3.3 Variant types

Si consideri un nodo in un albero binario: può essere una foglia *oppure* un nodo interno con due figli. Analogamente, all'interno di un compilatore un nodo di un albero sintattico (parse tree) può rappresentare una variabile, oppure un simbolo di funzione, oppure una costante,... In questi casi, cioè quando un termine ha un tipo scelto tra un insieme di possibili tipi, si usano i *variant types*. Facciamo un esempio nel caso in cui la scelta si riduca a due possibili tipi.

Rappresentiamo un address-book che contenga sia indirizzi fisici che indirizzi virtuali.

```
T_IndFisico = {nome: String, indir: String}
T_IndVirtual = {login:String, email:String}
```

Abbiamo rappresentato gli indirizzi, sia fisici che virtuali, come due tipi record, ognuno con due campi di tipo stringa. Per comodità abbiamo chiamato questi tipi record rispettivamente `T_IndFisico` e `T_IndVirtual`. Definiamo ora un nuovo tipo, a partire dai due tipi record definiti sopra, e per comodità lo chiamiamo `T_Indirizzo`.

```
T_Indirizzo = <fisico: T_IndFisico, virtuale: T_IndVirtual>
```

Il tipo `T_Indirizzo` è un *variant type* che permette di manipolare in modo uniforme elementi di tipo `T_IndFisico` e `T_IndVirtual`. Le due opzioni sono distite tramite le etichette `fisico` e `virtuale`. Un esempio di termine di tipo `T_Indirizzo`, è il seguente:

```
<fisico = {nome="pippo", indir="topolinia"}> : T_Indirizzo
```

Sui termini di tipo variant si definisce l'operazione di pattern matching, che permette di usare il termine distinguendo tra le possibili opzioni a seconda delle diverse etichette. Ad esempio la seguente funzione prende un indirizzo (fisico o virtuale) e ne estrae il nome:

```
def getNome(a:T_Indirizzo) : String = a match {
     case  <fisico=x> => x.nome
     case  <virtuale=y> => y.login
}

def getAll(a:List[T_Indirizzo]) : List[String] = {
    var l = new List[String]
    for (x <- a) l.add(getNome(x))
}
```

Più formalmente, aggiungere i variant types significa estendere il linguaggio nel modo seguente, dove ℓ_i sono etichette:

$$M ::= ... \mid \langle \ell = M \rangle \mid M \text{ match } \{\text{case } \ell_i = x_i \Rightarrow M_i\,^{i\in 1..n}\} \quad v ::= ... \mid \langle \ell = v \rangle \quad T ::= ... \mid \langle \ell_i : T_i^{i\in 1..n} \rangle$$

Nel termine che effettua il pattern matching le variabili x_i sono legate nei temini M_i. Più formamlmente, la definizione delle variabili libere si estende con i seguenti due casi:

$$\text{fv}(\langle \ell = M \rangle) = \text{fv}(M) \qquad \text{fv}(M \text{ match } \{\text{case } \ell_i = x_i \Rightarrow M_i\,^{i\in 1..n}\}) = \text{fv}(M) \cup \bigcup_{i=1,..,n} \text{fv}(M_i) \backslash \{x_i\}$$

EXERCISE 3.5. Definire la sostituzione $M\{x := N\}$ nel linguaggio esteso con redord e variants. □

La relazione di valutazione e le regole di tipo vanno estese con le regole seguenti:

(MATCH)
$$\frac{j \in 1..n}{\langle l_j = v_j \rangle \ \mathsf{match}\ \{\mathsf{case}\ \ell_i = x_i \ \Rightarrow\ M_i\ ^{i\in 1..n}\} \longrightarrow M_j\{x_j := v_j\}}$$

(RED MATCH)
$$\frac{M \longrightarrow M'}{M\ \mathsf{match}\ \{\mathsf{case}\ \ell_i = x_i \ \Rightarrow\ M_i\ ^{i\in 1..n}\} \longrightarrow M'\ \mathsf{match}\ \{\mathsf{case}\ \ell_i = x_i \ \Rightarrow\ M_i\ ^{i\in 1..n}\}}$$

(VARIANT)
$$\frac{M \longrightarrow M'}{\langle \ell = M \rangle \longrightarrow \langle \ell = M' \rangle}$$

Ad esempio sia M_s il termine $\langle l_1 = 3\rangle$ match $\{\mathsf{case}\ \ell_2 = x \ \Rightarrow\ x + 1\}$, allora $M_s \not\longrightarrow$, cioè M_s è un termine stuck, un errore di programmazione. Si consideri inoltre il seguente termine, che indichiamo M_f, e che è molto simile alla funzione `getNome` definita prima:

$$\mathsf{fn}\, x : \langle \ell_1{:}\mathsf{Nat}, \ell_2{:}\mathsf{Bool}\rangle.\ x\ \mathsf{match}\ \{\mathsf{case}\ \ell_1{=}x_1 \Rightarrow \mathsf{true},\ \mathsf{case}\ \ell_2{=}x_2 \Rightarrow x_2\}$$

L'applicazione $(M_f\ \langle \ell_1{=}5\rangle)$ evolve in due passi al valore true. Le seguenti regole di tipo indicano anche che si tratta di un termine ben tipato.

(TYPE VARIANT)
$$\frac{\Gamma \vdash M : T_j \quad j \in 1..n}{\Gamma \vdash \langle \ell_j = M\rangle : \langle \ell_i : T_i\ ^{i\in 1..n}\rangle}$$

(TYPE MATCH)
$$\frac{\Gamma \vdash M : \langle \ell_i : T_i\ ^{i\in 1..n}\rangle \quad \forall i \in 1..n\ \ \Gamma, x_i : T_i \vdash M_i : T}{\Gamma \vdash M\ \mathsf{match}\ \{\mathsf{case}\ \ell_i = x_i \ \Rightarrow\ M_i\ ^{i\in 1..n}\} : T}$$

Nota che, analogamente alla regola per tipare il costrutto if-then-else, anche il typing del costrutto di case richiede che tutti i rami abbiano lo stesso tipo T. Si osservi infine che l'aggiunta dei variant types fa perdere la proprietà di unicità del tipo. Ad esempio il termine $\langle \ell_1 = 5\rangle$ può avere tipo $\langle \ell_1 : \mathsf{Nat}, \ell_2 : \mathsf{Bool}\rangle$ oppure $\langle \ell_1 : \mathsf{Nat}, \ell_2 : \mathsf{Bool} \to \mathsf{Bool}\rangle$, oppure infiniti altri tipi! Per maggiori dettagli si rimanda al capitolo 11 del testo.

Si noti come, affinché l'applicazione $(M_f\ \langle \ell_1{=}5\rangle)$ sia ben tipata è importante riuscire a derivare per il termine $\langle \ell_1 = 5\rangle$ il tipo $\langle \ell_1{:}\mathsf{Nat}, \ell_2{:}\mathsf{Bool}\rangle$, cosa che è possibile con la regola (TYPE VARIANT). D'altra parte, si può vedere facilmente che il termine stuck M_s non è ben tipato. Se infatti il giudizio $\emptyset \vdash M_s : T$ fosse derivabile, tale derivazione terminerebbe con la regola (TYPE MATCH), che richiederebbe una derivazione per i seguenti due giudizi:

1. $\emptyset \vdash \langle l_1 = 3\rangle : \langle \ell_2 : T_2\rangle$
2. $x : T_2 \vdash x + 1 : T$

Se da un lato il secondo giudizio è effettivamente derivabile con $T = T_2 = \mathsf{Nat}$, il primo giudizio non è derivabile.

EXERCISE 3.6. Dimostrare il teorema di safety per il linguaggio con i tipi variante □

EXERCISE 3.7. Un ulteriore esempio di variant type è il seguente tipo $List$, che rappresenta liste di interi. Si noti che il tipo è definito in modo ricorsivo:

$$\begin{array}{ll} \text{Tipo} & List \ = \ \langle\, nil : \mathsf{Unit},\ cons : (int * List)\,\rangle \\ \text{Valori} & \langle nil = \mathsf{unit}\rangle \quad \langle\, cons = (n, \langle nil = \mathsf{unit}\rangle)\,\rangle \end{array}$$

Definire il termine che corrisponde ad una lista che contiene due elementi, cioè due costanti intere. □

EXERCISE 3.8. Studiare una variante di questo linguaggio che ammette un caso di default nell'operazione di pattern matching. □

Restituire un valore opzionale. Molto spesso nei programmi si usano funzioni/metodi che possono ritornare o un valore sensato oppure nessun valore, e.g. una funzione che cerca un elemento in un insieme può restituire l'elemento trovato oppure restituire un valore che indica che quell'elemento non c'è. In questi casi quale sarà il tipo di ritorno della funzione? Ci vuole cioè un tipo in grado di rappresentare uniformemente un *"valore sensato"* e *"l'assenza di valore"*. Ci sono diverse possibilità:

- In Java il tipo classe `C` ha come valori possibili istanze della classe `C` ma anche il valore speciale `null`. Spesso dunque si usa il valore `null` per rappresentare l'*assenza di un oggetto*, e.g. `C find(String s, List<C> l)` cerca nella lista `l` se c'è un elemento che contiene la stringa `s`, se lo trova restituisce l'elemento (di tipo `C`), altrimenti restituisce `null`. La funzione va quindi usata nel modo seguente:

  ```
  C c = find(s,l);
  if (c!= null) print(c.info());
  else print("non trovato");
  ```

  ricordandosi cioè di fare sempre un controllo `c!= null` per evitare l'insorgere dell'eccezione `NullPointerException`. È fondamentale ricordarsi di programmare il controllo perché in Java `NullPointerException` è una `RuntimeException`, dunque la sua gestione non è obbligatoria e non è controllata dal compilatore.

  Osserviamo che con questo pattern `null` rappresenta un valore *eccecezionale*, che va gestito in modo appropriato. In genere invece il valore `null` si usa per rappresentare un riferimento non inizializzato, e `NullPointerException` rappresenta l'*errore logico* corrispondente alla dereferenziazione di un riferimento nullo. Stiamo cioè confondendo un errore con un'eccezione.

- È interessante osservare che l'uso di `null` come "nessun oggetto di tipo `C`" è talmente comodo e diffuso che il linguaggio C# estende questa possibilità anche ai tipi base introducendo i *nullable types*. Il tipo `Nullable<int>`, o `int?`, ha come possibili valori tutti gli interi più il valore `null`, che indica proprio "nessun intero".

- Riportiamo quanto ha dichiarato nel 2009 T.Hoare, inventore del riferimento `null`:
  I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. In recent years, a number of program analysers like PREfix and PREfast in Microsoft have been used to check references, and give warnings if there is a risk they may be non-null. More recent programming languages like Spec# have introduced declarations for non-null references. This is the solution, which I rejected in 1965. From "Null References: The Billion Dollar Mistake", QCon, London 2009.

- Un altro possibile approccio, è quello dei cosiddetti *Null objects*: data una classe `C`, si definisce una sottoclasse `NullC` di `C` che ridefinisce con corpo vuoto i metodi della superclasse. Gli oggetti (basta un singleton object) della sottoclasse rappresentano il valore "nessun oggetto di tipo `C`" e si usano al posto di `null`. In questo modo infatti, grazie al polimorfismo, si può sempre dereferenziare correttamente un riferimento di tipo `C`.

  ```
  class C {
     ...
     String info(){ return ... }
  }

  C find(String s, List<C> l){
     for (C c : l)
  ```

  ```
  class NullC extends C {
     ...
     String info(){ return "";}
  }
  ```

```
        if(c.info().equals(s)) then return c;
   return new NullC();
}

C c = find(s,l);
print(c.info());  // senza piu' bisogno del controllo
```

- Infine, una terza possibilità è data dall'uso degli *option types*, che sono a tutti gli effetti dei tipi variante. Dato un tipo qualsiasi T, il tipo $Option(T)$ rappresenta dei valori opzionali, nel modo seguente:

Tipo	$Option(T) = \langle none : \mathsf{Unit},\ some : T \rangle$	
Valori	$\langle none = \mathsf{unit} \rangle$	indica l'**assenza** di un valore di tipo T
	$\langle some = v \rangle$	indica la **presenza** del valore v di tipo T

  Si osservi in particolare che il tipo $Option(T)$ incapsula i valori di tipo T, cioè v e $\langle some = v \rangle$ sono due termini ben diversi che supportano operazioni ben distinte; in particolare, per usare un'operazione disponibile per il tipo T sul termine $\langle some = v \rangle$ bisogna prima estrarre il termine v. Come esempio vediamo come si definisce la funzione di ricerca usando la sintassi Scala per il tipo `Option[T]`:

  ```
  def find (s: String , l: List[C]) : Option[C] = {
   for (c <- l)
     if(c.info()==s)  return Some(c)
   return None
  }

  find(s,l) match {
    case Some(x) => print(x.info())
    case None => print("non trovato")
  }
  ```

  In particolare, il termine `find(s,l).info()` non è ben tipato. L'uso dei tipi opzione, cioè la distinzione tra il tipo T e $Option(T)$, riesce dunque a **trasformare un errore runtime** (i.e. la dereferenziazione di un riferimento nullo) **in un errore statico controllato dal sistema di tipi, i.e., dal compilatore**.

  Il tipo opzione è stato introdotto anche in Java8 con la classe generica `Optional<T>`. Ma Java8 non ha il pattern matching, quindi per estrarre il valore eventualmente incapsulato in un oggetto di tipo `Optional<T>` bisogna passare attraverso i metodi `boolean isPresent()` e `T get()` della classe `Optional<T>`. A questo punto però se su un oggetto opzionale vuoto ci si dimentica di invocare `isPresent()` prima di `get()` si ottiene la runtime exception `NoSuchElementException`, ritrovandosi esattamente come nel caso classico in cui si usa `null`. La classe Java8 `Optional<T>` e la classe Scala `Option[T]` mettono a disposizione dei metodi che permettono di usare oggetti opzionali in modo diverso usando l'approccio delle moandi.

- Ricordiamo la soluzione proposta dal linguaggio Swift, che offre una sintassi comoda per fare il cosiddetto *optional chaining*. L'istruzione `find(s,l)?.info()` ha il seguente significato:
  - se `find(s,l)=Some(c)` allora fa automaticamente l'unwrapping e restituisce `c.info()`;
  - se `find(s,l)=nil` allora restituisce direttamente `nil` evitando di chiamare `nil.info()`, che provocherebbe un errore.

  In altre parole, l'operatore `?` aiuta a "far fallire dolcemente" a `nil` le catene di optional values.