text{(SEQ)}\ \dfrac{\text{(ASSIGN+CONS)}\ \dfrac{}{\{\mathtt{x} < 0\}\ \mathtt{x=x-1}\ \{\mathtt{x} < 0[\mathtt{y}+1/\mathtt{y}]\}}\quad \text{(ASSIGN)}\ \dfrac{}{\{\mathtt{x} < 0[\mathtt{y}+1/\mathtt{y}]\}\ \mathtt{y=y+1}\ \{\mathtt{x} < 0\}}}{\{\mathtt{x} < 0\}\ \mathtt{x=x-1;y=y+1}\ \{\mathtt{x} < 0\}}}{\{\mathtt{x} < 0\}\ c\ \{\mathtt{false}\}}$$

L'istanziazione della regola (WHILE+CONS) è corretta perché le side condition sono banalmente verificate:

- $\mathtt{x} < 0 \Rightarrow \mathtt{x} < 0$,
- $(\mathtt{x} < 0 \wedge \mathtt{x} = 0) \Rightarrow \mathtt{false}$.

Inoltre, l'istanziazione della regola (ASSIGN+CONS) è corretta perché $\mathtt{x} < 0$ implica $\mathtt{x} < 0[\mathtt{y}+1/\mathtt{y}][\mathtt{x}-1/\mathtt{x}] = \mathtt{x} - 1 < 0$.

8.4 Asserzioni alla Hoare di correttezza totale

Le asserzioni alla Hoare *di correttezza totale* sono triple della forma $\{P\}\ c\ \{\Downarrow R\}$, dove su P, R e c valgono le stesse assunzioni e la stessa terminologia che nel caso delle asserzioni di correttezza parziale.
Intuitivamente, il significato di tali asserzioni è il seguente:

> se P vale nello stato iniziale (cioè prima di eseguire c), allora l'esecuzione di c deve terminare producendo uno stato finale su cui vale R.

Si noti che, in questo caso, perché l'asserzione sia valida si richiede che l'esecuzione del comando termini.

Esempio 8.7 Un esempio di asserzione alla Hoare di correttezza totale è il seguente.

$$\{\mathtt{x} = x_0 \wedge \mathtt{y} = y_0 \wedge x_0 \geq 0\}\ \mathtt{while\ (x} \neq \mathtt{0)\ \{\ x=x-1;y=y+1\ \}}\ \{\Downarrow \mathtt{x} = 0 \wedge \mathtt{y} = x_0 + y_0\}$$

Questa asserzione è intuitivamente valida. Infatti, se $x_0 \geq 0$ è facile vedere che l'esecuzione del comando termina e alla fine $\mathtt{x}$ ha sicuramente assunto il valore 0 e $\mathtt{y}$ ha incrementato il suo valore iniziale di una quantità uguale al valore iniziale di $\mathtt{x}$.

Validità. Diremo che un'asserzione alla Hoare di correttezza totale $\{P\}\ c\ \{\Downarrow R\}$ è *valida*, e scriveremo $\models \{P\}\ c\ \{\Downarrow R\}$, se e solo se:

> per ogni $s \in \mathit{State}$, se $P(s) = T$, allora $R([\![c]\!]\, s) = T$.

Si noti che se $[\![c]\!]\, s$ è indefinito, cioè il comando c non termina a partire dallo stato iniziale s, l'implicazione risulta falsa. Ciò corrisponde al fatto che stiamo considerando la correttezza totale.

Weakest precondition. Analogamente al caso parziale, tra tutte le precondizioni P che rendono valida un'asserzione alla Hoare di correttezza totale $\{P\}\ c\ \{\Downarrow R\}$, è di particolare interesse *la più debole*, cioè quella implicata da tutte le altre. Questa precondizione può essere vista come un insieme minimo di requisiti da richiedere per ottenere che il comando termini in uno stato in cui vale R. Formalmente, definiremo, per ogni c comando e R postcondizione, la *weakest precondition* (abbreviato wp) di c rispetto a R, denotata $wp(c, R)$, nel modo seguente:

> $wp(c, R)(s) = T$ sse $R([\![c]\!]\, s)$.

Si noti che, se $[\![c]\!]\, s$ è indefinito, cioè il comando c non termina a partire dallo stato iniziale s, la wp risulta falsa.
La definizione può anche essere data in maniera equivalente in termini di insiemi di stati:

> $\{wp(c, R)\} = \{s \mid R([\![c]\!]\, s)\}$.

Ossia, la wp di c rispetto a R è data da tutti gli stati iniziali per cui eseguendo c si ottiene uno stato finale su cui vale R.
Si può facilmente provare, analogamente al caso parziale, che $R([\![c]\!]\, s)$ è effettivamente la precondizione più debole per c rispetto a R. Inoltre, si vede facilmente che la caratterizzazione esplicita della wlp data precedentemente per i comandi di MINI$\mathcal{W}$ coincide con quella della wp.

Sistema di prova. Il sistema di prova nel caso totale differisce da quello nel caso parziale solo per la regola relativa al ciclo while. In questo caso infatti occorre anche garantire la terminazione del comando a partire dalla precondizione data.
Sia `while (`b`) { `c` }` un comando while e INV un'asserzione. Una *funzione di terminazione* per `while (`b`) { `c` }` e INV è una funzione $t\colon State \to P$ con P insieme parzialmente ordinato (vedi Sez.A.2 nell'Appendice) che soddisfa le seguenti due condizioni:

1. $t(\{INV \wedge [\![b]\!]\}) \subseteq N$ con N sottoinsieme noetheriano di P;
2. per ogni s in $\{INV \wedge [\![b]\!]\}$, $t([\![c]\!]\, s) < t(s)$.

Il sistema di prova di Hoare per la correttezza totale è dato in Fig.15. Le regole sono analoghe a quelle date per la correttezza parziale, tranne la regola del while.

(SKIP) $$\frac{}{\{R\}\ \texttt{skip}\ \{\Downarrow R\}}$$

(ASSIGN) $$\frac{}{\{R[[\![e]\!]/id]\}\ id = e\ \{\Downarrow R\}}$$

(SEQ) $$\frac{\{P\}\ C_1\ \{\Downarrow Q\} \quad \{Q\}\ C_2\ \{\Downarrow R\}}{\{P\}\ C_1 ; C_2\ \{\Downarrow R\}}$$

(IF) $$\frac{\{P \wedge [\![B]\!]\}\ C_1\ \{\Downarrow R\} \quad \{P \wedge \neg [\![B]\!]\}\ C_2\ \{\Downarrow R\}}{\{P\}\ \texttt{if (}B\texttt{)}\{C_1\}\ \texttt{else}\ \{C_2\}\ \{\Downarrow R\}}$$

(WHILE) $$\frac{\{INV \wedge [\![B]\!]\}\ C\ \{\Downarrow INV\}}{\{INV\}\ \texttt{while (}B\texttt{)}\ \{\ C\ \}\ \{\Downarrow INV \wedge \neg [\![B]\!]\}}$$
$\exists t\colon State \to P$ funzione di terminazione per `while (`b`) { `c` }` e INV

(CONS) $$\frac{\{P'\}\ c\ \{\Downarrow R'\}}{\{P\}\ c\ \{\Downarrow R\}} \quad \begin{array}{l} P \Rightarrow P' \\ R' \Rightarrow R \end{array}$$

Figura 15: Sistema di prova di Hoare per la correttezza totale

Teorema 8.8 Il sistema di prova di Hoare per la correttezza totale dato in Fig.15 è sound e completo rispetto alla validità, cioè

$$\models \{P\}\ c\ \{\Downarrow R\} \text{ sse} \vdash \{P\}\ c\ \{\Downarrow R\}.$$

Prova di soundness. Analogamente a quanto visto per il caso parziale, dobbiamo provare che ogni (istanziazione di) metaregola è sound nel senso che se le premesse sono valide allora la conseguenza è valida. Diamo solo la prova per la regola (WHILE); per le altre metaregole la prova è analoga a quella data nel caso parziale.

While Il fatto che se vale INV prima di eseguire `while (`b`) { `c` }` e il comando termina, allora vale $INV \wedge \neg [\![b]\!]$ si può provare in modo analogo a quanto visto per il caso parziale. Dobbiamo quindi provare che se vale INV prima di eseguire `while (`b`) { `c` }`, allora il comando termina. Supponiamo per assurdo che il comando non termini. Consideriamo allora la seguente successione di elementi di P (l'insieme parzialmente ordinato codominio della funzione di terminazione t):

- $p_0 = t(s)$,
- $p_1 = t([\![c]\!]\, s)$,
- $p_2 = t([\![c]\!]([\![c]\!]\, s))$,
- $\ldots$,
- $p_i = t([\![c]\!]^i(s))$,
- $\ldots$

Si vede facilmente che, per ogni $i \geq 0$, $[\![c]\!]^i(s) \in \{INV \wedge [\![b]\!]\}$. Infatti, per ogni $i \geq 0$, $[\![c]\!]^i(s) \in \{[\![b]\!]\}$ perché in caso contrario l'esecuzione del comando while terminerebbe, contro l'ipotesi. A questo punto, si prova per induzione aritmetica che, per ogni $i \geq 0$, $[\![c]\!]^i(s) \in \{INV\}$, dato che vale $INV(s)$ per ipotesi ed è valida la premessa della regola (WHILE).

Allora, in base alla prima condizione che una funzione di terminazione deve soddisfare, si ha che, per ogni $i \geq 0$, $p_i \in N$ con N insieme noetheriano. Inoltre, in base alla seconda condizione che una funzione di terminazione deve soddisfare, si ha che, per ogni $i \geq 0$, $p_{i+1} < p_i$. Abbiamo quindi costruito una catena discendente infinita di elementi di N, il che è impossibile.

Prova di completezza. Analogamente a quanto visto per il caso parziale, è sufficiente provare che

(*) dato un comando c e una postcondizione R si ha $\vdash \{wp(c, R)\}\ c\ \{\Downarrow R\}$.

La prova di (*) è per induzione sulla struttura dei comandi di MINI$\mathcal{W}$. Diamo solo la prova per la regola (WHILE); per le altre metaregole la prova è analoga a quella data nel caso parziale.

While Sia $INV = wp(\texttt{while (}b\texttt{) \{ }c\texttt{ \}}, R)$. Allora $\vdash \{INV\}$ `while (`b`) { `c` }` $\{\Downarrow R\}$ si ottiene con un albero di prova analogo a quello visto per il caso parziale; dobbiamo però in questo caso dare anche una funzione di terminazione che ci consenta di istanziare la regola (WHILE). Poiché $\{INV\}$ `while (`b`) { `c` }` $\{\Downarrow R\}$ è per definizione valida, sappiamo che, per ogni stato s per cui vale $wp(c, R)(s)$, il comando termina dopo un certo numero di iterazioni, sia N_s, con $N_s \geq 0$. Definiamo allora t come segue:

$t\colon State \to \mathbb{N}$
$t(s) = N_s$, se $s \in \{INV \wedge [\![b]\!]\}$,
$t(s)$ = un valore arbitrario, per esempio 0, altrimenti.

Allora t verifica le due condizioni richieste per essere una funzione di terminazione. Infatti la prima vale banalmente poiché $\mathbb{N}$ è noetheriano. La seconda vale poiché, per ogni s in $\{INV \wedge [\![b]\!]\}$, essendo per definizione $[\![\texttt{while (}b\texttt{) \{ }c\texttt{ \}}]\!]\, s = [\![\texttt{while (}b\texttt{) \{ }c\texttt{ \}}]\!]([\![c]\!]\, s)$, deve essere $N_s = N_{[\![c]\!]\, s} + 1$.

Mostriamo ora un esempio di prova di validità di un'asserzione di correttezza totale utilizzando il sistema di Hoare.

Esempio 8.9 Consideriamo l'asserzione

$$\{\mathtt{x} = x_0 \wedge \mathtt{y} = y_0 \wedge x_0 \geq 0\}\ c\ \{\Downarrow \mathtt{x} = 0 \wedge \mathtt{y} = x_0 + y_0\}$$

dove c indica il comando `while (x` $\neq$ `0) { x=x-1; y=y+1 }`.
Un albero di prova per quest'asserzione è il seguente, dove come invariante del ciclo abbiamo scelto

$$INV = \mathtt{x} + \mathtt{y} = x_0 + y_0 \wedge \mathtt{x} \geq 0$$

e come funzione di terminazione $t\colon State \to \mathbb{Z}$ definita da $t(s) = s(\mathtt{x})$.

$$\text{(WHILE+CONS)}\ \frac{\text{(SEQ)}\ \dfrac{\text{(ASSIGN+CONS)}\ \overline{\{INV \wedge \mathtt{x} \neq 0\}\ \mathtt{x=x-1}\ \{\Downarrow INV[\mathtt{y}+1/\mathtt{y}]\}}\quad \text{(ASSIGN)}\ \overline{\{INV[\mathtt{y}+1/\mathtt{y}]\}\ \mathtt{y=y+1}\ \{\Downarrow INV\}}}{\{INV \wedge \mathtt{x} \neq 0\}\ \mathtt{x=x-1; y=y+1}\ \{\Downarrow INV\}}}{\{\mathtt{x} = x_0 \wedge \mathtt{y} = y_0 \wedge x_0 \geq 0\}\ c\ \{\Downarrow \mathtt{x} = 0 \wedge \mathtt{y} = x_0 + y_0\}}$$

L'istanziazione della regola (WHILE+CONS) è corretta perché le side condition sono banalmente verificate:

- $\mathtt{x} = x_0 \wedge \mathtt{y} = y_0 \wedge x_0 \geq 0 \Rightarrow INV$,
- $(INV \wedge \mathtt{x} = 0) \Rightarrow (\mathtt{x} = 0 \wedge \mathtt{y} = x_0 + y_0)$;
- t è una funzione di terminazione per c e INV perché è facile vedere che:
  - $t(\{INV \wedge \mathtt{x} \neq 0\}) \subseteq \mathbb{N}$,
  - per ogni s in $\{INV \wedge \mathtt{x} \neq 0\}$, $t([\![\mathtt{x=x-1; y=y+1}]\!]\, s) < t(s)$.

Inoltre, l'istanziazione della regola (ASSIGN+CONS) è corretta perché $INV \wedge \mathtt{x} \neq 0$ implica $INV[\mathtt{y}+1/\mathtt{y}][\mathtt{x}-1/\mathtt{x}] = \mathtt{x}+\mathtt{y} = x_0 + y_0 \wedge \mathtt{x} - 1 \geq 0$.

8.5 Esercizi

Notazioni Utilizziamo `if (`b`)`$\{c\}$ come abbreviazione per `if (`b`)`$\{c\}$ `else` $\{$`skip`$\}$. $Pari($`x`$)$, con `x` identificatore, indica il predicato sugli stati vero sse il valore di `x` è pari. Assumiamo di avere in MINI$\mathcal{W}$ il tipo degli array di interi (indiciati da 1 a una costante positiva data, con le operazioni e notazioni usuali), e il tipo delle stringhe (su un certo alfabeto) con le operazioni: `Empty`, `Hd`, `Tl` e c con l'usuale significato. Si noti che in presenza di operazioni parziali (come `Hd` e `Tl`) a un'espressione booleana b di MINI$\mathcal{W}$ corrisponde un predicato sullo stato $[\![b]\!]$ che vale vero se se l'espressione è vera, falso se l'espressione è falsa *oppure non definita*.

1. Sia $c =$ `while (x` $\neq$ `0) { x=x-1; y=y+1 }`, x_0, y_0 due costanti intere. Si provi che:
   (a) $\models \{\mathtt{x} = x_0 \wedge \mathtt{y} = y_0\}\ c\ \{\mathtt{x} = 0 \wedge \mathtt{y} = y_0 + x_0\}$ (soluzione data precedentemente),
   (b) $\models \{\mathtt{x} = 0 \wedge \mathtt{y} = y_0\}\ c\ \{\Downarrow \mathtt{y} = y_0\}$,
   (c) $\models \{\mathtt{x} < 0\}\ c\ \{\mathtt{false}\}$ (soluzione data precedentemente),
   (d) $\models \{\mathtt{x} = x_0 \wedge \mathtt{y} = y_0 \wedge x_0 \geq 0\}\ c\ \{\Downarrow \mathtt{x} = 0 \wedge \mathtt{y} = y_0 + x_0\}$ (soluzione data precedentemente).
2. Si provi, direttamente dalla definizione, utilizzando la wp e utilizzando il sistema di Hoare che è valida $\{\mathtt{x} > 0 \vee \mathtt{y} > 0\}$ `if (x>0)`$\{$`z=x`$\}$ `else` $\{$`z=y`$\}$ $\{\Downarrow \mathtt{z} > 0\}$.
3. Si trovi c che renda valide entrambe le asserzioni

   $\{\mathtt{x} < 7\}\ c\ \{\Downarrow \mathtt{y} = 0\}$,
   $\{\mathtt{x} \geq 7\}\ c\ \{\mathtt{false}\}$.
4. Si trovi P che renda valida $\{\mathtt{x} = x_0 \wedge \mathtt{y} = y_0 \wedge P\}$ `while (y` $\neq$ `x) { x=x+1; y=y-1 }` $\{\Downarrow \mathtt{x} = \mathtt{y}\}$.
5. Si trovi un comando c che renda valide entrambe le asserzioni

   $\{Pari(\mathtt{x}) \wedge \mathtt{x} \geq 0\}\ c\ \{\Downarrow \mathtt{y} = 0\}$,
   $\{\neg Pari(\mathtt{x}) \vee \mathtt{x} < 0\}\ c\ \{\mathtt{false}\}$.
6. Si trovi la condizione P necessaria e sufficiente perché valga

   $\{P \wedge \mathtt{x} = x_0 \wedge \mathtt{y} = y_0\}$ `s=0;while (x!=y)`$\{$`x=x-1; s=s+1`$\}$ $\{\Downarrow \mathtt{s} = x_0 - y_0\}$.
7. Scrivere un algoritmo iterativo che calcoli l'n-simo termine della successione di Fibonacci (n costante positiva) e provarne formalmente la correttezza (la successione di Fibonacci è definita induttivamente da $f_0 = 1$, $f_1 = 1$, $f_{i+2} = f_i + f_{i+1}$ per $i \geq 0$).
8. Si completi il seguente algoritmo iterativo scegliendo b in modo che esso risulti corretto rispetto alla specifica indicata (`A` è un array di dimensione n, con n costante positiva). La postcondizione può essere espressa informalmente dicendo che tutti gli elementi dispari devono precedere tutti quelli pari.

   $\{\mathtt{first} = 1 \wedge \mathtt{last} = n\}$

   ```
   while (B) {
     z=A[last];
     if (Pari(z)){last=last-1} else {A[last]=A[first];A[first]=z;first=first+1}
   }
   ```

   $\{\Downarrow \forall i, j \in [1..n].Pari(\mathtt{A[i]}) \wedge \neg Pari(\mathtt{A[j]}) \Rightarrow j < i\}$
9. Si verifichi la validità della seguente asserzione:

   $\{\mathtt{x} = \mathtt{y}\}$ `if (x>=0)`$\{$`if (y>=0)`$\{$`x=x+2`$\}\}$ `else`$\{$`if (y<0)`$\{$`y=y-2`$\}\}$ $\{\Downarrow \mathtt{x} = \mathtt{y} + 2\}$.

   Si valuti inoltre, detto c il comando sopra, $wp(c, \mathtt{x} = \mathtt{y} + 2)$.
10. Si consideri il seguente algoritmo iterativo che ordina in modo crescente gli elementi di un array `A` di dimensione n (n costante positiva fissata).

    $\{\mathtt{i} = 1\}$

```
while (i<n) {
  if (A[i]>A[i+1]) {swap(A,i);i=1}
  else {i=i+1}
}
```

$\{\Downarrow \forall i \in [1..n-1].\mathtt{A}[i] \leq \mathtt{A}[i+1]\}$

Si è assunto che `swap(A,i)` sia un comando che scambia tra loro l'elemento di posto `i` e il successivo. Si provi la correttezza totale dell'algoritmo rispetto alla specifica indicata, formalizzando le asserzioni su `swap(A,i)` necessarie nel corso della prova.

11. Il seguente algoritmo fornisce, data una stringa $\bar{w}$, la stringa rovesciata $\bar{w}^R$.

$\{\mathtt{rev} = \mathtt{Empty} \wedge \mathtt{w} = \bar{w}\}$

```
while (w!=Empty) {rev=Cons(Hd(w),rev);w=Tl(w)}
```

$\{\Downarrow \mathtt{rev} = \bar{w}^R\}$

Si provi formalmente la correttezza totale dell'algoritmo rispetto alla precondizione e postcondizione indicate.

12. Il seguente algoritmo controlla se una stringa $\bar{w}$ è *palindroma* (cioè $\bar{w} = \bar{w}^R$).

$\{\mathtt{pal} \wedge \mathtt{w} = \bar{w}\}$

```
while (w!=Empty && pal) {pal=Hd(w)==Last(w);w=Tl(w);w=CancelLast(w)}
```

$\{\Downarrow \mathtt{pal} = (\bar{w} = \bar{w}^R)\}$

Si è assunto di avere a disposizione due funzioni ausiliarie (parziali) `Last` e `CancelLast` che data una stringa restituiscono rispettivamente l'ultimo elemento e la stringa privata dell'ultimo elemento.

L'algoritmo dato *non* è corretto rispetto alla specifica indicata. Si motivi tale fatto dando un controesempio. Si modifichi l'algoritmo in modo da renderlo corretto e si provi la correttezza della versione modificata.

13. Si consideri il seguente algoritmo iterativo (x_0, y_0 costanti intere).

$\{\mathtt{diff} = 0 \wedge \mathtt{x} = x_0 \wedge \mathtt{y} = y_0\}$

```
while (x!=y) {
  if (x>y){x=x-1} else {x=x+1};
  diff=diff+1
}
```

$\{\Downarrow \mathtt{diff} = \mid x_0 - y_0 \mid\}$

Se ne provi formalmente la correttezza totale rispetto alla specifica indicata utilizzando il sistema di Hoare.

14. Si provi formalmente che vale (con `x`, `y` variabili intere e k costante intera positiva)

$\{\mid \mathtt{x} - \mathtt{y} \mid = k\}$

```
if (x>y){x=x-1};
if (y>x){y=y-1}
```

$\{\Downarrow \mid \mathtt{x} - \mathtt{y} \mid = k - 1\}$

15. Si consideri il seguente comando c, dove `x`, `y` sono stringhe.

$c =$ `while (x!=Empty) { Push(Hd(x),y);Pop(x) }`

Si provi che, assumendo che i comandi `Push(Hd(x),y)` e `Pop(x)` siano corretti rispetto a opportune asserzioni (specificare quali), vale la seguente asserzione:

$\{\mathtt{x} = \bar{x} \wedge \mathtt{y} = \mathtt{Empty}\}\ c\ \{\Downarrow \mathtt{y} = \bar{x}^R\}$

dove $\bar{x}$ è una costante di tipo stringa.

16. Si trovi $wp(\mathtt{if\ (x>=0)\{if\ (y<0)\{z=x\}\ else\ \{if\ (y>=0)\{z = x\}\}}, \mathtt{z = y})$.

Riferimenti bibliografici

[1] B. C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

A Appendice

A.1 Relazioni e funzioni parziali

Def. A.1 [Relazioni e funzioni parziali] Dati due insiemi A e B, una *relazione* R da A in B è un sottoinsieme di $A \times B$; una *relazione su* A è una relazione da A in A. Se $(a, b) \in R$ si scrive anche $a\ R\ b$. Una *funzione (parziale)* da A in B è una relazione f da A in B che gode della proprietà di univocità, cioè per ogni $a \in A$ esiste al più un $b \in B$ tale che $(a, b) \in R$; tale b, se esiste, è denotato con $f(a)$. Se f è una funzione da A in B si scrive $f\colon A \to B$; $A \to B$ denota l'insieme delle funzioni da A in B. Una funzione f da A in B si dice *totale* se $f(a)$ esiste per ogni $a \in A$.

Quando non specificato esplicitamente, per "funzione" intenderemo sempre funzione parziale (quindi non necessariamente totale). Si noti che quando si considera un uguaglianza $e_1 = e_2$ tra espressioni in cui compaiono funzioni parziali, non è scontato quale sia il significato dell'uguaglianza nel caso in cui e_1, e_2 o entrambe sono non definite. Noi assumeremo di interpretare tale uguaglianza nel senso cosiddetto *forte*, cioè $e_1 = e_2$ vale se e_1 ed e_2 sono entrambe definite ed uguali, oppure sono entrambe indefinite.

Def. A.2 [Sostituzione] Data una funzione $f\colon A \to B$, due elementi $a \in A$, $b \in B$, $f[b/a]$ indica la funzione da A in B così definita:

$f[b/a](a) = b$, $f[b/a](a') = f(a')$ per $a' \neq a$.

Abbrevieremo $f[a_1/b_1] \ldots [a_n/b_n]$ con $f[a_1/b_1 \ldots a_n/b_n]$ e $\emptyset[a_1/b_1 \ldots a_n/b_n]$ con $[a_1/b_1 \ldots a_n/b_n]$.

Def. A.3 [Chiusura riflessiva e transitiva] Se R è una relazione su A, la *chiusura transitiva* di R è la relazione R^+ su A definita come segue:

per ogni $a, a' \in A$, aR^+a' sse $a = a_0Ra_1, a_1Ra_2, \ldots, a_{n-1}Ra_n = a'$, per qualche $a_1, \ldots, a_{n-1}$.

La *chiusura riflessiva e transitiva* di R è la relazione $R^\star$ su A definita da: $R^\star = R^+ \cup \{(a, a) \mid a \in A\}$.
Equivalentemente, $R^\star$ può essere definita induttivamente come segue:

- aRa per ogni $a \in A$,
- se aRa', allora aR^+a',
- se aR^+a' e $a'R^+a''$, allora aR^+a'';

Def. A.4 [Famiglia di insiemi] Una *famiglia di insiemi* (o semplicemente *famiglia*) indiciata su S (o S-famiglia) è una funzione totale che associa ad ogni $s \in S$ un insieme. Se A è una S-famiglia di insiemi, per ogni $s \in S$ l'insieme associato ad s si indica A_s, ed A si scrive anche $\{A_s\}_{s \in S}$.

Si noti la differenza tra una famiglia ed un insieme; ad esempio la famiglia A indiciata su a, b, c definita da $A_a = \mathbb{Z}$, $A_b = \mathbb{B}$, $A_c = \mathbb{Z}$ è diversa dall'insieme $\{A_a, A_b, A_c\} = \{\mathbb{Z}, \mathbb{B}\}$.
Le usuali operazioni sugli insiemi (ad esempio unione, intersezione, differenza) si estendono alle famiglie, componente per componente (è necessario che le famiglie coinvolte siano indiciate tutte sullo *stesso* insiemi di indici).
In modo esattamente analogo ad una famiglia di insiemi si definisce una famiglia indiciata su S (o S-famiglia) di funzioni $f = \{f_s\}_{s \in S}$.

Def. A.5 [Segnatura] Una *segnatura (eterogenea)* è una coppia (S, O) dove S è un insieme di simboli detti *sort* (o anche *tipi*, *indici*) O è una $(S^\star \times S)$-famiglia di insiemi i cui elementi sono detti *simboli di operazione*. Per ogni $w \in S^\star, s \in S$, se $op \in O_{(w,s)}$ si scrive $op\colon w \to s$ e si dice che op ha *arità* w e *tipo* s; la coppia (w, s) si chiama *funzionalità* di op. Per i simboli di operazioni ad arità Λ (*simboli di costante*) scriveremo anche $op\colon \to s$ invece di $op\colon \Lambda \to s$.

Def. A.6 [Algebra] Sia $\Sigma = (S, O)$ una segnatura. Un'*algebra A su Σ* consiste di

- per ogni $s \in S$, un insieme A_s detto *carrier* (*supporto*) di tipo s;

- per ogni simbolo di operazione $op\colon w \to s$ in O, $w = s_1 \dots s_n$, una funzione $op^A\colon A_{s_1} \times \dots \times A_{s_n} \to A_s$ detta l'*interpretazione* di op in A.

Def. A.7 [Omomorfismo] Sia $\Sigma = (S, O)$ una segnatura, A, B due algebre su Σ. Un *omomorfismo* da A in B è una famiglia di funzioni $\{f_s\colon A_s \to B_s\}_{s \in S}$ tale che:

> per ogni simbolo di operazione $op\colon s_1 \dots s_n \to s$ in O,
> per ogni $a_1 \in A_{s_1}, \dots, a_n \in A_{s_n}$,
> $f_s(op(a_1, \dots, a_n)) = op(f_{s_1}(a_1), \dots, f_{s_n}(a_n))$

Si noti che in caso di indefinitezza l'uguaglianza sopra va interpretata in senso *forte* (vedi sopra).

A.2 Insiemi parzialmente ordinati ben fondati

Un insieme *parzialmente ordinato* (o *poset*, per *partially ordered set*) è una coppia $(P, \leq)$ dove P è un insieme e $\leq$ è un *ordine parziale* su P, cioè una relazione binaria su P tale che, per ogni $x, y, z \in P$:

- $x \leq x$ ($\leq$ è *riflessiva*),
- se $x \leq y$ e $y \leq x$ allora $x = y$ ($\leq$ è *antisimmetrica*),
- se $x \leq y$ e $y \leq z$ allora $x \leq z$ ($\leq$ è *transitiva*).

Dato $\leq$, possiamo definire la relazione binaria $<$ su P di *ordine parziale stretto* come segue:

> $x < y$ sse $x \leq y$ e $x \neq y$.

Una relazione di ordine parziale stretto su P è una relazione binaria su P tale che, per ogni $x, y, z \in P$:

- $x \not< x$,
- se $x < y$, allora $y \not< x$,
- se $x < y$ e $y < z$ allora $x < z$.

È facile vedere che è anche sempre possibile definire un ordine parziale a partire da un ordine parziale stretto; quindi è indifferente dare l'uno o l'altro.
Una *catena (strettamente) discendente infinita* in un insieme parzialmente ordinato $(P, <)$ è un sottoinsieme C di P tale che, per ogni $x \in C$, esiste $y \in C$ tale che $y < x$.
Un insieme P parzialmente ordinato è *noetheriano* (*ben fondato*) se non contiene catene discendenti infinite.