Linguaggi di Programmazione

(dispense)

Pietro Cenciarelli
Università di Roma, "La Sapienza"
cenciarelli@di.uniroma1.it

26 settembre 2019

Indice

1 Introduzione **4**

I Nozioni Preliminari **10**

2 Linguaggi **10**
2.1 Automi 10
2.2 Grammatiche 10
2.3 Semantica 10

3 Struttura e rappresentazione **11**
3.1 Induzione 11
3.2 Co-induzione 21
3.3 I tipi di dato nei linguaggi di programmazione 24

II Paradigmi di Programmazione **26**

4 Il Paradigma Funzionale **28**
4.1 *Exp*: un semplice linguaggio funzionale 28
4.1.1 Sintassi e semantica 28
4.1.2 Equivalenza operazionale 31
4.1.3 Valutazione *lazy* 31
4.1.4 Valutazione lazy con scoping statico 32
4.1.5 Semantiche equivalenti 33
4.2 *Fun*: un linguaggio con funzioni 35
4.2.1 Sintassi e semantica 35
4.2.2 Il lambda calcolo 37
4.2.3 I numeri di Church 38

4.3 Semantica comparata 40
4.3.1 Statico VS. dinamico 40
4.3.2 Eager VS. lazy 43

5 Il Paradigma Imperativo **44**
5.1 *Imp*: un semplice linguaggio imperativo 44
5.1.1 Sintassi e semantica 44
5.2 *All*: un linguaggio con procedure 46
5.2.1 Sintassi 46
5.2.2 Semantica call-by-value 46
5.2.3 Semantica call-by-reference 48
5.2.4 Semantica call-by-name 48

6 Il Paradigma ad Oggetti **49**
6.1 *Mini-Java* 49
6.1.1 Sintassi 50
6.1.2 Semantica statica 51
6.1.3 Semantica dinamica 51

III Sistemi dei Tipi **57**

7 I Tipi nei Linguaggi Funzionali **61**
7.1 Il sistema $F1$ 61
7.2 Il Polimorfismo 63
7.3 Il sistema $F2$ 64

8 Polimorfismo in ML **66**
8.1 Il sistema dei tipi di Fun_τ 67
8.2 Inferenza dei tipi in ML 69

9 L'isomorfismo di Curry-Howard **71**
9.1 Tipi dipendenti 71
9.2 Tipi e proposizioni 72
9.3 Verifica di un programma funzionale 72

IV La Correttezza di Programmi **74**

10 La Correttezza nei Linguaggi Imperativi **76**
10.1 Il metodo delle invarianti 76
10.2 La Logica di Hoare 79
10.3 Correttezza della divisione intera 82

11 La Correttezza nei Linguaggi Funzionali **84**
11.1 Verifica di specifiche equazionali 84
11.2 Correttezza della moltiplicazione per due 89

12 Conclusioni **91**

A Soluzione degli esercizi **95**

1 Introduzione

Su un argomento il dibattito politico era il più acceso nel paese di Lilliput: se le uova alla coque andassero mangiate aprendone il guscio dal lato piccolo o dal grande. I sostenitori del lato piccolo si chiamavano "little-endians" (*piccolapuntisti*) , i loro avversari "big-endians" (*grossapuntisti*). La diatriba si è riaperta nell'era del computer, dove little-endians sono i sostenitori delle architetture n cui la rappresentazione interna di un numero incomincia dalla cifra meno significativa (Intel e Digital) e big-endians i fautori della più significativa (Motorola, IBM e SUN). A tutt'oggi non cè un vincitore: nel linguaggio C, il programma

```
int main(void)
{
  int i = 1;
  char *c = &i;
  printf("%d", *c);
}
```

stampa 1 su un'architettura little-endian, 0 su una big-endian. Dunque, alla domanda

> "qual è il significato di un programma?"

non è ragionevole rispondere "è l'effetto che esso produce quando eseguito su un computer", perché... su quale computer?

La *Semantica* è la disciplina che studia il *significato* di un linguaggio. Nel caso specifico dei linguaggi di programmazione la semantica può essere definita *formalmente*. Una tale definizione costituisce una *specifica* non ambigua del linguaggio, e dunque uno strumento per il progettista del linguaggio ed un riferimento per l'implementatore e per il programmatore. Una semantica formale consente inoltre l'applicazione di strumenti matematici nello studio del linguaggio. In particolare essa consente:

- la *sintesi* di programmi da specifiche formali (per esempio espresse in UML). Questo viene fatto per garantire lo sviluppo di software *corretto* o per automatizzare la programmazione. In particolare è spesso è possibile generare automaticamente un compilatore o un interprete a partire da una specifica formale del linguaggio;
- l'*analisi* di programmi, ovvero la verifica formale di proprietà; ad esempio per applicazioni critiche.
- lo studio della *correttezza* di compilatori, sia in fase di traduzione che di ottimizzazione; in generale, questo richiede una nozione di *equivalenza* fra programmi.

Sono possibili divesi approcci alla definizione di equivalenza. Si può, ad esempio, decidere di considerare equivalenti due programmi quando producono

lo stesso risultato, oppure quando è possibile sostituire l'uno con l'altro in ogni contesto di programma senza così produrre effetti "visibili". In genere queste due definizioni non descrivono la stessa relazione.

Esercizio* **1.1** Sono equivalenti le seguenti versoni (una ricorsiva e l'altra iterativa) della funzione fattoriale?

```
int fact (int n) {
   if (n==0) return 1;
   else return n * fact(n-1);
}

int fact_iter (int n) {
   if (n==0) return 1;
   int i, result = 1;
   for (i=1; i<=n; i++)
      result = result * i;
   return result;
}
```

□

Semantica statica

Gli oggetti matematici hanno un *tipo*. Questa nozione si trova alle fondamenta della matematica stessa: distinguendo tra insiemi "grandi" e "piccoli", ad esempio, si evita di incappare in paradossi quale "l'insieme di tutti gli insiemi che non contengono se stessi come elemento", una versione insiemistica del barbiere che rade tutti e soli coloro che non si radono da soli (*chi raderà il barbiere?!*).

Nel trattare i programmi come oggetti matematici, una semantica formale deve dunque assegnare un tipo ai termini del linguaggio. Per *semantica statica* di un linguaggio di programmazione si intende il suo sistema dei tipi, ovvero l'insieme delle regole che consentono di dare un tipo ad espressioni, comandi ed altri costrutti di programmazione.

La determinazione dei tipi non è un puro esercizio di virtuosismo matematico. Lo scopo principale di un sistema dei tipi è infatti quello di prevenire l'occorrenza di errori durante l'esecuzione di un programma. Esempi di tali errori vanno dall'applicazione di funzioni ad argomenti inappropriati, al riferimento illegale della memoria, al tentativo di dividere un numero per zero. Nei diversi linguaggi di programmazione sono stati adottati approcci diversi a questo problema: vi sono linguaggi *esplicitamente* tipati come il Pascal o L'Algol, *implicitamente* tipati, come ML o Haskell, o *non* tipati affatto, come il LISP o l'Assembler. Per garantire la sicurezza, ovvero il non insorgere di errori "inaspettati", linguaggi non tipati ad alto livello come il LISP prevedono un meccanismo di controllo dei tipi (type checking) dinamico (ovvero a tempo di esecuzione).

Semantica dinamica

Assegna un significato ai programmi di un linguaggio in base al loro comportamento durante l'*esecuzione*. La descrizione del significato può essere *informale* (e non necessariamente per questo imprecisa), come avviene nei tradizionali manuali di programmazione, o *formale*, sfruttando cioè nozioni e tecniche della matematica o della logica. Abbiamo appena discusso le motivazioni dell'approccio formale. Vi sono anche esempi di approcci intermedi.

Si distingue tradizionalmente tra lo stile *operazionale* alla semantica dinamica, quello *denotazionale* e quello *assiomatico*.

Semantica operazionale. Il significato di un programma viene descritto da una *relazione* matematica tra i termini del linguagio che rappresenta una nozione di *valutazione*. Oltre ai termini della sintassi, generalmente astratta, tale relazione coinvolge spesso anche entità semantiche, ovvero rappresentazioni matematiche di un agente della computazione o del suo stato (ambiente, memoria...).

Una forma "primitiva" di semantica operazionale consiste nella formalizzazione di un *interprete* ovvero di un computer astratto (per esempio una macchina di Turing) su cui eseguire i programmi: il significato di un programma p è la *computazione* dell'interprete su p. La semantica basata sul processore astratto *Simplesem* usato in [GJ89] fornisce un esempio di questo tipo di semantica. Il punto debole di questo approccio è che il significato di un programma dipende da qualcosa di "esterno" al linguaggio, come ad esempio il funzionamento specifico di Simplesem, ovvero dal modello di computazione scelto.

L'approccio *strutturale* alla semantica operazionale consiste nel definire la relazione di valutazione solo in base alla struttura sintattica dei programmi. Ad esempio, sia $Env \times Term$ l'insieme delle "configurazioni", dove $Env = Var \stackrel{fin}{\to} Const$ è l'insieme degli *ambienti* che associano variabili a costanti, e $Term$ è l'insieme dei termini del linguaggio; una relazione $\to$ tra configurazioni può essere definita da regole del tipo: se $E_1, e_1 \to E_2, e_1'$ allora $E_1, e_1 + e_2 \to E_2, e_1' + e_2$. In tal modo, il significato della espressione `(4+x)-3` nell'ambiente $\{(\mathtt{x}, 2)\}$ è la computazione:

$$\{(\mathtt{x},2)\}, \mathtt{(4+x)-3} \to \{(\mathtt{x},2)\}, \mathtt{(4+2)-3} \to \{(\mathtt{x},2)\}, \mathtt{6-3} \to \{(\mathtt{x},2)\}, \mathtt{3}$$

Una relazione di valutazione definita come sopra è detta "a piccoli passi", perché modella passi elementari della computazione. Una forma alternativa di semantica operazionale strutturale è quella dei "grandi passi", dove la relazione di valutazione associa termini a *risultati*. A questa seconda specie appartiene la semantica presentata in [edi(6-8)], dove ne vengono discussi anche i vantaggi rispetto agli altri approcci.

Semantica denotazionale. Il significato di un programma non è determinato da una relazione sulla sintassi che lega termini a termini, come nell'approccio

operazionale, ma mappando i termini del linguaggio su oggetti matematici: numeri, elementi di strutture algebriche e così via. Per esempio, se $\rho = (\mathtt{x}, 2)$ è l'ambiente *semantico* che associa alla variabile `x` il *valore* 2 (nota la differenza con la costante `2` della semantica operazionale), allora:

$$[\![\mathtt{(4+x)-3}]\!]_\rho = [\![\mathtt{4+x}]\!]_\rho - [\![\mathtt{3}]\!]_\rho = 3$$

dove con $[\![t]\!]_\rho$ si intende il significato di t nell'ambiente ρ. Questo esempio mostra la *composizionalità* della semantica denotazionale: il significato di un'espressione complessa come `(4+x)-3` è una funzione del significato delle sue componenti più semplici: `(4+x)` e `3`.

In generale la semantica denotazionale è più *astratta* di quella operazionale. Nel caso precedente, ad esempio, la seconda ci dà una informazione sull'ordine di valutazione degli operandi che non risulta dalla prima.

Semantica assiomatica. Il significato di un programma p è determinato nell'ambito di una teoria assiomatica dall'insieme delle proposizioni vere per p. In generale, questa semantica consente un approccio "meccanico" alla dimostrazione formale di proprietà di programmi. Ad esempio, data una opportuna assiomatizzazione degli operatori `if`, `for`, `:=`, e così via, è possibile derivare formalmente la seguente proposizione:

$$\{\mathtt{n} \geq 0\}\ \mathtt{fact(n);}\ \{\mathtt{res} = \mathtt{n}!\}$$

dove $\{A\}\, p\, \{B\}$ viene letta: "se il programma p, eseguito in uno stato che soddisfa la proprietà A, termina, allora lo stato da esso prodotto soddisfa la proprietà B".

Le caratteristiche di un linguaggio

Nei manuali di programmazione il significato dei costrutti di un linguaggio viene di solito descritto facendo uso di nozioni *semantiche* quali quella di *variabile*, *istruzione*, *sottoprogramma* e così via. Di ciascuno di questi oggetti è possibile specificare degli *attributi*. Ad esempio:

- di una variabile:
  - il nome (ma esistono anche variabili *anonime*, cioè senza nome);
  - l'area di memoria in cui è mantenuto il suo valore;
- di un sottoprogramma:
  - il nome;
  - i parametri formali con il loro tipo;
  - la modalità di passaggio dei parametri;
- di un'istruzione:

– le azioni ad essa associate.

Valori di tipo appropriato possono essere associati agli attributi delle entità di un programma tanto durante la compilazione quanto durante l'esecuzione. L'associazione di un valore ad uno o più attributi di una entità si chiama *binding* (legame).

Un fattore importante nell'analisi di un linguaggio di programmazione è il *tempo di binding*. Attributi diversi di una stessa entità possono essere legati in tempi diversi, ed un legame può essere modificato durante l'esecuzione. Ad esempio: una variabile può essere legata al suo tipo a tempo di compilazione ed al suo valore a tempo di esecuzione. Un legame si dice *statico* se è stabilito prima dell'esecuzione e non può essere cambiato in seguito, *dinamico* altrimenti.

Ecco alcuni esempi di binding statico e dinamico

- in Fortran ed in Ada il tipo INTEGER è legato al momento della *definizione* del linguaggio, mentre in Pascal il tipo *integer* può essere ridefinito dal programmatore, e quindi legato alla sua rappresentazione a tempo di *compilazione*.

- In Pascal una variabile viene legata al suo tipo a tempo di compilazione. In APL questo avviene a tempo di esecuzione: l'istruzione

  $$A \leftarrow B + C$$

  è corretta solo se, al momento dell'*esecuzione*, B e C contengono valori numerici, o sono matrici con lo stesso numero di dimensioni e grandezza. In generale, linguaggi in cui il legame di tipo è statico (Fortran, Pascal, Modula 2, Ada ecc.) sono anche detti *fortemente tipati*; quelli con legame dinamico (LISP, Snobol, APL, ecc.) sono anche detti *non tipati*.

- L'allocazione di una zona di meoria per una variabile può avvenire staticamente (eseguita dal compilatore) o dinamicamente (eseguita a run time, sia automaticamente che dal programma, come con `new` in Java).

- Il collegamento di una variabile con un valore è tipicamente dinamico. Una variabile con legame statico al valore è una *costante simbolica* (per distinguerla dalle costanti "letterali", *literal*). In Pascal:

  ```
  const pi = 3.14;
  circonferenza = 2 * pi * raggio;
  ```

  In Pascal una costante viene tradotta dal compilatore nel suo valore. Le costanti dell'Algol 69 sono più dinamiche: il loro valore viene determinato a tempo di esecuzione, valutando una espressione che può contenere variabili. In modo simile le variabili di tipo `final` di Java.

Nei linguaggi tipati dinamicamente il controllo sui tipi può avvenire tanto per evitare errori di inconsistenza, quanto per determinare le azioni stesse da eseguire: nell'esempio dell'APL sopra riportato, la somma di matrici è cosa diversa dalla somma di valori semplici. È dunque necessario che i *descrittori* delle diverse entità di programma, ovvero l'informazione sui valori dei loro attributi (incluso il tipo), esistano a tempo di esecuzione, e possano essere dinamicamnete modificati alla creazione di un nuovo legame. Altri linguaggi, come ad esempio il Pascal, sono progettati in modo da consentire la scomparsa (almeno parziale) dei descrittori a tempo di esecuzione.

Da questa descrizione emerge un principio generale: linguaggi con legami dinamici sono orientati all'*interpretazione*, perché non esiste prima dell'esecuzione informazione sufficiente per generare il codice da eseguire. Al contrario, linguaggi con legami statici sono orientati alla *traduzione*.

Parte I
Nozioni Preliminari

2 Linguaggi

2.1 Automi

2.2 Grammatiche

2.3 Semantica

3 Struttura e rappresentazione

How much is two?

Paul Halmos - Naive Set Theory

3.1 Induzione

Per *sintassi* di un linguaggo di programmazione si intende generalmente un insieme di regole formali che definiscono le *stringhe* (ovvero le sequenze finite di simboli) del linguaggio. La sintassi è spesso definita usando la notazione BNF, ovvero mediante una grammatica *libera dal contesto*: il linguaggio è l'insieme delle stringhe di simboli terminali generabili mediante una *derivazione*, ovvero applicando le produzioni della grammatica. Va notato che le grammatiche libere dal contesto non sono generalmente adeguate ad esprimere vincoli sintattici quali la dichiarazione delle variabili prima dell'uso o la correttezza di una espressione dal punto di vista dei tipi.

Una derivazione getta luce sulla *struttura* di una stringa, ovvero determina un *albero sintattico*. L'informazione sulla struttura, come ad esempio la precedenza da assegnare agli operatori nell'interpretazione di una espressione aritmetica, è necessaria per associare un significato agli elementi di un linguaggio (*semantica*). Una grammatica si dice *ambigua* se esiste più di un albero sintattico per una stessa stringa. (Per una introduzione alle nozioni di *grammatica*, *derivazione*, *albero sintattico*... vd. [HMU03].)

Affinché un compilatore o un interprete possano *riconoscere* le stringhe di un linguaggio, è necessario che questo sia definito da una grammatica non ambigua. Esistono tecniche per disambiguare una grammatica o per trasformarne le produzioni in modo da rendere più efficiente il riconoscimento, al costo però di complicare, a volte in modo drammatico, gli alberi sintattici con struttura di scarsa rilevanza semantica. Per ovviare a questo inconveniente si ricorre, nello studio della semantica, all'uso della *sintassi astratta*.

Come vedremo, la sintassi astratta non è altro che una *definizione induttiva*. Il lettore ha probabilmente familiarità con l'*induzione matematica*, una tecnica per la dimostrazione di proprietà sui numeri naturali, e con definizioni induttive, quale ad esempio quella della funzione fattoriale. In questa sezione mostreremo che tecnica per definire funzioni o dimostrare proprietà sui numeri naturali, può essere generalizzata ed applicata ad altri oggetti quali ad esempio gli alberi o, appunto, i termini di un linguaggio di programmazione. Il risultato della nostra investigazione sarà l' *induzione strutturale*, un principio generale di cui l'induzione matematica rappresenta un caso particolare. Iniziamo da ciò che conosciamo: i numeri naturali.

I numeri naturali

Nel 1894 il matematico italiano Giuseppe Peano (1858 - 1932) propose una presentazione assiomatica dell'insieme N dei numeri naturali:

1. $0 \in N$;

2. se $n \in N$, allora $succ(n) \in N$;

3. non esiste nessun $n \in N$ tale che $0 = succ(n)$;

4. se $succ(n) = succ(m)$ allora $n = m$;

5. se $S \subseteq N$ è tale che $0 \in S$ ed inoltre $succ(n) \in S$ se $n \in S$, allora $S = N$.

Un *modello* per questi assiomi è un insieme M equipaggiato con un elemento 0_M ed una funzione "successore" $succ_M : M \to M$ che soddisfano questi assiomi.

Esercizio 3.1 Siano $0_\aleph = \emptyset$, $1_\aleph = \{0_\aleph\}$, $2_\aleph = \{0_\aleph, 1_\aleph\}\ldots$ Ad esempio, espandendo $3_\aleph$ otteniamo: $\{\{\,\}, \{\{\,\}\}, \{\{\,\}, \{\{\,\}\}\}\}$. Sia $\aleph = \{0_\aleph, 1_\aleph, 2_\aleph, 3_\aleph \ldots\}$. La funzione $succ_\aleph$ fa ciò che ci si aspetta: $succ_\aleph(0_\aleph) = 1_\aleph$, $succ_\aleph(1_\aleph) = 2_\aleph$, $succ_\aleph(2_\aleph) = 3_\aleph$ e così via. Questa rappresentazione dei numeri naturali è dovuta al matematico ungherese John von Neumann (1903 - 1957). Verificare che $\aleph$ soddisfa gli assiomi. □

In realtà esistono infiniti modelli per gli assiomi di Peano (provate ad immaginarne uno nuovo). Quale di essi è "quello giusto"? Vediamo prima quali sono quelli sbagliati, ovvero cerchiamo di capire che cosa è un modello e che cosa non lo è. Per farlo, immaginiamo di giocare un gioco interattivo con un alieno in visita da un lontano pianeta dove fossero sconosciuti i numeri naturali: noi glieli descriviamo e, a seconda della sua risposta, raffiniamo la nostra descrizione fino a che l'alieno non coglie il concetto.

Incominciamo così: i naturali formano un insieme N dotato di un elemento 0 e di un'operazione $succ : N \to N$ (assiomi 1 e 2). L'alieno propone subito una soluzione economica: $N = \{0\}$, e $succ(0) = 0$. Non va bene, rispondiamo noi, perché 0 non è il successore di nessuno (assioma 3). Sulla base di questa nuova informazione, l'alieno propone allora: $N = \{0, 1\}$, dove $succ(0) = succ(1) = 1$. Niente da fare: non esistono due numeri distinti che hanno lo stesso successore (assioma 4). L'alieno capisce allora che i naturali sono tanti, e propone: $N = \{0, 1, 2, \ldots, 1326, \ldots \heartsuit\}$, $succ(0) = 1$, $succ(1) = 2 \ldots succ(1326) = 1327$ e $succ(\heartsuit) = \heartsuit$". Troppi!

Esercizio 3.2 Verificare che l'ultimo insieme proposto dell'alieno effettivamente verifica i primi quattro assiomi di Peano. Per escludere questo modello potremmo richiedere che nessun elemento di N possa essere successore di se stesso. Questo non sarebbe tuttavia risolutivo: con quale "strano" insieme potrebbe allora rispondere l'alieno? □

L'ultimo insieme proposto dall'alieno non soddisfa il quinto assioma. Infatti, sia $N_\top = \{0, 1, 2, \ldots, 1326, \ldots \heartsuit\}$ e sia $S = \{0, 1, 2, \ldots, 1326, \ldots\}$. Abbiamo che $0 \in S$ e inoltre $n \in S$ implica $succ(n) \in S$. Dato che $S \subseteq N_\top$, il quinto assioma imporrebbe $S = N_\top$ (ciò che non è, dato che $\heartsuit \notin S$). A questo punto l'alieno propone $\aleph$ e noi ci dichiariamo soddisfatti.

Perché siamo pronti ad accettare $\aleph$ come modello dei naturali ma non gli altri insiemi proposti? Anzitutto perché non vogliamo rinunciare a definire funzioni

per induzione (ovvero per casi, uno per lo zero ed uno per il successore), cosa che nessuno delle altre proposte ci consente di fare. Consideriamo infatti l'usuale definizione della funzione fattoriale:

$$\begin{array}{rcl} \mathit{fact}(0) & = & 1 \\ \mathit{fact}(\mathit{succ}(n)) & = & \mathit{succ}(n) * \mathit{fact}(n). \end{array}$$

Se, violando il quinto assioma, scegliamo $N_\top$ come modello dei naturali questa definizione risulta incompleta, perché non dice quanto vale $\mathit{fact}(\heartsuit)$. Analogamente, la seguente definizione del predicato "essere pari":

$$\begin{array}{rcl} \mathit{is_even}(0) & = & \mathit{true} \\ \mathit{is_even}(\mathit{succ}(n)) & = & \mathit{not}(\mathit{is_even}(n)) \end{array}$$

darebbe risultati contraddittori se, in barba al quarto assioma, adottassimo il secondo modello dell'alieno, dove $\mathit{succ}(1) = 1$; dalle equazioni conseguirebbe infatti $\mathit{true} = \mathit{false}$.

Considerazioni analoghe valgono per le dimostrazioni. Considerate infatti lo schema dell'*induzione matematica*:

$$\frac{P(0) \qquad P(n) \Rightarrow P(n+1))}{\forall\, n\,.\, P(n)}$$

Volendo usare questo schema per dimostrare, ad esempio, che tutti i naturali sono numeri primi, dovremmo scegliere come $P(n)$ la proposizione "n è un numero primo". Ora, se adottassimo $N_\top$ come modello dei naturali, questo schema ci consentirebbe di dimostrare erroneamente che nessun elemento di $N_\top$, è uguale a $\heartsuit$, incluso ahimé $\heartsuit$ stesso (si scelga come $P(n)$ la proposizione "n è diverso da $\heartsuit$").

Va notato che lo schema dell'induzione matematica altro non è che il quinto assioma di Peano (che viene infatti chiamato *principio di induzione*) espresso usando la nozione di predicato al posto di quella di sottoinsieme. Infatti, un predicato P sui numeri naturali può essere visto come un insieme $S_P \subseteq N$, ovvero quel sottoinsieme dei numeri naturali che soddisfano la proprietà espressa dal predicato. L'assioma 5 ci dice che se $0 \in S_p$ (ovvero, se vale $P(0)$) ed inoltre se $n \in S_p$ implica $\mathit{succ}(n) \in S_P$ (ovvero $P(n)$ implica $P(n+1)$), allora $S_P = N$ (ovvero, per ogni $n \in N$ vale $P(n)$).

Torniamo ora alla domanda iniziale: quale tra gli infiniti modelli degli assiomi di Peano è "il vero" insieme dei naturali? La risposta è: *non importa, dato che tutti i modelli sono fra loro isomorfi.* Se, insieme a von Neumann, l'alieno preferisce $\aleph$, faccia pure: come chiariremo ora, è solo una faccenda di *rappresentazione.*

Algebre

L'insieme N dei numeri naturali, con l'operazione di somma e lo 0, elemento neutro dell'operazione, formano un'algebra detta *monoide.* La somma lavora su

due argomenti e viene perciò detta *binaria*, o di *arietà* 2. Questo si indica simbolicamente attaverso una *segnatura*: $N \times N \to N$. Altre strutture algebriche sono, ad esempio, i *gruppi*. Il gruppo Z degli interi possiede, fra le altre, un'operazione $Z \to Z$, di arietà 1, che associa ad ogni intero z il suo *complemento* $-z$. Vi sono poi strutture algebriche le cui operazioni usano (anche) argomeni *esterni* all'algebra stessa: sull'algebra L delle *liste di interi*, ad esempio, è definita un'operazione $cons : L \times Z \to L$ che aggiunge un intero ad una lista: $cons(\langle 2, 5\rangle, 7) = \langle 7, 2, 5\rangle$. In questo caso si dice che l'operazione è *parametrica in* Z. Le definizioni che seguono consentono di trattare in modo uniforme gli esempi fin qui riportati.

Una *segnatura algebrica* è costituita da un insieme I di nomi di funzione e, per ogni $i \in I$, un numero $\alpha_i \geq 0$ ed una sequenza $K_i = \langle K_{i1}, \ldots K_{in_i}\rangle$ di insiemi, i domini degli eventuali parametri esterni dell'operazione. Indichiamo con $|K_i|$ la lunghezza della sequenza K_i. Il numero α_i rappresenta, a meno di parametri esterni, l'arietà di i, che sarà dunque $\alpha_i + |K_i|$. Per brevità di notazione scriveremo semplicemente I per indicare una segnatura. Abuseremo anche della notazione denotando con K_i l'insieme $K_{i1} \times \cdots \times K_{in_i}$, mentre A^n indica come sempre il prodotto cartesiano di un insieme A per se stesso n volte.

Un'*algebra* (A, γ) di segnatura I è una struttura matematica costituita da un insieme A, chiamato *carrier* o insieme *sottostante* dell'algebra, ed una famiglia $\gamma = \{\gamma_i\}_{i \in I}$ di funzioni

$$\gamma_i : A^{\alpha_i} \times K_i \to A$$

chiamate *operazioni fondamentali* dell'algebra[1]. Quando il contesto ci consentirà di farlo senza ambiguità, useremo semplicemente A o γ per indicare un'algebra (A, γ).

La somma, nel monoide dei naturali, è un'operazione di arietà 2 senza parametri esterni ($\alpha_+ = 2$), mentre *cons*, nell'algebra delle liste, è un'operazione di arietà 2 con un parametro esterno ($\alpha_{cons} = 1$). Ora, cosa succede se, per qualche operazione i della segnatura, si ha $\alpha_i = 0$ (e, ad esempio, non vi sono parametri esterni)? In altre parole: cos'è una funzione *nullaria*, cioè con arietà 0? Consideriamo la coppia $(7, 3)$. Essa è un elemento dell'insieme $N \times N = N^2$. La tripla $(9, 2, 6)$ è un elemento di $N \times N \times N = N^3$. Dovremo allora convenire che $()$ sia un elemento dell'insieme N^0, l'unico! Dunque, indicando con $\mathbf{1}$ questo insieme, in analogia col mondo dei numeri dove $n^0 = 1$, si ha:

$$N^0 = \{()\} = \mathbf{1}.$$

Una funzione nullaria su un insieme A avrà dunque segnatura $\mathbf{1} \to A$. Per semplicità assumeremo che, per ogni insieme X, l'equazione $\mathbf{1} \times X = X$ valga in senso stretto. Dunque un'operazione γ_i di un'algebra A con $\alpha_i = 0$ e, ad esempio, un solo parametro esterno K avrà segnatura $K \to A$ e non $\mathbf{1} \times K \to A$.

[1] Algebre con parametri esterni sono dette *eterogenee*. Nei libri di algebra (vedi ad esempio [Lan93, BS81]) si considerano in genere operazioni senza parametri esterni. Questi rendono tutto un po' più complicato, ma sono qui necessari per trattare con strumenti algebrici strutture dati quali liste o alberi o, come vedremo oltre, i termini di una sintassi astratta.

Una funzione nullaria su un insieme A può essere vista come un *elemento* di A. Ad esempio, prendiamo $A = \{\heartsuit, \diamondsuit, \clubsuit, \spadesuit\}$, che ha quattro elementi. Prendiamo poi l'insieme delle funzioni da $\mathbf{1}$ a A. Quanti elementi ha? Quattro. Contiamoli: c'è una funzione, chiamiamola $f_\heartsuit$, definita come segue: $f_\heartsuit(\,()\,) = \heartsuit$; poi c'è $f_\diamondsuit$ tale che $f_\diamondsuit(\,()\,) = \diamondsuit$ ... e così via. Morale: un qualunque insieme A è isomorfo all'insieme di funzioni da $\mathbf{1}$ in A, e questo ci consentirà d'ora in poi di trattare gli elementi di un insieme come funzioni.

Un *omomorfismo* $(A, \gamma) \to (B, \delta)$ tra due algebre con la stessa segnatura I è una funzione $h : A \to B$ tale che, per ogni $i \in I$ con $\alpha_i = n$ e m parametri esterni,

$$h(\gamma_i(a_1, \ldots, a_n, k_1, \ldots k_m)) \quad = \quad \delta_i(h(a_1), \ldots, h(a_n), k_1, \ldots k_m). \qquad (1)$$

Un *isomorfismo* è un omomorfismo bigettivo. Quello che segue è un esempio di omomorfismo. Un non-esempio è dato nell'esercizio 3.7.

Esempio. Sia W l'insieme di tutte le parole su un alfabeto A, ovvero le sequenze finite di elementi di A. W ha una ovvia struttura di monoide, con l'operazione binaria $\mathit{append} : A \times A \to A$ di concatenazione (che è associativa, come richiesto in un monoide), e un elemento neutro $\langle\,\rangle : \mathbf{1} \to A$, la parola vuota. La funzione $\mathit{length} : W \to N$ che associa ad ogni parola la sua lunghezza soddisfa le equazioni $\mathit{length}(\langle\,\rangle) = 0$ e $\mathit{length}(\mathit{append}(w, w')) = \mathit{length}(w) + \mathit{length}(w')$, ed è dunque un omomorfismo nel monoide $(N, +, 0)$. □

Vi sono algebre, dette *induttive*, che hanno un ruolo importante nella nostra trattazione. Vediamo di cosa si tratta.

Sia $f : A^n \times K \to A$ una operazione su A con parametri esterni in $K = K_1 \times \cdots \times K_m$. Un insieme $S \subseteq A$ si dice *chiuso* rispetto ad f quando $a_1, \ldots, a_n \in S$ implica $f(a_1, \ldots, a_n, k_1, \ldots, k_m) \in S$.

Esempio. Indichiamo con Z l'insieme dei numeri interi e con N l'insieme dei naturali. Sia

$$f : Z \times N \to Z$$

la funzione $f(z, n) = z^n$. L'insieme $D \subseteq Z$ dei numeri dispari è chiuso rispetto ad f perché una qualunque potenza di un numero dispari è dispari. L'insieme $P \subseteq Z$ dei numeri pari no! Perchè?

Esercizio 3.3 Quale condizione deve verificare una funzione $f : \mathit{bool} \to Z$ affinché l'insieme $P \subseteq Z$ dei numeri pari sia chiuso rispetto ad essa? □

Definizione 3.1 *Un'algebra* (A, γ) *si dice* induttiva *quando:*

1. *le* γ_i *sono tutte* iniettive*;*
2. *le loro immagini sono* disgiunte*, e*

3. *soddisfano il* principio di induzione*, ovvero: se un insieme $S \subseteq A$ è chiuso rispetto a ciascuna delle γ_i allora $S = A$.*

Le γ_i sono chiamate costruttori *di A. Un costruttore γ_i è chiamato* base *se $\alpha_i = 0$.* □

Esercizio* **3.4** Trovare un insieme di costruttori per i booleani $\{true, false\}$. La funzione *not*, lei da sola, soddisfa i requisiti? □

Esercizio 3.5 Dimostrare che ogni algebra induttiva non vuota ha almeno un costruttore base. Dimostrare che ogni algebra induttiva non vuota con un costruttore non base è necessariamente infinita. □

Teorema 3.1 *Sia A un'algebra induttiva. Per ogni algebra B con stessa segnatura esiste un unico omomorfismo $A \to B$.*

Corollario 3.1 *Due algebre induttive A e B con la stessa segnatura sono necessariamente isomorfe.*

Dimostrazione. Se $f : A \to B$ e $g : B \to A$ sono i due omomorfismi ottenuti applicando il Teorema 3.1 rispettivamente ad A e B, gli omomorfismi composti $g \cdot f : A \to A$ e $f \cdot g : B \to B$ devono coincidere necessariamente (per la condizione di unicità) con le identità, da cui l'isomorfismo. □

Nella sezione 11.1 mostreremo che il teorema 3.1 costituisce il fondamento dello schema *a clausole* che si usa abitualmente in matematica per definire funzioni su insiemi induttivi, e che il linguaggio SML adotta per definire funzioni su `datatype` (vedi sezione 3.3). Il suo corollario, noto nella sua versione categoriale come *lemma di Lambek*, gioca invece un ruolo importante in questa sezione, perché ci consente di distinguere fra struttura e rappresentazione. Mentre in informatica l'implementazione che scegliamo per un tipo di dato può avere un forte impatto sull'efficienza degli algoritmi che manipolano i dati, in matematica si tende a considerare irrilevante ciò che distingue oggetti isomorfi. Il corollario 3.1 ci dice dunque che ciò che in un insieme induttivo è matematicamente rilevante è la struttura, che è data dalla sua segnatura; tutto il resto è rappresentazione.

Torniamo all'insieme N dei numeri naturali. I primi due assiomi di Peano ci danno una segnatura algebrica:

$$\begin{aligned} 0 : &\quad \mathbf{1} \to N \\ succ : &\quad N \to N. \end{aligned}$$

Gli altri assiomi di Peano, 3, 4 e 5, corrispondono esattamente alle condizioni (rispettivamente) 2, 1 e 3 della Definizione 3.1. Questo ci dice che N è un'algebra induttiva, e ciò è quanto basta per definire (induttivamente) funzioni e dimostrare teoremi (per induzione). Poi, per il lemma di Lambek, fa poca differenza rappresentare N come $\{0, 1, \dots\}$ o come $\{\{\}, \{\{\}\}, \dots\}$.

Esercizio 3.6 Mostrare che esiste un isomorfismo algebrico tra l'insieme dei numeri naturali e l'insieme $P = \{0, 2, 4, \dots\}$ dei numeri pari, dove $0_P = 0$ e $succ_P(n) = n + 2$. $\square$

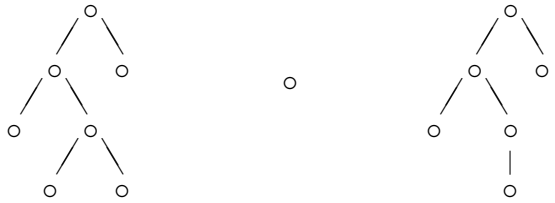
Esercizio 3.7 La funzione $f : N_\top \to \aleph$ tale che $f(\heartsuit) = 0_\aleph$, $f(0) = 1_\aleph$, $f(1) = 2_\aleph$ è *biunivoca* ma non è un isomorfismo di algebre. Perché? $\square$

Esercizio 3.8 Scesi sulla terra, gli alieni ci parlano di algebra universale e, in particolare, di due insiemi, $\square$ e $\triangle$, definiti induttivamente da costruttori con le seguenti segnature. C'è qualcosa di simile, chiedono, sulla Terra?

- $\square$
  $t : \mathbf{1} \to \square$
  $f : \mathbf{1} \to \square$
- $\triangle$
  $l : \mathbf{1} \to \triangle$
  $b : \triangle \times \triangle \to \triangle$

$\square$

Vediamo ora un altro esempio di algebra induttiva, quella degli *alberi binari.* Traiamo ispirazione dalla definizione induttiva di N: un numero naturale è *zero*, oppure il successore di un numero naturale. Analogamente, un albero binario è una *foglia*, oppure una *radice* con un sottoalbero (binario) destro ed uno sinistro. Ecco due esempi di alberi binari ed un non-esempio:



Questi sono solo disegni. Per spiegare ad un alieno la nozione di albero binario, si potrebbe dare la seguente definizione ricorsiva: l'insieme *B-trees* degli alberi binari è *quello* definito induttivamente da due costruttori con la seguente segnatura:

$$\begin{aligned} \textit{leaf} : &\quad \mathbf{1} \to \textit{B-trees} \\ \textit{branch} : &\quad \textit{B-trees} \times \textit{B-trees} \to \textit{B-trees} \end{aligned}$$

(...sì, è proprio $\triangle$!). L'alieno è ora libero di immaginare gli alberi come meglio crede e di scegliere l'interpretazione che preferisce per i simboli *leaf* e *branch*: il lemma di Lambek garantise che, qualunque sia la scelta, la sua rappresentazione sarà isomorfa alla nostra. Ora possiamo usare lo schema induttivo per definire funzioni su *B-trees*. Nella sezione 11.1 torneremo sul fondamento di questo schema; qui ci limiteremo a metterlo in pratica per definire la funzione *leaves*,

che conta le foglie di un albero binario, procedendo per casi, uno per ciascun costruttore, proprio come abbiamo fatto per il fattoriale:

$$\begin{array}{rcl} \mathit{leaves}\,(\mathit{leaf}) & = & 1 \\ \mathit{leaves}\,(\mathit{branch}\,(t_1, t_2)) & = & \mathit{leaves}\,(t_1) + \mathit{leaves}\,(t_2). \end{array}$$

Inoltre, così come il quinto assioma di Peano ci mette a disposizione l'induzione matematica per dimostrare proprietà dei naturali, la terza condizione della Definizione 3.1 ci mette a disposizione l'induzione *strutturale*. Funziona come quella matematica: per dimostrare che una proprietà vale per ogni elemento di un insieme A definito induttivamente si considera ciascun costruttore $\gamma : A^n \times K_1 \times \ldots K_m \to A$ e, assumendo che la proprietà valga per $a_1, \ldots a_n$, si dimostra che, per qualunque $k_1, \ldots k_m$, essa vale per $\gamma(a_1, \ldots a_n, k_1, \ldots k_m)$. Come esempio dimostriamo la seguente proposizione:

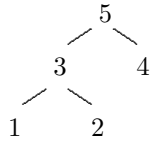
Proposizione 3.1 *Un albero binario con n foglie ha $2n - 1$ nodi.*

Dimostrazione. Un albero binario della forma *leaf*(()), cioè l'albero composto di una sola foglia, ha effettivamente $2 * 1 - 1 = 1$ nodi. Ora, assumiamo che la proprietà valga per gli alberi b_1 e b_2; dobbiamo dimostrare che vale per l'albero $b = \mathit{branch}(b_1, b_2)$. Sia $n_1 + n_2$ il numero di foglie di b, dove le foglie di b_1 sono n_1 e quelle di b_2 sono n_2. Per ipotesi b_1 ha $2\,n_1 - 1$ nodi e b_2 ne ha $2\,n_2 - 1$. Dunque b avrà $(2\,n_1 - 1) + (2\,n_2 - 1) + 1$ nodi, ovvero $2(n_1 + n_2) - 1$ come volevasi dimostrare. □

Esercizio 3.9 Dimostrare la Proposizione 3.1 per induzione sul numero di nodi. Ovviamente si può, ma c'è una complicazione: la scelta ingenua di "ogni albero binario con n foglie ha $2n-1$ nodi" come proprietà $P(n)$ non funziona (perché?). Abbiamo bisogno della cosiddetta *induzione completa*. □

Esercizio 3.10 Dimostrare che un albero binario con n nodi interni ha $n + 1$ foglie. □

Esercizio* 3.11 Consideriamo alberi binari con nodi etichettati da numeri naturali. Eccone uno:



Definire l'algebra induttiva *BN-trees* di questi alberi. Definire un'algebra di uguale segnatura sull'insieme S delle sequenze finite di numeri naturali in modo che la funzione f : *BN-trees* $\to S$ che associa a ciascun albero la sequenza di etichette ottenuta con una visita *depth-first* sia un omomorfismo. Applicando f all'all'albero dell'esempio, ci aspettiamo di ottenere la sequenza $\langle 5, 3, 1, 2, 4 \rangle$. □

Esercizio* **3.12** Dato un insieme A, indichiamo con A-*list* l'insieme delle liste finite di elementi di A. Indichiamo con () la lista vuota e con l_i l'i-esimo elemento di una lista l. Definire induttivamente il seguente insieme L:

$$\begin{array}{l} L_0 = \{()\} \\ L_{n+1} = L_n \cup L_n\text{-}\mathit{list} \\ L = \bigcup_{n \in N} L_n \end{array}$$

kkk □

Alberi come grafi Nella teoria dei grafi un albero è un grafo non orientato nel quale due vertici sono connessi da uno ed un solo cammino. Un albero è definito dunque in termini di una coppia (V, E) dove V è un *insieme* di vertici (assumiamolo finito) ed E è un *insieme* di archi con particolari proprietà. In particolare, un albero *radicato* è un albero nel quale ad un vertice viene conferito uno speciale status di "radice". Da questa investitura deriva un orientamento naturale degli archi (che consiste nell'allontanarsi dalla race) ed una relazione padre-figlio. Se aggiungiamo poi il vincolo che ogni padre abbia esattamente due figli, otteniamo gli alberi binari. Che rapporto c'è fra gli alberi binari così ottenuti e gli elementi di *B-trees*? Ebbene: ne sono la versione *commutativa*! Notiamo infatti che mentre, per l'iniettività dei costruttori, $\mathit{branch}(t_1, t_2) \neq \mathit{branch}(t_2, t_1)$ per $t_1 \neq t_2$, negli alberi-grafo non c'è alcun ordinamento fra i figli di un nodo, e non ha dunque senso parlare di sottoalbero destro e sinistro. Per ottenere gli alberi-grafo (binari), dobbiamo dunque *quozientare B-trees* rispetto alla congruenza indotta dall'equazione $\mathit{branch}(t_1, t_2) = \mathit{branch}(t_2, t_1)$. Gli alberi-grafo binari corrispondono allora alle classi di equivalenza indotte su *B-trees* dalla relazione che esprime la commutatività di *branch*.

Quella degli "alberi commutativi" è un esempio di *teoria algebrica*, ovvero una teoria i cui assiomi sono equazioni. Altre teorie algebriche sono quella dei monoidi, dei gruppi, degli anelli, dove le equazioni impongono, ad esempio, l'associatività della moltiplicazione. Anche per algebre di questo tipo esiste un equivalente del teorema 3.1, ma questo ci porta oltre i limiti della nostra trattazione.

Sintassi Astratta

Sia L il linguaggio (un insieme di *stringhe*) generato dalla seguente grammatica.

$$\mathit{Exp} \quad ::= \quad \texttt{0} \mid \texttt{1} \mid \ldots \mid \mathit{Exp}\ \texttt{+}\ \mathit{Exp} \mid \mathit{Exp}\ \texttt{*}\ \mathit{Exp}$$

Supponiamo di voler definire una funzione $\mathit{eval} : L \to N$ che valuta le espressioni del linguaggio. Se e_1 ed e_2 sono stringhe, indichiamo con e_1`+`e_2 la stringa ottenuta concatenando le tre stringhe e_1, `"+"` ed e_2. Ad esempio, `"4"+"7"` = `"4+7"`.

$$
\begin{array}{rcl}
eval(\texttt{0}) & = & 0 \\
eval(\texttt{1}) & = & 1 \\
\ldots & & \\
eval(e_1 \texttt{ + } e_2) & = & eval(e_1) \ + \ eval(e_2) \\
eval(e_1 \texttt{ * } e_2) & = & eval(e_1) \ * \ eval(e_2)
\end{array}
$$

Attenzione alla differenza tra `0` e 0, tra `+` e +, e così via. Ora, quanto fa $eval(\texttt{3 * 2 + 1})$? Dipende:

$$
\begin{array}{l}
eval(\texttt{3 * 2 + 1}) = eval(\texttt{3}) * eval(\texttt{2 + 1}) = 3 * 3 = 9, \text{ oppure} \\
eval(\texttt{3 * 2 + 1}) = eval(\texttt{3 * 2}) + eval(\texttt{1}) = 6 + 1 = 7.
\end{array}
$$

Ciò che abbiamo tentato di fare è procedere per casi, così come avevamo fatto per *fact* o per *leaves* nelle sezioni precedenti. Ciò che rende però accettabile la definizione di *fact* ma non quella di *eval* è che la prima applica uno schema induttivo di definizione su un insieme definito induttivamente (N), mentre la seconda tenta di applicare lo stesso schema ad un insieme (L) che induttivo non è. In effetti, ciascuna produzione della grammatica definisce un operatore su L. Ad esempio, `Exp ::= Exp + Exp` definisce l'operatore binario $Plus : L \times L \to L$ tale che $Plus(\texttt{"3"}, \texttt{"2 + 1"}) = \texttt{"3 + 2 + 1"}$. Analogamente, per il prodotto: $Times(\texttt{"3"}, \texttt{"2 + 1"}) = \texttt{"3 * 2 + 1"}$, e così via. Questi operatori sintattici tuttavia non sono i *costruttori* del linguaggio, ovvero non definiscono L induttivamente. In particolare *Plus* e *Times* non hanno immagini disgiunte, visto che $Times(\texttt{"3"}, \texttt{"2 + 1"}) = Plus(\texttt{"3 * 2"}, \texttt{"1"})$, da cui l'ambiguità nella definizione di *eval*.

La *sintassi astratta* di un linguaggio è una definizione *induttiva* di un insieme T di *termini*. Per esempio, la sintassi astratta del linguaggio delle espressioni è data dalla seguente segnatura:

$$
\begin{array}{rl}
zero : & \mathbf{1} \to T \\
one : & \mathbf{1} \to T \\
\ldots & \ldots \\
plus : & T \times T \to T \\
times : & T \times T \to T
\end{array}
$$

Ora la domanda è: come sappiamo che un'algebra induttiva con questa segnatura *esiste*? La seguente costruzione ci assicura in tal senso.

Data una segnatura I, definiamo un'algebra (T, τ) su I:

$$
\begin{array}{rcl}
T_0 & = & \emptyset \\
T_{n+1} & = & T_n \cup \{\, (i, t_1, \ldots, t_{\alpha_i}, k_1, \ldots, k_{|K_i|}) \mid i \in I,\ t_j \in T_n, \text{ per } j = 1 \ldots \alpha_i, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad \text{e } k_l \in K_l, l = 1 \ldots |K_i| \,\} \\
T & = & \bigcup_{n \geq 0} T_n
\end{array}
$$

Le operazioni τ_i sono definite come segue:

$$\tau_i(t_1, \ldots, t_n, k_1, \ldots, k_n) = (i, t_1, \ldots, t_n, k_1, \ldots, k_m).$$

Va notato che, in accordo con quanto visto nell'esercizio 3.5, se non vi sono costruttori base in τ, T è vuoto. Gli elementi di T sono chiamati *termini*, e T *algebra dei termini* su I.

Teorema 3.2 (T, α) *è un'algebra induttiva.*

Torniamo alla nostra grammatica per le esperssioni. Abbiamo visto che essa definisce un'algebra (non induttiva) sul linguaggio L. Il teorema 3.2 ci dà la sua sintassi astratta, i cui *termini* possono ora essere valutati senza ambiguità da una funzione *eval* correttamente definita dalle seguenti clausole.

$$\begin{array}{rcl} eval(zero(()) & = & 0 \\ eval(one(()) & = & 1 \\ \ldots & \ldots & \\ eval(plus(t_1, t_2)) & = & eval(t_1) + eval(t_2) \\ eval(times(t_1, t_2)) & = & eval(t_1) * eval(t_2) \end{array}$$

3.2 Co-induzione

Nell'esercizio 3.12 abbiamo definito l'algebra *BN-trees* degli alberi binari etichettati da numeri naturali; ricordiamone la segnatura:

$$N \xrightarrow{leaf} BN\text{-}trees \xleftarrow{branch} N \times BN\text{-}trees \times BN\text{-}trees$$

Ci sono poi gli alberi binari (etichettati) *infiniti*, cioè quelli che hanno un numero infinito di nodi. Indichamo con B^∞-*trees* l'insieme di tutti gli alberi binari, finiti e infiniti. Chiaramente *B-trees* $\subseteq B^\infty$-*trees* e le operazioni *leaf* e *branch* si estendono banalmente a B^∞-*trees*. Ad esempio, c'è l'albero in cui ogni sottoalbero è infinito e i cui nodi hanno tutti etichetta 7, chiamiamolo $\mathbf{7}$; poi c'è, l'albero fatto da una sola foglia etichettata con 7, chiamiamolo $\mathbf{7}$; poi c'è $branch(7, \mathbf{7}, \mathbf{7})$, e così via.

Considerate ora la seguente variante del famoso *lemma di König*:

Teorema. Ogni albero infinito in B^∞-*trees* ha almeno un cammino infinito.

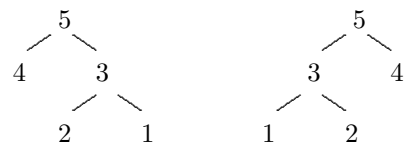
Dimostrazione. ($\ddot{\frown}$) Sia $t \in B^\infty$-*trees* un albero infinito. Sicuramente t è della forma $branch(n, t_1, t_2)$, ed assumiamo induttivamente che la proprietà valga per t_1 e t_2. Poiché t è infinito, sicuramente lo deve essere anche almeno uno dei due sottoalberi, diciamo t_1. Applicando l'ipotesi induttiva, t_1 ha un cammino infinito, e dunque anche t ha un cammino infinito. □

Cosa c'è che non va in questo argomento? C'è che, se l'*argomento* fosse corretto, allora potremmo dimostrare allo stesso modo anche il suo contrario:

Teorema. ($\ddot{\frown}$) Nessun albero infinito in B^∞-*trees* ha cammini infiniti.

Dimostrazione. ($\ddot{\frown}$) Sia $t \in B^\infty$-*trees* un albero infinito. Sicuramente t è della forma $\mathit{branch}\,(n, t_1, t_2)$, ed assumiamo induttivamente che la proprietà valga per t_1 e t_2. Ogni cammino di t è finito perché è ottenuto aggiungendo un solo passo ad un cammino di t_1 o di t_2, che è finito per ipotesi. $\square$

Entrambe le dimostrazioni (una di una proposizione vera, l'altra di una palesemente falsa) applicano erroneamente un argomento induttivo ad un insieme, B^∞-*trees*, che induttivo non è. Analogamente, lo schema di definizione "per casi" considerato nella Sezione **??** non funziona per B^∞-*trees*. Supponiamo, ad esempio, di voler definire una funzione $\mathit{flip} : B^\infty\text{-}\mathit{trees} \to B^\infty\text{-}\mathit{trees}$ che restituisce una versione speculare dell'albero in input. Dunque, applicando *flip* all'albero di sinistra nella seguente figura ci aspettiamo di ottenere come risultato l'albero sulla destra.



La seguente definizione funziona bene per alberi finiti ma, proprio come nella sezione **??** *fact* non definiva il fattoriale su $N_\top$, non definisce *flip* su tutto l'insieme B^∞-*trees*.

$$\begin{aligned}
&\mathit{flip}\,(\mathit{leaf}(n)) = \mathit{leaf}(n)\\
&\mathit{flip}\,(\mathit{branch}\,(n, t_1, t_2)) = \mathit{branch}\,(n, \mathit{flip}\,(t_2), \mathit{flip}\,(t_1))
\end{aligned}$$

Non tutto è perduto. Insiemi come B^∞-*trees* vengono detti *co-induttivi* e godono, come i cugini induttivi, di una proprietà di "minimalità" (ricordate nella Sezione **??**? N è minimale nel senso in cui $N_\top$ non lo è) che ci consente di definire funzioni (e dimostrare proprietà). Ecco di cosa si tratta.

Data una famiglia $\{R_i\}_{i=1\ldots n}$ di relazioni di equivalenza $R_i \subseteq A_i \times A_i$ otteniamo una relazione $R_\times$ sull'insieme prodotto $A_1 \times \cdots \times A_n$, come segue: $(a_1, \ldots a_n)\, R_\times (b_1, \ldots b_n)$ se e solo se $a_i\, R_i\, b_i$, per ogni i. Analogamente, la relazione R_+ sull'insieme somma (unione disgiunta) $A_1 + \cdots + A_n$ è definita da: $a\, R_+ b$ se e solo se esiste un i tale che $a, b \in A_i$ e $a\, R_i\, b$.

Ora, sia δ una famiglia di funzioni della forma

$$\begin{aligned}
\delta_1 &: A \to B_1\\
&\vdots\\
\delta_m &: A \to B_m
\end{aligned}$$

dove le B_i sono insiemi ottenuti per somme e prodotti a partire da una famiglia $\{X_i\}$ di insiemi "base", tra cui (eventualmente) A; ad esempio, $(C \times A) + (A \times D)$

è un possibile B_i. A partire da una relazione di equivalenza R su A definiamo una famiglia di relazioni "base" $R_i \subseteq X_i \times X_i$ come segue: R_i è R se $X_i = A$, altrimenti R_i è l'identità. Applicando ricorsivamente la costruzione per somme e prodotti descritta sopra, otteniamo una relazione $\hat{R}_j \subseteq B_j \times B_j$, per ogni $j = 1 \ldots m$. Una relazione di equivalenza $R \subseteq A \times A$ si dice *congruenza* rispetto alla famiglia δ quando $a\,R\,b$ implica $\delta_j(a)\,\hat{R}_j\,\delta_j(b)$, per ogni j. Diciamo che A è definito *co-induttivamente* dalla famiglia δ quando la sola congruenza rispetto a δ è l'identità. In tal caso le funzioni δ_i sono chiamate *osservatori* o, per simmetria con i tipi induttivi, *distruttori*.

Esercizio 3.13 Dimostrare che B^∞-*trees* è un insieme co-induttivo. Quali sono i distruttori? □

Eccoli:

$$N \xleftarrow{root} \textit{B-trees} \xrightarrow{split} \mathbf{1} + (\textit{B-trees} \times \textit{B-trees}).$$

La funzione *root* restituisce l'etichetta della radice, mentre *split* restituisce () se l'albero è una foglia, o la coppia dei figli destro e sinistro altrimenti. La minimalità di B^∞-*trees* consiste nel fatto che non esistono due alberi diversi che non possano essere distinti applicando ricorsivamente gli osservatori un numero finito di volte. Questa proprietà giustifica un schema di definizione di funzioni "duale" rispetto a quello per induzione: invece di definire quanto vale la funzione sul risultato di ciascun costruttore, si specifica cosa producono gli osservatori se applicati al risultato della funzione:

$$\begin{array}{rcl} \textit{root}\,(\textit{flip}\,(t)) & = & \textit{root}\,(t) \\ \textit{split}\,(\textit{flip}\,(t)) & = & \texttt{case}\ (\textit{split}\,(t))\ \texttt{of} \\ & & \quad () : () \\ & & \quad (t_1, t_2) : (\textit{flip}\,(t_2), \textit{flip}\,(t_1)). \end{array}$$

Essendo ogni albero *univocamente* determinato dalle osservazioni che su di esso si possono fare, la definizione risulta corretta. Va notato che la definizione co-induttiva di B^∞-*trees* non parla della struttura dei suoi elementi (foglie? rami?) ma solo degli esperimenti (osservazioni) che su di essi si possono fare. Analogamente, possiamo "smontare" gli alberi mediante i distruttori, ma non operare su di essi in base alla loro struttura. Questo è il principio ispiratore, nel mondo dei linguaggi di programmazione, della nozione di tipo di dato astratto (vedi Sezione 3.3).

Esempio. Definiamo uno *stream* come una sequenza di numeri naturali infinita "a destra". Ad esempio: $s = \langle 7, 100, 84, \ldots \rangle$ è uno stream. L'insieme degli stream è definito co-induttivamente dai seguenti distruttori:

$$N \xleftarrow{first} \textit{Streams} \xrightarrow{rest} \textit{Streams}.$$

dove, ad esempio $\textit{first}\,(s) = 7$ e $\textit{rest}\,(s) = \langle 100, 84, \ldots \rangle$. Si definisce facilmente una funzione $\textit{cons} : N \times \textit{Streams} \to \textit{Streams}$ che appiccica un naturale in testa a uno stream:

$$\begin{array}{rcl} \mathit{first}\,(\mathit{cons}\,(n,s)) & = & n \\ \mathit{rest}\,(\mathit{cons}\,(n,s)) & = & s. \end{array}$$

□

Non esempio. Sia $\mathcal{O}$ l'insieme delle coppie $(X, \leq_X)$ dove X è un sottoinsieme infinito di N e $\leq_X$ è un ordinamento (totale) di X con la proprietà che non esistono catene discendenti *infinite* $n_1 \geq_X n_2 \geq_X n_3 \geq_X \ldots$. Per certi versi $\mathcal{O}$ assomiglia a *Streams*; ad esempio, ogni $(X, \leq_X) \in \mathcal{O}$ ha un "primo" elemento (esiste sempre un elemento minimo rispetto all'ordinamento $\leq_X$. Dimostratelo!) e un "resto", ottenuto togliendo da X il primo elemento. Sull'insieme $\mathcal{O}$, come sugli stream, sono pertanto definite due operazioni:

$$N \xleftarrow{\mathit{first}} \mathcal{O} \xrightarrow{\mathit{rest}} \mathcal{O}.$$

L'insieme $\mathcal{O}$ non è però co-induttivamente definito da *first* e *rest*. Perché? (Spunto: ricordate $N_\top$?)

3.3 I tipi di dato nei linguaggi di programmazione

Abbiamo visto nella Sezione 3.2 che gli elementi degli insiemi co-induttivi possono essere "osservati", ma non manipolati in base alla loro struttura, come avviene invece per i cugini induttivi. Nella programmazione si usa a volte l'espressione *tipo di dato astratto* per indicare un tipo, generalmente definito dal programmatore, in cui rimane mascherata (*incapsulamento*) parte dell'implementazione, ovvero della struttura interna dei dati, e dotato di operazioni che possono invece accedere all'implementazione, e sono dunque le sole che garantiscono la consistenza interna dei dati.

Nel linguaggio Java l'incapsulamento avviene mediante l'uso dei *modificatori d'accesso* `private` e `protected`. Lo *Standard ML* consente di definire tipi di dato astratto attraverso il costrutto `abstype`, oppure mediante un uso opportuno di *strutture* e *segnature*, che fanno parte del suo *sistema dei moduli*.

Va notato che lo *Standard ML* supporta la definizione "per casi" di funzioni su `datatype`, che possono dunque considerarsi come tipi induttivi, ma non la definizione "per osservazioni" su `abstype`. Non è ancora nato un linguaggio *commerciale* che supporta a pieno la nozione di tipo co-induttivo. Tali tipi sono però stati implementati da Eduardo Gimenez nel metalinguaggio di *Coq*, un sistema di assistenza alle dimostrazioni sviluppato presso l'INRIA, l'istituto francese per la ricerca informatica.

L'algebra induttiva *B-trees* degli alberi binari della sezione 3.1 può essere rappresentata in SML sotto forma di *datatype*:

$$\begin{array}{rl} \textit{datatype}\ \text{B-trees} = & \textit{leaf of}\ \text{unit} \\ & \mid \textit{branch of}\ \text{B-trees} \times \text{B-trees} \end{array}$$

Il tipo unit rappresenta in SML quello che abbiamo indicato con **1** nella sezione 3.1, ed ha () come unico valore. La definizione è composta da due clausole

separate dalla barra verticale "|", una per ciascun costruttore dell'algebra, *leaf* e *branch*. Dei costruttori l'SML richiede di specificare soltanto il dominio (nel nostro caso rispettivamente unit e B-trees $\times$ B-Trees). Il codominio, che per definizione è l'algebra stessa, viene lasciato implicito. Dunque, ad esempio, la seconda clausola, *branch of* B-trees$\times$B-trees, specifica che *branch* è una funzione di tipo B-Trees $\times$ B-trees $\rightarrow$ B-trees. Va notato che, nel caso di costruttori con arietà zero, come ad esempio *leaf* per i B-tree, l'SML consente di omettere il dominio unit del costruttore, come si vede nella soluzione dell'esercizio 3.14. L'SML consente poi di definire per casi le funzioni sui datatype, proprio come abbiamo fatto nella sezione 3.1. Ad esempio, la funzione *leaves* può essere programmata come segue:

$$\begin{array}{l} \mathit{fun}\ \ \mathit{leaves}\,(\mathit{leaf}) = 1 \\ \quad \mid \mathit{leaves}\,(\mathit{branch}\,(t_1, t_2)) = \mathit{leaves}\,(t_1) + \mathit{leaves}\,(t_2) \end{array}$$

Esercizio* **3.14** Definire in SML un datatype `BoolExp` per rappresentare espressioni booleane, ed una funzione di valutazione `valu: BoolExp -> bool` che restituisce il valore di verità di una espressione. □

Parte II
Paradigmi di Programmazione

Questa sezione affronta alcune tematiche relativa alla semantica dinamica dei linguaggi di programmazione. In particolare, vengono studiate diverse regole di scoping e strategie di valutazione dei parametri nei paradigmi di programmazione imperativa, funzionale e ad oggetti.

Lo *scoping* di un linguaggio è quell'insieme di regole che determinano la visibilità di una variabile all'interno di un programma, che consentono cioè di associare una variabile a ciascun *riferimento* (ovvero uso della variabile mediante un identificatore).

Lo scoping può essere:

- *Statico*: i riferimenti ad una variabile sono risolti in base alla struttura sintattica del programma, tipicamente in base ad una dichiarazione esplicita o implicita.
- *Dinamico*: i riferimenti sono risolti in base allo stato di esecuzione del programma, per esempio: una dichiarazione estende il suo effetto fino a che non si incontra un'altra dichiarazione di variabile con lo stesso nome. Questo avviene in APL e in LISP.

Uno stesso linguaggio può adottare regole di scoping diverse per entità di tipo diverso. In Java, per esempio, il riferimento ad un *campo* di un oggetto (ovvero la variabile a cui punta il riferimento) viene risolto staticamente, mentre il riferimento ad un metodo viene risolto dinamicamente. In generale, per linguaggi ad oggetti (ad esempio C^{++} ed Eiffel) l'espressione "scoping dinamico" (o binding dinamico) viene applicata ad indicare che il dispatching dei metodi avviene a tempo di esecuzione (per default in Java ed Eiffel, per metodi *virtual* in C^{++}).

Esempio:

```
integer x;
x := 7;

procedure pippo
   begin
   x := 5;
   end;

procedure pluto
   begin
   integer x;
   pippo;
   end;
```

```
pluto;
```

Dopo l'esecuzione della procedura `pluto` la variabile `x` vale 5 in un linguaggio con scoping statico, mentre vale 7 in uno con scoping dinamico.

4 Il Paradigma Funzionale

Nel discorso matematico l'astrazione è pervasiva. Ne è un esempio, forse il più elementare, il concetto di *variabile*, ovvero l'astrazione di un valore. Nella sezione 4.1 introduciamo un semplice linguaggio di espressioni aritmetiche, *Exp*, che comprende l'apparato linguistico minimo per poter studiare la nozione di variabile di un linguaggio di programmazione funzionale, quali ad esempio SML, Haskell e LISP. Arricchendo questo linguaggio con la nozione di *funzione* (ovvero l'astrazione di un termine rispetto ad una variabile) otterremo poi un linguaggio estremamente espressivo, *Fun* (sezione 4.2), dove impareremo a codificare booleani, liste, numeri e gran parte dell'aritmetica. *Exp* e *Fun* ci forniranno il contesto linguistico per mettere a confronto il *binding statico* delle variabili, adottato in SML e Haskell, e quello *dinamico*, adottato in certe implementazioni del LISP e di *S*, un linguaggio largamente usato per il calcolo statistico. Studieremo inoltre due strategie di valutazione dei programmi: quella *eager* (sollecita, in inglese si usa anche *strict*), adottata in SML e LISP, e quella *lazy* (pigra), usata in Haskell e in *S*. Incrociando questi due aspetti ortogonali nella semantica dei linguaggi funzionali, regole di scoping e strategie di valutazione, otteniamo la seguente tabella a quattro caselle, ciascuna con un suo linguaggio rappresentante a fornire riscontro sperimentale alla trattazione teorica.

	statico	**dinamico**
eager	SML	Emacs LISP
lazy	Haskell	S-Plus

4.1 *Exp*: un semplice linguaggio funzionale

[RISCRIVERE!] Nel presentare *Exp* come embrione di linguaggio funzionale, abbiamo usato l'aggettivo "puro" perché, in questo contesto, una variabile rappresenta formalmente un qualcosa di diverso dalla variabile di un linguaggio di programmazione reale: mentre quest'ultima è usata tipicamente per l'operazione di *assegnamento*, ed è fatta dunque per "contenere" un *valore*, una variabile viene vista in *Exp* come *segnaposto* per un qualunque *termine*, ed è usata tipicamente nell'operazione di *sostituzione*. Con uno slogan: l'assegnamento agisce su valori, mentre la sostituzione agisce su termini, i quali rappresentano *computazioni* che non necessariamente producono un valore.

4.1.1 Sintassi e semantica

Assumiamo che siano dati un insieme *Var* di variabili ed un insieme di costanti, entrambi numerabili. Useremo le *meta*variabili x, y... per indicare generiche variabili, k_1, k_2... per le costanti ed M, N... per i termini del linguaggio. L'insieme *Exp* di tali termini è definito induttivamente dalla seguente sintassi astratta:

$$
\begin{array}{lcl}
k & ::= & 5 \,|\, 40 \,|\, \ldots \\
M & ::= & k \,|\, x \,|\, M + N \,|\, \mathit{let}\ x = M \text{ in } N
\end{array}
$$

Usando la sintassi di SML, abbiamo indicato con $\mathit{let}\ x = M$ in N termini derivanti dall'applicazione di un operatore di sintassi astratta con segnatura:

$$\mathit{let} : \mathit{Var} \times \mathit{Exp} \times \mathit{Exp} \to \mathit{Exp}.$$

Intuitivamente un tale termine rappresenta un segmento di codice che comprende la definizione di una variabile *locale* x, inizializzata al *valore* dell'espressione M, ed un corpo N che potrà contenere riferimenti ad x. Per esempio, la valutazione del termine $\mathit{let}\ x = 3 + 2$ in $x + x$ crea una variabile x, il cui valore, $3 + 2$, viene usato localmente nell'espressione $x + x$ per calcolare il risultato, 10. Questa spiegazione del costrutto *let* descrive informalmente quanto avviene in SML; esploreremo più avanti soluzioni semantiche alternative, in particolare rispetto a località e valore.

Vi sono due riferimenti ad x nell'espressione $\mathit{let}\ x = 3 + 2$ in $x + x$. Si dice in questo caso che l'espressione contiene due *occorrenze* della variabile. Per la x che compare testualmente subito dopo il *let* (ovvero come primo argomento del costruttore *let*) parleremo invece di *dichiarazione*. Ora: quante occorrenze di x troviamo nell'espressione $\mathit{let}\ x = 3\ \mathit{in}\ x + \mathit{let}\ x = 2\ \mathit{in}\ x + x$? Dipende, perché questa espessione (esercizio: quanto fa?) contiene in realtà *due* variabili distinte con lo stesso nome e, per capire quante volte occorra ciascuna di esse, dobbiamo essere più specifici sulla struttura dell'espressione. Ad esempio, esplicitando con l'uso di parentesi la sua sintassi astratta, possiamo dare di questa espressione la seguente interpretazione: $\mathit{let}\ x = 3\ \mathit{in}\ (x + ((\mathit{let}\ x = 2\ \mathit{in}\ x) + x))$, dalla valutazione della quale ci aspettiamo il risultato 8. In questo caso, la variabile x il cui valore è 3 occorre due volte, mentre quella con valore 2 una soltanto. Al fine di individuare le variabili in gioco dobbiamo perciò esplicitare la struttura del termine e *legare* occorrenze a dichiarazioni.

Si dice che una occorrenza di una variabile x è *libera* in un termine t quando non compare nel corpo N di nessun sottotermine di t della forma $\mathit{let}\ x = M$ in N. Ogni occorrenza libera di x in un termine N si dice *legata* (*bound*) alla dichiarazione di x nel termine $\mathit{let}\ x = M$ in N. Lo *scope* di tale dichiarazione è l'insieme delle occorrenze libere di x in N. In generale, con *scope di una variabile* si intende la porzione di programma all'interno della quale una variabile può essere riferita. Due espressioni si dicono *alfa-equivalenti* se sono identiche a meno di ridenominazione di variabili legate. Ad esempio: l'espressione $\mathit{let}\ x = 1$ in $x + 1$ è alfa-equivalente a $\mathit{let}\ y = 1$ in $y + 1$.

Per esercizio, definiamo induttivamente una funzione $\mathit{free} : \mathit{Exp} \to \mathcal{P}(\mathit{Var})$, che restituisce l'insieme delle variabili che occorrono libere in una espressione:

$$
\begin{array}{rcl}
\mathit{free}\,(k) & = & \emptyset \\
\mathit{free}\,(x) & = & \{x\} \\
\mathit{free}\,(M + N) & = & \mathit{free}\,(M) \cup \mathit{free}\,(N) \\
\mathit{free}\,(\mathit{let}\ x = M\ \mathit{in}\ N) & = & \mathit{free}\,(M) \cup (\mathit{free}\,(N) - \{x\}).
\end{array}
$$

Domini semantici. La semantica di *Exp* viene presentata nello stile *operazionale* (*structural operational semantics*: *SOS*) usando la nozione di *ambiente*. Un ambiente è una funzione parziale che associa dei *valori* ad un insieme *finito* di variabili:

$$Env = Var \xrightarrow{fin} Val$$

Per questo semplice linguaggio l'insieme *Val* dei "valori", ovvero l'insieme dei possibili *risultati* della valutazione di un termine, coincide con l'insieme delle costanti. In generale, usiamo la metavariabile v per indicare generici valori.

Scriviamo gli ambienti come insiemi di coppie. Per esempio, l'ambiente E in cui z vale 3 ed y vale 9 è indicato con $\{(z,3),(y,9)\}$. Come richiesto dalla definizione, il dominio $dom(E)$ di questo ambiente è un sottoinsieme finito $\{y,z\}$ di *Var*. La *concatenazione* di E_1 ed $E_2 \in Env$ è l'ambiente, scritto E_1E_2, il cui dominio è $dom(E_1) \cup dom(E_2)$, e tale che:

$$(E_1E_2)(x) = \begin{cases} E_2(x) & \text{se } x \in dom(E_2) \\ E_1(x) & \text{altrimenti} \end{cases}$$

Per esempio, $(\{(z,3),(y,9)\}\,\{(z,4)\})(z) = 4$, mentre $(\{(z,3),(y,9)\}\,\{(z,4)\})(x)$ è indefinito.

Regole. La semantica operazionale di *Exp* è una relazione

$$\rightsquigarrow \quad \subseteq \quad Env \times Exp \times Val.$$

Un'asserzione di appartenenza $(E,M,v) \in \rightsquigarrow$ viene chiamata *giudizio operazionale* (in inglese: *operational judgement*) e si scrive: $E \vdash M \rightsquigarrow v$. Questa relazione è definita dal seguente insieme di regole.

$$[const] \quad E \vdash k \rightsquigarrow k$$

$$[var] \quad E \vdash x \rightsquigarrow v \quad (\text{se } E(x) = v)$$

$$[plus] \quad \frac{E \vdash M \rightsquigarrow v \quad E \vdash N \rightsquigarrow w}{E \vdash M + N \rightsquigarrow u} \quad (\text{se } u = v + w)$$

$$[let] \quad \frac{E \vdash M \rightsquigarrow v \quad E\{(x,v)\} \vdash N \rightsquigarrow v'}{E \vdash let\ x = M\ in\ N \rightsquigarrow v'}$$

La seguente proposizione si dimostra banalmente per induzione sulla struttura dei termini.

Proposizione 1 *Ogni programma in Exp* termina*, ovvero: per ogni termine $M \in Exp$ ed ambiente $E \in Env$ tale che $free(M) \subseteq dom(E)$ esiste un valore v tale che $E \vdash M \rightsquigarrow v$ è derivabile.*

4.1.2 Equivalenza operazionale

Definiamo *operazionalmente equivalenti* due programmi M ed N tali che, per ogni ambiente E e valore v, $E \vdash M \rightsquigarrow v$ se e solo se $E \vdash N \rightsquigarrow v$. L'equivalenza operazionale, che indichiamo con

$$M \sim N,$$

è chiaramente una relazione di equivalenza tra programmi.

Esempio. Assumendo che la variabile y non sia libera nel programma L, dimostriamo l'equivalenza

$$\mathit{let}\ x = (\mathit{let}\ y = M\ \mathit{in}\ N)\ \mathit{in}\ L \sim \mathit{let}\ y = M\ \mathit{in}\ (\mathit{let}\ x = N\ \mathit{in}\ L).$$

Supponiamo che E e v siano tali che $E \vdash \mathit{let}\ x = (\mathit{let}\ y = M\ \mathit{in}\ N)\ \mathit{in}\ L \rightsquigarrow v$. Devono dunque esistere due valori v' e v'' che rendano possibile la seguente derivazione.

$$\frac{\dfrac{E \vdash M \rightsquigarrow v'' \quad E\{(y, v'')\} \vdash N \rightsquigarrow v'}{E \vdash \mathit{let}\ y = M\ \mathit{in}\ N \rightsquigarrow v'} \quad E\{(x, v')\} \vdash L \rightsquigarrow v}{E \vdash \mathit{let}\ x = (\mathit{let}\ y = M\ \mathit{in}\ N)\ \mathit{in}\ L \rightsquigarrow v}$$

L'assunzione che y non sia libero in L e che $E\{(x, v')\} \vdash L \rightsquigarrow v$ sia derivabile ci consente di asserire che $E\{(y, v''), (x, v'))\} \vdash L \rightsquigarrow v$ sia derivabile, e dunque, sfruttando le altre ipotesi della precedente derivazione:

$$\frac{E \vdash M \rightsquigarrow v'' \quad \dfrac{E\{(y, v'')\} \vdash N \rightsquigarrow v' \quad E\{(y, v''), (x, v')\} \vdash L \rightsquigarrow v}{E\{(y, v'')\} \vdash \mathit{let}\ x = N\ \mathit{in}\ L \rightsquigarrow v}}{E \vdash \mathit{let}\ y = M\ \mathit{in}\ (\mathit{let}\ x = N\ \mathit{in}\ L) \rightsquigarrow v}$$

Allo stesso modo si dimostra l'implicazione inversa. □

4.1.3 Valutazione *lazy*

Sia M il programma $\mathit{let}\ x = \ N\ \mathit{in}\ 1$, dove:

$$N = \sum_{i = \log e}^{\sqrt{|\pi|^4}} (\log \pi^{i/2} \cdot \int_{\sqrt{2}}^{e} (i \cdot x)!\ dx)$$

Perché calcolare N se il risultato di M sarà comunque 1? Presentiamo ora un approccio *lazy* (pigro) alla valutazione delle espressioni del linguaggio *Exp*, che consiste nel valutare un termine solo quando (e *se*) ce n'è veramente bisogno. In questa nuova semantica, che chiameremo appunto *lazy*, la valutazione di N nel termine $\mathit{let}\ x = N\ \mathit{in}\ M$ viene rimandata fino al momento in cui, *eventualmente*, ad M non serva il valore di x. Per contrasto chiameremo *erger* (sollecita, chiamata anche *greedy*, avida) la semantica introdotta nella sezione 4.1.1.

Domini semantici. Quei termini che l'approccio lazy lascia non valutati fino a che un valore non sia richiesto, come l'N dell'esempio precedente, possono essere conservati in un "ambiente pigro". Estendiamo dunque la nozione di ambiente come segue:

$$Env_L = Var \xrightarrow{fin} Exp$$

Regole. La modifica appena apportata alla nozione di ambiente ci obbliga a riscrivere non solo la regola del *let*, ma anche quella per le variabili:

$$[var]_L \quad \frac{E \vdash M \rightsquigarrow v}{E \vdash x \rightsquigarrow v} \quad (\text{se } E(x) = M)$$

$$[let]_L \quad \frac{E\{(x, M)\} \vdash N \rightsquigarrow v}{E \vdash let\ x = M\ in\ N \rightsquigarrow v}$$

Tutte le altre regole rimangono invariate.

Va notato che, nella semantica *lazy*, non sempre si risparmia "tempo di calcolo". Nel termine $let\ x = N\ in\ (x + x + x)$, l'espressione N viene calcolata tre volte con l'approccio lazy e una sola nell'approccio erger!

(Non) equivalenza tra semantiche. La semantica lazy del linguaggio *Exp* data in questa sezione differisce da quella eager della sezione 4.1.1 solo a livello "implementativo", consentendo a volte di far meno calcoli, o esite forse un qualche termine che produce risultati diversi a seconda di quale semantica usiamo per valutarlo? Eccone ad esempio uno:

$$let\ x = 2\ in\ (let\ y = x\ in\ (let\ x = 3\ in\ y + 1)). \tag{2}$$

Le due semantiche viste finora non sono perciò equivalenti.

Esercizio 4.1 Verificare l'affermazione precedente valutando (2) nell'ambiente vuoto con ciascuna semantica. □

Esercizio 4.2 "*Io sono colui che sono*" (Es 3,14) dice Dio a Mosè sul monte Sinai. Tradotto in SML:

$$let\ x = x\ in\ x.$$

Valutare questo termine con la semantica lazy. □

4.1.4 Valutazione lazy con scoping statico

Cosa porta le regole della sezione (4.1.1) a produrre, nella valutazione del termine (2), un risultato diverso rispetto a quello ottenuto applicando le regole di (4.1.3)? Nella semantica eager viene associato ad y il *valore* che la variabile x ha al momento della dichiarazione "$let\ y = x$", cioè 2, ed è questo che viene usato nella valutazione di $y + 1$. Al contrario, nella semantica lazy, viene associata ad y l'*espressione* x che, al momento della valutazione di $y + 1$, vale 3 in virtù

della dichiarazione *let* $x = 3$. Si tratta perciò di una questione di gestione degli ambienti, che ha poco a che vedere con pigrizia e solerzia nella valutazione. È possibile infatti adottare un approccio lazy nella valutazione di (2) che produca però lo stesso risultato delle regole eager.

Per far ciò è però necessario modificare nuovamente la nozione di ambiente, in modo da potervi accomodare, oltre ad intere espressioni, anche gli ambienti in cui valutarle:

$$Env_{LS} = Var \xrightarrow{fin} (Exp \times Env_{LS}).$$

Regole. Ecco le regole della semantica lazy con *scoping statico* del linguaggio *Exp*. Le distingueremo da quelle presentate nella sezione 4.1.3, che costituiscono la semantica lazy con scoping *dinamico*.

$$[var]_{LS} \quad \frac{E' \vdash M \rightsquigarrow v}{E \vdash x \rightsquigarrow v} \quad (\text{se } E(x) = (M, E'))$$

$$[let]_{LS} \quad \frac{E\{(x, (M, E))\} \vdash N \rightsquigarrow v}{E \vdash let\ x = M\ in\ N \rightsquigarrow v}$$

Tutte le altre regole rimangono invariate.

Esercizio 4.3 Valutare l'espressione (2) con le regole di questa sezione confrontando il risultato con quanto ottenuto nell'esercizio della sezione 4.1.3. □

La differenza fra approccio statico e dinamico al binding delle variabili, che discuteremo nella sezione 4.3.1, emerge qui nel contesto della semantica lazy. Non è possibile produrre qualcosa di analogo all'esempio dell'esercizio 4.3 in un contesto eager per il linguaggio *Exp* (perché ?). Per far ciò dobbiamo aspettare la sezione 4.2, dove introdurremo un linguaggio sintatticamente appena più ricco, ma semanticamente assai più potente.

4.1.5 Semantiche equivalenti

Di nuovo: la versione lazy con scoping statico della semantica di *Exp* differisce da quella data in (4.1.1) solo a livello "implementativo", consentendo a volte all'una di far meno calcoli dell'altra, o esite un qualche termine M che produce 3 se valutato sollecitamente e 4 se pigramente? Per rispondere a questa domanda dobbiamo prima mettere in relazione le due nozioni di ambiente che abbiamo introdotto per le due semantiche.

Equivalenza di ambienti. Notiamo innanzitutto che l'equazione con cui abbiamo definito Env_{LS} nella sezione 4.1.4, ammette in realtà più di una soluzione; cioè, esistono diversi insiemi X tali che $X = Var \xrightarrow{fin} (Exp \times X)$. Ad esempio, esiste una soluzione che contiene ambienti "alti", ovvero *infinitamente annidati*, come ad esempio

$$\overset{\infty}{E} = \{(x, (x, \{(x, (x, \{\ldots\}))\}))\},$$

ed altre che contengono solo ambienti "bassi". La cosa non ci disturba; scegliete voi la soluzione che più vi piace. Attenzione però: se ammettete $\overset{\infty}{E}$ in Env_{LS} allora il tentivo di valutare: $\overset{\infty}{E} \vdash x \rightsquigarrow ?$ vi farà entrare in un loop infinito (nel qual caso diciamo che la valutazione di x *non termina*).

Diciamo che un ambiente eager $E \in Env$ è *equivalente* ad un ambiente lazy $E' \in Env_{LS}$ quando, per ogni variabile x, $E(x) = v$ se e solo se esistono M ed E'' tali che $E'(x) = (M, E'')$ e $E'' \vdash M \rightsquigarrow v$. Nota che ambienti "alti" possono essere equivalenti a bassissimi ambienti eager. Ad esempio, $\overset{\infty}{E}$ è equivalente a $\emptyset$.

□

Proposizione 2 *Sia M un termine di Exp, sia v un valore e siano $E \in Env$ ed $E' \in Env_{LS}$ ambienti equivalenti, con $free(M) \subseteq dom(E)$; il giudizio operazionale $E \vdash M \rightsquigarrow v$ è derivabile nella semantica eager se e solo se $E' \vdash M \rightsquigarrow v$ è derivabile nella semantica lazy con scoping statico.*

Dimostrazione. La dimostrazione procede per induzione sulla struttura di M. Per le costanti, e per termini della forma $M + N$ la proposizione si dimostra vera banalmente. Per una variabile x: sia E un ambiente eager equivalente ad uno lazy E'; se $E \vdash x \rightsquigarrow v$, allora deve essere $E(x) = v$ e dunque, per l'equivalenza degli ambienti, $E'(x) = (N, E'')$ per qualche N ed E'' tali che $E'' \vdash N \rightsquigarrow v$. Usando questo giudizio operazionale come premessa, possiamo applicare la regola $[var]_{LS}$ per concludere $E' \vdash x \rightsquigarrow v$ come richiesto. Il ragionamento inverso (se $E' \vdash x \rightsquigarrow v \ldots$) è del tutto analogo. Supponiamo infine che $E \vdash let\ x = M\ in\ N \rightsquigarrow v$ sia derivabile nella semantica eager, e che E sia equivalente ad E'. La regola $[let]$ richiede che esista un valore v' tale che $E \vdash M \rightsquigarrow v'$. Dunque, essendo M un termine più elementare di $let\ x = M\ in\ N$, si può assumere induttivamente che $E' \vdash M \rightsquigarrow v'$, e dunque che $E'\{(x, (M, E'))\}$ sia equivalente a $E\{(x, v')\}$. La regola $[let]$ richiede inoltre che $E\{(x, v')\} \vdash N \rightsquigarrow v$ sia derivabile. Essendo N un termine più elementare di $let\ x = M\ in\ N$ e per l'equivalenza appena stabilita, possiamo ancora assumere induttivamente che $E'\{(x, (M, E'))\} \vdash N \rightsquigarrow v$, e dunque, applicando $[let]_{LS}$, concludere $E' \vdash let\ x = M\ in\ N \rightsquigarrow v$ come richiesto. Vice versa, supponiamo che $E' \vdash let\ x = M\ in\ N \rightsquigarrow v$ sia derivabile nella semantica lazy, e che E sia equivalente ad E'. La regola $[let]_{LS}$ richiede che $E'\{(x, (M, E'))\} \vdash N \rightsquigarrow v$ sia derivabile. Inoltre, dato che $free(let \ldots) \subseteq dom(E)$, per ipotesi, la Proposizione 1 ci dice che esiste un v' tale che $E \vdash M \rightsquigarrow v'$, e dunque l'equivalenza di E ed E' implica l'equivalenza di $E\{(x, v')\}$ ed $E'\{(x, (M, E'))\}$. Possiamo perciò assumere induttivamente che $E\{(x, v')\} \vdash N \rightsquigarrow v$ e concludere, usando $[let]$ che $E \vdash let\ x = M\ in\ N \rightsquigarrow v$. □

La condizione posta sulle variabili libere di M nella Proposizione 2 serve. Senza di essa, il giudizio operazionale $\emptyset \vdash let\ x = y\ in\ 5 \rightsquigarrow 5$, derivabile nella semantica lazy, violerebbe l'enunciato.

Una conseguenza della Proposizione 2 è che ogni espressione *chiusa* di *Exp*, ovvero senza variabili libere, *termina* nella semantica lazy statica. È possibile

dire di più: ogni espressione, chiusa o no, termina se valutata in un ambiente basso. Dato che le regole di inferenza preservano la bassezza degli ambienti, la non terminazione può essere indotta soltanto "artificialmente" nel contesto della semantica lazy statica iniziando la valutazione in un ambiente alto. Questo non è vero nella semantica *lazy dinamica* (provate a valutare $\emptyset \vdash \mathit{let}\ x = x\ \mathit{in}\ x$ nell'ambiente vuoto), il che pone seri dubbi sulla sensatezza di una tale semantica. Nella Sezione 4.2.1 vedremo che, in presenza di programmi che non terminano, l'equivalenza tra approccio lazy ed approccio eager decade. Il seguente esercizio individua un altro fenomeno computazionale che invalida questa equivalenza: gli *effetti collaterali*.

Esercizio 4.4 Sia Exp^{++} il linguaggio ottenuto estendendo linguaggio *Exp* con un costrutto $x := M$ per l'assegnamento. Come in Java, il risultato della valutazione di $x := M$ consiste nel valore v prodotto dalla valutazione di M, ma ha anche un effetto collaterale: modifica l'ambiente assegnando v alla variabile x. Definire una semantica operazionale lazy ed una eager (entrambe con scoping statico) per Exp^{++} e fornire un termine la cui valutazione produca 3 se valutato sollecitamente e 4 se pigramente. □

4.2 *Fun*: un linguaggio con funzioni

Il linguaggio *Fun* estende *Exp* con la nozione di *funzione*. Dato un termine t con una variabile libera x è possibile costruire, per *astrazione* su x, la funzione ϕ_t che, dato un *valore* v per x, restituisce il risultato della valutazione di $[v/x]t$. Mentre per un matematico dell'Ottocento una tale funzione avrebbe vissuto solo nel metalinguaggio matematico, a partire dal secolo scorso l'operazione di astrazione viene precipitata nel *linguaggio oggetto* e ϕ_t riceve quel riconoscimento di "oggetto matematico" di cui invece ha sempre goduto, ad esempio, il numero 5. In *Fun* ϕ_t si esprime come $\mathit{fn}\, x \Rightarrow t$; in questo termine la variabile x rappresenta il valore d'ingresso della funzione, e prende il nome di *parametro*.

4.2.1 Sintassi e semantica

La sintassi astratta del linguaggio *Fun* è la seguente:

$$\begin{array}{lcl} k & ::= & 5 \,|\, 40 \,|\, \ldots \\ M & ::= & k \,|\, x \,|\, M + N \,|\, \mathit{let}\ x = M \text{ in } N \,|\, \mathit{fn}\ x \Rightarrow M \,|\, M\ N \end{array}$$

L'operatore di sintassi astratta *fn* ha segnatura:

$$\mathit{fn} : \mathit{Var} \times \mathit{Fun} \to \mathit{Fun}$$

Nel termine $\mathit{fn}\,(x, M)$, che scriviamo $\mathit{fn}\ x \Rightarrow M$, ogni occorrenza di x libera nel secondo argomento, M, è legata al primo argomento, il parametro x. Useremo la notazione $\mathit{fn}\ x_1\, x_2 \ldots x_n \Rightarrow M$ per indicare il termine

$$\mathit{fn}\ x_1 \Rightarrow (\mathit{fn}\ x_2 \Rightarrow \ldots (\mathit{fn}\ x_n \Rightarrow M) \ldots).$$

Domini semantici. L'insieme *Val* dei valori non coincide più con l'insieme delle costanti. È necessario infatti introdurre un nuovo tipo di valore per rappresentare *funzioni*. Un tale valore, che chiamiamo *chiusura*, ha la forma (x, M, E), dove x è una variabile, M è un termine ed E è un ambiente, e rappresenta la funzione che mappa un valore v nel risultato della valutazione di M nell'ambiente $E\,\{(x, v)\}$.

$$\mathit{Val} = \{5, 40, \dots\} \cup (\,\mathit{Var} \times \mathit{Fun} \times \mathit{Env})$$

Regole. Nella semantica eager con scoping statico del linguaggio *Fun* le regole per lambda astrazione ed applicazione sono le seguenti:

$$[\mathit{fn}] \quad E \vdash \mathit{fn}\ x \Rightarrow M \rightsquigarrow (x, M, E)$$

$$[\mathit{appl}] \quad \frac{E \vdash M \rightsquigarrow (x, M', E') \quad E \vdash N \rightsquigarrow v \quad E'\{(x, v)\} \vdash M' \rightsquigarrow v'}{E \vdash M\,N \rightsquigarrow v'}$$

A queste si aggiungono le regole [*const*], [*var*], [*plus*] e [*let*] della Sezione 4.1.1, che *Fun* eredita da *Exp*. Ecco invece la regola dell'applicazione per una semantica con scoping dinamico:

$$[\mathit{appl}]_d \quad \frac{E \vdash M \rightsquigarrow (x, M') \quad E \vdash N \rightsquigarrow v \quad E\{(x, v)\} \vdash M' \rightsquigarrow v'}{E \vdash M\,N \rightsquigarrow v'}$$

Esercizio 4.5 Prendendo ispirazione da quanto fatto con *Exp*, definire le regole operazionali per ciascuna versione di *Fun* non ancora considerata: eager dinamica, lazy statica e lazy dinamica. □

Esercizio 4.6 Valutare il termine

$$\mathit{let}\ x = 7\ \mathit{in}\ ((\mathit{fn}\ y \Rightarrow (\mathit{let}\ x = 3\ \mathit{in}\ y\ 9))\,(\mathit{fn}\ z \Rightarrow x))$$

con regole di binding statico e dinamico. Cambia qualcosa? □

Esercizio 4.7 Trovare un termine del linguaggio *Fun* la cui valutazione con regole di scoping statico dia risultati diversi se mediante l'approccio eager o lazy. Provate. Se non riuscite . . . leggete oltre! □

Il problema posto dall'esercizio precedente si risolve osservando che è possibile in *Fun* scrivere programmi che non terminano. In particolare, il programma

$$(\mathit{fn}\,x \Rightarrow x\,x)\,(\mathit{fn}\,x \Rightarrow x\,x) \tag{3}$$

richiede, tanto nella semantica eager che in quella lazy, che la sua stessa valutazione avvenga come premessa della regola operazionale che lo valuta, mandando così "in loop" l'esecuzione della semantica. Tanto il sistema dei tipi di

SML quanto quello di Haskell rigettano il programma (3) come non tipabile (torneremo su questo punto nella Sezione III). Ovviamente entrambi i linguaggi consentono di scrivere un'espressione, chiamiamola ω, la cui valutazione non termina. Usando ω è facile distinguere la strategia eager da quella lazy:

$$\mathit{let}\ x = \omega \text{ in } 7$$

produce come risultato 7 in Haskell (scoping statico, valutazione lazy) mentre non termina in SML (scoping statico, valutazione eager).

4.2.2 Il lambda calcolo

Avremmo potuto esprime sintassi e semantica di *Fun* in termini di un linguaggio più elementare, quello del *lambda calcolo*. I suoi termini, chiamati *lambda termini*, sono:

$$M \quad ::= \quad x \,|\, \mathit{fn}\ x \Rightarrow M \,|\, M\ N$$

Del lambda calcolo esistono molte versioni in letteratura. Quella da noi adottata usa una notazione "alla ML" (*fn* invece di λ) ed una semantica "call-by-value" basata sugli ambienti, le cui regole sono: [*var*], [*fn*] ed [*appl*]. [Dire che lambda fu introdotto da Church nel 1936 e usa normalmente una semantica call-by-name.]

La regola [*let*] di *Fun* può essere effettivamente *derivata* da [*fn*] ed [*appl*]. Infatti, [*let*] consente di derivare $E \vdash \mathit{let}\ x \Rightarrow M\ \mathit{in}\ N \rightsquigarrow v$ dalle premesse $E \vdash M \rightsquigarrow v'$ e $E\,\{(x, v')\} \vdash N \rightsquigarrow v$. Ma, *traducendo* il termine $\mathit{let}\ x = M\ \mathit{in}\ N$ come $(\mathit{fn}\ x \Rightarrow N)\,M$, è possibile derivare $E \vdash (\mathit{fn}\ x \Rightarrow N)\,M \rightsquigarrow v$ dalle stesse premesse e dall'*assioma* $E \vdash \mathit{fn}\ x \Rightarrow N \rightsquigarrow (x, N, E)$, usando [*fn*]:

$$\frac{E \vdash \mathit{fn}\ x \Rightarrow N \rightsquigarrow (x, N, E) \quad E \vdash M \rightsquigarrow v' \quad E\,\{(x, v')\} \vdash N \rightsquigarrow v}{E \vdash (\mathit{fn}\ x \Rightarrow N)\,M \rightsquigarrow v}$$

Qualunque programma di *Fun* che non usi numeri e somme pu essere dunque "compilato", ovvero tradotto, nel lambda calcolo. La traduzione risulta inoltre *corretta* perché conforme alle specifiche semantiche del linguaggio. Come vedremo nella prossima sezione, la traduzione può in realtà essere estesa all'intero *Fun*, inclusi numeri (costanti) e somma. Nel seguito di questa sezione vedremo invece che, oltre ai numeri, molte altre strutture dati e costrutti comunemente usati nella programmazione possono essere tradotti nel lambda calcolo, estendendo la traduzione virtualmente ad ogni linguaggio di programmazione. Vediamo di seguito una rappresentazione di *booleani* e *sequenze*, con alcune operazioni e costrutti ad essi associati.

Iniziamo con i booleani:

$$
\begin{array}{lcl}
\mathit{true} & \equiv & \mathit{fn}\ x\, y \Rightarrow x \\
\mathit{false} & \equiv & \mathit{fn}\ x\, y \Rightarrow y \\
\mathit{not} & \equiv & \mathit{fn}\ z\, x\, y \Rightarrow z\, y\, x \\
B\,?\,M\,:\,N & \equiv & B\, M\, N
\end{array}
$$

Le prime due clausole definiscono le costanti *true* e *false* come funzioni a due argomenti, la prima restituisce il primo, la seconda il secondo. Per convincerci che questa rappresentazione abbia senso, abbiamo poi definito l'operazione *not* ed un costrutto $B\,?\,M\,:\,N$ per le *espressioni condizionali* che, come nel linguaggio `C` o in *Java*, restituisce il valore di M o di N a seconda che B sia vero o falso. Intuitivamente, *not* si aspetta come primo argomento un booleano z e restituisce quella funzione (un booleano) che fa lo stesso lavoro di z ma scambiando l'ordine degli argomenti. È dunque facile verificare che il termine *not true* si comporta precisamente come *false*. Ad esempio, $(\mathit{not}\,\mathit{true})\,5\,7$ dà lo stesso risultato di *false* 5 7, ovvero 7. Similmente, *not false* si comporta come *true*. Nella sezione 11 introdurremo una teoria equazionale del lambda calcolo nella quale, a supporto della nostra codifica dei booleani, potranno essere facilmente derivate le seguenti equazioni:

$$
\begin{array}{rcl}
\mathit{not}\ \mathit{true} & = & \mathit{false} \\
\mathit{not}\ \mathit{false} & = & \mathit{true} \\
\mathit{true}\,?\,M\,:\,N & = & M \\
\mathit{false}\,?\,M\,:\,N & = & N
\end{array}
$$

Esercizio* **4.8** Definire gli operatori *and* and *or* sui booleani. □

Quanto alle sequenze, esse possono essere rappresentate nel lambda calcolo come funzioni della forma:

$$\mathit{fn}\ x \Rightarrow x\, M_1\, M_2 \ldots M_n,$$

che indicheremo con la notazione: $\langle M_1,\, M_2, \ldots M_n\rangle$. L'$i$-esima proiezione di una sequenza di lunghezza n data dalla funzione $\mathit{fn}\ z \Rightarrow (z\,(\mathit{fn}\ x_1\, x_2 \ldots x_n \Rightarrow x_i))$, che indichiamo con $P_{n,i}$. Anche qui possibile verificare l'equazione

$$P_{n,i}\,\langle M_1,\, M_2, \ldots M_n\rangle = M_i.$$

Esercizio* **4.9** Scrivere in *Fun* una funzione F che concatena due liste:

$$F\langle M_1,\, M_2, \ldots M_m\rangle\langle N_1,\, N_2, \ldots N_n\rangle = \langle M_1,\, M_2, \ldots M_m,\, N_1,\, N_2, \ldots N_n\rangle.$$

□

4.2.3 I numeri di Church

Vi sono diversi modi di rappresentare i numeri naturali nel lambda calcolo. Quello che presentiamo ora dovuto ad Alonzo Church.

Quello di Church è un mondo fatto di funzioni. È dunque naturale che per lui *due* volesse dire: applicare una funzione x due volte ad un argomento y, ovvero

$$\mathit{fn}\ x\, y \Rightarrow x\,(x\, y).$$

In generale, dunque, indicando con $M^n N$ il termine $M(M(\ldots(M\,N)\ldots))$, in cui M ripetuto n volte, un numero n viene rappresentato con il termine

$$c_n \equiv \mathit{fn}\ x\, y \Rightarrow x^n\, y.$$

Notiamo che Church rappresenta *false* come zero, come in `C`. Imbarazzante. Seguendo comunque l'idea di Church, la funzione successore dovrà ricevere un numero z in ingresso e restituire quel numero che applica x a y una volta in più di z:

$$\mathit{succ} \equiv \mathit{fn}\ z\, x\, y \Rightarrow x\,(z\, x\, y).$$

Analogamente, la somma di z e w è quel numero che applica la funzione x w volte a y e poi passa il risultato a z che applica x altre z volte:

$$\mathit{plus} \equiv \mathit{fn}\ z\, w\, x\, y \Rightarrow z\, x\,(w\, x\, y).$$

Affinché tutto questo abbia un senso dobbiamo convincerci che con questi numeri ed operazioni "di Church" si possa effettivamente *calcolare*, ovvero che l'aritmetica di Church sia *corretta*. In che senso, dunque, $\mathit{plus}\, c_2\, c_5$ "fa" 7? Certamente $\mathit{plus}\, c_2\, c_5$ è sintatticamente molto diverso da c_7. Possiamo tuttavia definire una funzione $\mathit{dechurch}\,(_)$ di interpretazione dei numeri di Church nei numeri naturali tale che

$$\mathit{dechurch}\,(\mathit{plus}\, c_2\, c_5) = \mathit{dechurch}\,(c_7) = 7.$$

Lo facciamo nel linguaggio *Fun* stesso:

$$\mathit{dechurch}\,(M) = M\ (\mathit{fn}\ x \Rightarrow x + 1)\ 0.$$

Possiamo poi usare la semantica operazionale per fare i calcoli: facile verificare che $\mathit{dechurch}\,(\mathit{plus}\, c_2\, c_1) \rightsquigarrow 3$. Più in generale, indicando con $\mathit{church}\,(M)$ la traduzione nell'aritmetica di Church di una espressione numerica M (in *Fun*), otteniamo il seguente risultato di correttezza:

$$\mathit{dechurch}\,(\mathit{church}\,(M)) \rightsquigarrow k \text{ se e solo se } M \rightsquigarrow k.$$

Esercizio 4.10 Qual è la differenza fra $\mathit{dechurch}\,(M)$ e $M\ \mathit{succ}\ c_0$? □

Esercizio* 4.11 Scrivere nel lambda calcolo una funzione che calcola la moltiplicazione di due numeri di Church. □

Scritta la moltiplicazione di numeri di Church, dovrebbe essere immediato scrivere il fattoriale. Non è così, e la difficoltà non sta tanto nei numeri di Church quanto nella funzione fattoriale stessa. Vediamo perché.

La somma di due numeri n ed m si ottiene iterando n volte la funzione successore a partire da m. Analogamente il prodotto si ottiene iterando la somma; l'elevamento a potenza iterando il prodotto. Otteniamo una interessante successione:

$$\begin{array}{ll} \mathit{plus} & \equiv \mathit{fn}\ u, v \Rightarrow u\ \mathit{succ}\ v \\ \mathit{times} & \equiv \mathit{fn}\ u, v \Rightarrow u\ (\mathit{plus}\ v)\ c_0 \\ \mathit{power} & \equiv \mathit{fn}\ u, v \Rightarrow u\ (\mathit{times}\ v)\ c_1 \end{array}$$

Questa semplice strategia non si applica direttamente al calcolo del fattoriale, dove, ad ogni passo dell'iterazione, si applica al risultato del passo precedente *una funzione diversa*: il fattoriale di 7 si ottiene applicando la funzione $\mathit{fn}\ x \Rightarrow 7 * x$ al fattoriale di 6, che si ottiene applicando la funzione $\mathit{fn}\ x \Rightarrow 6 * x$ al fattoriale di 5, e così via. Nella sezione 11.1 affronteremo questo problema nel contesto di una teoria generale dell'iterazione; qui ci limitiamo a mostrare una implementazione della funzione fattoriale ispirata alla tecnica proposta teorico della ricorsione Stephen Kleene per calcolare il predecessore di un numero di Church.

Esercizio* **4.12** Scrivere nel lambda calcolo una funzione *pred* che calcola il predecessore di un numero di Church. (*Suggerimento*: definire un termine S tale che $S\,\langle c_m, c_n\rangle = \langle c_{m+1}, c_m\rangle$. Notare che $c_n\,S\,\langle c_0, c_0\rangle = \langle c_n, c_{n-1}\rangle$.) □

Ecco una implementazione in SML della funzione fattoriale che ricalca il procedimento adottato nella soluzione dell'esercizio 4.12 per il calcolo del predecessore:

$$\begin{array}{ll} \texttt{fun}\ \mathit{fact}\ n = & \texttt{let val}\ \mathit{step} = (\mathit{fn}\ (x, y) \Rightarrow (\mathit{succ}\ x, \mathit{times}\ (\mathit{succ}\ x)\ y))\ \texttt{in} \\ & (\texttt{let val}\ (a, b) = (n\ \mathit{step}\ (c_0, c_1))\ \texttt{in}\ b\ \texttt{end})\ \texttt{end}; \end{array}$$

4.3 Semantica comparata

4.3.1 Statico VS. dinamico

Nella semantica con scoping statico la valutazione del programma

$$(\mathit{fn}\ x \Rightarrow (\mathit{let}\ y = 6\ \text{in}\ (x\ 1)))\ (\mathit{fn}\ z \Rightarrow y)$$

fallisce. In SML, che adotta appunto tale forma di scoping, l'interprete restituisce un messaggio di errore: y è unbound. Lo stesso avviene in *Common Lisp*, anch'esso con scoping statico, dove il programma si traduce come segue:

$$((\texttt{lambda}\ (x)\ (\texttt{let}\ ((y\ 5))\ (\texttt{funcall}\ x\ 1)))(\texttt{lambda}\ (z)\ y)).$$

Se eseguiamo però questo programma in *Emacs Lisp*, che adotta invece regole di scoping dinamico, il programma termina con successo, con risultato 5. Spiegano i manuali che, mentre nello scoping statico, chiamato perciò anche

scoping lessicale, l'associazione fra un nome ed una variabile vale soltanto nel contesto lessicale, ovvero nel testo, della funzione o blocco in cui la variabile è definita (il che impedisce alla variabile y, definita in $fn\ x \Rightarrow (let\ y \ldots)$, di essere riferita nel corpo della funzione $fn\ z \Rightarrow y$), nello scoping dinamico tale associazione vale invece finché la funzione o blocco sono in esecuzione (il che consente il riferimento a y, perché avviene mentre il corpo del *let* è ancora in esecuzione). Il nostro esempio mostra come lo scoping dinamico consenta di scrivere codice in cui compaiono identificatori indefiniti, che potranno riferirsi ad entità eventualmente diverse a seconda del contesto locale in cui il codice viene eseguito. Nel nostro esempio la funzione $fn\ z \Rightarrow y$ usa un nome y che, a tempo di esecuzione, verrà associato alla variabile locale di una seconda funzione, $fn\ x \Rightarrow (\ldots)$, cui la definizione della prima non appartiene. Questo comportamento conferisce flessibilità ai linguaggi con scoping dinamico, e costituisce infatti una caratteristica fondamentale di alcuni linguaggi ad oggetti, fra i quali Java, dove il riferimento ai metodi viene risolto dinamicamente (vedi sezione 6.1.3).

Nello scoping dinamico può avvenire però anche l'opposto di quanto appena descritto: variabili perfettamente definite nel loro contesto lessicale possono uscirne a tempo di esecuzione e risultare indefinite. Consideriamo il seguente esempio:

$[(fn\ x \Rightarrow (fn\ y \Rightarrow x\ y))\ (fn\ z \Rightarrow z)]\ 5$, che in Lisp si scrive:

(`funcall` (`funcall` (`lambda` (x) (`lambda` (y) (`funcall` x y))) (`lambda` (z) z)) 5).

In Common Lisp questo programma fa 5, mentre in Emacs Lisp esso restituisce un errore: x è unbound. Possiamo verificare questo fenomeno con la nostra semantica operazionale: valutando il termine $(fn\ x \Rightarrow (fn\ y \Rightarrow x\ y))\ (fn\ z \Rightarrow z)$ con le regole dello scoping dinamico vediamo che x sfugge al suo scope:

$$\emptyset \vdash (fn\ x \Rightarrow (fn\ y \Rightarrow x\ y))\ (fn\ z \Rightarrow z) \rightsquigarrow (y, x\ y),$$

otteniamo cioè un risultato che presenta una variabile libera, x, che non compare nell'ambiente (vuoto) in cui avviene valutazione. Teoria e pratica sono dunque in accordo.

Secondo l'autorevole opinione di John Reynolds, il binding dinamico sarebbe oggi "largamente ed inequivocabilmente considerato un errore di design" [Rey98, 11.7]. A motivare una bocciatura tanto perentoria è la violazione, nel bibding dinamico, del principio secondo cui *renaming should preserve meaning*, ovvero dell'alfa-equivalenza (vedi sezione 4.1.1). Il programma

$[fn\ x \Rightarrow ((fn\ y \Rightarrow x\ 3)\ 5)]\ (fn\ z \Rightarrow y)$, che in Lisp si scrive:

(`lambda` (x) (`funcall` (`lambda` (y) (`funcall` x 3)) 5)) (`lambda` (z) y)),

produce un errore in SML e Common Lisp (y unbound), mentre restituisce 5 in Emacs Lisp. Fin qui nulla di nuovo rispetto all'esempio introduttivo di questa sezione. Notando però che (`lambda` (y) (`funcall` x 3)), ovvero $fn\ y \Rightarrow x\ 3$, è alfa-equivalente a (`lambda` (w) (`funcall` x 3)), proviamo a sostituire il primo

termine con il secondo nel nostro programma. Ora anche Emacs Lisp si arrabbia e ci dice che y è unbound.

Un altro argomento spesso addotto a sfavore dello scoping dinamico è la violazione da parte di quest'ultimo della *referential transparency*. La nozione di referential transparency è stata introdotta dal filosofo americano Willard Van Orman Quine in [Qui64] e ripresa nel contesto della computer science da Christopher Strachey e Peter Landin. In [Str67], Strachey ne dà la sequente definizione:

> "... if we wish to find the value of an expression which contains a sub-expression, the only thing we need to know about the subexpression is its value."

Questa formulazione della referential transparency viene anche chiamata *estensionalità*. Il seguente programma Lisp contiene tre espressioni `let` annidate. Nella seconda la variabile `f` viene legata alla funzione senza parametro `lambda` () x, che restituisce il valore della variabile x.

(`let` ((x 1)) (`let` ((f (`lambda` () x))) (+ (`funcall` f) (`let` ((x 2)) (`funcall` f))))).

La sotto-espressione (`funcall` f), ovvero l'invocazione della funzione f, occorre due volte. Adottando regole di scoping dinamico, le due occorrenze producono risultati diversi, e non perché sia cambiato il *valore* della variabile x (siamo infatti in un contesto funzionale senza assegnamenti) ma perché, in violazione alla referential transparency, ciascuna invocazione fa accesso (usando lo stesso identificatore x) a variabili diverse.

In realtà, quasi nulla nel mondo è referentially transparent e, nonostante sia considerata uno dei principi ispiratori della programmazione funzionale, questa proprietà risulta violata non appena si osservano fenomeni computazionali quali side-effect, non-determinismo o anche, più semplicemente, la non-terminazione. Ad esempio, sia M il termine (3) della sezione 4.2.1; anche nello scoping *statico*, l'espressione *let* $x = M$ in 2 termina felicemente con valore 2 se valutata con semantica lazy. La referential transparency è stata qui violata, perché il sottotermine M non può essere sostituito con il suo valore, non avendone alcuno! All'incirca per lo stesso motivo, allontanandoci per un attimo dalla computer science, il paradosso di Epimenide, *questo enunciato è falso* (anche noto come *paradosso del mentitore*), vìola la referential transparency: se tentiamo di sostituire l'espressione "questo enunciato" con il suo valore, entriamo in una sorta di sostituzione infinita:

> "... "..." è falso" è falso.

Esistono versioni referenzialmente trasparenti di questo pardosso, come vedremo nell'introduzione alla Parte III.

Il binding dinamico è comune a diversi linguaggi progettati nei primi anni '60, quali Snobol, APL e il primo Lisp (il primo interprete, scritto da Steve

Russell, risale al 1960), oltre a dialetti successivi come Logo (1967) e il già citato Emacs Lisp (Richard Stallman, 1985). Questa scelta nel design di Emacs Lisp fu fatta per rendere il text editor Emacs (dove il Lisp è appunto usato per programmare funzionalità ed estensioni) eseguibile su computer con memoria di un solo megabyte: il binding dinamico consente infatti implementazioni estremamente compatte. Altri esempi di binding dinamico: Bash e le macro di TeX. Vi sono poi linguaggi, come il Common Lisp e Perl, che consentono al programmatore di scegliere fra binding statico e dinamico nella definizione di una variabile.

Lo scoping statico è usato in C, nei linguaggi della famiglia ALGOL, come Pascal, Modula2 ed Ada, e nei linguaggi funzionali moderni come Scheme, ML e Haskell.

4.3.2 Eager VS. lazy

L'esempio dell'esercizio 4.2 pone seri dubbi sulla combinazione di semantica lazy e scoping dinamico, perché fa emergere di un fenomeno computazionale come la non terminazione da una semplice definizione di variabile invece che da costrutti di controllo tipicamente previsti allo scopo, come il `while` o la ricorsione. Questa combinazione si ritrova però in natura, in particolare in *S-Plus*, un'implementazione commerciale del linguaggio *S* sviluppato alla fine degli anni '70 presso i Bell Labs, e oggi largamente usato nel calcolo statistico.

5 Il Paradigma Imperativo

5.1 *Imp*: un semplice linguaggio imperativo

Abbiamo visto in *Fun* un meccanismo *eager* per la valutazione dei parametri di una funzione ed un meccanismo *lazy*. Una situazione analoga esiste per le procedure in linguaggi imperativi. Prenderemo dunque in considerazione due modalità eager per la valutazione dei parametri di una procedura, il passaggio per *valore* e quello per *reference*, ed una modalità lazy, il passaggio per *nome*. Per comprendere questi meccanismi abbiamo bisogno della nozione di *locazione* (una astrazione sulla nozione di indirizzo di memoria), che introduciamo in questa sezione studiando un linguaggio imperativo, *Imp*, fatto di assegnamenti e semplici costrutti di controllo. Nella prossima sezione vedremo una estensione di *Imp* con procedure ed array, *All* (*Algol-like language*), che può essere considerato come il nucleo dell'Algol.

5.1.1 Sintassi e semantica

Come nelle precedenti sezioni, $x, y \in Var$ e $k \in Val$ indicano generiche variabili e costanti. Le metavariabili M ed N variano ora sull'insieme *Exp* delle espressioni del linguaggio, un sottoinsieme di quelle trattate nella Sezione 4.1. Inoltre, $p, q \in Imp$ indicano generici programmi. *Val*, *Exp* ed *Imp* sono definiti induttivamente dalla seguente sintassi astratta:

$$
\begin{array}{lcl}
k & ::= & 0 \,|\, 1 \,|\, \ldots \,|\, true \,|\, false \\
M & ::= & k \,|\, x \,|\, M + N \,|\, M < N \\
p & ::= & skip \,|\, p; q \,|\, if\ M\ then\ p\ else\ q \,|\, while\ M\ do\ p \,| \\
 & & var\ x = M\ in\ p \,|\, x := M
\end{array}
$$

Intuitivamente, *skip* è il programma che non fa niente, *if-then-else* e *while* sono i ben noti costrutti di controllo, *var* rappresenta la dichiarazione (ed inizializzazione) di una nuova variabile e := rappresenta l'assegnamento di un valore ad una variabile già definita.

Domini semantici. L'insieme *Val* dei valori è come in *Exp*. Assumiamo che sia dato un insieme numerabile *Loc* di locazioni e che le *meta*variabili l_1, $l_2 \ldots$ varino su tale insieme. L'insieme *Env* degli *ambienti* e *Store* delle *memorie* sono definiti come segue:

$$
\begin{array}{lcl}
Env & = & Var \xrightarrow{fin} Loc \\
Store & = & Loc \xrightarrow{fin} Val
\end{array}
$$

sui quali varieranno le metavariabili $E \in Env$ ed $S \in Store$. La composizione di memorie è definita in modo analogo agli ambienti.

Regole. La semantica operazionale di *Imp* definisce *due* relazioni di valutazione: una per le espressioni M, che producono un valore senza avere side-effects, ed una per i programmi p, che non producono valori ma cambiano la memoria:

$$\overset{M}{\rightsquigarrow} \subseteq \mathit{Env} \times \mathit{Exp} \times \mathit{Store} \times \mathit{Val}$$
$$\overset{p}{\rightsquigarrow} \subseteq \mathit{Env} \times \mathit{Imp} \times \mathit{Store} \times \mathit{Store}$$

In genere scriveremo queste relazioni omettendo gli indici M e p: il contesto chiarirà quale è intesa. Le regole sono:

$$[\mathit{const}] \quad E \vdash k,\ S \rightsquigarrow k$$

$$[x] \quad E \vdash x,\ S \rightsquigarrow v \quad (\text{se } E(x) = l \text{ ed } S(l) = v)$$

$$[\mathit{plus}] \quad \frac{E \vdash M, S \rightsquigarrow v_1 \quad E \vdash N, S \rightsquigarrow v_2}{E \vdash M + N, S \rightsquigarrow v} \quad (\text{se } v_1 + v_2 = v)$$

$$[\mathit{min}\,1] \quad \frac{E \vdash M, S \rightsquigarrow v_1 \quad E \vdash N, S \rightsquigarrow v_2}{E \vdash M < N, S \rightsquigarrow \mathit{true}} \quad (\text{se } v_1 < v_2)$$

$$[\mathit{min}\,2] \quad \frac{E \vdash M, S \rightsquigarrow v_1 \quad E \vdash N, S \rightsquigarrow v_2}{E \vdash M < N, S \rightsquigarrow \mathit{false}} \quad (\text{se } v_1 \geq v_2)$$

$$[\mathit{skip}] \quad E \vdash \mathit{skip},\ S \rightsquigarrow S$$

$$[\mathit{seq}] \quad \frac{E \vdash p, S \rightsquigarrow S' \quad E \vdash q, S' \rightsquigarrow S''}{E \vdash p; q, S \rightsquigarrow S''}$$

$$[\mathit{if}\,1] \quad \frac{E \vdash M, S \rightsquigarrow \mathit{true} \quad E \vdash p, S \rightsquigarrow S'}{E \vdash \mathit{if}\ M\ \mathit{then}\ p\ \mathit{else}\ q, S \rightsquigarrow S'}$$

$$[\mathit{if}\,2] \quad \frac{E \vdash M, S \rightsquigarrow \mathit{false} \quad E \vdash q, S \rightsquigarrow S'}{E \vdash \mathit{if}\ M\ \mathit{then}\ p\ \mathit{else}\ q, S \rightsquigarrow S'}$$

$$[\mathit{while}\,1] \quad \frac{E \vdash M, S \rightsquigarrow \mathit{true} \quad E \vdash p, S \rightsquigarrow S' \quad E \vdash \mathit{while}\ M\ \mathit{do}\ p, S' \rightsquigarrow S''}{E \vdash \mathit{while}\ M\ \mathit{do}\ p, S \rightsquigarrow S''}$$

$$[\mathit{while}\,2] \quad \frac{E \vdash M, S \rightsquigarrow \mathit{false}}{E \vdash \mathit{while}\ M\ \mathit{do}\ p, S \rightsquigarrow S}$$

$$[\mathit{var}] \quad \frac{E \vdash M, S \rightsquigarrow v \quad E\,\{(x, l)\} \vdash p, S\,\{(l, v)\} \rightsquigarrow S'}{E \vdash \mathit{var}\ x = M\ \mathit{in}\ p, S \rightsquigarrow S'} \quad (\text{dove } l \text{ è nuova})$$

$$[\mathit{assign}] \quad \frac{E \vdash M, S \rightsquigarrow v}{E \vdash x := M, S \rightsquigarrow S\,\{(l, v)\}} \quad (\text{se } l = E(x))$$

[...Spiegare!]

Esercizio. Usando la semantica operazionale, definire una relazione di equivalenza $\cong$ sul linguaggio tale che l'insieme quoziente $Imp/_{\cong}$ sia un monoide rispetto a *skip* e composizione sequenziale (;).

5.2 *All*: un linguaggio con procedure

Presentiamo ora il linguaggio *All*, che estende *Imp* con array e procedure, e fornisce un semplice impianto linguistico per studiare alcuni meccanismi di passaggio dei parametri che si ritrovano in molti linguaggi della famiglia *Algol* (del secondogenito della famiglia, classe 1960, C.A.R. Hoare ha detto: "ecco un linguaggio così avanti rispetto al suo tempo che non solo migliora i suoi predecessori, ma anche quasi tutti i suoi successori"). Possiamo in effetti considerare *All* come il nucleo di un linguaggio *Algol-like*.

5.2.1 Sintassi

Il linguaggio *Exp* delle espressioni di *All* estende quello di *Imp* con un costrutto sintattico per indicizzare array. Risulterà inoltre conveniente distinguere tra le espressioni quelle "assegnabili", ovvero quelle che sono ammesse alla sinistra di un assegnamento. Indicheremo con *LExp* (per *l*eft *exp*ressions) il loro insieme ed useremo la metavariabile V per indicarne un generico elemento. Gli array vengono creati ed inizializzati dal costrutto $arr\ x = [M_1, \ldots M_n]\ in\ p$. Per semplificare la trattazione non ammettiamo array "senza nome", né perciò espressioni del tipo $[88, 66, 44](2)$.

$$
\begin{array}{lcl}
k & ::= & 5 \mid 40 \mid \ldots \mid true \mid false \\
V & ::= & x \mid x\,[M] \\
M & ::= & k \mid V \mid M + N \mid M < N \\
p & ::= & skip \mid p; q \mid if\ M\ then\ p\ else\ q \mid while\ M\ do\ p \mid \\
 & & var\ x = M\ in\ p \mid arr\ x = [M_0, \ldots M_n]\ in\ p \mid V := M \mid \\
 & & proc\ y(x)\ is\ p\ in\ q \mid call\ y(M)
\end{array}
$$

Oltre all'introduzione di array e procedure, va notato che, in *All*, l'assegnamento assume una forma più generale che in *Imp*, consentendo di operare anche su *elementi* di array.

5.2.2 Semantica call-by-value

Gli ambienti devono fare spazio a due nuovi tipi di oggetto. Le variabili dichiarate come array saranno associate nell'ambiente a sequenze finite e non vuote di locazioni. Indichiamo con Loc^+ l'insieme di tali sequenze. Nota che la dichiarazione di una singola variabile x mediante il costrutto *var* assocerà ora ad x una sequenza $\langle l \rangle$ di lunghezza 1 che, per continuità con *Imp*, scriveremo omettendo le parentesi. Le variabili dichiarate come procedure saranno associate a

chiusure, in modo del tutto analogo a quanto accade in *Fun*. Dunque, mentre l'insieme *Val* dei valori e quello *Store* delle memorie rimangono invariati rispetto ad *Imp*, per gli ambienti si ha:

$$Env = Var \xrightarrow{fin} Loc^{+} \cup (Var \times All \times Env)$$

Introduciamo una nuova relazione di valutazione, specifica per le espressioni assegnabili (ripetiamo le altre per chiarezza):

$$\begin{array}{l} \overset{V}{\rightsquigarrow} \subseteq Env \times LExp \times Store \times Loc \\ \overset{M}{\rightsquigarrow} \subseteq Env \times Exp \times Store \times Val \\ \overset{p}{\rightsquigarrow} \subseteq Env \times All \times Store \times Store \end{array}$$

Le regole [*const*], [*plus*], [*min* 1], [*min* 2], [*skip*], [*seq*], [*if* 1], [*if* 2], [*while* 1] e [*while* 2] rimangono invariate rispetto ad *Imp*. A queste si aggiungono:

$$[loc\,1] \quad E \vdash x,\ S \overset{V}{\rightsquigarrow} l \quad (\text{se } E(x) = l)$$

$$[loc\,2] \quad \frac{E \vdash M, S \overset{M}{\rightsquigarrow} m}{E \vdash x[M], S \overset{V}{\rightsquigarrow} l_m} \quad (\text{se } E(x) = \langle l_0, \ldots, l_n \rangle \text{ e } 0 \leq m \leq n)$$

$$[ref] \quad \frac{E \vdash V, S \overset{V}{\rightsquigarrow} l}{E \vdash V, S \overset{M}{\rightsquigarrow} v} \quad (\text{se } S(l) = v)$$

$$[assign] \quad \frac{E \vdash M, S \overset{M}{\rightsquigarrow} v \quad E \vdash V, S \overset{V}{\rightsquigarrow} l}{E \vdash V := M, S \overset{p}{\rightsquigarrow} S\,\{(l, v)\}}$$

$$[arr] \quad \frac{E \vdash M_0, S \rightsquigarrow v_0 \ \ldots E \vdash M_n, S \rightsquigarrow v_n \quad E\,\{(x, (l_0, \ldots l_n))\} \vdash p, S\,\{(l_i, v_i)\} \rightsquigarrow S'}{E \vdash arr\ x = [M_0, \ldots M_n]\ in\ p, S \rightsquigarrow S'} \quad (*)$$

$$(*) \text{ dove } l_i \text{ è nuova, per } i = 0 \ldots n$$

$$[proc] \quad \frac{E\,\{(y, (x, p, E))\} \vdash q, S \rightsquigarrow S'}{E \vdash proc\ y(x)\ is\ p\ in\ q, S \rightsquigarrow S'}$$

$$[call]_{value} \quad \frac{E \vdash M, S \overset{M}{\rightsquigarrow} v \quad E'\,\{(x, l)\} \vdash p, S\,\{(l, v)\} \rightsquigarrow S'}{E \vdash call\ y(M), S \rightsquigarrow S'} \quad (*)$$

$$(*) \quad \text{se } E(y) = (x, p, E') \text{ e } l \text{ è nuova}$$

Va notata la scomparsa di [*x*]. Per ottenere il *valore* referenziato da una variabile *x* sono necessari ora due passi: [*loc* 1] e [*ref*].

5.2.3 Semantica call-by-reference

Si impone che l'espressione che si passa come argomento ad una chiamata per riferimento deve essere di tipo assegnabile. Ecco dunque la regola della chiamata, che va a sostituire $[call]_{value}$. Tutto il resto rimane come prima.

$$[call]_{reference} \quad \frac{E \vdash V, S \overset{V}{\rightsquigarrow} l \quad E'\{(x,l)\} \vdash p, S \rightsquigarrow S'}{E \vdash call\ y(V), S \rightsquigarrow S'} \quad (\text{se } E(y) = (x, p, E'))$$

Esercizio. Valutare il seguente programma con la semantica call-by-value e con quella call-by-reference. Cosa cambia?

$$var\ x = 5\ in\ proc\ y(z)\ is\ z := 1\ in\ y(x)$$

5.2.4 Semantica call-by-name

Come per la chiamata per riferimento, l'espressione che si passa come argomento ad una chiamata per nome deve essere di tipo assegnabile. Tale espressione deve inoltre rimanere non valutata fino a quando l'esecuzione del corpo della procedura non richieda la valutazione del parametro. E' dunque naturale tenere tale espressione nell'ambiente, come già si era fatto nella semantica lazy di *Exp*. Dobbiamo dunque estendere la nozione di ambiente:

$$Env = Var \xrightarrow{fin} Loc^{+} \cup (Var \times All \times Env) \cup (LExp \times Env)$$

Questa modifica alla nozione di ambiente comporta una ulteriore regola per accedere alla locazione di una variabile che funge da parametro di procedura. Apparte questa e la regola della chiamata, che sostituisce le precedenti, tutto il resto rimane inalterato.

$$[loc\,3] \quad \frac{E' \vdash V, S \overset{V}{\rightsquigarrow} l}{E \vdash x, S \overset{V}{\rightsquigarrow} l} \quad (\text{se } E(x) = (V, E'))$$

$$[call]_{name} \quad \frac{E'\{(x,(V,E))\} \vdash p, S \rightsquigarrow S'}{E \vdash call\ y(V), S \rightsquigarrow S'} \quad (\text{se } E(y) = (x, p, E'))$$

Esercizio. Valutare il seguente programma con la semantica call-by-reference e con quella call-by-name. Cosa cambia?

$$var\ x = 0\ in\ arr\ y = [3,4,5]\ in\ proc\ z(w)\ is\ (x := x+1; u := w)\ in\ call\ z(y[x])$$

6 Il Paradigma ad Oggetti

Introduzione. Citare "A Theory of Objects" [AC96].

6.1 *Mini-Java*

In questa sezione introduciamo un sottoinsieme molto elementare del linguaggio ad oggetti Java, che chiamiamo Mini-Java. Nel linguaggio Mini-Java abbiamo incluso quanto basta per studiare alcuni degli aspetti caratterizzanti della tecnologia ad oggetti e del Java in particolare: la nozione di oggetto, con il riferimento `this` a se stesso e `super` al *super-oggetto*, l'ereditarietà, il binding *dinamico* dei metodi (e statico per gli attributi). Di Mini-Java studiamo tanto la semantica statica (la teoria dei tipi) che quella dinamica.

Giochiamo con statico e dinamico nel contesto ad-oggetti, che combina elementi di programmazione procedurale tipica dei linguaggi imperativi con la capacità di manipolare oggetti di ordine superiore, tipica dei linguaggi funzionali.

Connessioni: tipo di dato astratto e co-induzione. Ovvero un tipo di dato corredato di operazioni (metodi-estrattori possono essere visti come osservatori).

Esempio. Attributi statici, metodi dinamici:

```
class A {
    int f = 1;
    int get_f () {
        return f;
    }
}

class B extends A {
    int f = 2;
    int get_f () {
        return f;
    }
}

class esempio {
    public static void main (String [] args) {
        A b = new B();
        System.out.println(b.f);
        System.out.println(b.get_f());
    }
}
```

Notare che Smalltalk non consente l'accesso diretto agli attributi (come se fossero sempre privati). Per osservarli dobbiamo usare estrattori come `get_f`, e dunque non possiamo giocare su statico e dinamico come in Java.

6.1.1 Sintassi

Usiamo le metavariabili $C, D \in \mathit{Class}$ per indicare nomi di classe, $f \in \mathit{Field}$ per i nomi di attributo, m per nomi di metodo, $x, y \in \mathit{Var}$ per le variabili ed $n \in \texttt{int}$ per gli interi. Se V è una metavariabile, indichiamo con $[V]$ una sequenza, eventualmente vuota, di termini della categoria sintattica associata a V.

$$
\begin{array}{rcl}
T, X, Y & ::= & \texttt{int} \mid C \\
U & ::= & \texttt{class}\ C\ \texttt{extends}\ D\ \{\,[T\ f = e;]\ [M]\,\} \\
M & ::= & T\ m\ (\,[X\ x]\,)\ \{\,[Y\ y]\ [\,s\,]\,\} \\
s & ::= & x = e;\ \mid e.f = d;\ \mid \texttt{return}\ e; \\
e, d & ::= & x \mid n \mid \texttt{this} \mid \texttt{super} \mid \texttt{null} \mid \texttt{new}\ C\,(\,) \mid e.f \mid e.m(\,[d\,]\,)
\end{array}
$$

La dichiarazione U di una classe comprende la clausola `extends`, un insieme di dichiarazioni di attributi con relativa inizializzazione, $T\ f = e$, e di metodi M. Assumiamo che esista una classe predefinita `Object` senza attributi ne metodi. Il corpo di un metodo è costituito da un insieme di dichiarazioni di variabili locali, $Y\ y$, e da un insieme di *comandi*, per i quali usiamo la metavariabile s. Va notato che, in questa versione minimale di Java, gli assegnamenti non sono espressioni ma comandi. Un *programma* è costituito da un insieme di dichiarazioni di classi. Si assume che una di esse contenga un metodo `main`. Non essendo previsti metodi `void`, l'esecuzione del `main` produce un valore, che è in risultato del'intero programma. Ecco un esempio.

Esempio di programma Mini-Java. Quanto fa?

```
class A extends Object {
    int f = this.m();
    int m () {return 3;}
}

class B extends A {
    int g = this.m();
    int m () {return this.g;}
}

class weird extends Object {
    int main () {
        B b = new B();
        return b.f;
    }
}
```

□

6.1.2 Semantica statica

La semantica statica di Mini-Java è data da un insieme di asserzioni che assegnano un tipo alle espressioni *significative* del linguaggio ("$2.f$" non lo è) e controllano che metodi e classi siano ben formati. Un *contesto di tipi* è una sequenza $\Gamma \in (\mathit{Var}^+ \times \mathit{Types})^*$, dove $\mathit{Var}^+ = \mathit{Var} \cup \{\texttt{this}\}$ e $\mathit{Types} = \mathit{Class} \cup \{\texttt{int}, \top\}$ è l'insieme dei tipi, che include $\top$, il tipo dell'espressione `null`. Una virgola indica la concatenazione di contesti. La metavariabile z indica un generico elemento di Var^+. Scriviamo $C \prec D$ se C estende D *direttamente*, ovvero se la dichiarazione di C è `class` C `extends` D $\{\dots\}$. Indichiamo con $\leq$ la chiusura riflessiva e transitiva di $\prec$, aumentata, per ogni classe C, dall'associazione $\top \leq C$.

Nella semantica statica usiamo due funzioni di servizio, *ftype* ed *mtype*, facilmente definibili dalla sintassi astratta del linguaggio. In particolare, $\mathit{ftype}\,(f, C)$ restituisce il tipo dichiarato per il campo f nella classe C. Se f non è definito in C, *ftype* cerca la dichiarazione più vicina risalendo la gerarchia delle classi. Analogamente, $\mathit{mtype}\,(m, C, (T_1, \dots T_n))$ denota il tipo T del risultato del più vicino metodo di nome m i cui parametri siano *compatibili* con $T_1, \dots T_n$, ovvero la cui dichiarazione sia della forma $T\ m\ (T'_1\, x_1, \dots T'_n\, x_n)\ \{\dots\}$ con $T_i \leq T'_i$, per $1 \leq i \leq n$.

Le asserzioni che assegnano un tipo alle espressioni hanno la forma

$$\Gamma \vdash e : T.$$

Il controllo dei tipi per dichiarazioni di metodi e classi avviene mediante asserzioni della forma $C \vdash M\ \mathit{ok}$ e $\vdash U\ \mathit{ok}$. Usiamo la metavariabile a per indicare una generica *left espressione*, ovvero una variabile o una espressione del tipo $e.f$. Le regole della semantica statica sono riportate nella tabella 1.

Abusando leggermente della sintassi atratta, abbiamo semplificato le regole assumendo che il comando `return` sia l'ultimo del corpo di un metodo. Nota inoltre che abbiamo adottato un approccio "permissivo" nel controllo sulle dichiarazioni dei metodi. Java fa un ulteriore controllo: se la dichiarazione di un metodo in una classe C sovrascrive (*overriding*) un metodo dichiarato in una superclasse di C, allora i parametri delle due dichiarazioni devono coincidere.

6.1.3 Semantica dinamica

Le regole della semantica operazionale di Mini-Java sono riportate nella tabella 2. Prima di affrontarle definiamo i domini semantici degli oggetti, dei valori, degli ambienti e delle memorie.

Un *oggetto* è costituito da un insieme di attributi, il valore dei quali può essere modificato da assegnamenti. Dunque, in prima approssimazione, un oggetto può essere rappresentato da un mapping finito di attributi in locazioni. Dato però

$$\Gamma_1, z : T, \Gamma_2 \vdash z : T$$

$$\Gamma_1, \texttt{this} : C, \Gamma_2 \vdash \texttt{super} : D \quad (C \prec D)$$

$$\Gamma \vdash \texttt{null} : \top$$

$$\Gamma \vdash n : \texttt{int}$$

$$\Gamma \vdash \texttt{new}\ C\,(\,) : C$$

$$\frac{\Gamma \vdash e : C}{\Gamma \vdash e.f : T} \quad (\mathit{ftype}\,(f, C) = T)$$

$$\frac{\Gamma \vdash e : C \quad \Gamma \vdash e_i : T_i}{\Gamma \vdash e.m(e_1, \ldots e_n) : T} \quad (\mathit{mtype}\,(m, C, (T_1, \ldots T_n)) = T)$$

$$\frac{\Gamma \vdash a_i : T_i \quad \Gamma \vdash e_i : T'_i \quad \Gamma \vdash d : T'}{C \vdash T\ m\ (\,[X\,x]\,)\ \{\,[Y\,y]\,[a = e;\,]\ \texttt{return}\ d\,;\,\}} \quad (*)$$

$$(*) = (T'_i \leq T_i,\ T' \leq T \text{ and } \Gamma = \{(x_i, X_i), (y_i, Y_i), (\text{this}, C)\}$$

$$\frac{\{(\text{this}, C)\} \vdash e_i : T'_i \quad \Gamma \vdash M_i\ \mathit{ok}}{\vdash \texttt{class}\ C\ \texttt{extends}\ D\ \{\,[\,T\,f = e;\,]\,[M]\,\}\ \mathit{ok}} \quad (T'_i \leq T_i)$$

Tabella 1: Semantica statica di Mini-Java

che un oggetto possiede anche gli attributi (eventualmente omonimi) dei suoi avi, per individuare un attributo dobbiamo specificarne, oltre al nome, anche la classe dell'avo cui l'attributo si riferisce. Inoltre, dato che, come vedremo, l'informazione sul tipo di un oggetto è significativa *a tempo di esecuzione*, aggiungiamo anche questa informazione alla rappresentazione operazionale di un oggetto. Dunque, indicando con *Loc* l'insieme delle locazioni, l'insieme *Obj* degli oggetti è definito come segue:

$$o \in Obj = Class \times (Class \times Field \xrightarrow{fin} Loc).$$

Notazione. Se $o = (C, F)$ è un oggetto, dove C è la sua classe, e $C \leq C'$, denotiamo con $o|_{C'}$ la coercizione di o ad elemento di C', ovvero: $o|_{C'} = (C', F')$, dove $F'(D, f) = F(D, f)$ se $D \geq C'$, altrimenti $F'(D, f)$ è indefinito. Scriviamo inoltre $o(D, f)$ intendendo la locazione $F(D', f)$, dove $D' \geq D$ è la prima classe contenente una dichiarazione di f incontrata risalendo a partire da D la gerarchia delle classi. □

Un oggetto è un *valore*, ovvero il possibile risultato della valutazione di un'espressione. Allo stesso modo, sono valori gli interi e `null`, il riferimento "a nessun oggetto" (nota: in Java `null` non ha tipo `Object`, anzi... non ha tipo tout court!). Introduciamo inoltre un valore fittizio, che indichiamo con $*$, che rappresenta il risultato della valutazione di un blocco di comandi quando non vi compare un comando `return`. A partire dai valori, definiamo inoltre il dominio semantico degli *ambienti* e degli *heap*:

$$\begin{aligned} v \in Val &= \texttt{int} \cup Obj \cup \{\texttt{null}, *\}, \\ E \in Env &= Var^{+} \xrightarrow{fin} Val, \\ S \in Heap &= Loc \xrightarrow{fin} Val. \end{aligned}$$

Regole. Durante l' esecuzione di un programma, l'interprete Java usa l'informazione sui tipi di espressioni ed oggetti inferita dalla semantica statica (vedi, ad esempio, la regola per il riferimento ad un attributo). Assumiamo dunque che Mini-Java venga "compilato" in una struttura sintattica dove ad ogni espressione sia associato il tipo dedotto applicando le regole della Sezione 6.1.2. Indichiamo con $e : T$ una espressione arricchita dal suo tipo. Per semplificare la notazione, tuttavia, rappresentiamo esplicitamente il tipo solo quando esso gioca un ruolo significativo nell'applicazione delle regole.

Assumiamo inoltre che la compilazione traduca le dichiarazioni di attributo che compaiono nel corpo di una classe ($[T\ f = e;]$) in una sequenza di comandi ($[\text{this}.f = e;]$), in modo che tanto la creazione di un nuovo oggetto (quando vengono appunto inizializzati gli attributi) che l'esecuzione del corpo di un metodo possano avvenire usando le stesse regole operazionali. In particolare, la funzione *cbody*, che possiamo pensare come prodotta dal compilatore, associa a ciascun nome di classe la sequenza dei comandi di inizializzazione degli attributi che vi sono definiti.

La funzione *mbody* si occupa del *dispatching* dinamico dei metodi. In particolare, se una classe D contiene una dichiarazione $T\ m\ (\,[X\ x]\,)\ \{\,[Y\ y]\ [s]\,\}$ con parametri $[X\ x]$ *compatibili* (vedi Sezione 6.1.2) con una tupla $t = (T_1, \ldots T_n)$ di tipi, se inoltre $C \leq D$ e nessun'altra classe D' diversa da D e tale che $C \leq D' \leq D$ contiene la dichiarazione di un metodo m anch'esso con parametri compatibili con t, allora $mbody\,(C, m, t) = ([x], [s])$. La coppia $([x], [s])$ è qualcosa di analogo a ciò che nel Capitolo 4.2.1 abbiamo chiamato *chiusura* di una funzione. Va notato che, dopo il controllo dei tipi, non rimane traccia nella chiusura di un metodo delle dichiarazioni di variabile locale. Queste verranno inserite nell'ambiente da un eventuale assegnamento (vedi la regola per l'assegnamento), mentre un riferimento ad una variabile non precedentemente assegnata genererà un errore a tempo di esecuzione (ovvero il fallimento della semantica operazionale).

In Java la costruzione di un oggetto di classe C mediante l'invocazione di un costruttore avviene in quattro passi successivi:

1. viene allocata memoria per gli attributi dell'oggetto; ciascun attributo viene inizializzato con il *valore di default* associato al suo tipo (0 per gli interi, `null` per gli oggetti...);

2. viene invocato il costruttore *del padre* di C:

3. a ciascun attributo f dichiarato in C, con una dichiarazione $T\ f = e$, viene assegnato il risultato della valutazione di e;

4. viene eseguito il corpo del costruttore di C.

In Mini-Java l'espressione `new` $C\,(\,)$ costruisce un nuovo oggetto della classe C e lo restituisce come risultato. Questo avviene inizialmente con l'invocazione della funzione *alloc*, che è responsabile del passo 1. In particolare, dato un oggetto o ed una classe C, *alloc* associa tanti attributi ad o quanti sono gli attributi dichiarati nella classe C ed inizializza il loro valore in memoria con il valore di default di ciascun attributo. Più formalmente, $alloc\,(o, C)$ è una coppia (o', S), dove o' è l'estensione di o con gli attributi di C e valori nello heap S, ovvero: $o'\,(D, f) = l$ se e solo se $o\,(D, f) = l$ (ovvero (D, f) individua un attributo già presente in o), oppure $C = D$, l'attributo f è dichiarato in C con tipo T, l è una nuova locazione ed $S(l)$ è il valore di default del tipo T.

Il secondo passo nella creazione di un oggetto è l'invocazione del costruttore del padre. Questo, a sua volta, invocherà la *alloc*, che aggiungerà all'oggetto gli attributi del padre, e così via fino a raggiungere la classe `Object` (per la creazione dei cui oggetti esiste una regola operazionale specifica). Affinchè un costruttore possa aggiungere attributi ad un oggetto in costruzione è dunque necessario che questo gli venga passato come argomento. Nelle regole operazionali questo avviene associando al termine `new` sotto forma di indice l'oggetto in costruzione. In particolare, assumiamo che il compilatore traduca i termini della forma `new` $C\,(\,)$ in $\texttt{new}_o\ C\,(\,)$, dove $o = (C, \emptyset)$ è l'oggetto vuoto (ovvero inizialmente senza attrubuti) della classe C.

La seconda premessa della regola per `new` esegue i comandi di inizializzazione dei campi valutando l'espressione associata a ciascun campo nella sintassi (facoltativa in Java, obbligatoria per noi). Non avendo incluso dichiarazioni di costruttori nella sintassi di Mini-Java, il quarto passo nella creazione di un oggetto non compare nelle regole operazionali.

Le ultime quattro regole operazionali consentono di valutare blocchi di comandi: due per gli assegnamenti (ad attributi ed a variabili locali), una per il comando `return` ed una per il blocco vuoto ϵ, la cui valutazione produce il valore speciale $*$.

Esercizio. Scrivere in Mini-Java un programma la cui valutazione dimostri che il passaggio dei parametri avviene per valore. □

$$E \vdash z, S \rightsquigarrow v, S \quad (E(z) = v)$$

$$E \vdash \texttt{super} : C, S \rightsquigarrow o|_C, S \quad (E(\texttt{this}) = o)$$

$$E \vdash \texttt{null}, S \rightsquigarrow \texttt{null}, S$$

$$E \vdash n, S \rightsquigarrow n, S$$

$$E \vdash \texttt{new}_o\ \texttt{Object}\,(\,), S \rightsquigarrow o, S$$

$$\frac{\emptyset \vdash \texttt{new}_{o'}\, D\,(\,), S\, S' \rightsquigarrow o'', S'' \quad \{(\texttt{this}, o'')\} \vdash [\,s\,], S'' \rightsquigarrow *, S'''}{E \vdash \texttt{new}_o\, C\,(\,), S \rightsquigarrow o'', S'''} \quad (\star)$$

$$(\star) = (\mathit{alloc}\,(C, o) = (o', S'), C \prec D \text{ and } \mathit{cbody}\,(C) = [\,s\,])$$

$$\frac{E \vdash e, S \rightsquigarrow o, S'}{E \vdash (e : C).f, S \rightsquigarrow v, S'} \quad (o(C, f) = l \text{ and } S'(l) = v)$$

$$\frac{E \vdash e, S \rightsquigarrow (C, F), S' \quad E \vdash e_i, S_{i-1} \rightsquigarrow v_i, S_i \quad E' \vdash [s], S_n \rightsquigarrow v, S''}{E \vdash e.m\,(e_1 : T_1, \ldots e_n : T_n), S \rightsquigarrow v, S''} \quad (\star\star)$$

$$(\star\star) = (\mathit{mbody}\,(C, m, (T_1, \ldots T_n)) = ([x], [s]), S_0 = S' \text{ and } E' = \{(x_i, v_i)\})$$

$$\frac{E \vdash e, S \rightsquigarrow o, S' \quad E \vdash d, S' \rightsquigarrow v, S'' \quad E \vdash [\,s\,], S''\, \{(l, v)\} \rightsquigarrow v', S'''}{E \vdash (e : C).f = d\,; [\,s\,], S \rightsquigarrow v', S'''} \quad (\star\star\star)$$

$$(\star\star\star) = (o\,(C, f) = l)$$

$$\frac{E \vdash e, S \rightsquigarrow v, S' \quad E\, \{(x, v)\} \vdash [\,s\,], S' \rightsquigarrow v', S''}{E \vdash x = e; [\,s\,], S \rightsquigarrow v', S''}$$

$$\frac{E \vdash e, S \rightsquigarrow v, S'}{E \vdash \texttt{return}\ e; [\,s\,], S \rightsquigarrow v, S'}$$

$$E \vdash \epsilon, S \rightsquigarrow *, S$$

Tabella 2: Semantica dinamica di Mini-Java

Parte III
Sistemi dei Tipi

Nei primi anni del '900 uno sciame sismico di paradossi investì le fondazioni della matematica. La scossa più famosa fu quella del 1901, quando Bertrand Russell propose l'antinomia, nella teoria degli insiemi (naive) di Cantor, di un insieme R che contiene tutti gli insiemi che non sono elementi di se stessi:

$$R = \{X \ : \ X \notin X\}.$$

È facile convincersi che, per questo insieme, $R \in R$ vale se e solo se $R \notin R$. Il problema, in definizioni come quella di R appena data, è che non basta enunciare una proprietà $P(X)$ (nel caso specifico $X \notin X$) affinche l'insieme $\{X \ : \ P(X)\}$ di tutte le "cose" X che la soddisfano esista; dobbiamo anche dire da dove queste cose intendiamo andarle a pescare. Questo è ciò che prescrive la teoria *assiomatica* degli insiemi: dato un insieme A, ed una proprietà $P(X)$ esiste di un insieme

$$\{X \in A \ : \ P(X)\}$$

che contiene esattamente gli elementi di A che soddisfano P. Questo viene chiamato *assioma di comprensione ristretta* (restricted set comprehension) o *assioma di specificazione*. Nota che l'antinomia di Russel scompare usando la versione ristretta dell'assioma di comprensione: dell'insieme $\{X \in A \ : \ X \notin X\}$ possiamo semplicemente dire che esso non appartiene ad A senza incappare in contraddizione alcuna. Inoltre, dato che questo vale comunque si scelga A, se ne conclude che, dato un qualunque insieme, c'è sempre qualcosa che non vi appartiene, ovvero che *che l'universo non esiste*!

Al di là delle conclusioni metafisiche che se ne possono trarre, il paradosso di Russel ammonisce, nel fondare teorie matematiche, ad essere espliciti sulla natura delle cose di cui si parla. Questo vale naturalmente anche per *operare* sulle cose: come sanno bene i fisici, non si possono sommare due metri e tre ampere, cosí come non si possono uguagliare, in una equazione, una forza e una velocità. Quelle che in fisica si chiamano *dimensioni*, in informatica si chiamano *tipi*.

I programmi operano sui dati e producono (a volte) risultati. A volte, tipocamente nei linguaggi funzionali, dati e risultati possono essere a loro volta programmi. Consideriamo ad esempio il lambda calcolo introdotto nella sezione 4.2.2 e chiamiamo programmi i suoi termini chiusi, ovvero quelli senza variabili libere. Nello spirito della semantica denotazionale e memori dell'ingenuità appena commessa nella definizione dell'insieme R, ci chiediamo se, dato un programma M, esista effettivamente un oggetto matematico, indichiamolo con $[\![M]\!]$, che ne rappresenti astrattamente il significato. Supponiamo di andare a pescare tale significato in un insieme A_0 di oggetti matematici. Dovendo poi dar conto di termini del tipo $M\,N$, dovremo convenire che l'oggetto denotato da $[\![M]\!]$ debba essere una sorta di funzione che, pescando i suoi input da un insieme

A_1 di oggetti matematici, cui eventualmente appartiene $[\![N]\!]$, restituisca oggetti in un qualche insieme A_2, cui eventualmente appartiene $[\![M\ N]\!]$.

Un *sistema dei tipi* per un linguaggio di programmazione è un insieme di regole che consentono di dare un tipo ad espressioni, comandi ed altri costrutti del linguaggio. Un linguaggio si dice *tipato* se per esso è definito un tale sistema; altrimenti si dice *non tipato*. Il processo che porta alla determinazione di un tipo per i termini di un linguaggio si chiama *controllo dei tipi* (*type checking*).

Un linguaggio si dice *fortemente tipato* se il tipo di tutte le variabili è determinato a tempo di compilazione, altrimenti si dice *dinamicamente tipato* (se comunque esiste una nozione *non triviale* di tipo) o *non tipato*. Tra i inguaggi del primo genere ci sono il Fortran, l'ML, il Pascal, il Modula 2 e l'Ada; tra quelli del secondo lo Snobol e l'APL, mentre il LISP appartiene alla terza categoria. In un linguaggio fortemente tipato lo spazio di memoria richiesto per contenere il valore di ciascuna variabile durante l'esecuzione è completamente determinato a tempo di compilazione. In Pascal l'informazione sui tipi esiste solo a tempo di compilazione e scompare dai descrittori a tempo di esecuzione.

Esempio. In APL l'*esecuzione* dell'assegnamento

```
A <- 5
```

definisce `A` come una variabile intera (non vi sono dichiarazioni esplicite). Successivamente ad `A` potrebbe venir assegnato un array a due dimensioni. Dunque l'esecuzione di

```
A[2:3] <- 8
```

(assegnamento alla seconda riga, terza colonna di `A`) è corretta solo se *in quel momento* `A` è un tale array. Questo richiede controllo *dinamico* dei tipi. □

Lo scopo di un sistema dei tipi è quello di evitare che durante l'esecuzione di un programma occorrano errori quali, ad esempio, l'applicazione di funzioni ad argomenti inappropriati o il riferimento illegale della memoria. Tale obiettivo può comunque essere raggiunto in un linguaggio non tipato introducendo controlli dinamici (*dynamic checking*). Va notato che anche in un linguaggi con controlo statico dei tipi possono rendersi necessari controlli a tempo di esecuzione. L'esempio tipico a questo proposito è il controllo sugli indici di un array. Quando nessun programma genera errori a tempo di esecuzione il linguaggio si dice *corretto* (*sound*).

Distinguiamo due generi di errore: i cosiddetti errori *trapped*, ovvero quelli che provocano il fallimento della computazione, e gli errori *untrapped*, che non presentano sintomi immediatamente visibili, e sono perciò più insidiosi. Al primo tipo appartengono, ad esempio, la divisione per zero o un accesso proibito alla memoria; al secondo la lettura erronea di una porzione non significativa della memoria. Un linguaggio in cui nessun programma legale può generare errori untrapped si dice *sicuro* (*safe*). Il "C" è il campione dei linguaggi *unsafe*: un

integer overflow, ad esempio, non genera alcuna reazione visibile nell'esecuzione di un programma.

Dato che errori quali il tentativo di accedere ad un array fuori dai suoi limiti non sono catturabili staticamente, spesso l'obiettivo della correttezza completa non sembra ragionevole. Alcuni sistemi dei tipi riescono dunque solo a garantire l'assenza di errori appartenenti ad un insieme "proibito", che include *tutti* gli errori untrapped ed *una parte* dei trapped. Un linguaggio in cui ogni programma legale è libero da errori proibiti si dice *strongly checked*. A questa categoria possono ovviamente appartenere anche linguaggi non tipati. Chiamando *Sound*, *Strongly-Checked* e *Safe* rispettivamente gli insiemi dei linguaggi sound, strongly checked e safe, abbiamo dunque le seguenti inclusioni:

$$\textit{Sound} \subseteq \textit{Strongly-Checked} \subseteq \textit{Safe}.$$

Vi sono linguaggi che non garantiscono nemmeno la sicurezza. Linguaggi tipati ma non sicuri si dicono *weakly checked*, ovvero: a controllo debole. *Pascal* appartiene a questa categoria. Va notato che, dal punto di vista della mantenibilità del codice, un linguaggio tipato, anche se debolmente, è superiore ad uno non tipato, anche se sicuro. La seguente tabella fornisce esempi di linguaggi in varie combinazioni di tipaggio e sicurezza.

	Typed	*Untyped*
Safe	ML	LISP
Unsafe	C	Assembler

La definizione di una *teoria dei tipi*, ovvero un sistema dei tipi dato sotto forma di regole *formali* di deduzione, permette, ove sia definita anche una semantica formale, di dimostrare matematicamente proprietà dinamiche del linguaggio. Considerando "legali" solo quei programmi ai quali è possibile assegnare un tipo nella teoria, il cosiddetto *teorema di correttezza* (attenzione: correttezza del sistema dei tipi, non del linguaggio) afferma che nessun programma legale genera mai errori proibiti. Un altro risultato che mette in relazione la semantica statica con quella dinamica è quello che afferma che il tipo di un termine non cambia durante la valutazione: se t è di tipo T e $t \Rightarrow v$, allora anche v è di tipo T. In genere la dimostrazione avviene per induzione strutturale.

Tipi e replicatori. Un *quine* è un programma che stampa il proprio stesso codice. Il termine è stato introdotto da D. R. Hoftadter [Hof79], dal nome del filosofo già citato nella sezione 4.3.1 che ha proposto una versione referenzialmente trasparente del paradosso di Epimenide [Qui66]:

> "yields falsehood when preceded by its quotation" yields falsehood when preceded by its quotation.

In Lisp, che ha il backquote, è facile scrivere un quine (...o no?! attenzione al "quote"!):

((`lambda` (x) (`list` x (`list` '`quote` x))) '(`lambda` (x) (`list` x (`list` '`quote` x))))

In SML è più complicato. . . è una questione di tipi?

7 I Tipi nei Linguaggi Funzionali

I *tipi* compaiono nella computer science con i primi linguaggi "ad alto livello"; il *Fortran* ad esempio, che è del 1957, ha booleani, interi, reali... Non è tuttavia che con i linguaggi funzionali degli anni '70 (*ML*, *Miranda*...) che la teoria dei tipi acquista ...

La teoria dei tipi origina dalla logica, dove i tipi furono introdotti nei primi del 1900 per ovviare ai paradossi che proliferavano in quegli anni, quando i barbieri radevano tutti coloro che non radono se stessi (*chi rade il barbiere?*).

Nei linguaggi funzionali i *tipi* possono essere visti *specifiche* di programmi nello stesso senso in cui lo sono le asserzioni in una tripla di Hoare.

7.1 Il sistema $F1$

In letteratura viene spesso chiamato *lambda calcolo tipato*. È un sistema del *prim'ordine*, ovvero senza parametrizzazione ed astrazione sui tipi. Tale è il sistema dei tipi di *Pascal* e *Algol68*. Va notato che, questi linguaggi consentono di passare funzioni e procedure come argomenti a sottoprogrammi. Funzioni che accettano funzioni come argomento vengono dette *di ordine superiore*. Questa nozione non va confusa con quella di *tipo* di ordine superiore, che affronteremo nella prossima sezione.

Sintassi. Usiamo la metavariabile K per indicare un generico tipo costante (per esempio *int*, *bool*...), A, B... per indicare elementi di *Types*, l'insieme dei tipi del linguaggio, M, N... per indicare elementi di *Terms*, i termini *grezzi*, k per indicare una generica costante di tipo K, ed x per indicare un generico elemento dell'insieme numerabile *Var* delle variabili del linguaggio.

$$\begin{array}{rcl} \mathit{Types} & ::= & K \mid A \to B \\ \mathit{Terms} & ::= & k \mid x \mid \lambda\, x : A \,.\, M \mid M\, N \end{array}$$

Gli elementi di *Terms* non sono ancora termini legali di $F1$, perché non necessariamente tipabili nella teoria, da cui l'aggettivo "grezzi". Per esempio, non è ben tipato il termine $(\lambda\, x : \mathit{int} \,.\, x)\,(\lambda\, x : \mathit{int} \,.\, x)$. In sistemi dei tipi più complessi una stessa differenza esiste tra i tipi grezzi e quelli legali (derivati formalmente mediante regole) del linguaggio.

Judgements. Usiamo la metavariabile Γ per indicare un generico *contesto dei tipi*, ovvero una *lista ordinata* $x_1 : A_1, x_2 : A_2, \ldots x_n : A_n$ di variabili x_i *tutte distinte*, ciascuna associata ad un tipo A_i. Indichiamo con $\emptyset$ il contesto vuoto, dove $n = 0$.

Il sistema $F1$ permette di dedurre *asserzioni di tipo* (*typing judgements*), ovvero clausole della forma:

$$\Gamma \vdash M : A$$

che si legge: "M è un termine legale di tipo A nel contesto dei tipi Γ".

Regole di inferenza.

1. $\Gamma \vdash k : K$

2. $\Gamma_1, x : A, \Gamma_2 \vdash x : A$

3. $$\frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}$$

4. $$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda\, x : A\,.\,M : A \to B}$$

Le prime due regole sono assiomi, cioè regole senza premesse. In particolare, la prima assegna a ciascuna costante il suo tipo; la seconda assegna ad una variabile il tipo ad essa associato nel contesto. La terza regola dice che applicando una "funzione" di tipo $A \to B$ ad un "argomento" del giusto tipo A, si ottiene un "risultato" di tipo B. La regola 4 consente di assegnare un tipo freccia $A \to B$ a termini della forma $\lambda\, x : A\,.\,M$ (chiamati *lambda-astrazioni*) in un contesto Γ. Tale tipo sarà corretto se sarà possibile derivare il tipo B per il "corpo" M della lambda astrazione, nel contesto Γ arricchito dell'associazione tra il "parametro" x con il suo tipo A. Va notato che questa premessa richiede che $\Gamma, x : A$ sia effettivamente un contesto, e dunque che x non compaia in Γ. Questo implica che non si possono tipare in $F1$ termini come

$$\lambda\, x : K \to K\,.\,(\lambda\, x : K\,.\,x) \tag{4}$$

in cui si annidino due lambda astrazioni sulla *stessa* variabile. Questo non deve essere considerato un problema dato che è possibile comunque assegnare correttamente un tipo a termini che sono alfa-equivalenti a (4), quale, per esempio: $\lambda\, x : K \to K\,.\,(\lambda\, y : K\,.\,y)$. In alcune presentazioni di simili sistemi di tipi, termini alfa-equivalenti vengono spesso considerati *sintatticamente* identici!

Esercizio. Dimostrare che se $\Gamma \vdash M : A$ è derivabile in $F1$, allora tutte le variabili libere di M sono in Γ.

Esercizio. Ricordate $fn\, x \Rightarrow x\, x$? Esiste un tipo A tale che $\lambda\, x : A\,.\,x\, x$ risulti ben tipato? □

No! Supponiamo infatti che $\emptyset \vdash \lambda\, x : A\,.\,x\, x : A \to B$ sia derivabile in $F1$ per qualche tipo B. In base alla regola 4 dovrà essere derivabile $x : A \vdash x\, x : B$ che richiede, in base alla regola 3, la premessa $x : A \vdash x : A \to B$. Ma allora, in base alla regola 2, i tipi A e $A \to B$ devono coincidere, il che è impossibile perché il secondo non potrà che avere almeno una freccia in più del primo.

7.2 Il Polimorfismo

I sistemi dei tipi *del second'ordine* catturano aspetti dei linguaggi di programmazione quali:

- il polimorfismo, come in ML;
- i moduli di programma, come in Modula2 e ML;
- i tipi di dato astratto, come in Smalltalk, CLU, Ada e ML.

I sistemi dei tipi del second'ordine estendono quelli del prim'ordine con *variabili di tipo*. Nella Sezione 7.3 presenteremo un sistema in cui questa estensione avviene nella forma più generale, dove cioè compaiono operatori di astrazione sui tipi, analoghi alla lambda astrazione vista per $F1$, tanto al livello dei *tipi* quanto a quello dei *termini*. Cosa vuol dire "astrarre su un tipo"?

Nel sistema $F1$ i termini vengono tipati *in un contesto* dei tipi: il judgement $x : \mathit{int} \vdash x+1 : \mathit{int}$ asserisce che $x+1$ è un termine legale di tipo *int* nel contesto che assegna il tipo *int* alla variabile x. In un sistema dei tipi del *second'ordine* le variabili agiscono da segnaposto non solo per i termini, ma anche per i *tipi*. È dunque possibile scrivere un tipo come $X \to \mathit{int}$ in cui la variabile X segna il posto per un tipo arbitrario. Inoltre, come vedremo, l'operazione di astrazione si può estendere alle variabili di tipo, tanto per formare *termini* quanto per formare *tipi*. Nel sistema $F2$ descritto nella prossima sezione si usano simboli diversi nei due casi: $\Lambda\, X\,.\, t$ e $\forall\, X\,.\, A$ indicano il termine ed il tipo ottenuti per astrazione su X rispettivamente dal termine t e dal tipo A.

Così come il termine $\lambda\, x : \mathit{int}\,.\, x+1$ può essere *applicato* ad uno di tipo *int*, un termine della forma $\Lambda\, X\,.\, t$ può essere *istanziato* con un tipo.

Esempio. Che genere di oggetto matematico denota un termine del tipo $\forall\, X\,.\,(X \to (\mathit{bool} \to \mathit{bool}))$? Intuitivamente un tale oggetto potrebbe essere, ad esempio, la funzione che restituisce sempre l'identità sui booleani, qualunque sia il suo input. Accettiamo l'esistenza di un tale oggetto, ed indichiamo dunque con $\forall\, X\,.\,(X \to (\mathit{bool} \to \mathit{bool}))$ il suo tipo. Ora possiamo assegnare un tipo a termini di *Fun* che non potevamo tipare in $F1$. Ad esempio, usando $\forall\, X\,.\,(X \to (\mathit{bool} \to \mathit{bool}))$ per x, ci aspettiamo di poter dare un senso a

$$\lambda\, x\,.\,((x\ 5)\,((x\ \mathit{true})\ \mathit{false}))$$

che non può essere tipato in $F1$. Nessun tipo di $F1$ potrebbe infatti essere associato alla variabile x in modo da rendere contemporaneamente tipabili i sottotermini $(x\ 5)$ ed $(x\ \mathit{true})$.

Esempio. Il termine $\Lambda\, X\,.\,\lambda\, x : X\,.\, x$ denota l'"identità polimorfa". La sua istanziazione sugli interi, $(\Lambda\, X\,.\,\lambda\, x : X\,.\, x)\ \mathit{int}$ è *equivalente* (nozione che non definiremo formalmente) all'identità intera: $\lambda\, x : \mathit{int}\,.\, x$.

Esempio. Sia e una eccezione in un ipotetico linguaggio polimorfo con eccezioni e sia t il termine $\Lambda\, Y\, .\, \lambda\, x : X\, .\, \mathit{raise}_Y(e)$. È ragionevole aspettarsi che, nel sistema dei tipi di tale linguaggio, il termine t abbia tipo $\forall\, Y\, .\, X \to Y$, e che $t\ \mathit{int}$, di tipo $X \to \mathit{int}$, sia equivalente a $\lambda\, x : X\, .\, \mathit{raise}_{\mathit{int}}(e)$. □

Applicazione ed istanziazione condividono la stessa sintassi (si indicano entrambe mediante la giustapposizione di termini) ma sono operazioni di diversa natura. Per esempio, l'identità polimorfa $\Lambda\, X\, .\, \lambda\, x : X\, .\, x$ *non* è (immediatamente) una funzione, e può essere applicata solo dopo essere stata istanziata. Il suo tipo è $\forall\, X\, .\, X \to X$ che *non* è un tipo freccia.

7.3 Il sistema $F2$

Noto anche come *lambda calcolo polimorfo* o *del second'ordine*, o *System F*.

Sintassi. Estendiamo le convenzioni sulle metavariabili adottate per $F1$ convenendo che X indichi una generica variabile di *tipo*.

$$\begin{array}{rcl} \mathit{Types} & ::= & K \mid X \mid A \to B \mid \forall\, X\, .\, A \\ \mathit{Terms} & ::= & k \mid x \mid \lambda\, x : A\, .\, M \mid M\, N \mid \Lambda\, X\, .\, M \mid M\, A \end{array}$$

Nel tipo $\forall\, X\, .\, A$, così come nel termine $\Lambda\, X\, .\, M$, gli operatori di astrazione $\forall$ e Λ legano le occorrenze libere di X rispettivamente in A e in M.

Judgements. Come in $F1$.

Regole di inferenza. Le seguenti regole si aggiungono a quelle di $F1$.

1. $$\frac{\Gamma \vdash M : A}{\Gamma \vdash \Lambda\, X\, .\, M : \forall\, X\, .\, A} \quad (\text{se } X \notin \mathit{free}(\Gamma))$$

2. $$\frac{\Gamma \vdash M : \forall\, X\, .\, A}{\Gamma \vdash M\, B : [B/X]\, A}$$

La prima regola consente di costruire termini polimorfi astraendo su una variabile di tipo. Tale astrazione, però, può avvenire soltanto a condizione che la variabile su cui si astrae non compaia libera nel contesto. Dimenticando questa condizione si giungerebbe a tipare termini sensa senso, come dimostra il seguente esempio.

Esempio. Applicando la regola 1, ma senza rispettare la condizione laterale, costruiamo un *nuovo* numero intero, di cui il maestro delle elementari non ci aveva mai parlato. Sia M il termine bruto

$$\Lambda\, X\, .\, \lambda\, x : X\, .\, ((\Lambda\, X\, .\, x)\ \mathit{int})$$

L'applicazione erronea di 1 ci consente di dedurre il giudizio:

$$\emptyset \vdash M : \forall X \,.\, (X \to \mathit{int})$$

(esercizio: individuare il "passo falso" sull'albero di derivazione). Da tale premessa è facile derivare:

$$\emptyset \vdash (M\ (\mathit{int} \to \mathit{int}))\,(M\ \mathit{int}) : \mathit{int}$$

Ora, questo termine è definito nell'ambiente vuoto, ovvero ha in sé tutta l'informazione che gli serve per esistere. Ma dunque, che intero denota? □

La seconda regola, che riguarda l'istanziazione di un termine polimorfo con un tipo, si usa la nozione di sostituzione: $[B/X]\,A$ è il termine ottenuto sostituendo ogni occorrenza *libera* di X in A con B. Questa operazione può richiedere la ridenominazione di alcune variabili (di tipo) legate per evitare la *cattura* di X.

Esempio. Indichiamo con $\equiv$ la relazione di alfa-equivalenza. L'esempio che segue mostra il fenomeno della cattura:

$$\forall Z \,.\, (Y \to Y) \to Z \equiv [(Y \to Y)/X]\,\forall Y \,.\, X \to Y \neq \forall Y \,.\, (Y \to Y) \to Y.$$

□

Il termine $t \equiv \lambda x \,.\, ((x\ 5)\ ((x\ \mathit{true})\ \mathit{false}))$ della sezione 7.2 può essere tipato in $F2$. Per rendere precisa questa affermazione dobbiamo introdurre una funzione *untype* che associa ad ogni termine di $F2$ la sua versione non tipata:

$$\begin{array}{ll} \mathit{fun} & \mathit{untype}\ k = k \\ & \mid \mathit{untype}\ x = x \\ & \mid \mathit{untype}\ \mathit{fn}\ a \Rightarrow b \end{array}$$

[QUESTO VA RISCRITTO IN MODO ORGANICO, PARLANDO DELL'APPROCCIO ALLA CHURCH E ALLA CURRY AL LAMBDA CALCOLO TIPATO]

Esercizio. Indichiamo con $\overline{n}$ la codifica di Church di un numero naturale n. Ricordiamo che in ML $\overline{n}$ ha tipo `('a -> 'a) -> ('a -> 'a)`. Abbiamo visto che, dati due numeri di Church $\overline{n}$ ed $\overline{m}$, il termine $\overline{n}(\overline{m})$ denota il numero di Church $\overline{m^n}$. Dunque, una funzione ML che dato $\overline{n}$ calcolasse $\overline{n^n}$ avrebbe la forma `fn x => x x`. Sappiamo tuttavia che tale termine non è tipabile in ML. È possibile scrivere una tale funzione in $F2$?

8 Polimorfismo in ML

In ML i termini di cui si compone un programma *possono* non contenere alcuna informazione sui tipi. Ad esempio, sono legali in ML tanto il termine `fn x : int => x` quanto `fn x => x`. Il tipo di un termine viene calcolato dall'interprete mediante un sistema di inferenza:

```
- fn x : int => x;
> val it = fn : int -> int

- fn x => x;
> val it = fn : 'a -> 'a
```

In questo esempio, il tipo `'a -> 'a` indica che la funzione `fn x => x` è *polimorfa*, ed è dunque corretto applicarla ad argomenti di tipo diverso:

```
- (fn x => x) 5;
> val it = 5 : int

- (fn x => x) true;
> val it = true : bool
```

Che relazione esiste tra il sistema dei tipi di ML ed il sistema $F2$ descritto nella Sezione 7.3? Notiamo per prima cosa che l'espressione $\lambda x \,.\, x$ non è un termine legale in $F2$, dove l'identità polimorfa si scrive: $\Lambda X \,.\, \lambda x : X \,.\, x$. In base a tale corrispondenza, al tipo `'a -> 'a` dovrebbe corrispondere il $F2$ il tipo $\forall X \,.\, X \to X$, e dunque, là dove in ML scriviamo `(fn x => x) 5`, in $F2$ scriveremmo $((\Lambda X \,.\, \lambda x : X \,.\, x)\ \mathit{int})\ 5$.

Allo scopo di studiare il polimorfismo in ML, ci limiteremo a considerarne un semplice sottolinguaggio, che chiameremo Fun_τ, ottenuto intersecando ML con il linguaggio *Fun* che abbiamo introdotto nella sezione 4.2. Dunque, Fun_τ è costituito da tutti quei programmi di *Fun* ai quali è possibile assegnare un tipo nel sistema dei tipi di ML. L'estensione della nostra discussione al caso in cui, come in ML, l'annotazione esplicita dei tipi è facoltativa risulterà ovvia.

È possibile mappare i programmi di Fun_τ in $F2$, come abbiamo fatto sopra per l'identità polimorfa. Chiamando I (come *i*nterpretazione) questa mappatura, un termine t di Fun_τ può essere visto come la forma "abbreviata" di $I(t)$. Il sistema di inferenza dei tipi dell'interprete di ML, che andremo a considerare più sotto, svolge il lavoro di I. Naturalmente non è possibile estendere l'interpretazione all'intero linguaggio *Fun*. Ad esempio, il termine $5\,(\mathit{fn}\ x \Rightarrow x)$ non è tipabile.

Nel descrivere l'interpretazione dei tipi ci avvarremo ancora del simbolo di astrazione $\forall$, che viene generalmente omesso dagli interpreti ML. Questo ci consentirà di chiarire meglio il "grado" di polimorfismo di ML. In particolare, vedremo che questo linguaggio ha un sistema dei tipi del *second'ordine* in cui l'astrazione sui tipi è consentita solo al livello *più esterno* della struttura sintattica di un tipo. Questo vuol dire che esistono in ML funzioni polimorfe,

ma *non* funzioni capaci di manipolare oggetti polimorfi (ovvero: riceverli come argomenti o produrli come risultato).

Sintassi dei tipi. Usiamo la metavariabile X per indicare un generico elemento dell'insieme infinito (numerabile) *TypeVar* delle *variabili di tipo*. Usiamo le metavariabili τ e σ per indicare rispettivamente un generico *tipo primitivo* ed un generico *schema* (di tipo).

$$\begin{array}{rcl} \tau & ::= & K \mid X \mid \tau_1 \to \tau_2 \\ \sigma & ::= & \tau \mid \forall X \,.\, \sigma \end{array}$$

Si chiama *monotipo* (*mono-type*) uno schema in cui non compaiono variabili di tipo. Uno schema della forma $\forall X_1 \,.\, \forall X_2 \ldots \forall X_n \,.\, \tau$ è scritto in forma abbreviata come $\forall X_1 \, X_2 \, X_n \,.\, \tau$.

Non esempio. $(\forall X \,.\, X \to X) \to (\forall X \,.\, X \to X)$ non è un tipo legale in ML.

Inserire la discussione su $\lambda x.x$ dagli appunti della lezione .

8.1 Il sistema dei tipi di Fun_τ

I sequenti, *judgements*, del nostro sistema dei tipi sono della forma $\Gamma \vdash M : \sigma$, dove Γ è un *contesto dei tipi*, ovvero una lista di associazioni $x : \sigma$ in cui nessuna variabile x è ripetuta, M è un termine e σ è uno schema. I termini M sono essenzialmente quelli del linguaggio Fun_τ. Dato un contesto Γ, indichiamo con $free(\Gamma)$ l'insieme delle variabili di tipo che compaiono libere in almeno un tipo in Γ.

Definizione 3 *Uno schema tipo $\sigma' = \forall Y_1 \ldots Y_m \,.\tau'$ di Fun_τ si dice* istanza generica (generic instance) *di uno schema $\sigma = \forall X_1 \ldots X_n \,.\tau$, e si scrive $\sigma > \sigma'$, se $\tau' = [\tau_i / X_i]\,\tau$ per n tipi primitivi τ_i, e nessuna Y_j occorre libera in σ.*

Le regole di inferenza del sistema dei tipi di Fun_τ sono le seguenti:

1. $\Gamma_1, x : \sigma, \Gamma_2 \vdash x : \sigma$

2. $$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \sigma'} \quad (\sigma > \sigma\,')$$

3. $$\frac{\Gamma \vdash M : \sigma}{\Gamma \vdash M : \forall X \,.\, \sigma} \quad (X \notin free(\Gamma))$$

4. $$\frac{\Gamma \vdash M : \sigma \quad \Gamma, x : \sigma \vdash N : \sigma'}{\Gamma \vdash \mathit{let}\ x = M\ \mathit{in}\ N : \sigma'}$$

5. $$\frac{\Gamma, x : \tau' \vdash M : \tau}{\Gamma \vdash \mathit{fn}\ x \Rightarrow M : \tau' \to \tau}$$

6. $$\frac{\Gamma \vdash M : \tau' \to \tau \quad \Gamma \vdash N : \tau'}{\Gamma \vdash M\,N : \tau}$$

Le regole 1, 4, 5 e 6 sono autoesplicative. Le regole 2 e 3 mostrano come, in Fun_τ, un termine possa avere più di un tipo, cosa che non accade né in $F1$ nè in $F2$. La regola 2 consente di assegnare a qualunque termine M di tipo σ un qualunque tipo σ' che sia una istanza generica di σ, e dunque di usare M in qualunque contesto in cui ci si aspetta un σ'. È grazie a questa regola che riusciamo a assegnare un tipo al termine $(\mathit{fn}\ x \Rightarrow x)\,5$. La terza regola ci consente di astrarre il tipo di un termine su una qualunque variabile di tipo, a condizione però che questa non compaia libera nel contesto dei tipi. È grazie a questa regola che riusciamo ad assegnare il tipo $\forall X\,.\,X \to X$ a $\mathit{fn}\ x \Rightarrow x$. Questa regola ha una condizione laterale, che è analoga alla 1 del sistema $F2$. Violando questa condizione, ad esempio, si potrebbe derivare:

$$\frac{x : X \vdash x : X}{x : X \vdash x : \forall X\,.\,X}\ (\mathsf{no!})$$

da cui sarebbe facile inferire il sequente $\vdash \mathit{fn}\ x \Rightarrow x\,x : \forall X, Y.\,X \to Y$. Istanziando opportunamente questo schema si deduce facilmente il sequente:

$$\vdash (\mathit{fn}\ x \Rightarrow x\,x)\,(\mathit{fn}\ x \Rightarrow x\,x) : \forall X\,.\,X$$

Accettare questo termine in Fun_τ sarebbe una faccenda piuttosto incresciosa per almeno due motivi. L'esistenza di un termine *chiuso* di tipo $\forall X\,.\,X$, implicherebbe l'esistenza di almeno un "abitante" per *ogni* tipo. Pensando ai tipi come a delle specifiche, o insiemi, questo suggerirebbe che il nostro sistema dei tipi è inconsistente, in quanto esisterebbe, ad esempio, un programma che soddisfa la specifica $\{x \mid \mathit{false}\}$ (ovvero che appartiene all'insieme vuoto). La seconda ragione di sconforto è che, non ostante in Fun_τ si riescano a tipare piu' termini che in $F1$, in virtù del polimorfismo, come in $F1$ tutti i programmi terminano.

Nota. Nella Sezione 4.2.2 abbiamo proposto di tradurre (da Fun_τ nel lambda calcolo) il termine $\mathit{let}\ x = M\ \mathit{in}\ N$ come $(\mathit{fn}\,x \Rightarrow N)\,M$. La cosa funziona operazionalmente, ma non con i tipi! Infatti, mentre il termine

$$\mathit{let}\ x = (\mathit{fn}\,x \Rightarrow x)\ \mathit{in}\ (x\,x)$$

è tipabile in ML, come abbiamo già visto $(\mathit{fn}\ x \Rightarrow x\,x)\,(\mathit{fn}\,x \Rightarrow x)$ non lo è. Che sta succedendo?

8.2 Inferenza dei tipi in ML

Dunque un termine può avere più di un tipo in Fun_τ. Qual è dunque quello scelto dall'interprete del linguaggio? Stabiliremo ora che tale tipo, calcolato applicando l'algoritmo $\mathcal{W}$ descritto qui di seguito, è il più generale possibile.

Sostituzione. Una *sostituzione* di tipi è un elemento dell'insieme

$$TypeVar \xrightarrow{fin} PrimTypes,$$

ovvero una funzione V che mappa variabili di tipo in tipi primitivi, e tale che $V(X) \neq X$ solo per un numero *finito* di $X \in TypeVar$. Abusando della notazione, indichiamo con $V(\sigma)$ il tipo risultante dall'applicazione della sostituzione V al *tipo* σ, ovvero dalla sostituzione in σ di X con $V(X)$, per ogni variabile X *libera* in σ. Tale sostituzione può richiedere la ridenominazione di variabili legate per evitare la cattura. La *composizione* di due sostituzioni W e V è la sostituzione $W\,V$ che mappa X in $W(V(X))$. Una sostituzione V *unifica* due tipi σ e σ' se $V(\sigma) = V(\sigma')$.

Esercizio. Dimostrare che, per ogni sostituzione V, dati due tipi σ e σ', se $\sigma' \leq \sigma$ allora $V\,\sigma' \leq V\,\sigma$.

Proposizione 4 *(Teorema di unificazione di Robinson) Esiste un algoritmo U che, dati due tipi primitivi τ_1 e τ_2, restituisce una sostituzione V o altrimenti fallisce. Inoltre:*

- *se $U(\tau_1, \tau_2) = V$ allora:*
  - *$V(X) \neq X$ solo se X compare in τ_1 o τ_2, ed inoltre*
  - *V unifica τ_1 e τ_2;*
- *se esiste un unificatore W di τ_1 e τ_2, allora $U(\tau_1, \tau_2) = V$ per un V tale che $W = Z\,V$, per una qualche sostituzione Z.* □

Algoritmo $\mathcal{W}$ di inferenza dei tipi.

- *Input*: un contesto Γ ed un termine M;
- *Output*: (ove non fallisca) una coppia (V, τ), dove V è una sostituzione e τ un tipo primitivo tali che:

$$V\,\Gamma \vdash M : \tau$$

Prima di definire W, chiariamo il senso della sostituzione V restituita dall'algoritmo, là dove ci saremmo aspettati di ricevere soltanto un tipo.

Esempio. Dato il contesto $\Gamma = (x : X, y : Y)$ ed il termine $y\,x$, ci chiediamo quale sia il tipo τ tale che $\Gamma \vdash y\,x : \tau$. Ciò che l'algoritmo ci consente di stabilire è che:

$$x : X, y : X \to Z \vdash y\,x : Z.$$

Oltre a restituirci un tipo, $\mathcal{W}$ ci dice anche qualcosa sul tipo Y che Γ assegnava ad y, e cioè che deve essere un tipo freccia e che questo deve avere X come sorgente. Ciò è espresso dalla sostituzione $\{Y = X \to Z\}$, in base alla quale il tipo dell'intero termine è Z. □

L'output $W(\Gamma, M) = (V, \tau)$ è definito per casi come segue:

- se $M = x$ e $x : \forall\, X_1 \ldots X_n \,.\, \tau'$ è in Γ, allora V è la sostituzione identica e $\tau = [Y_i/X_i]\tau'$, dove le Y_i sono variabili *nuove*, per $i = 1 \ldots n$;
- se $M = M_1\, M_2$, e inoltre:
  - $\mathcal{W}(\Gamma, M_1) = (V_1, \tau_1)$ e
  - $\mathcal{W}(V_1\Gamma, M_2) = (V_2, \tau_2)$ e
  - $U(V_2\tau_1, \tau_2 \to Y) = W$, dove Y è una nuova variabile di tipo,

  allora $V = W(V_2\, V_1)$ e $\tau = W(Y)$;
- se M è il termine *fn* $x \Rightarrow N$, e inoltre $\mathcal{W}((\Gamma, x : X), N) = (V_1, \tau_1)$, dove X è una nuova variabile di tipo, allora $V = V_1$ e $\tau = V_1(X) \to \tau_1$;
- se M è il termine *let* $x = M$ *in* N, e inoltre $\mathcal{W}(\Gamma, M) = (V_1, \tau_1)$ e $\mathcal{W}((V_1\Gamma, x : \overline{V_1\Gamma}(\tau_1)), N) = (V_2, \tau_2)$, allora $V = V_1V_2$ e $\tau = \tau_2$.

Proposizione 5 *Se $\mathcal{W}(\Gamma, M) = (V, \tau)$ allora $V\,\Gamma \vdash M : \tau$ è derivabile.*

Dimostrazione. Per induzione sulla struttura di M. □

Definizione 6 *La* chiusura *di uno schema σ rispetto ad un contesto Γ, scritta $\overline{\Gamma}(\sigma)$, è il tipo $\forall\, X_1, \ldots X_n \,.\, \sigma$ dove le X_i sono* tutte *le variabili di tipo che sono libere in σ ma non in Γ.*

Notando che da $\Gamma \vdash M : \tau$ si può inferire $\Gamma \vdash M : \overline{\Gamma}(\tau)$, abbiamo che $\mathcal{W}(\Gamma, M) = (V, \tau)$ implica $V\,\Gamma \vdash M : \overline{V\,\Gamma}(\tau)$. Quest'ultimo tipo è chiamato lo *schema* calcolato da $\mathcal{W}$.

Definizione 7 *Uno schema σ_p si dice* principale *per un contesto Γ ed un termine M quando:*

- *$\Gamma \vdash M : \sigma_p$, ed inoltre*
- *se $\Gamma \vdash M : \sigma$ allora $\sigma_p > \sigma$.*

Proposizione 8 *Se $\Gamma \vdash M : \sigma$ è derivabile nel sistema dei tipi di Fun_τ, allora $\mathcal{W}(\Gamma, M)$ produce* uno *schema principale.*

9 L'isomorfismo di Curry-Howard

Possiamo pensare semanticamente i programmi del linguaggio *Imp* come funzioni parziali da stato a stato, dove, in assenza di costrutti che producono aliasing (ad esempio la call-by-reference), possiamo adottare come nozione di *stato* una semplice associazione di valori a variabili. Dunque, lo stato risultante dall'esecuzione del programma composto $p; q$ a partire dallo stato iniziale s è $q(p(s))$. In questa prospettiva, che avvicina il paradigma imperativo a quello funzionale, una tripla di Hoare $\{A\}p; q\{B\}$ può essere vista come una condizione di ingresso/uscita della funzione denotata da $p; q$, e la regola di inferenza *comp* può essere letta come un controllo che la condizione di uscita di p soddisfi la condizione di ingresso di q.

Volendo spingere oltre l'analogia tra linguaggi imperativi e funzionali, possiamo considerare una asserzione A come un *tipo*, l'insieme degli stati che soddisfano A, e considerare la deduzione di una specifica come un controllo sulla consistenza dei tipi. In questa sezione sfrutteremo questa idea presentando un sistema per la verifica della correttezza di linguaggi funzionali basato su quella che è nota in letteratura come la teoria dei *tipi dipendenti*. Nel corso degli anni sono state proposte diverse teorie dei tipi, con relativi sistemi formali [Mar84, Coq89, GTL89, Bar92], e ne sono state studiate le connessioni con altrettanti sistemi logici [Bar92, How80]. In questo capitolo ci baseremo sulla presentazione della teoria dei tipi di Per Martin-Löf [Mar84].

Esiste una correlazione tra logica e *tipi*, nel senso che un tipo può essere visto come un insieme di valori che soddisfano una proposizione, e viceversa una proposizione può essere interpretata come l'insieme dei valori che la soddisfano. In questo capitolo mostriamo che, sfruttando questa relazione, diversi sistemi dei tipi per linguaggi funzionali possono essere descritti semplicemente considerando sistemi logici di diversa complessità.

9.1 Tipi dipendenti

Vi sono funzioni per le quali è possibile associare uno specifico codominio a ciascun elemento del dominio. Consideriamo come esempio la funzione f che mappa n nella sequenza dei primi $n + 2$ numeri di Fibonacci:

$$f(0) = \langle 1, 1\rangle, \quad \ldots \; f(3) = \langle 1, 1, 2, 3, 5\rangle \ldots$$

Indicando con $S(n)$ l'insieme delle sequenze di naturali lunghe $n + 2$, abbiamo che $f(n) \in S(n)$ per ogni $n \in N$. Volendo dunque specificare la dipendenza del codominio della funzione dai valori di input, indicheremo il tipo di f con l'espressione:

$$\Pi n : N \,.\, S(n).$$

In teoria dei tipi espressioni di questa forma sono chiamate *prodotti dipendenti*. Intuitivamente, il tipo $\Pi x : A \,.\, B(x)$ denota dunque l'insieme delle funzioni che, dato in ingresso un x in A, restituiscono un valore in $B(x)$. Nel caso l'insieme $B(x)$ non dipenda da x, ovvero sia lo stesso per ogni x, allora considereremo tale espressione come sinonimo del più usuale $A \to B$.

REGOLA DEL PI Usiamo la formulazione dalla teoria dei tipi dipendenti di Per Martin-Löf [Mar84].

9.2 Tipi e proposizioni

Nel capitolo X abbiamo collegato il quinto assioma di Peano con l'induzione matematica, notando che una proprietà logica P può essere vista come un insieme, quello appunto degli oggetti che soddisfano P. Elaborando sul tema, tali oggetti possono essere visti come esempi (in teoria dei tipi si usa il termine "*testimoni*") di "P-ità". Ad esempio, il numero sette è un testimone (canonico) della *disparità*.

L'idea che dimostrare significhi esibire un testimone è alla base di una scuola di pensiero nota come *logica costruttiva* ed iniziata da *L.E.J. Brouwer* [Bro87]. Quello che segue è un famoso esempio di dimostrazione *non* costruttiva.

Teorema 9.1 *Esistono due numeri irrazionali u e v tali che u^v è razionale.*

Dimostrazione. Se $\sqrt{2}^{\sqrt{2}}$ è razionale allora $u = v = \sqrt{2}$. Altrimenti $u = \sqrt{2}^{\sqrt{2}}$ e $v = \sqrt{2}$. □

Per Brouwer questa non è una prova: per ammettere una testimonianza (nel caso specifico dell'esistenza di una coppia di irrazionali) bisogna presentare il testimone davanti al giudice!

Il punto di vista filosofico di Brouwer, noto come *intuizionismo*, è stato ulteriormente elaborato da Heyting (un suo studente) e Kolmogorov, che svilupparono un approccio costruttivo alla matematica ed una interpretazione dei connettivi logici basata su un calcolo delle prove (dimostrazioni). Ad esempio, una prova dell'implicazione $A \supset B$ è costituita da una costruzione (funzione) che trasforma prove di A in prove di B. Questa interpretazione ha dato origine al cosiddetto paradigma delle *proposizioni come tipi* (*propositions-as-types*), che collega logica e teoria dei tipi, in cui una proposizione viene vista come il tipo delle sue dimostrazioni. Curry e Howard, in particolare, hanno messo in luce l'isomorfismo che esiste fra le formule di un sistema logico ed i tipi di una teoria dei tipi. Secondo questo isomorfismo, ad esempio, il tipo $\Pi\, x : A \,.\, B$ corrisponde alla formula $\forall\, x : A \,.\, B$, ed un termine $\lambda\, x : A \,.\, M$ di un tale tipo rappresenta una prova (canonica) della formula secondo l'interpretazione di Heyting.

REGOLA DELL'UGUAGLIANZA

9.3 Verifica di un programma funzionale

Nella sezione X abbiamo introdotto uno schema di definizione di funzioni sui numeri naturali mediante due clausole: una specifica il valore della funzione su *zero*; l'altra specifica il suo valore sul *successore* di n, usando eventualmente il valore che la funzione assume in n. Un esempio canonico di questo schema di definizione è la funzione *fattoriale*, il cui tipo è $N \to N$. Questo schema

si estende ai tipi dipendenti. Ad esempio, la funzione di Fibonacci specificata sopra è definita come segue.

Esplicitando le componenti di una sequenza $s \in S(n)$ come $\langle s_0, s_1, \ldots s_{n+1} \rangle$, sia d la funzione che, dato $n \in N$ e data $s \in S(n)$, restituisce la sequenza $\langle s_0, s_1, \ldots s_{n+1}, s_n + s_{n+1} \rangle \in S(n+1)$. Il tipo di d è $\Pi n : N \,.\, (S(n) \to S(n+1))$. Usando d la definizione di f è dunque:

$$\begin{array}{rcl} f(0) & = & \langle 1, 1 \rangle \\ f(succ\,(n)) & = & d\,(n, f(n)). \end{array}$$

Lo schema di definizione appena descritto è formalizzato nella teoria dei tipi dipendenti [Mar84] dalla seguente regola generale:

- assimatizzazione naturali
- programma per x+x

Parte IV
La Correttezza di Programmi

> On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. Over the following days, the Director General of ESA and the Chairman of CNES set up an independent Inquiry Board. The crash was found to be caused by the following sequence of events (summary): 1. An overflow occurred in the Inertial Reference System (SRI) computer when converting a 64-bit floating point to 16-bit signed integer value. 2. There was no handler for that specific ovewrflow. The default handler (wrongly) shut down the SRI unit. 3. The shutdown caused the SRI to output a core dump on the bus. The main computer interpreted the core dump as flight data, causing such a violent trajectory correction that the rocket disintegrated. The investigation team concluded that the designers of the computer system put protections against hardware faults but did not take into account software faults.

Questo brano è tratto da un comunicato stampa emesso da una commissione d'inchesta istituita dall'ESA in seguito all'esplosione dell'*Ariane 5* nel giugno 1996. Il missile era al suo primo lancio, dopo un periodo di sviluppo durato 10 anni e costato sette miliardi di dollari. Il valore complessivo dell'Ariane e del suo carico è stato stimato essere di cinquecento milioni di dollari.

Chi sviluppa software sa che si verificano spesso, durante l'esecuzione di un programma, condizioni anomale, dovute per esempio ad interazioni non previste tra sottoprogrammi diversi, e che inducono il sistema a conportamenti indesiderati. Questa considerazione, a fronte dell'utilizzo sempre maggiore del software nel controllo di applicazioni *critiche* (tanto in termini di costo che di sicurezza), ci induce alla ricerca di criteri matematici per stabilire la *correttezza* di un programma.

Semantica assiomatica. The question of whether a computer program is correct requires a more basic question to be answered: what is the meaning of a program? Only when we have a precise, mathematically tractable description of a program can we *prove* that it does what it is meant to do. The discipline of computer science which studies the meaning of programs is called programming language *semantics*.

Many approaches to semantics have been proposed over the years. In section **??** we described one, *operational semantics*, based on the notion of *computation*, that is a sequence of steps of program execution by some abstract machine. Below we describe an approach to programming language semantics based on logics, which has immediate applications in the study of program correctness. In this approach the meaning of programming constructs, like assignments or

`while`-loops, is described by logical formulae. Such formulae can be deduced according to some rules of reasoning, called *inference rules*, form some primitive statements, whose truth is accepted "a priori," called *axioms*. From that the name of **axiomatic semantics**. In section **??** we introduce a formal notion of *deduction*; in section 10.2 we give the axiomatic semantics of a simple language (say, the language of Java statements reduced to the bone), and in section 10.3 we show an example where the semantics is used to prove the correctness of a program formally.

10 La Correttezza nei Linguaggi Imperativi

Below we introduce the technique of *loop invariants*, based on the principle of induction (section **??**), which can be used to give a mathematical argument for the correctness of a program with loops. In section **??** we shall *formalise* this technique and present a logic for reasoning about program correctness formally, that is by means of formal deduction.

10.1 Il metodo delle invarianti

The correctness of a programs is often expressed as a statement about the content of certain program variables, for example "the final content of `result` is $a + b$." If a program only consisted in a sequence of assignments it would be easy to trace down the content of each program variable at any point of execution. However, when the program involves loops, for example, there is no general way of telling how many times the loop will be executed and hence of predicting, by just glancing at the program, what value would a certain variable `x` have at some point of execution, if `x` is assigned in a loop.

A possible solution to this problem is to try and make a statement about the effects of the loop, which should hold *no matter how many times the loop is executed*. This statement should be *strong* enough so as to help us, when combined with information gathered from the rest of the program, to prove our program correct. Let's put this idea at work.

Suppose that some property involving certain program variables can be shown to hold at the end of an iteration through the loop under the assumption that it holds at the beginning of the iteration. Such a property is called **loop invariant** because its truth is not affected by iterations through the loop. If an invariant P can be shown to hold when entering the loop for the first time, then we know that P will hold when, *if ever*, exiting the loop for the last time, no matter how many times we iterate. In fact, let $Q(n)$ be the property:

> "P holds after n iterations."

Then $Q(1)$ holds by hypothesis. Moreover, assume that $Q(n)$ holds. This implies that when entering the loop for the $(n+1)$-th time (which is indeed immediately after having finished the n-th iteration), P will also hold. Then, since P is invariant, we can deduce that $Q(n+1)$ holds. Therefore, by induction, we can conclude that $Q(n)$ holds for all n. Now, if the program ever exits the loop, it will be after k iterations, for some $k \in N$. But then P holds because $Q(k)$ holds.[2]

Summing up, the general technique for proving the correctness of a program with a loop is the following: first, state a property P which, if true when the program abandons the loop, would help you to prove the correctness of the

[2] In fact, this kind of reasoning is called "partial correctness," because correctness is proven under the condition that the program terminates. There are techniques for proving the "total" correctness of a program, but we shall not explore them here.

program. Then try to show that P is a loop invariant and that it holds at the beginning of the loop. This is enough (as argued above) to conclude that P will hold when (if ever) exiting the loop for the last time. Of course one loop may have several loop invariants. For example $5 = 5$ is a loop invariant for just about any loop, but not a very useful one! How to find the most useful invariant for each problem is a matter of practice. In the following sections we make practice with two examples.

Correttezza della moltiplicazione egizia. Il Papiro di Rhind (circa 1650 a.C.) descrive, fra molte altre cose matematiche, l'algoritmo usato dagli antichi egizi per svolgere la moltiplicazione. Supponiamo di essere nel 1650 a.C. e di voler moltiplicare 138 per 45. Noi si inizia con: *cinque per otto*. Cerchiamo dunque fra i nostri papiri quello con la tabellina, ma ahimé non lo troviamo! Dobbiamo aspettare non già Pitagora, che non ha inventato affatto la tavola pitagorica, ma il monaco romano Vittore d'Aquitania, che la introdusse nel V secolo dell'era cristiana. Con un po' di sforzo riusciamo comunque a calcolare cinque per otto: fa quaranta, scrivo zero e porto quattro. Ahimé no: *scrivo zero* non si può fare perché lo zero ancora non è stato ancora inventato! (I romani nemmeno avevano un simbolo per indicare lo zero!) Procediamo comunque; finito di moltiplicare il 5, scendiamo di una riga e iniziamo a moltiplicare il 4, incolonnandoci rispetto alla riga precedente, ma *scalando a sinistra di una posizione*. Ma ahimè neanche questo sappiamo fare, perché il sistema posizionale per la rappresentazione dei numeri fu inventato 1500 anni più tardi in Cina. Ecco dunque l'algoritmo egizio, senza tabellina, senza zero, senza rappresentazione posizionale e... in Java!

```
public class Aegypt {
  public static void main(String[] args) {

    int a = 45;
    int b = 138;
    int x = a, y = b, res = 0;

    /* Let INVARIANT be the following property:
       y >= 0 && x * y + res = a * b */

    while (y >= 1) { // INVARIANT holds here
      if  (y % 2 == 0) {
        x = x + x;
        y = y/2;
      }
      else {
        res = res + x;
        y = y - 1;
      } // INVARIANT holds here
    }   // INVARIANT holds here and moreover y < 1.
```

```
    System.out.print("Ahmes says: ");
    System.out.println(a + " times " + b + " is " + res);
  }
}
```

When referring to a while statement "`while (`E`)` S" we shall call "iteration" one evaluation of the condition expression E possibly followed by the execution of the *body* S.

We now prove the correctness of the above program. More precisely we prove that the value of `res` printed by the method `main` is

$$\texttt{res} = \texttt{a * b}.$$

After initialization of `a`, `b`, `x`, `y` and `res`, the property

```
y >= 0 && x * y + res = a * b
```

holds. This property is called "INVARIANT" in the comment part of the program. Now suppose that INVARIANT holds when the program is about to execute the test `y >= 1`. Suppose that this test is successful and that `y` is even. We enter the "then" branch of the `if` where `x` is doubled and `y` is divided by two. Clearly, after executing these two operations the value of `x * y` is the same as before and so the truth of

```
x * y + res = a * b
```

is unaffected. Similarly dividing a positive number by two still yields a positive number and hence also `y >= 0` still holds. So, at the end of the iteration, INVARIANT would still hold. In case `y` is odd, and the "else" branch of the `if` statement is executed, the truth of INVARIANT is also preserved, because:

```
x * y + res = (x + x) * (y -1) + res
```

and moreover `y - 1 >= 0` since, being inside the loop, we must have passed the test `y >= 1`. On the other hand, if the test `y >= 1` evaluates to false, then the loop is immediately exited, and INVARIANT, which we assumed true at the beginning of the iteration, would still be true. We can then conclude that INVARIANT deserves its name, that is that it is a loop invariant. Then, by the inductive argument given in the previous section, we conclude that, *if* the program ever exits the loop, INVARIANT would hold.

In fact we know more: if the program exits the loop, it means that `y` is less than 1 (the test must fail in order to exit the loop). Hence, by the invariant, it must be

$$0 \leq \texttt{y} < 1$$

that is $\texttt{y} = 0$. But then `x * y` is 0 and, by the invariant, `res = a * b` after the `while` as required.

Fibonacci. Il seguente programma Java stampa il primo numero maggiore di 1000 nella sequenza di Fibonacci. Quale specifica ne esprime la correttezza?

```
public class Fibo {
  public static void main (String[] args) {
    int prev = 0, curr = 1;
    int sum;
    while ((sum = prev + curr) <= 1000) {
      prev = curr;
      curr = sum;
    }
    System.out.print(sum + " is the first Fibonacci");
    System.out.println(" number greater than 1000.");
  }
}
```

10.2 La Logica di Hoare

Diamo ora una semantica assiomatica del linguaggio *Imp*. Per nostra comodità, distinguiamo tra espressioni *numeriche*, per le quali useremo le lettere M, $N\ldots$ ed espressioni *booleane*, per le quali useremo A, $B\ldots$, cui aggiungiamo l'implicazione $\supset$. Inoltre, eliminiamo per semplicità dal linguaggio dei programmi il costrutto *var* per la dichiarazione di variabili.

$$
\begin{array}{lcl}
M, N & ::= & 0 \,|\, 1 \,|\, \ldots \,|\, x \,|\, M + N \\
A, B & ::= & \mathit{true} \,|\, \mathit{false} \,|\, A \supset B \,|\, M < N \,|\, M = N \\
p & ::= & \mathit{skip} \,|\, p; q \,|\, x := M \,|\, \mathit{if}\ B\ \mathit{then}\ p\ \mathit{else}\ q \,|\, \mathit{while}\ B\ \mathit{do}\ p
\end{array}
$$

Delle espressioni booleane abbiamo dato una versione minimalista con un unico operatore logico. Gli altri possono infatti essere derivati. La negazione $\neg A$, ad esempio, può essere considerata un'abbreviazione di $A \supset \mathit{false}$. Analogamente scriviamo $A \vee B$ al posto di $\neg A \supset B$, e $A \wedge B$ per $\neg(A \supset \neg B)$, o anche per $\neg(\neg A \vee \neg B)$, che fa lo stesso. Per quanto riguarda gli operatori relazionali, scriviamo $M \geq N$ per $\neg(M < N)$, e così via.

Intuitively, `skip` is the "do nothing" command; $x := e$ is the assignment of e to x, and $C1; C2$ is the composite command which first performs $C1$ and then $C2$. The `while` and the `if` commands are just like in Java, and in fact the above language can be considered as a simple sublanguage of the Java statements. An example of a program in this language is the following, which computes the integer division of x by y, putting the result in a and the remainder in b. For clarity we introduce parentheses, which are not really part of the language (we are dealing with *abstract* syntax here, so we are not concerned with ambiguity). Moreover, we write "$b \geq y$" for "$b > y \vee b = y$" and so on.

$b := x;$
$a := 0;$
`while` $(b \geq y)$ `do` $(b := b - y;\ a := a + 1)$

How can we claim that this program computes integer division? Above we just relied on our intuitive understanding of assignment and `while`, but so far we are unable to make this statement precise, unless some abstract machine is defined, like the one we introduced in section **??**, to actually *run* our program and check the result. Another possibility is to use logic.

Below we show how to combine logics with programs so that statemets such as "program C computes the integer division of x by y" can be expressed formally and *proven* by applying rules of inference, like we did in section **??** for abstract propositions. Then, the mathemetical meaning of a program C (its axiomatic semantics) can be considered to be the set of properties which can be proven about C.

Let a **formula** be a proposition or a *triple* as in the following grammar:

$$\mathit{Form} \quad ::= \quad \mathit{Prop} \mid \{\mathit{Prop}\}\ \mathit{Com}\ \{\mathit{Prop}\}$$

The propositions P and Q in the formula $\{P\}\ C\ \{Q\}$ are called respectively **precondition** and **postcondition**. The intuitive meaning of such a triple is: "if P holds before running the program C, then Q holds after C terminates, if ever." This interpretation of Hoare triples is called *partial correctness*, as the postcondition is meant to hold *provided* C terminates.

The rules for deriving such statements are the ones of the propositional calculus given in section **??** together with the three groups of rules, *arithmetical axioms*, *general rules* and *special rules* described below. The resulting system of formal deduction is known as **Hoare logic** (or, more precisely, Hoare-Floyd).

We define the arithmetical axioms to be the set of all propositions which we know to hold in arithmetic, such as $1 + 1 = 2$ and $7 > 4$, and moreover all the *tautologies* involving $=$, $<$ and $>$, such as $x = x$, $x < x+1$ and so on. Of course the proposition $x = 5$ (don't confuse that with an assignment in Java!) is *not* an arithmetical axiom since its truth depends on the value of the variable x.

General rules. The following inference rules are called *general* because they apply to any program C. As before, names are attached to each rule.

$$\frac{}{\{P\}\ C\ \{\mathit{true}\}}\ (\mathit{true})$$

$$\frac{}{\{\mathit{false}\}\ C\ \{P\}}\ (\mathit{false})$$

$$\frac{P \supset Q \qquad \{Q\}\ C\ \{R\}}{\{P\}\ C\ \{R\}}\ (\mathit{str})$$

$$\frac{\{P\}\ C\ \{Q\} \qquad Q \supset R}{\{P\}\ C\ \{R\}}\ (\textit{weak})$$

$$\frac{\{P\}\ C\ \{Q_0\}\ \ldots\ \{P\}\ C\ \{Q_n\}}{\{P\}\ C\ \{Q_0 \wedge \cdots \wedge Q_n\}}\ (\textit{and})$$

$$\frac{\{P_0\}\ C\ \{Q\}\ \ldots\ \{P_n\}\ C\ \{Q\}}{\{P_0 \vee \cdots \vee P_n\}\ C\ \{Q\}}\ (\textit{or})$$

The rules (*true*) and (*false*) say that *true* and *false* are respectively trivial postcondition and trivial preconditions of any program. Indeed *true* is always true after running a program, no matter what the preconditions are, and similarly for *false*. The rules (*str*) and (*weak*) allow respectively to strenthen the precondition and to weaken the postcondition of a formula. The (*and*) and (*or*) rules are obvious.

Special rules. The following inference rules are called *special* because they are specific of each program construct.

$$\frac{}{\{P\}\ \texttt{skip}\ \{P\}}\ (\textit{skip})$$

$$\frac{}{\{[E/x]P\}\ x := E\ \{P\}}\ (\textit{assign})$$

$$\frac{\{P \wedge Q\}\ C_1\ \{R\} \qquad \{P \wedge \neg Q\}\ C_2\ \{R\}}{\{P\}\ \texttt{if}\ Q\ \texttt{then}\ C_1\ \texttt{else}\ C_2\ \{R\}}\ (\textit{if})$$

$$\frac{\{P \wedge Q\}\ C\ \{P\}}{\{P\}\ \texttt{while}\ Q\ \texttt{do}\ C\ \{P \wedge \neg Q\}}\ (\textit{while})$$

$$\frac{\{P\}\ C_1\ \{Q\} \qquad \{Q\}\ C_2\ \{R\}}{\{P\}\ C_1; C_2\ \{R\}}\ (\textit{comp})$$

The rule (*skip*) says that *skip* does not change the world: what was true before its execution is still true after.

The rule for assignment requires some attention. Given a proposition P, we write $[E/x]P$ the proposition obtained by replacing each occurrence of x in P with E. For example, if P is the proposition $y = x + 7$ and E is the expression $x + 1$, then $[E/x]P$ stands for the proposition $y = x + 1 + 7$. Now, it is clear that if $y = x + 1 + 7$, then, after performing the assignment $x := x + 1$ we will have $y = x + 7$.

Esercizio 10.1 Derivare la tripla $\{x = 1\}\ x := x + 1\ \{x = 2\}$. □

Let Q be an arbitrary condition and let P be some condition which holds at some point during computation. At that point, either Q holds or it doesn't. In the first case the command `if` Q `then` C_1 `else` C_2 would execute C_1, while in the second it would execute C_2. Suppose that we know that running C_1 when P and Q hold will lead to a situation where R holds, and that the same happend if we run C_2 when P holds and Q does not, then R will surely hold after running `if` Q `then` C_1 `else` C_2. This argument explains the rule (*if*).

A similar argument justifies the (`while`) rule. In this rule the proposition P plays the role of *loop invariant*. Then, the rule is just a formalization of the argument extensively explained in section **??**.

The rule for composition is self explaining.

10.3 Correttezza della divisione intera

Nella sezione 10.2 abbiamo introdotto la Logica di Hoare con l'esempio di un programma, chiamiamolo p, che calcola la divisione intera di due numeri x e y:

$$b := x;\ a := 0;\ \texttt{while}\ (b \geq y)\ \texttt{do}\ (b := b - y;\ a := a + 1).$$

Prima di dimostrare formalmente la correttezza di questo programma, dobbiamo esprimere formalmente la proprietà che vogliamo dimostrare, ovvero tradurre la frase "il programma p calcola la divisione intera di x per y" in una formula della logica. Questo richiede una certa attenzione. Considerando che il programma lascia in a il risultato della divisione ed in b il resto, potremmo in prima approssimazione adottare la tripla $\{true\}\ p\ \{a \cdot y + b = x\}$ come specifica della correttezza. Ci accorgiamo però subito (ce ne accorgiamo?) che, se x è negativo, p non risponde correttamente. Il programma è stato infatti pensato per x ed y positivi. Dato però che p va in loop quando $y \leq 0 \leq x$, la precondizione $x \geq 0$ (che non parla di y) sarà sufficiente per dimostrare la correttezza *parziale* del programma.

Esercizio 10.2 Sarebbe dunque soddisfacente assumere $\{x \geq 0\}\ p\ \{a \cdot y + b = x\}$ come enunciato della correttezza di p? □

Scegliamo seguente formula come specifica di p:

$$\{x \geq 0\}\ p\ \{a \cdot y + b = x \wedge b \geq 0 \wedge b < y\}. \tag{5}$$

(Perché vogliamo $b \geq 0$ nella postcondizione? Perché ci vogliamo $b < y$?) Va notato che il programma $a := 0;\ b := 0;\ x := 0;\ y := 1$ soddisfa anch'esso la specifica (5), pur non calcolando affatto la divisione intera di x per y. Per impedire una tale situazione dovremmo poterci riferire, nella postcondizione, ai valori x' ed y' che le variabili x ed y avevano nella precondizione (ovvero *prima* dell'esecuzione del programma), e dunque scrivere $\{a \cdot y' + b = x' \wedge b \geq 0 \wedge b < y'\}$. L'apice ($'$) viene comunemente usato a questo scopo nella Logica di Hoare. Nel prosieguo di questa sezione prenderemo comunque per buona la specifica (5).

Esercizio 10.3 Scrivere la specifica del programma $tmp := x; x := y; y := tmp$ che esegue lo *swap* delle variabili x ed y. $\square$

Per dimostrare (5) incominciamo col dedurre:

$$\dfrac{\dfrac{}{x \geq 0 \supset (x = x \wedge x \geq 0)} \qquad \dfrac{}{\{x = x \wedge x \geq 0\}\ b := x;\ \{b = x \wedge b \geq 0\}}\,(assign)}{\{x \geq 0\}\ b := x;\ \{b = x \wedge b \geq 0\}}\,(str)$$

Indichiamo con A l'asserzione $ay + b = x \wedge b \geq 0$. Ora, proprio come sopra:

$$\dfrac{\dfrac{}{(b = x \wedge b \geq 0) \supset (0 \cdot y + b = x \wedge b \geq 0)} \qquad \dfrac{}{\{0 \cdot y + b = x \wedge b \geq 0\}\ a := 0;\ \{A\}}}{\{b = x \wedge b \geq 0\}\ a := 0;\ \{A\}}$$

Possiamo ora applicare (*comp*) alle triple così ottenute:

$$\dfrac{\{x \geq 0\}\ b := x;\ \{b = x \wedge b \geq 0\} \qquad \{b = x \wedge b \geq 0\}\ a := 0;\ \{A\}}{\{x \geq 0\}\ b := x;\ a := 0;\ \{A\}} \tag{6}$$

Deriviamo poi:

$$\{A\}\ \texttt{while}\ (b \geq y)\ \texttt{do}\ (b := b - y;\ a := a + 1)\ \{B\} \tag{7}$$

dove B è l'asserzione $ay + b = x \wedge b \geq 0 \wedge b < y$. Questo si ottiene applicando la regola (*while*):

$$\dfrac{\{ay + b = x \wedge b \geq 0 \wedge b \geq y\}\ b := b - y;\ a := a + 1\ \{ay + b = x \wedge b \geq 0\}}{\{ay + b = x \wedge b \geq 0\}\ \texttt{while}\ (b \geq y)\ \texttt{do}\ (b := b - y;\ a := a + 1)\ \{B\}}$$

La premessa di questa derivazione si ottiene in modo analogo a quanto fatto in precedenza. Infine, otteniamo (5) applicando ancora (*comp*) a (6) e (7).

11 La Correttezza nei Linguaggi Funzionali

Nella Sezione 10.2 abbiamo presentato un sistema formale, la Logica di Hoare, per dimostrare la correttezza di programmi imperativi nel linguaggio *Imp*. Presenteremo ora qualcosa di analogo per il paradigma funzionale, estendendo *Fun*, il cuore dello SML, con un costrutto che permette di definire induttivamente funzioni su N. Avendo trattato la teoria delle algebre induttive nella sua generalità, quanto faremo può essere esteso immediatamente a funzioni su `datatype`, ovvero la forma che tali algebre assumono nello SML. Lo schema induttivo è supportato da diversi linguaggi di programmazione funzionali e consente un approccio formale alla verifica della correttezza di programmi basato sulla *logica equazionale*.

Questa sezione mostra come si possa andare parecchio avanti sul cammino della specifica e verifica di programmi funzionali usando soltanto tre ingredienti:

1. la regola beta,
2. le equazioni per la ricorsione e
3. le proprietà equazionali di riflessività, simmetria e transitività.

11.1 Verifica di specifiche equazionali

Nella sezione 3.1 abbiamo notato che l'insieme N dei numeri naturali forma un'algebra induttiva i cui costruttori sono 0 e *succ*. Abbiamo poi affermato che la natura induttiva di quest'algebra ci consente di definire funzioni su N (e questo vale per qualunque algebra induttiva) mediante un insieme di clausole equazionali, una per ogni costruttore. In questa sezione mostriamo che il fondamento di questo schema di definizione risiede precisamente nel teorema 3.1.

In SML il predicato *is_even* introdotto nella sezione 3.1 si scrive così:

$$\begin{array}{l} \mathit{fun}\ \ \mathit{is_even}\ 0 = \mathit{true} \\ \quad \mid \mathit{is_even}\ (\mathit{succ}\ n) = \mathit{not}\ (\mathit{is_even}\ n); \end{array}$$

Cosa ha a che fare questa definizione con il teorema 3.1? Per rispondere a questa domanda consideriamo la forma che il teorema 3.1 assume quando riferito all'algebra dei numeri naturali:

Proposizione 11.1 *Dati un insieme A, un elemento $a \in A$ ed una funzione $h : A \to A$, esiste una ed una sola funzione $f : N \to A$ tale che $f(0) = a$ e $f(\mathit{succ}\ n) = h(f(n))$.*

Osserviamo ora che le due clausole che definiscono la funzione *is_even* ci indicano un'algebra sui booleani che ha la stessa segnatura di N: la prima ci dà un elemento di *bool*, ovvero la funzione $\mathit{true} : \mathbf{1} \to \mathit{bool}$, mentre la seconda ci dice come calcolare $\mathit{is_even}\ (\mathit{succ}\, n)$ a partire da $\mathit{is_even}\ n$, ovvero la funzione $\mathit{not} : \mathit{bool} \to \mathit{bool}$. La proposizione 11.1 ci dice allora che esiste un'unica funzione

$f : \mathrm{N} \to \mathit{bool}$ che, sostituendo a con *true* e h con *not*, soddisfa le equazioni $f(0) = \mathit{true}$ e $f(\mathit{succ}\ n) = \mathit{not}(f(n))$. Il codice SML definisce dunque *is_even* come l'*unica* funzione f che soddisfa queste equazioni.

Indichiamo con ρ la funzione che, dati a e h come nella proposizione 11.1 ci restituisce l'omomorfismo f; dunque: $f = \rho\, a\, h$. La proprietà di omomorfismo è allora espressa in termini di ρ dalle seguenti equazioni:

$$\begin{array}{rcl} \rho\, a\, h\, 0 & = & a \\ \rho\, a\, h\, (\mathit{succ}\ n) & = & h(\rho\, a\, h\, n). \end{array}$$

Applicando ricorsivamente queste equazioni, abbiamo:

$$\rho\, a\, h\, n = \rho\, a\, h\, \overbrace{(\mathit{succ}\,(\ldots(\mathit{succ}}^{\text{n volte}}\, 0)\ldots) = h(\ldots(h\ a)\ldots),$$

ovvero, $\rho\, a\, h\, n$ itera n volte h a partire da a; ciò che vale a ρ il nome di *iteratore*. Possiamo ora esprimere qualunque numero di Church c_n (vedi Sezione 4.2.3) usando l'iteratore:

$$c_n h\, a = \rho\, a\, h\, n.$$

Questa equazione è stabilita formalmente nell'esercizio **??**.

Sacrificando la chiarezza espessiva della notazione a clausole adottata in SML, possiamo scrivere *is_even* in notazione funzionale usando l'iteratore:

$$\mathit{is_even} \equiv \rho\, \mathit{true}\, \mathit{not}.$$

Questa situazione non è tuttavia ancora del tutto soddisfacente. Consideriamo ad esempio l'algebra induttiva delle liste finite di numeri naturali definita dai costruttori $\mathit{empty} : \mathbf{1} \to \mathit{N\text{-}List}$, che restituisce la lista vuota, e $\mathit{cons} : \mathit{N\text{-}List} \times \mathrm{N} \to \mathit{N\text{-}List}$, che inserisce un naturale in testa ad una lista; ad esempio: $\mathit{cons}((1,3,2),4) = (4,1,3,2)$. Ecco *N-List* in SML sotto forma di datatype (vedi sezione 3.3) ed una funzione $\mathit{nicelist} : \mathrm{N} \to \mathrm{N\text{-}List}$ che, dato un naturale n restituisce la lista $(n-1,\ldots 0)$ dei primi n numeri naturali.

$$\begin{array}{rl} \mathit{datatype}\ \text{N-List} = & \mathit{empty}\ \mathit{of}\ \text{Unit} \\ & \mid \mathit{cons}\ \mathit{of}\ \text{N-List} \times \text{N}; \end{array}$$

$$\begin{array}{l} \mathit{fun}\ \ \mathit{nicelist}\,(0) = \mathit{empty}\,() \\ \quad \mid \mathit{nicelist}\,(\mathit{succ}\ n) = \mathit{cons}(\mathit{nicelist}\,(n), n). \end{array}$$

Contrariamente a quanto avveniva prima, dove per calcolare $\mathit{is_even}\,(\mathit{succ}\ n)$ avevamo solo bisogno di $\mathit{is_even}\,(n)$, ora il calcolo di $\mathit{nicelist}\,(\mathit{succ}\ n)$ richiede, oltre a $\mathit{nicelist}\,(n)$, anche n stesso. Lo stesso avviene per la funzione fattoriale, che in SML si scrive così:

$$\begin{array}{l} \mathit{fun}\ \ \mathit{fact}\ 0 = 1 \\ \quad \mid \mathit{fact}\ (\mathit{succ}\ n) = (\mathit{fact}\ n) * (\mathit{succ}\ n). \end{array}$$

Come fondamento per la definizione di funzioni come *fact* e *nicelist* enunciamo dunque una nuova versione della proposizione 11.1:

Teorema 11.1 *Dati un insieme A, un elemento $a \in A$ ed una funzione $h : A \to (N \to A)$, esiste una ed una sola funzione $f : N \to A$ tale che $f(0) = a$ e $f(succ\ n) = h\,(f(n))\,n$.*

La proposizione 11.1 consegue banalmente dal teorema 11.1: data una funzione $h : A \to A$, la funzione f della proposizione è ottenuta applicando il teorema alla funzione $fn\ x\ n \Rightarrow h\,(x) : A \to (\mathrm{N} \to A)$. La seguente dimostrazione mostra che vale anche l'implicazione inversa.

Il teorema 11.1 deriva dalla proposizione 11.1. (*Dimostrazione*) Dati un insieme A, un elemento $a \in A$ e una funzione $h : A \to (\mathrm{N} \to A)$, la coppia $(a, 0)$ e la funzione $\hat{h} : A \times \mathrm{N} \to A \times \mathrm{N}$ tale che $\hat{h}(x, n) = (h\,x\,n, succ\,(n))$ determinano sull'insieme $A \times \mathrm{N}$ un'algebra come richiesto della proposizione 11.1. Esiste dunque un'unica funzione $\hat{f} : \mathrm{N} \to A \times \mathrm{N}$ tale che $\hat{f}(0) = (a, 0)$ e, per ogni n, $\hat{f}(succ\,(n)) = \hat{h}(\hat{f}(n))$. Chiamiamo $f : \mathrm{N} \to A$ la funzione definita da $f(n) = a'$, dove $\hat{f}(n) = (a', m)$. Per induzione è facile verificare che, per ogni n, $\hat{f}(n) = (f(n), n)$. Dunque, per definizione, $f(0) = a$ e inoltre:

$$\begin{aligned}(f(succ\,(n)), succ\,(n)) &= \hat{f}(succ\,(n)) \\ &= \hat{h}(\hat{f}(n)) \\ &= \hat{h}(f(n), n) \\ &= (h\,(f(n))\,n, succ\,(n)),\end{aligned}$$

da cui consegue $f(succ\,(n)) = h\,(f(n))\,n$, come richiesto.

Rimane da dimostrare l'unicità. Supponiamo che esista una funzione g diversa da f, tale che $g(0) = a$ e $g(succ\,(n)) = h\,(g(n))\,n$. Sia $\hat{g} : \mathrm{N} \to A \times \mathrm{N}$ la funzione tale che $\hat{g}(n) = (g(n), n)$. Dato che $g \neq f$ e $g(0) = f(0)$, esiste un k tale che $g(succ\,(k)) \neq f(succ\,(k))$ da cui: $\hat{g}(succ\,(k)) = (g(succ\,(n)), succ\,(n)) \neq (f(succ\,(n)), succ\,(n)) = \hat{f}(succ\,(k))$. Dunque $\hat{g} \neq \hat{f}$. Inoltre, $\hat{g}$ soddisfa: $\hat{g}(0) = (a, 0)$ e $\hat{g}(succ\,(n)) = (g(succ\,(n)), succ\,(n)) = (h\,(g(n))\,n, succ\,(n)) = \hat{h}(\hat{g}(n))$, contro l'assunzione dell'unicità di $\hat{f}$. □

In generale dunque, per definire funzioni come il fattoriale, ci avvarremo di un operatore di ricorsione *rec* che, dato un elemento di un insieme A, ed una funzione $A \to (\mathrm{N} \to A)$ restituisce l'unica funzione $\mathrm{N} \to A$ che soddisfa le equazioni del Teorema 11.1. Questo operatore compare nel *sistema T* di Gödel [GTL89, capitolo 7] (*Dialectica Interpretation*, 1958). Come abbiamo appena visto esso è matematicamente equivalente all'iteratore ρ; nell'esercizio 11.4 deriveremo le equazioni del primo da quelle del secondo in un sistema formale basato sul linguaggio *Fun* della sezione 4.2 e dotato di *coppie*.

Introduciamo ora Fun_ρ, un linguaggio funzionale ottenuto aumentando *Fun* con il costrutto *rec*:

$$M \quad ::= \quad x \mid fn\ x \Rightarrow M \mid M\,N \mid 0 \mid succ\,(M) \mid rec\,(M, N).$$

Come già abbiamo fatto in 4.2.2, abbiamo eliminato dalla sintassi l'operatore di somma, che può essere qui definito ricorsivamente senza ricorrere ai numeri di Church. Ecco di seguito, oltre alla somma, le definizioni ricorsive del predicato *is_even* e della funzione fattoriale:

$$\begin{array}{rcl} \mathit{plus} & \equiv & \mathit{fn}\ x, y \Rightarrow \mathit{rec}\,(y, \mathit{fn}\ w\ z \Rightarrow \mathit{succ}\ w)\ x \\ \mathit{is_even} & \equiv & \mathit{rec}\,(\mathit{true}, \mathit{fn}\ y\ z \Rightarrow \mathit{not}\ y) \\ \mathit{fact} & \equiv & \mathit{rec}\,(1, \mathit{fn}\ y\ z \Rightarrow y * (\mathit{succ}\ \ z)) \end{array}$$

Presentiamo ora un sistema logico per la verifica formale dei programmi di Fun_ρ. Si tratta di un sistema *equazionale*, dove il solo predicato è l'uguaglianza: $M = N$. Nel presentare gli assiomi denoteremo con $[N/x]M$ il termine ottenuto sostituendo con N le occorrenze *libere* di x in M.

$$\mathit{fn}\ x \Rightarrow M = \mathit{fn}\ y \Rightarrow [y/x]M \tag{8}$$

$$(\mathit{fn}\ x \Rightarrow M)\ N = [N/x]M \tag{9}$$

$$\mathit{rec}\,(M, N)\,0 = M \tag{10}$$

$$\mathit{rec}\,(M, N)\,(\mathit{succ}\ \ L) = N(\mathit{rec}\,(M, N)\,L)\,L. \tag{11}$$

$$\frac{P(0) \qquad P(x) \Rightarrow P(\mathit{succ}\ x)}{\forall\, n\,.\, P(n)} \tag{12}$$

$$\frac{M = N \qquad M = N'}{N = N'} \tag{13}$$

$$\frac{M = N \qquad M' = N'}{M\,N = M'\,N'} \tag{14}$$

$$\frac{M = N}{\mathit{fn}\ x \Rightarrow M = \mathit{fn}\ x \Rightarrow N} \tag{15}$$

Affinché questo sistema sia consistente (ovvero non contraddittorio) dobbiamo fare un'assunzione sull'operatore di sostituzione. Ecco di cosa si tratta. Una sostituzione *brutale* di x con y in $\mathit{fn}\ y \Rightarrow x$ produce $\mathit{fn}\ y \Rightarrow y$, ovvero: $[y/x]\,\mathit{fn}\ y \Rightarrow x \overset{(?)}{\equiv} \mathit{fn}\ y \Rightarrow y$. Se adottassimo questa forma di sostituzione, potremmo però dedurre, ad esempio, che 5 è uguale a 3:

$$\begin{array}{l} 5 \overset{9}{=} (\mathit{fn}\ y \Rightarrow y)\ 5 \overset{9,14}{=} ((\mathit{fn}\ y \Rightarrow (\mathit{fn}\ y \Rightarrow y))\ 3)\ 5 \overset{9,14,15}{=} \\ \quad ((\mathit{fn}\ y \Rightarrow (\mathit{fn}\ x \Rightarrow \mathit{fn}\ y \Rightarrow x)\,y)\ 3)\ 5 \overset{9,14,15}{=} ((\mathit{fn}\ x \Rightarrow \mathit{fn}\ y \Rightarrow x)\ 3)\ 5 \overset{9,14}{=} \\ \quad (\mathit{fn}\ y \Rightarrow 3)\,5 \overset{9}{=} 3. \end{array}$$

In questo esempio, l'occorrenza sottolineata della variabile y nel termine $\mathit{fn}\ y \Rightarrow (\mathit{fn}\ x \Rightarrow \mathit{fn}\ y \Rightarrow x)\,\underline{y}$, che è legata alla lambda astrazione più esterna,

entra, a seguito di una sostituzione brutale, nello scope della lambda astrazione più interna. Si dice in questo caso che $\underline{y}$ viene *catturata*. Ci viene in aiuto l'assioma (8), noto in letteratura col nome di *regola alfa*, il quale dice che i nomi delle variabili legate non contano. Assumeremo dunque che l'operatore di sostituzione rinomini opportunamente eventuali variabili legare onde evitare la cattura. Nel nostro esempio, essendo $fn\ y \Rightarrow x$ alfa equivalente a $fn\ z \Rightarrow x$, si ha:

$$(fn\ x \Rightarrow fn\ y \Rightarrow x)\, y = [y/x]\, fn\ y \Rightarrow x = [y/x]\, fn\ z \Rightarrow x = fn\ z \Rightarrow y,$$

e 5 torna ad essere diverso da 3, come ci piace che sia.

Assumendo che l'operatore di sostituzione eviti la cattura delle variabili, l'equazione (9), nota in letteratura come *regola beta*, è in accordo con la semantica *lazy statica* del linguaggio *Fun* presentata nella sezione 4.2. Questo vuol dire che, per ogni costante n, $M = N = n$ se e solo se $M \rightsquigarrow n$ ed $N \rightsquigarrow n$. (Esercizio: perché abbiamo quantificato su n?)

Fun_ρ include infine le equazioni (10) e (11), che derivano direttamente dal Teorema 11.1, lo schema dell'induzione matematica (12) e tre regole di congruenza. La prima (13) ci dice che la relazione di uguaglianza è simmetrica e transitiva. Da questa regola, in congiunzione con la beta, si deriva anche riflessività: M=M. L'equazione (15) è nota in letteratura come *regola* ξ.

Esercizio 11.1 Mostrare che $M = M$ è derivabile da (9) e (13). Mostrare che valgono simmetria e transitività. □

Concludiamo questa sezione mostrando che le equazioni usate per definire la funzione fattoriale in SML sono derivabili da (10) e (11).

$$\begin{aligned}
fact\ 0 &\equiv rec\,(1, fn\ y\, z \Rightarrow y * (succ\ \ z))\, 0 \overset{(10)}{=} 1 \\
fact\ (succ\ \ n) &\equiv rec\,(1, fn\ y\, z \Rightarrow y * (succ\ \ z))\ (succ\ \ n) \overset{(11)}{=} \\
&\quad (fn\ y\, z \Rightarrow y * (succ\ \ z))(rec\,(1, fn\ y\, z \Rightarrow y * (succ\ \ z))\, n)\, n \overset{(9)}{=} \\
&\quad (rec\,(1, fn\ y\, z \Rightarrow y * (succ\ \ z))\, n) * (succ\ \ n) \equiv \\
&\quad (fact\ \ n) * (succ\ n).
\end{aligned}$$

Esercizio* 11.2 Usando la regola beta, dimostrare la proprietà *associativa* della somma *di Church* definita nella sezione 4.2.2, ovvero:

$$plus\ M\ (plus\ N\ L) = plus\ (plus\ M\ N)\ L.$$

□

Esercizio* 11.3 Verificare che le equazioni usate per definire la funzione *is_even* in SML sono derivabili da (10) e (11). □

Esercizio* **11.4** Supponiamo che Fun_ρ, invece dell'operatore *rec*, avesse l'iteratore ρ con le equazioni della proposizione 11.1. Supponiamo inoltre di avere termini della forma $\langle M, N\rangle$, che denotano coppie di valori, e due operatori di *proiezione*, π_1 e π_2 per i quali valgano le seguenti equazioni:

$$\begin{array}{rcl} \pi_1\langle M, N\rangle & = & M \\ \pi_2\langle M, N\rangle & = & N \\ \langle \pi_1 M, \pi_2 M\rangle & = & M. \end{array}$$

Definiamo *rec* come segue:

$$rec\,(M, N)\,L \equiv \pi_1\,(\rho\,\langle M, 0\rangle\;(fn\;\langle x, m\rangle \Rightarrow \langle N\,x\,m, succ\,(m)\rangle)\,L).$$

Verificare che le equazioni (10) e (11) sono derivabili. □

11.2 Correttezza della moltiplicazione per due

Abbiamo visto nella sezione 11.1 che la definizione a clausole di una funzione in SML ci guida nella sua definizione ricorsiva mediante l'operatore *rec*. Abbiamo anche visto (e verificato per il fattoriale e per la funzione *is_even*) che le clausole SML esplicitano equazioni derivabili dagli assiomi (9), (10) e (11). Useremo dunque queste equazioni per verificare formalmente la correttezza di programmi scritti in Fun_ρ. In questa sezione sviluppiamo un esempio di verifica formale per un semplice programma, *twice*, che restituisce il doppio del numero che riceve in ingresso:

$$twice \equiv rec\,(0, fn\;y\,z \Rightarrow succ\,(succ\;z))$$

In SML:

$$\begin{array}{l} fun\;\;twice\;\;0 = 0 \\ \quad \mid twice\;\;(succ\;n) = succ\,(succ\,(twice\;\;n)). \end{array}$$

Verifichiamo la correttezza di questo programma derivando la seguente specifica: $twice\,n = plus\,n\,n$, dove *plus* è la funzione di somma definita nella sezione 11.1, e che in SML si scrive:

$$\begin{array}{l} fun\;\;plus\;\;0\;\;n = n \\ \quad \mid plus\;\;(succ\;m)\;\;n = succ\,(plus\;\;m\;\;n). \end{array}$$

Deriviamo innanzi tutto la commutatività della somma:

Lemma 11.1 *Per ogni n ed m, $plus\;\;n\;\;m = plus\;\;m\;\;n$.*

Dimostrazione. Scriviamo $P\,(n)$ per la formula $\forall m\,.\,plus\;\;n\;\;m = plus\;\;m\;\;n$. Dimostriamo $\forall n\,.\,P\,(n)$ per induzione su n.

1. Dimostriamo $P\,(0)$, ovvero $\forall m\,.\,plus\;\;0\;\;m = plus\;\;m\;\;0$, per induzione su m.
   (a) Per $m = 0$ l'equazione è l'identità.

(b) Assumendo l'equazione $plus\ 0\ m = plus\ m\ 0$, abbiamo:

$$\begin{aligned} plus\ 0\ (succ\ m) = succ\ m = succ(plus\ 0\ m) = \\ succ\ (plus\ m\ 0) = plus\ (succ\ m)\ 0. \end{aligned}$$

2. Dimostriamo che $P(n)$ implica $P(succ\ n)$. Chiamando α l'ipotesi induttiva, ovvero $\forall m\,.\,plus\ n\ m = plus\ m\ n$, usiamo l'induzione su m per dimostrare la proposizione $\forall m\,.\,plus\ (succ\ n)\ m = plus\ m\ (succ\ n)$.

   (a) Per $m = 0$ la proposizione vale per quanto dimostrato in 1.

   (b) Chiamando β l'ipotesi induttiva, ovvero $plus\ (succ\ n)\ m = plus\ m\ (succ\ n)$, abbiamo:

$$\begin{aligned} plus\ (succ\ n)\ (succ\ m) = succ\ (plus\ n\ (succ\ m)) \stackrel{\alpha}{=} \\ succ(plus\ (succ\ m)\ n) = \\ succ\ (succ\ (plus\ m\ n)) \stackrel{\alpha}{=} \\ succ\ (succ\ (plus\ n\ m)) = \\ succ\ (plus\ (succ\ n)\ m) \stackrel{\beta}{=} \\ succ\ (plus\ m\ (succ\ n)) = \\ plus\ (succ\ m)\ (succ\ n). \end{aligned}$$

□

Esercizio 11.5 Dove viene usata la regola beta (9) nella dimostrazione precedente? □

Teorema 11.2 *Per ogni n, $twice\ n = plus\ n\ n$.*

Dimostrazione. Per induzione. Passo base: $twice\ 0 = plus\ 0\ 0$ vale immediatamente dalle definizioni. Chiamando α il Lemma 11.1 e β l'ipotesi induttiva, ovvero $twice\ n = plus\ n\ n$, abbiamo:

$$\begin{aligned} twice\ (succ\ n) = succ\ (succ\ (twice\ n)) \stackrel{\beta}{=} \\ succ(succ\ (plus\ n\ n)) = \\ succ\ (plus\ (succ\ n)\ n) \stackrel{\alpha}{=} \\ succ\ (plus\ n\ (succ\ n)) = \\ plus\ (succ\ n)\ (succ\ n). \end{aligned}$$

□

12 Conclusioni

In questo corso di *Linguaggi di Programmazione* abbiamo considerato tre paradigmi di programmazione: funzionale, imperativo e ad oggetti, isolando per ciascuno di essi un linguaggio minimale, che può essere considerato come il nucleo di linguaggi "paradigmatici", quali *SML*, "C" e *Java*. Di ciascuno di essi abbiamo studiato la semantica statica (la disciplina dei tipi) e quella dinamica, quest'ultima tanto dal punto di vista *operazionale* che da quello *assiomatico*.

L'approccio formale allo studio dei linguaggi di programmazione fornisce una visione astratta ed allo stesso tempo rigorosa di concetti fondamentali, quali ad esempio quello di *scoping* (statico o dinamico) delle variabili, di *valutazione* (eager o lazy) dei parametri o di *tipo* (eventualmente polimorfo) delle espressioni. Questi concetti si incarnano in forme diverse nei diversi linguaggi di programmazione, ed è dunque importante astrarre dalle forme specifiche per comprendere cosa appartiene al concetto e cosa è idiosincrasia del linguaggio.

In queste nota abbiamo affrontato aspetti *dinamici* (il modello di esecuzione), aspetti *statici* (la disciplina dei tipi) ed aspetti di *specifica* dei linguaggi di programmazione, ed abbiamo esplorato alcune delle relazioni che legano questi aspetti. In particolare, l'uso di metodi formali ci ha consentito di affrontare il tema della *correttezza* dei programmi, il quale, insieme a quello della *complessità*, costituisce un elemento imprescindibile per qualunque corso di studi superiori di informatica.

Un approccio al problema della correttezza è quello del *model checking*, che si basa sull'esplorazione, più o meno esaustiva, dello spazio degli stati di un programma. Il problema di questo approccio è che tale spazio può essere troppo grande per poter essere visitato "con la forza bruta". Ad esempio, nessun model checker attualmente in uso è capace di stabilire che il codice

```
int x = 1;
while (x < 1000000000000)
   x = x+1;
```

produce uno stato in cui il valore di `x` è maggiore di zero, ciò che si dimostra in modo banale nella logica di Hoare.

La logica di Hoare consente di verificare mediante una derivazione formale specifiche scritte nel linguaggio della logica del primo ordine. Ad esempio, la formula $a\,y + b = x \wedge b \geq 0 \wedge b < y$ specifica la correttezza di un programma che calcola quoziente (intero) e resto della divisione di x per y. Ogni logica deve avere un modello (non banale), ovvero un mondo dove sia possibile discriminare ciò che è vero da ciò che non lo è. La logica di Hoare, che parla di programmi, viene tipicamente *validata* rispetto alla semantica operazionale del linguaggio, cosa che abbiamo fatto per il linguaggio *Imp*. Va notato che la semantica operazionale è di per sé sufficiente per verificare *equivalenze* di programmi, ovvero formule di una logica molto elementare, il cui solo predicato è l'uguaglianza. Il

problema dell'equivalenza è cruciale nella costruzione di ottimizzatori di codice. Un tipico esempio di ottimizzazione consiste nello spostare fuori da un ciclo assegnamenti a variabili non usate nel ciclo.

```
while (x < 7) {                y = 1;
   y = 1;              ⟶      while (x < 7)
   x = x+1;}                     x = x+1;
```

Naturalmente l'equivalenza di questi programmi dipende fortemente dalle assunzioni sul linguaggio di programmazione: non sarebbe corretta, ad esempio, se il linguaggio consentisse l'*aliasing* o computazioni *concorrenti*.

La logica di Hoare parla di programmi imperativi. La verifica formale di programmi *funzionali* è spesso limitata, nei libri di testo, a semplici teorie equazionali. Qui abbiamo voluto adottare un approccio più generale per mettere in luce la connessione tra verifica formale ed *inferenza dei tipi*, la seconda essendo un caso speciale della prima. In particolare abbiamo studiato un sistema di deduzione per formule del tipo $p : \{x \mid \phi(x)\}$, la quale asserisce che il programma p soddisfa la proprietà espressa dalla formula ϕ. Restringendo il nostro sistema alle sole formule del tipo $p : \{x \mid x \text{ è un intero}\}$, ovvero $p : int$, abbiamo ottenuto un sistema dei tipi, chiamato *semplice*, per ML_λ, ovvero il nucleo di SML con annotazione esplicita dei tipi.

In genere, i sistemi dei tipi disciplinano i programmi escludendone alcuni che pure rispettano la sintassi "bruta". Nel caso di ML_λ, ad esempio, non è possibile tipare il termine $\perp = (fn\ x \Rightarrow x\,x)\,(fn\ x \Rightarrow x\,x)$ ovvero, non è possibile trovare tre tipi A, B e C tali che

$$\vdash (fn\ x : A \Rightarrow x\,x)\,(fn\ x : B \Rightarrow x\,x) : C$$

sia derivabile nel sistema dei tipi. È possibile dimostrare che, se un termine di ML_λ è tipabile, allora la sua esecuzione nella la semantica operazionale termina (ciò che $\perp$ non fa). Questa proprietà si conserva estendendo il sistema dei tipi semplice con il *polimorfismo*.

Nel linguaggio comune il polimorfismo viene spesso associato ai linguaggi ad oggetti dove, ad esempio, un metodo con parametro di classe C_1 può essere invocato con un argomento di classe C_2 quando questa sia sottoclasse di C_1. Indichiamo questa relazione con $C_2 \leq_{sc} C_1$ e chiamiamo polimorfismo *di sottoclasse* questo genere di disciplina dei tipi. In queste note abbiamo considerato una forma di polimorfismo, nota come *alla ML*, in cui una nozione naturale di "sottotipo", quella di *istanza generica*, deriva direttamente dalla struttura sintattica dei tipi. I termini che denotano tipi possono contenere variabili e queste possono essere quantificate ($\forall$). Recentemente sono state proposte estensioni del linguaggio Java dove le classi possono essere parametrizzate mediante variabili di tipo (*classi generiche*).

Scriviamo $\sigma_1 \leq_{ig} \sigma_2$ per indicare che σ_1 è una istanza generica di σ_2. Se vale questa relazione, allora un programma di tipo σ_2 può essere usato in un contesto

che richiede un programma di tipo σ_1. Va notato che questo è esattamente il contrario di quanto accade con $\leq_{sc}$, Ad esempio, il contesto

$$\mathit{let}\ x = [\,_\,]\ \mathit{in}\ (x\,5)\,(x\ \mathit{true})$$

richiede che alla variabile x venga legata una funzione polimorfa, per esempio di tipo $\forall\, X, Y\,.\, X \to Y \to Y$, e non sarebbe corretto sostituire al segnaposto $[\,_\,]$ la funzione $\mathit{fn}\ x \Rightarrow \mathit{fn}\ y \Rightarrow x + y$ di tipo $\mathit{int} \to \mathit{int} \to \mathit{int}$, istanza generica del precedente.

In queste note abbiamo riservato particolare particolare attenzione al polimorfismo per il forte impatto pragmatico che esso ha sulla programmazione. In particolare, la possibilità di programmare moduli polimorfi, tanto nella forma di sottoclasse che alla ML, favorisce la *riusabilità* e dunque la *mantenibilità* del codice.

Ringraziamenti

Ringrazio i miei studenti che nel corso degli anni hanno contribuito con le loro osservazioni a migliorare la trattazione ed arricchirla di contenuti. Fra gli altri voglio citare:

il dream-team [expand], Marco Mulas, ...

A Soluzione degli esercizi

1.1 No. Differiscono, ad esempio, se applicate ad interi negativi.

3.4 Per i costruttori, vedi l'esercizio 3.8. L'insieme vuoto, sottoinsieme *proprio* dei booleani, è chiuso rispetto a *not*, in violazione al principio di induzione.

3.12

$$N \xrightarrow{leaf} \textit{BN-trees} \xleftarrow{branch} N \times \textit{BN-trees} \times \textit{BN-trees}$$

$$N \xrightarrow{l} S \xleftarrow{b} N \times S \times S$$

Indichiamo con $s_1 \cdot s_2$ la sequenza ottenuta concatenando due sequenze di naturali s_1 ed s_2; ad esempio: $\langle 1,2,3\rangle \cdot \langle 4,5\rangle = \langle 1,2,3,4,5\rangle$. Definiamo:

$$\begin{aligned} &l(n) = \langle n\rangle \\ &b(n, s_1, s_2) = \langle n\rangle \cdot s_1 \cdot s_2 \end{aligned}$$

Verifichiamo che $f : \textit{BN-trees} \to S$, definita come nel testo, sia un omomorfismo:

$$\begin{aligned} &f(\textit{leaf}(n)) = \langle n\rangle = l(n) \\ &f(\textit{branch}(n, t_1, t_2)) = \langle n\rangle \cdot f(t_1) \cdot f(t_2) = b(n, f(t_1), f(t_2)) \end{aligned}$$

3.14 (*Espressioni booleane in SML*)

```
datatype BoolExp = mytrue
                 | myfalse
                 | mynot of BoolExp
                 | myand of BoolExp * BoolExp

    fun valu mytrue = true
       | valu myfalse = false
       | valu mynot b = not (valu b)
       | valu myand (b1, b2) = (valu b1) andalso (valu b2)
```

4.8 (*Operatori logici sui booleani di Church*)

$$\begin{aligned} \textit{and} &\equiv \textit{fn}\ z\, w\, x\, y \Rightarrow z\,(w\, x\, y)\, y \\ \textit{or} &\equiv \textit{fn}\ z\, w\, x\, y \Rightarrow z\, x\,(w\, x\, y) \end{aligned}$$

4.9 (*Concatenazione di liste*)

$$\textit{fn}\ u\, v\, x \Rightarrow v\,(u\, x)$$

4.11 (*Moltiplicazione di numeri di Church*)

$$\textit{fn}\ u\, v\, x \Rightarrow u\,(v\, x)$$

4.12

$$\begin{array}{lcl} S & \equiv & fn\ z \Rightarrow \langle succ\,(P_{2,1}\,z), P_{2,1}\,z\rangle \\ pred & \equiv & fn\ z \Rightarrow P_{2,2}\,(z\,S\,\langle c_0, c_0\rangle) \end{array}$$

11.2

$$\begin{aligned} plus\ M\ (plus\ N\ L) &= fn\ x\,y \Rightarrow M\,x\,((fn\ z\,w \Rightarrow N\,z\,(L\,z\,w))\,x\,y) \\ &= fn\ x\,y \Rightarrow M\,x\,(N\,x\,(L\,x\,y)) \\ &= fn\ x\,y \Rightarrow (fn\ z\,w \Rightarrow M\,z\,(N\,z\,w))\,x\,(L\,x\,y) \\ &= plus\ (plus\ M\ N)\ L. \end{aligned}$$

11.3 L'equazione (10) ci dà immediatamente il caso base della definizione a clausole di *is_even*, mentre (11) ci dà il passo induttivo:

$$\begin{aligned} is_even\ 0 &\equiv rec\,(true, fn\ y\,z \Rightarrow not\ y)\ 0 \overset{(10)}{=} true \\ is_even\ (succ\ \ n) &\equiv rec\,(true, fn\ y\,z \Rightarrow not\ y)\ (succ\ \ n) \overset{(11)}{=} \\ &\quad (fn\ y\,z \Rightarrow not\ y)(rec\,(true, fn\ y\,z \Rightarrow not\ y)\,n)\,n \overset{(9)}{=} \\ &\quad not\ (rec\,(true, fn\ y\,z \Rightarrow not\ y)\,n) \equiv not\,(is_even\ \ n). \end{aligned}$$

11.4 Definiamo $rec\,(M, N)\,L \equiv \pi_1 F(L)$, dove:

$$F(L) \equiv \rho\,\langle M, 0\rangle\ (fn\ \langle x, m\rangle \Rightarrow \langle N\,x\,m, succ\,(m)\rangle)\,L.$$

Notiamo che $F(0) = \langle M, 0\rangle$ e inoltre:

$$\begin{aligned} F(succ\ (n)) &\equiv \rho\,\langle M, 0\rangle\,(fn\ \langle x, m\rangle \Rightarrow \langle N\,x\,m, succ\,(m)\rangle)\,succ\,(n) \\ &= (fn\ \langle x, m\rangle \Rightarrow \langle N\,x\,m, succ\,(m)\rangle)\,F(n) \\ &= (fn\ \langle x, m\rangle \Rightarrow \langle N\,x\,m, succ\,(m)\rangle)\,\langle \pi_1 F(n), \pi_2 F(n)\rangle \\ &= \langle N\,(\pi_1 F(n))\,(\pi_2 F(n)), succ\,(\pi_2 F(n))\rangle. \end{aligned}$$

Lemma A.1 *Per ogni n, $\pi_2\,F(n) = n$*

Dimostrazione. L'equazione $\pi_2\,F(n) = n$ vale per $n = 0$. Infatti:

$$\pi_2\,F(0) = \pi_2\,\langle M, 0\rangle = 0.$$

Inoltre, $\pi_2\,F(n) = n$ implica $\pi_2\,F(succ\,(n)) = succ\,(n)$; infatti:

$$\begin{aligned} \pi_2\,F(succ\ (n)) &= \pi_2\langle N\,(\pi_1 F(n))\,(\pi_2 F(n)), succ\,(\pi_2 F(n))\rangle \\ &= succ\,(\pi_2 F(n)) = succ\,(n). \end{aligned}$$

□

Deriviamo ora le equazioni per *rec*:

$$rec\,(M, N)\,0 \equiv \pi_1\,F(0) = \pi_1\,\langle M, 0\rangle = M.$$

E inoltre:

$$\begin{aligned} rec\,(M,N)\,succ\,(n) &\equiv \pi_1\,F(succ\,(n)) \\ &= \pi_1\,\langle N\,(\pi_1 F(n))\,(\pi_2 F(n)), succ\,(\pi_2 F(n))\rangle \\ &= N\,(\pi_1 F(n))\,(\pi_2 F(n)) = N(rec\,(M,N)\,n)\,n. \end{aligned}$$

$\square$

Riferimenti bibliografici

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, New York, 1996. ISBN 0-387-94775-2.

[AGM92] S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors. *Computational structures*, volume 2 of *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, England, 1992.

[Bar92] Henk Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science*. Oxford University Press, 1992.

[Bro87] L.E.J. Brouwer. Intuitionism and formalism - 1912. In P. Benacerraf and H. Putnam, editors, *Philosophy of Mathematics. Selected Readings*. Cambridge University Press, 1987.

[BS81] S. Burris and H. P. Sankappanavar. *A Course in Universal Algebra*. Graduate Texts in Mathematics. Springer-Verlag, Berlin, 1981. `http://www.math.uwaterloo.ca/~snburris/htdocs/UALG/univ-algebra.pdf`.

[Coq89] Thierry Coquand. On the analogy between propositions and types. In G. P. Huet, editor, *Logical Foundations of Functional Programming*. Addison-Wesley, Reading, MA, 1989.

[GTL89] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge Tracts in Theoretical Computer science. Cambridge University Press, 1989.

[HMU03] J. E. Hopcroft, R. Motwani, and J.D. Ullman. *Automi, Linguaggi e Calcolabilità*. Addison-Wesley, 2003.

[Hof79] Douglas R. Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books, Inc., New York, NY, USA, 1979.

[How80] W. A. Howard. The formulae as types notion of construction. In J. P. Seldin and J. R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, New York, 1980. Unpublished manuscript of 1969.

[Lan93] S. Lang. *Algebra*. Addison-Wesley, 1993.

[Mar84] Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[Qui64] Willard Van Orman Quine. *Word and Object (Studies in Communication)*. The MIT Press, 1964.

[Qui66] Willard Van Orman Quine. *The Ways of Paradox, and other Essays*. Random House, 1966.

[Rey98] J.C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998. ISBN 0521594146.

[Str67] Christopher Strachey. Fundamental concepts in programming languages. Int. Summer School in Computer Programming, Copenhagen, Denmark (unpublished manuscript), 1967.

Indice analitico

alfa-equivalenza, 28, 40
algebra, 14

booleani
    nel lambda calcolo, 36

cattura delle variabili, 86
chiusura
    di un metodo, 53
    di una funzione, 35
Church
    numeri di, 37, 64, 84
Curry-Howard (isomorfismo di), 71

endians (big/little), 4
Epimenide (paradosso di), 41
espressioni
    condizionali, 37
estensionalità, 41

induzione
    completa, 18
    matematica, 13, 86
    principio di, 13, 15
    strutturale, 18
intuizionismo, 71
iterazione, 84

König (lemma di), 21

Lambek, J. (lemma di), 16

Martin-Löf, Per, 71
metodo
    dispatching dinamico, 40, 53
Mosè, 31

Peano, Giuseppe, 11

Quine, Willard Van Orman, 41, 58

referential transparency, 41, 58
replicatori, 58
Russel, Bertrand, 56

scope, 28
segnatura algebrica, 14
semantica, 4
    assiomatica, 7
    denotazionale, 6, 56
    operazionale, 6
sequenze
    nel lambda calcolo, 37
sintassi
    astratta, 19

teoria algebrica, 18
tipo
    dipendente, 70
    ordine di un, 60

variabile
    libera/legata, 28, 34