3 Estensioni del linguaggio

3.1 Il tipo Unit

Nei linguaggi funzionali come ML,Haskell, ed anche in Scala, esiste il tipo primitivo Unit, che ha un unico valore, che noi rappresentiamo con unit (in ML/Scala si indica con `()`, in Haskell `()` rappresenta sia il tipo che il suo unico valore). Nei linguaggi funzionali ogni funzione deve restituire un valore; quando la funzione provoca solo dei side-effects. e.g., degli assegnamenti a delle variabili o operazione di I/O, il tipo di ritorno di questa funzione è Unit. Ad esempio in Scala la funzione (metodo) di stampa è `println(x:Any):Unit`, inoltre l'assegnamento resituisce sempre `()` (mentre il C/Java restituisce il valore assegnato).

Introdurre questo nuovo tipo nel nostro linguaggio/sistema di tipi, significa aggiungere il termine unit alla sintassi del linguaggio e alla sintassi dei valori. Non serve estendere la semantica del linguaggio con nuove regole di valutazione, va invece aggiunto il tipo Unit alla sintassi dei tipi, e una nuova regola di tipo:

$$M := \ldots \mid \mathsf{unit} \qquad v := \ldots \mid \mathsf{unit} \qquad T := \ldots \mid \mathsf{Unit}$$

$$\frac{}{\Gamma \vdash \mathsf{unit} : \mathsf{Unit}} \ (\textsc{Unit})$$

Come esempio osserviamo come il termine unit può essere usato per ritardare la valutazione di una funzione. Usando una semantica call-by-value e un linguaggio con funzioni higher order possiamo riderfinire il costrutto per le asserzioni nel modo seguente

```
var assertionsEnabled = true

def assert(predicate: Unit -> Boolean)=
     if(assertionsEnabled && (predicate(unit))==false)
        throw new AssertionError

....assert(fun x:Unit.saldoConto()>0)...
```

In questo modo il predicato diventa una funzione, non più un'espressione booleana. Anche nella semantica call-by-value, il parametro attuale di `assert` è già un valore, quindi non viene valutato, mentre il vero predicato boolean si valuta solo dopo aver applicato il valore `unit` al parametro `predicate`, cosa che avviene quindi solo se il flag indica che le asserzioni sono attive.

Quindi, più in generale, mentre il termine $(\mathsf{fn}\, x{:}T.M)\ N$ valuta N prima di invocare la funzione, il termine

$$(\mathsf{fn}\, y{:}\mathsf{Unit}{\to}T.\ M\{x{:=}(y\ \mathsf{unit})\})\ \ \mathsf{fn}\, z{:}\mathsf{Unit}.N$$

invoca subito la funzione e valuta N solo quando effettivamente viene usato.

Si ottiene cosí l'effetto della chiamata lazy in un linguaggio con semantica call-by-value *con in più* funzioni higher order e tipo Unit. Vale la pena osservare che nei linguaggi imperativi la semantica lazy è particolarmente subdola, perché, data una chiamata di funzione, non è facile capire esattamente quando avvengono i side effects provocati dalla valutazione effetiva dei parametri attuali.

Precisazione: La corretta sintassi Scala è

```
def assert(predicate: ()=>Boolean)=
     if(assertionsEnabled && (predicate())==false)
        throw new AssertionError

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

e lo stesso esempio in Java 8 si scrive

```
public static void assert(predicate: Supplier<Boolean>) {
     if(assertionsEnabled && (predicate.get())==false)
        throw new AssertionError;
```

```
     else
        return;
}

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

Si ricordi infine che in Scala resta più naturale e dunque più corretto, usare il passaggio dei parametri by name, mentre in Java 8 non c'è questa scelta.

Come ultimo esempio indichiamo come si può codificare il costrutto iterativo while con sole variabili e funzioni higher-order.

```
def WHILE(condition: =>Boolean, command: =>Unit) :Unit =
  if(condition){
     command
     WHILE(condition, command)
  }
  else ()

var a=0
WHILE(a<4, {print(a); a=a+1})
```

L'ultima chiamata di funzione produce effettivamente la stampa di quattro valori interi. Si noti come la semantica call-by-name sia indispensabile per permettere che la condizione sia ri-valutata ed il comando ri-eseguito ad ogni chiamata ricorsiva. Inoltre, poiché si tratta di una funzione *ricorsiva-in coda*, la sua implementazione puó essere "efficiente quanto" quella del costrutto nativo.

3.2 Coppie, tuple e record

La struttura dati più semplice contenuta in un linguaggio, anche in Scala, è la coppia, che si generalizza al concetto di tupla e di record, i.e. tupla etichettata (strutture del C). I nuovi termini del linguaggio sono le coppie di termini (M_1, M_2) e i due operatori di proiezione $M.\_1$ e $M.\_2$. I nuovi valori sono le coppie di valori: (v_1, v_2) e il nuovo tipo è il cosiddetto *tipo prodotto* $T_1 * T_2$. Diamo la semantica delle coppie con le seguenti regole, che corrispondono ad una strategia di valutazione "da sinistra a destra", e che richiede di valutare completamente entrambi i termini dela coppia prima di "usare" la coppia ad esempio in una proiezione o come parametro di una funzione.

(PAIR 1)
$$\frac{}{(v_1, v_2).\_1 \longrightarrow v_1}$$

(PAIR 2)
$$\frac{}{(v_1, v_2).\_2 \longrightarrow v_2}$$

(PROJECT 1)
$$\frac{M \longrightarrow M'}{M.\_1 \longrightarrow M'.\_1}$$

(PROJECT 2)
$$\frac{M \longrightarrow M'}{M.\_2 \longrightarrow M'.\_2}$$

(EVAL PAIR 1)
$$\frac{M_1 \longrightarrow M_1'}{(M_1, M_2) \longrightarrow (M_1', M_2)}$$

(EVAL PAIR 2)
$$\frac{M_2 \longrightarrow M_2'}{(v, M_2) \longrightarrow (v, M_2')}$$

(PAIR)
$$\frac{\Gamma \vdash M_1 : T_1 \quad \Gamma \vdash M_2 : T_2}{\Gamma \vdash (M_1, M_2) : T_1 * T_2}$$

(PROJECT 1)
$$\frac{\Gamma \vdash M : T_1 * T_2}{\Gamma \vdash M.\_1 : T_1}$$

(PROJECT 2)
$$\frac{\Gamma \vdash M : T_1 * T_2}{\Gamma \vdash M.\_2 : T_2}$$

EXERCISE 3.1. Discutere quali altri regole/strategie di riduzione sono possibili per i termini coppia. □

EXERCISE 3.2. Scrivere la valutazione dei termini $(4 - 1,\ \mathsf{if\ true\ then\ false\ else\ false}).\_1$ e $(\mathsf{fn}\, x : \mathsf{Nat} * \mathsf{Nat}.x.\_2)\ (4 - 2,\ 3 + 1)$. □

Le coppie di valori si generalizzano alle tuple di valori, indicate con $(M_i^{i\in 1...n})$. Si noti che esiste anche la tupla con un solo elemento, e.g., (5), che è ben diversa dal semplice elemento 5: la differenza tra questi due termini sta nelle operazioni che si possono applicare su di loro, ad esempio $(5)+1$ è un termine scorretto (mal tipato e stuck) mentre $(5).\_1+1$ è corretto (ben tipato ed evolve al valore finale 6.

Vediamo un esempio concreto di uso del tipo coppia: nel linguaggio Scala la classe `Array[T]` contiene il metodo `partition` che prende come parametro un predicato, cioè una funzione che restituisce un valore booleano, e restituisce una coppia di array che contengono rispettivamente gli elementi dell'array di invocazione che soddisfano, risp. non soddisfano, il predicato:

```
def partition(p: T=>Boolean): (Array[T],Array[T]) // metodo di Array[T]

val people: Array[Person] = ... //inizializzazione
val (minors, adults) = people.partition(x => x.age<18)
```

Si osservi come il tipo di ritorno di `partition` sia il tipo coppia e come la variabile `(minors,adults)` di tipo (inferito) coppia sia inizializzata tramite un meccanismo di pattern matching. Si osservi inoltre come il parametro passato al metodo `partition` sia una funzione anonima (anonymous function literal) del tipo corretto.

Se agli elementi delle tuple si aggiungono delle etichette (prese da un insieme predefinito di etichette, che assumiano esistere), si ottengono i **record (strutture)**, ad esempio:

$$\begin{array}{lcl} \{\mathsf{pippo}=\mathsf{true},\ \mathsf{pluto}=2\} & : & \{\mathsf{pippo}:\mathsf{Bool},\ \mathsf{pluto}:\mathsf{Nat}\} \\ \{\mathsf{numConto}=1234,\ \mathsf{saldo}=1000\} & : & \{\mathsf{numConto}:\mathsf{Nat},\ \mathsf{saldo}:\mathsf{Nat}\} \\ \{\mathsf{a}=3,\ \mathsf{b}=\mathsf{fn}\,x.\mathsf{fn}\,y.x+y\} & : & \{\mathsf{a}:\mathsf{Nat},\ \mathsf{b}:\mathsf{Nat}\to\mathsf{Nat}\to\mathsf{Bool}\} \end{array}$$

L'ultimo record indicato assomiglia ad un oggetto con un campo dati a e un "metodo" b. In realtà il campo b è una funzione che non può accedere direttamente al campo a, mentre un metodo di un oggetto ha accesso allo stato dell'oggetto. Alla fine della sezione discuteremo un po' su come rappresentare gli oggetti tramite record. I record assomigliano superficialmente ad oggetti JSON, che usano stringhe per etichettare i diversi campi. In questo linguaggio però i campi dato possono contenere anche funzioni mentre non è possibile definire array.

Da un punto di vista formale, il linguaggio va eseso nel modo seguente:

$$M ::= \ldots \mid \{\ell_i = M_i{}^{i\in 1..n}\} \mid M.\ell \qquad v ::= \ldots \mid \{\ell_i = v_i{}^{i\in 1..n}\} \qquad T ::= \ldots \mid \{\ell_i : T_i{}^{i\in 1..n}\}$$

La sintassi dei record è la seguente: $\{\ell_i = M_i{}^{i\in 1..n}\}$, in cui si assume che le n etichette siano distinte. Le proiezioni sono definite usando il nome dell'etichetta corrispondente: $M.\ell$. La defeinizione delle variabili libere e della sostituzione si estende in modo naturale ai nuovi termini perché la loro introduzione non aggiunge alcun binder per le variabili. La semantica è data dalle seguenti regole:

$$\text{(SELECT)}\ \frac{j\in 1..n}{\{\ell_i = v_i{}^{i\in 1..n}\}.\ell_j \longrightarrow v_j} \qquad \text{(EVAL SELECT)}\ \frac{M\longrightarrow M'}{M.\ell\longrightarrow M'.\ell}$$

$$\text{(EVAL RECORD)}\ \frac{M_j\longrightarrow M'_j}{\{\ell_i = v_i{}^{i\in 1..j-1}, \ell_j = M_j, \ell_k = M_k{}^{k\in j+1..n}\}\longrightarrow\{\ell_i = v_i{}^{i\in 1..j-1}, \ell_j = M'_j, \ell_k = M_k{}^{k\in j+1..n}\}}$$

Per semplicità assumiamo (diversamente da alcuni linguaggi di programmazione) che l'ordine dei campi di un record sia rilevante, assumiamo cioè che i termini $\{\mathsf{a}=2,\ \mathsf{b}=1\}$ e $\{\mathsf{b}=1, \mathsf{a}=2\}$ siano distinti. Diamo infine le regole di tipo per i record:

$$\text{(TYPE RECORD)}\quad \frac{\forall i \in 1..n \;\; \Gamma \vdash M_i : T_i}{\Gamma \vdash \{\ell_i = M_i \;^{i\in 1..n}\} : \{\ell_i : T_i \;^{i\in 1..n}\}} \qquad \text{(TYPE SELECT)}\quad \frac{\Gamma \vdash M : \{\ell_i : T_i \;^{i\in 1..n}\} \;\; j \in 1..n}{\Gamma \vdash M.\ell_j : T_j}$$

Si osservi che, estendendo il linguaggio con i termini coppie e/o records, il teorema di safety va ridimostrato.

Theorem 3.3 (Safety)**.** *Se M è un termine chiuso e ben tipato, allora non evolve in un termine stuck.* □

Lo statement del teorema rimane identico, anche la definizione di termine stuck rimane identica, ma il nuovo linguaggio ha termini in più, regole di semantica e di typing in più, e anche termini stuck in più. Ad esempio $\mathsf{true}.\_2$ oppure $\{\mathsf{a} = M_1, \mathsf{b} = M_2\}.\mathsf{c}$ sono nuovi "errori di programmazione".

Possiamo in qualche modo simulare il comportamento degli oggetti? Proviamo ad aggiungere un metodo per depostare soldi nel conto corrente:

- Sia $obj = \{\mathsf{numConto}{=}1234,\ \mathsf{saldo}{=}1000,\ \mathsf{deposita} = \mathsf{fn}\, x_{this}{:}T_o.\mathsf{fn}\, y{:}\mathsf{Nat}.x_{this}.\mathsf{saldo}{+}y\}$ allora $obj.\mathsf{deposita}(obj)(50)$, dove ho passato esplicitamente l'oggetto stesso come valore del parametro $this$ deposita 50 euro nel conto? No: $obj.\mathsf{deposita}(obj)(50) \longrightarrow^* obj.\mathsf{saldo} + 50 \longrightarrow^* 1050$
- In un linguaggio puramente funzionale non ci sono nemmeno side effects sullo stato degli "oggetti", quindi dobbiamo far restituire al metodo deposita un nuovo oggetto con lo stato aggiornato:

$$\begin{aligned} obj = \{&\mathsf{numConto} = 1234,\ \mathsf{saldo} = 1000,\\ &\mathsf{deposita} = \mathsf{fn}\, x_{this}{:}T_o.\mathsf{fn}\, y{:}\mathsf{Nat}.\{\quad \mathsf{numConto} = x_{this}.\mathsf{numConto},\\ &\qquad\qquad\qquad\qquad\quad\ \mathsf{saldo} = x_{this}.\mathsf{saldo} + y,\\ &\qquad\qquad\qquad\qquad\quad\ \mathsf{deposita} = x_{this}.\mathsf{deposita}\}\ \} \end{aligned}$$

  allora $obj.\mathsf{deposita}(obj)(50) \longrightarrow^* \{\mathsf{numConto} = obj.\mathsf{numConto},\ \mathsf{saldo} = obj.\mathsf{saldo}{+}50,\ \mathsf{deposita} = obj.\mathsf{deposita}\} \longrightarrow^* \{\mathsf{numConto} = 1234,\ \mathsf{saldo} = 1050,\ \mathsf{deposita} = \mathsf{fn}\, x_{this}.\mathsf{fn}\, y.\{...\}\}$ come volevasi.
- Qual è il tipo T_o di obj? $T_o = \{\mathsf{numConto} : \mathsf{Nat},\ \mathsf{saldo} : \mathsf{Nat},\ \mathsf{deposita} : T_o \to \mathsf{Nat} \to T_o\}$ che è un tipo ricorsivo, non ammesso nel nostro sistema di tipi semplici. Nota che un tipo ricorsivo serve ogni volta che si passa/si usa il parametro $this$ e ogni volta che si vuole produrre un oggetto con stato modificato. Esistono linguaggi fondazionali ad hoc per gli oggetti es. ς-calcolo di Abadi e Cardelli[3] è un linguaggio object-based con semantica funzionale, e.g $\{\mathsf{numConto} = 1234,\ \mathsf{saldo} = 1000,\ \mathsf{deposita} = \varsigma(x)\mathsf{fn}\, y.x.\mathsf{saldo}{+}{=}y\}$ restituisce un nuovo oggetto con il saldo aggiornato. Studieremo invece in seguito il linguaggio Featherweight Java, class-based, sia con semantica funzionale che imperativa. L'uso dei cosiddetti *Functional Objects* è sempre piú diffuso nei linguaggi di programmazone moderni, poiché permette di combinare efficacemente i vantaggi della programmazione ad oggetti con quelli della programmazione funzionale.

EXERCISE 3.4. Dimostrare il teorema di safety per il linguaggio contenente interi, booleani, funzioni e records. □

[3] A theory of objects. M. Abadi, L. Cardelli. Springer, 1996.