Corso: Fondamenti, Linguaggi e Traduttori

Paola Giannini

Realizzazione del type-checking

Aggiunte delle classi Symbol Table e `NodeConvert`

- Abbiamo bisogno di definire la Symbol Table (questa la definirete in un package `symbolTable`) e per definire le sue entry
- si deve definire la classe `Attributes` (per il momento avrà solo il campo `LangType`, poi durante la generazione del codice avremo anche un campo che ci identifica il registro associato all'identificatore).
- Per segnalare l'inserimento di una conversione useremo un nuovo nodo `NodeConvert` (che estende `NodeExpr` e contiene una espressione che indicherà (per la fase di generazione del codice) che deve essere fatto un cast (da `Int` a `Float`) al risultato dell'espressione.

Aggiunte ai nodi e `TypeDescriptor`

- A tutti i nodi e quindi a `NodeAST` dobbiamo aggiungere
  - un campo `resType` che ci dice il tipo risultante dall'analisi del nodo. Il tipo del campo è
  - `TypeDescriptor`: un tipo enumerato con `Int`, `Float`, `Void` e `Error`. Usiamo `Void` per i nodi dichiarazione e istruzione corretti, `Int` e `Float` per i nodi espressione corretti (e che quindi hanno uno dei due tipi) e `Error` per un nodo scorretto (o con sottoparti scorrette).
  - Fare il type-checking significherà partire da un AST di un programma in cui tutti i nodi hanno `resType` con valore `null` e modificare l'AST in modo tale che tutti i nodi il campo `resType` abbiano un valore di tipo del tipo enumerato `TypeDescriptor`.
- Inoltre al `NodeId` dobbiamo aggiungere
  - un campo `definition` che riferisce agli attributi della Symbol Table associata all'identificatore (di tipo `Attributes`).

Come calcolare `resType` (in generale effettuare computazioni sui i nodi del AST)

- Per calcolare `resType` dobbiamo visitare l'AST del programma (anche la generazione del codice verrà effettuata processando l' AST).
- Vediamo due modi possibili (ce ne sono certamente altri),
  1. il primo concettualmente più semplice ma che ha lo svantaggio di richiedere una modifica significativa ai nodi del AST
  2. il secondo usa un pattern di programmazione e permette la separazione fra computazione da fare e nodi del AST

Calcolo resType con metodo dei nodi.

- Aggiungere ad ogni nodo un metodo per ogni tipo algoritmo che vogliamo fare.
- In NodeAST il metodo sarà astratto, mentre per i nodi concreti specificherà cosa fare:

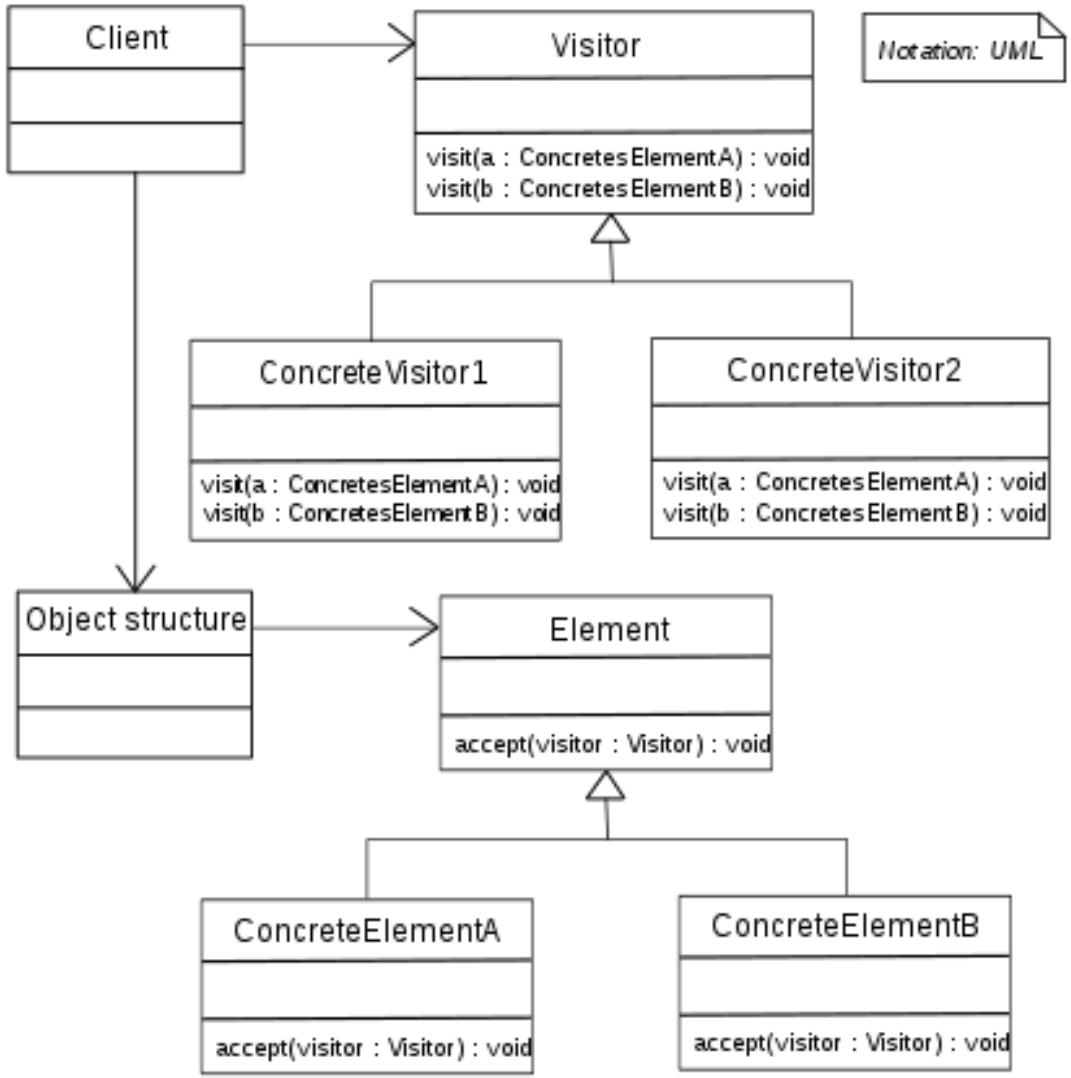
```
public abstract class NodeAST {
  private TypeDescriptor resType;

  public abstract void calResType(); // assegna a resType il tipo del sottoalbero
  public TypeDescriptor getResType() {
    return resType;
  }

public class NodeBinOp extends NodeExpr {
  private LangOper op; private NodeExpr left; private NodeExpr right;
        ....................
  public void calResType(){
    left.calResType; // calcola il tipo della sotto-espressione di sinistra
    right.calResType; // calcola il tipo della sotto-espressione di destra
    if (left.getResType()== .....     right.getResType()== ......) //controlli opportuni
       ..........
    resType = ..... // assegnamento al campo
  }
}
```

- Se vogliamo aggiungere ulteriori algoritmi (ad esempio controllo che le variabili siano inizializzate prima di essere usate, o trovare le porzioni di codice che non sono raggiungibili, ecc.) dobbiamo andare ad aggiungere codice ai nodi del AST!!!

Uso il pattern **Visitor** (comportamentale)



Separa computazione e oggetti su cui si fa.

- Il pattern visitor ci permette di separare la computazione da fare sui nodi dalla definizione dei nodi.
- In questo modo altri tipi di computazione possono essere aggiunti senza modificare la definizione dei nodi.
- Caratterizziamo astrattamente le computazioni definendone una interfaccia.

```
public interface IVisitor {
    public abstract  void visit (NodeProgram node);
    public abstract  void visit (NodeId node);
    public abstract void visit(NodeDcl node);
    public abstract void visit(NodeBinExpr node);
    public abstract void visit(NodePrint node);
         ...............
}
```

Notare l'uso dell' overloading. Una classe concreta che implementa `IVisitor` implementa tutti i metodi!

Visitor concreti

```
public class TypeCheckingVisitor implements IVisitor {

   public void visit(NodeBinOp node) {
     node.getLeft().accept(this);
     node.getRight().accept(this);
     // controllo specifico  per l'espressione
     }

   public void visit(NodeAss node) {
     node.getExpr().accept(this);
     node.getId().accept(this);
     // controllo specifico  per l'assegnamento
     }
   // I metodi visit per gli altri nodi concreti
}

public class CodeGeneratorVisitor implements IVisitor {

   public void visit(NodeBinOp node)  {
      node.getLeft().accept(this);
      node.getRight().accept(this);
     // generazione codice per l'espressione
     }

   public void visit(NodeAss node) throws TypeException {
      node.getExpr().accept(this);
      node.getId().accept(this);
      // generazione codice  per l'assegnamento
     }
   // I metodi visit per gli altri nodi concreti.......
 }
```

Come vengono eseguite le computazioni specificate da `visit`?

- Come si esegue il visitor su un nodo?
- Il nodo deve implementare il metodo astratto `accept(IVisitor visitor)`

```
public abstract class NodeAST {
  public abstract void accept(IVisitor visitor);
  // altri metodi di NodeAST
}

//Le sottoclassi concrete di NodeAST devono ridefinire accept
public class NodeBinOp extends NodeExpr{
   public void accept(IVisitor visitor) {
      visitor.visit(this);
   }
   // Gli altri metodi di NodeBinOp
}
......
```

In questo modo si realizza quello che si chiama double dispatch. Cioè

`node.accept(visitor)  ==>  visitor.visit(node)`

esegue l'overloading del metodo `visit` che ha come parametro il tipo dinamico dell'oggetto `node` del tipo dinamico dell'oggetto `visitor`.

Le classi coinvolte nella specifica del TypeChecking Visitor

Interfacce/Classi astratte

```
public interface IVisitor {
    public abstract  void visit (NodeProgram node);
    public abstract  void visit (NodeId node);
    public abstract void visit(NodeDcl node);
    public abstract void visit(NodeBinExpr node);
    public abstract void visit(NodePrint node);
         ..............
}
```

```
public abstract class NodeAST {
    private TypeDescriptor resType;
    public TypeDescriptor getResType() {
        return resType;
    }
    public void setResType(TypeDescriptor resType) {
        this.resType = resType;
    }

    public abstract void accept(IVisitor visitor);
}
```

Classi concrete

```
public class TypeCheckingVisitor implements IVisitor {

   public void visit(NodeBinOp node) {
     node.getLeft().accept(this);
     node.getRight().accept(this);
     // controllo specifico  per l'espressione
     }

   public void visit(NodeAss node) {
     node.getExpr().accept(this);
     node.getId().accept(this);
     // controllo specifico  per l'assegnamento
     }
   // I metodi visit per gli altri nodi concreti
}
```

```
public class NodeBinOp extends NodeExpr{
   public void accept(IVisitor visitor) {
      visitor.visit(this);
   }
   // Gli altri metodi di NodeBinOp
}
```

- Come parte la visita

```
NodeProgram nP = new Parser(new Scanner(......)).parse();
nP.accept(new TypeCheckingVisitor());
```