1 Il mini linguaggio funzionale

1.1 Ancora su call-by-name vs call-by-value

La strategia call-by-value ha il vantaggio che evita di ripetere la valutazione degli stessi argomenti, e.g. dato $M = \mathsf{fn}\, x.x{+}x{+}x{+}x\ N$ conviene valutare il termine N usa sola volta prima di crearne 4 copie a seguito della sostituzione al posto di x. D'altra parte la strategia call-by-name evita di valutare parametri che non sono usati, e.g. $M' = \mathsf{fn}\, x.3{+}7\ N$.

Come ulteriore esempio, si consideri il termine Ω, che abbiamo visto non raggiungere mai un valore finale. Sia $M_1 = \mathsf{fn}\, x.\mathsf{fn}\, y.x$, cioè M_1 è la funzione che prende due parametri e restituisce il primo. Allora il termine $M_1\ 3\ \Omega$ riduce al valore 3 usando una strategia call-by-name, mentre usando una strategia call-by-value la valutazione non termina.

EXERCISE 1.1. Consideriamo le seguenti definizioni in Scala:

```
def square(x:Int):Int = x*x
def sumOfSquare(x:Int,y:Int):Int = square(x)+square(y)
```

- Descrivere i passi di riduzione dell'espressione `sumOfSquare(3,4)` secondo una strategia call-by-value analoga a quella definita nella sezione precedente. Descrivere inoltre la riduzione della stesa espressione secondo la strategia call-by-name.
- Descrivere i passi di riduzione dell'espressione `sumOfSquare(3,2+2)` secondo le strategie call-by-value e call-by-name.

□

EXERCISE 1.2. Si consideri la seguente definizione in Scala:

```
def test(x:Int, y:Int):Int = x* x
```

confrontare la velocità (cioè il numero di passi) di riduzione delle seguenti espressioni secondo le strategie CBV e CBN, indicando quale delle due è piú veloce

- `test(2,3)`
- `test(3+4,8)`
- `test(7,2*4)`
- `test(3+4,2*4)`

□

Consideriamo un esempio più concreto: implementiamo un costrutto `assert` per le asserzioni[1]:

```
var assertionsEnabled = true

def assert(predicate:Boolean)=
     if(assertionsEnabled && !predicate)
        throw new AssertionError

....assert(saldoConto()>0)...
```

La funzione `assert` consulta il flag che indica se le asserzioni sono abilitate. Se il flag è attivo, la funzione verifica che il predicato passato come parametro abbia un valore di verità `true`, in caso contrario solleva un'eccezione. Nel caso le asserzioni siano state disabilitate, la funzione non fa nulla.

Si osservi che il senso del costrutto `assert` è rispettato solo nel caso di una semantica call-by-name. Infatti, quando si valuta la chiamata della funzione, cioè l'istruzione `assert(saldoConto()>0)`, nel caso ci sia una strategia call-by-value, il predicato che viene passato come parametro viene subito valutato, senza verificare prima che il flag `assertionsEnabled` sia effettivamente positivo. In Scala per default si usa la strategia call-by-value, è possibile peró indicare che un parametro deve essere passato by name, ad esempio correggendo la funzione `assert` in `def assert(predicate: =>Boolean)`

[1] Tratto dal libro Programming in Scala, Artima eds.

EXERCISE 1.3. Definire in Scala una funzione `and` che si comporta come il costrutto logico `x&&y`. □

Osserviamo infine che un teorema classico dimostra (nel λ-calcolo) che la strategia di valutazione scelta non influisce sul valore terminale raggiunto da un termine. In altre parole, per i termini che evolvono ad un valore finale, tutte le strategie di valutazione confluiscono allo stesso valore finale. Si dimostra inoltre che se la valutazione di un termine termina con la strategia call-by-value, allora termina anche con la strategia call-by-name, mentre il viceversa non vale, e.g. $\mathsf{fn}\, x.\mathsf{fn}\, y.x \;\; 3 \;\; \Omega$.

2 I tipi semplici

I sistemi di tipi rappresentano un tool per *ragionare* sui programmi, garantendo che il comportamento di un programma rispetta una specifica data o un comportamento atteso. Più precisamente, lo scopo di un sistema di tipi per un linguaggio di programmazione è ottenere il seguente risultato:

> Se M è un programma ben tipato, allora M non produce errori runtime.

Il controllo che il programma M sia ben tipato è un controllo statico, effettuato cioè sul programma sorgente. La proprietà statica ha dunque un impatto sul comportamento dinamico del programma, che non produce errori durante la sua esecuzione. In questo senso quindi i sistemi di tipi sono tecniche di analisi statica dei programmi. Rispetto ad altre tecniche, i sistemi di tipi rappresentano un metodo formale "lightweight", facilmente automatizzabile. Queste tecniche stanno infatti alla base di checker automatici inseriti in compilatori, linkers e analizzatori di programmi, quindi spesso sono usati in modo trasparente al programmatore.

Si osservi inoltre che il risultato evidenziato sopra non può essere invertito, cioè i sistemi di tipi possono provare l'*assenza* di comportamenti errati, ma non la presenza di errori. Ci sono infatti programmi che non evolvono in errori ma che non sono ben tipati. In altre parole, i tipi fanno delle approssimazioni statiche sul comportamento dinamico di un programma, effettuando un'analisi corretta ma non completa (come in genere accade con le tecniche di analisi statica).

Abbiamo definito un linguaggio: la sintassi dei termini (cioè la sintassi dei programmi) e quella dei valori terminali, la semantica operazionale, cioè relazione di valutazione che indica come evolve runtime un programma. Definiamo ora quali sono i comportamenti errati che vogliamo elimiare con i tipi. I programmi errati sono quelli che *raggiungono* uno stato stuck, che non è uno stato finale ma non c'è nessuna regola di valutazione che ci dice come continuare la computazione. Di conseguenza, useremo i tipi per classificare le frasi di un programma a seconda del "tipo" di valore che le frasi computano. Ad esempio vogliamo dire staticamente che $\mathsf{if}\,\mathsf{true}\,\mathsf{then}\,\mathsf{false}\,\mathsf{else}\,\mathsf{true}$ ha tipo Bool e $(1+3)-(2-5)$ ha tipo Nat, senza valutarli. Alla fine arriveremo a dimostrare il seguente teorema:

Teorema di Safety: Se M è ben tipato allora $M \not\longrightarrow^* M'$ con M' stuck.

La dimostrazione si baserà sui due teoremi:

- Teorema di *Progressione*: un termine ben tipato non è stuck, cioè o è un valore (uno stato finale) oppure può essere ridotto ad un altro termine secondo le regole di transizione.
- Teorema di *Preservazione*: se un termine ben tipato si valuta in un altro termine, allora anche il termine risultante è ben tipato.

Sintassi dei tipi I tipi servono per classificare i **valori**, che nel nostro linguaggio sono le costanti, intere e booleane, e le funzioni. Quindi la sintassi dei tipi per il nostro linguaggio è la seguente:

Tipi T	::=	Bool	tipo dei valori booleani
	|	Nat	tipo dei valori interi
	|	$T \to T$	tipo delle funzioni

Ad esempio $\mathsf{Bool} \to \mathsf{Bool}$ è il tipo delle funzioni che prendono un booleano e restituiscono un booleano, mentre $(\mathsf{Bool} \to \mathsf{Bool}) \to (\mathsf{Bool} \to \mathsf{Bool})$ è il tipo delle funzioni che prendono come argomento una funzione da booleani in booleani, e restituiscono una funzione dello stesso tipo.

Definiamo ora una *relazione di typing* $M : T$ che classifica il termine M a seconda del tipo del valore in cui M si valuterà. Nota che non vogliamo dare un tipo proprio a quei termini che non si valutano ad alcun valore, che cioè raggiungono uno stato stuck.

Come faciamo a sapere che tipo di argomento passare alla funzione? In un linguaggio con type system si può scegliere se usare una sintassi tipata per i termini oppure no: possiamo annotare l'argomento di una funzione con il suo tipo (come ad es. in Java/C++), i.e. $\mathsf{fn}\, x{:}T.M$ oppure non farlo e lasciare che il tipo dell'argomento sia inferito dalla macchina analizzando il corpo della funzione (come in ML). Scegliamo la prima possibilità.[2] Inoltre, se M contiene una variabile libera x,e.g. $M = x + 1$, che tipo ha la variabile? La relazione di typing si baserà su un insieme di assunzioni di tipo es: $x : \mathsf{Nat} \vdash x + 1 : \mathsf{Nat}$. Indichiamo con Γ un *contesto*, cioè una lista di dichiarazioni di variabili, tra loro distinte. Indichiamo con $Dom(\Gamma)$ l'insieme delle variabili che appaiono in Γ. I giudizi di tipo saranno della forma $\Gamma \vdash M : T$ che indica che "in un contesto Γ il termine M ha tipo T". I giudizi di tipo sono derivabili con il seguente sistema di regole.

(TRUE)
$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}$$

(FALSE)
$$\frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}$$

(NAT)
$$\frac{}{\Gamma \vdash n : \mathsf{Nat}}$$

(SUM)
$$\frac{\Gamma \vdash M : \mathsf{Nat} \quad \Gamma \vdash N : \mathsf{Nat}}{\Gamma \vdash M + N : \mathsf{Nat}}$$

(MINUS)
$$\frac{\Gamma \vdash M : \mathsf{Nat} \quad \Gamma \vdash N : \mathsf{Nat}}{\Gamma \vdash M - N : \mathsf{Nat}}$$

(IFTHENELSE)
$$\frac{\Gamma \vdash M_1 : \mathsf{Bool} \quad \Gamma \vdash M_2 : T \quad \Gamma \vdash M_3 : T}{\Gamma \vdash \mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3 : T}$$

(VAR)
$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

(FUN)
$$\frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \mathsf{fn}\, x{:}T_1.M : T_1 \to T_2}$$

(APP)
$$\frac{\Gamma \vdash M : T_1 \to T_2 \quad \Gamma \vdash N : T_1}{\Gamma \vdash M\ N : T_2}$$

La regola (FUN) ha come premessa un giudizio che dice che il termine M ha tipo T_2 assumendo che la variabile che compare in M abbia tipo T_1. Questa assunzione viene aggiunta al contesto, assumendo che questa aggiunta produca un nuovo contesto ben formato. Ciò succede quando la variabile x non compare già nel dominio di Γ. Grazie all'α-conversione, possiamo sempre rinominare la variabile legata del termine M in modo tale che sia distinta dalle altre variabili che compaiono in Γ.

Esempi: dare la derivazione dei seguenti giudizi di tipo:

- $\varnothing \vdash \mathsf{if\ true\ then}\ 5 + 7\ \mathsf{else}\ 2 : \mathsf{Nat}$,
- $\varnothing \vdash \mathsf{fn}\, x{:}T.x : T \to T$,
- $\varnothing \vdash (\mathsf{fn}\, x{:}\mathsf{Bool}.x)\ \mathsf{true}\ :\ \mathsf{Bool}$,
- $f : \mathsf{Bool} \to \mathsf{Bool} \vdash f\ (\mathsf{if\ false\ then\ true\ else\ false})\ :\ \mathsf{Bool}$
- $f : \mathsf{Bool} \to \mathsf{Bool} \vdash \mathsf{fn}\, x{:}\mathsf{Bool}.f\ (\mathsf{if}\ x\ \mathsf{then\ false\ else}\ x)\ :\ \mathsf{Bool} \to \mathsf{Bool}$.

Nota che abbiamo usato le regole per fare *type checking*. I giudizi di tipo sono delle asserzioni sul tipo dei programmi, le regole di tipo sono implicazioni tra giudizi di tipo, le derivazioni sono deduzioni basate sulle regole di tipo.

Definition 2.1. Diciamo che un termine *chiuso* M è *ben tipato* se esiste un tipo T tale che il giudizio $\varnothing \vdash M : T$ sia derivabile. □

[2] Si noti che la semanica operazionale del linguagio tipato è analoga a quella del linguaggio non tipato: l'aggiunta della notazione di tipo non interferisce in alcun modo nella valutazione di un termine.

Come altre tecniche di analisi statica, anche i sistemi di tipo in genere sono imprecisi, o meglio incompleti, esistono cioè dei termini ragionevolmente corretti, a cui però non si riesce a dare un tipo. Ad esempio, al termine $\mathsf{if\ true\ then\ 0\ else\ false}$ non si riesce a dare un tipo (staticamente) anche se sicuramente questo termine si valuterà ad un numero. I tipi quindi fanno delle approssimazioni corrette. Si osservi che ad esmpio anche in C/C++ tutti i rami if/else del corpo di una funzione devono restituire valori dello stesso tipo. Ciò è importante perché se una funzione restituisse valori di tipo diverso, l'uso che il chiamante potrebbe fare di questa funzione sarebbe più limiato, o più complesso. Più che una limitazione, questa regola agevola quindi la scrittura di programmi più "maneggevoli" e più semplici da analizzare.

EXERCISE 2.2. Trovare un contesto Γ tale che $\Gamma \vdash f\ x\ y\ : \mathsf{Bool}$ sia derivabile. □

EXERCISE 2.3. Il giudizio $\Gamma \vdash x\ x\ : T$ è derivabile? Se sì trovare una derivazione per qualche Γ, T, altrimenti provare che non è derivabile. □