

Capitolo 4

Analisi Semantica

Le analisi lessicale e sintattica dei Capitoli 2 e 3 hanno verificato che il programma sorgente soddisfa delle regole di sintassi specificate, rispettivamente, da un insieme di token e da una grammatica. Inoltre, l'analisi sintattica ha costruito un albero di sintassi astratta che fornisce una visione ad alto livello della struttura del file sorgente (Figura 3.1).

Questo non significa che tutti i programmi che hanno superato con successo l'analisi sintattica, cioè senza generare alcun errore di sintassi, siano automaticamente dei programmi *corretti*, pronti cioè ad essere tradotti in codice oggetto ed eseguiti. Per esempio, basta prendere il programma della Figura 1.4 e modificare la linea `this.state := true` in `this.state := 3` per ottenere un programma che supera senza alcun problema sia l'analisi lessicale che quella sintattica, ma che non è *corretto*, poiché esso tenta di assegnare un valore intero (3) a un campo che può contenere solo valori di tipo booleano (`state`). Accorgersi di tali errori va ben al di là delle possibilità delle grammatiche libere dal contesto. Serve uno strumento alternativo, che sarà quello della discesa ricorsiva sull'albero di sintassi astratta del codice sorgente, alla ricerca di eventuali errori. Questa ricerca sarà l'oggetto di questo capitolo e prende il nome di *analisi semantica*.

L'esempio appena visto è quello di un *errore di tipo*. Ma l'analisi semantica non si limita alla ricerca di tali errori. In particolare, è normalmente compito dell'analisi semantica di:

1. identificare usi di espressioni incompatibili con i loro tipi statici (*errori di tipo*);
2. identificare occorrenze di variabili usate ma non dichiarate;
3. garantire che i comandi `break` e `continue` occorrano solo nello scope di un'istruzione iterativa;
4. garantire che un metodo non `void` termini sempre con un'istruzione `return exp`, indipendentemente dal percorso di esecuzione che viene seguito al suo interno, e che un metodo `void` non contenga comandi di tipo `return exp`;
5. garantire che non ci siano parti di codice che non possono mai essere eseguite e che sono quindi irraggiungibili e *inutili* (identificazione *del codice morto*);
6. identificare e annotare il tipo statico delle espressioni che occorrono in un programma (*inferenza dei tipi*);
7. identificare, per ogni accesso a un campo, la classe in cui il campo è definito;
8. identificare, per ogni istruzione `new Classe`, il costruttore di *Classe* che deve essere chiamato in tal punto a tempo di esecuzione, sulla base del tipo dei parametri passati al costruttore;
9. identificare, per ogni invocazione di metodo, la dichiarazione di metodo che deve venire chiamata a tempo di esecuzione, a meno di ridefinizioni nelle sottoclassi (nel qual caso si chiama una di tali ridefinizioni), sulla base del tipo dei parametri passati al metodo.

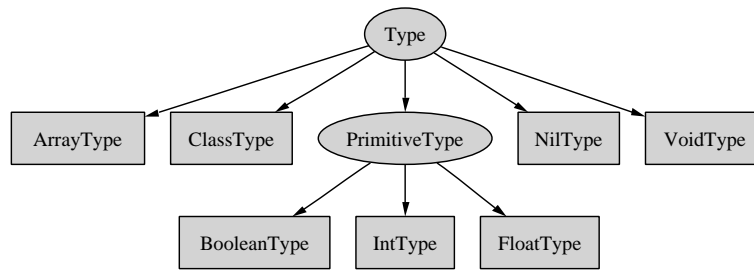


Figura 4.1: Le classi del package `types` usate per rappresentare i tipi semantici di Kitten.

Potremmo quindi dire che l'analisi semantica si occupa di garantire condizioni di correttezza elementari, senza le quali non ha neppure senso compilare il programma in codice oggetto (*verifica del codice*, punti 1–5) e di raccogliere informazione sul programma che si sta compilando, al fine di facilitare la successiva fase di generazione del codice oggetto (*annotazione del codice*, punti 6–9). Va detto che tale divisione è concettualmente utile ma non netta, dal momento che, per esempio, l'identificazione del costruttore chiamato da un'istruzione `new` (punto 8) è sì un'annotazione utile a generare il codice oggetto che effettua la chiamata a tale costruttore, ma è anche una verifica che tale costruttore esista realmente. L'insieme esatto dei compiti affidati all'analisi semantica è comunque molto variabile col linguaggio di programmazione considerato. Per esempio, molti linguaggi di programmazione ad oggetti (fra cui Java e Kitten) affidano all'analisi semantica anche il compito di

10. compilare, ricorsivamente, tutte le classi a cui si fa riferimento nel testo della classe in fase di compilazione.

Come ulteriore esempio, l'analisi semantica potrebbe garantire che quando una variabile è usata allora essa è già stata inizializzata, indipendentemente dal percorso di esecuzione che ha portato al punto in cui la variabile è usata. Tale controllo è effettuato da Java ma non da Kitten.

4.1 I tipi Kitten

Il concetto di *tipo* (Sezione 1.7) è al centro dell'analisi semantica (punti 1,4,6,7,8,9,10 della precedente enumerazione). Va subito notato che per *tipo* non intendiamo qui la sintassi astratta di una *espressione* di tipo, come nella Sezione 3.3.1. In quel caso, avevamo bisogno di un modo per rappresentare la *struttura* di una parte di codice che rappresenta un tipo Kitten. Si tratta adesso invece di rappresentare il tipo *semantico* delle espressioni Kitten, cioè una struttura dati con associate alcune operazioni che permettono, per esempio, di determinare se un tipo è un sottotipo di un altro o qual è il minimo sovratipo comune di due o più tipi, se esiste (Sezione 1.7) o quali siano i campi o i costruttori o metodi di un tipo classe. Per apprezzare ancora di più la differenza, basta osservare che due occorrenze dell'espressione `int` in due punti diversi di un file sorgente danno origine a due oggetti `IntTypeExpression` diversi, ma il loro tipo semantico è lo stesso, identico oggetto.

La distribuzione Kitten contiene il package `types`, al cui interno trovano posto delle classi che rappresentano i tipi *semantici* del linguaggio Kitten. La Figura 4.1 presenta la gerarchia di tali classi. Si noti che non esiste un tipo specifico per le stringhe, che sono invece considerate come un esempio di `ClassType`. Esamineremo fra poco tali classi. Per adesso, mostriamo come esista una relazione stretta fra le espressioni di tipo date nella sintassi astratta (Figura 3.4) e i loro tipi *semantici* (Figura 4.1). Per esempio, ogni oggetto di classe `absyn.IntTypeExpression` viene mappato in un oggetto di classe `types.IntType`. Tale funzione dalle espressioni di tipo al loro tipo semantico è mostrata in Figura 4.2. Le espressioni di tipo che rappresentano i tipi primitivi

```

τ[ ] : absyn.TypeExpression ↦ types.Type

τ[IntTypeExpression()] = Type.INT
τ[FloatTypeExpression()] = Type.FLOAT
τ[BooleanTypeExpression()] = Type.BOOLEAN
τ[VoidTypeExpression()] = Type.VOID
τ[ArrayTypeExpression(elementsType)] = ArrayType.mk(τ[elementsType])
τ[ClassTypeExpression(name)] = ClassType.mk(name)

```

Figura 4.2: La funzione di analisi semantica $\tau[_]$ per le espressioni di tipo Kitten.

vengono mappate in costanti della classe `types.Type`. Quelle che rappresentano gli array vengono mappate in tipi semantici di tipo `types.ArrayType` per il tipo semantico dei propri elementi. Tali tipi array sono costruiti tramite un *costruttore con memoria* `mk()`, cioè un metodo che costruisce un oggetto della classe specificata ma che ritorna sempre lo stesso oggetto se viene chiamato due volte con lo stesso parametro. L'implementazione di un costruttore con memoria si ottiene semplicemente usando un metodo `static` con una memoria statica. Infine, le espressioni di tipo che rappresentano un tipo classe vengono mappate in un oggetto di tipo `types.ClassType`, costruito anch'esso tramite un costruttore con memoria `mk()`. Si noti che tale costruttore accede al file Kitten che definisce la classe. Se tale file non esistesse o fosse sintatticamente o semanticamente scorretto, il costruttore `mk()` fornirebbe una classe minimale di default, sottoclasse diretta di `Object` e priva di campi, costruttori e metodi.

Vediamo adesso più da vicino le classi dei tipi semantici. La Figura 4.3 mostra la classe astratta `types/Types.java`. Essa definisce in primo luogo delle costanti per dei tipi di uso comune. Le sue sottoclassi dovranno instanziare il metodo `toString()` nonché il metodo `canBeAssignedTo()`. Quest'ultimo deve determinare se un tipo può essere assegnato a un altro, seguendo le regole della Sezione 1.7. Il metodo `subTypeOf()` è definito come un sinonimo di `canBeAssignedTo()`. Il metodo `canBeAssignedToSpecial()` deve comportarsi come `canBeAssignedTo()`, ma deve imporre ai tipi primitivi di essere sottotipi solo di se stessi. Inoltre ammette che `void` sia sottotipo di se stesso. Questo metodo è utile all'interno della classe `ArrayType` che vedremo fra un attimo, e nel determinare se il tipo di ritorno di una ridefinizione di un metodo è compatibile con quello del metodo ridefinito. Il metodo `leastCommonSupertype()` determina il minimo sovratipo comune a due tipi. Si noti che tale minimo sovratipo comune non esiste necessariamente. Per esempio, non c'è alcun sovratipo comune fra `int` e `boolean`. La definizione fornita dentro `types/Types.java` funziona per tutti i tipi primitivi, ma deve essere ridefinita per altri tipi, come vedremo. Infine, il metodo `equals()` determina se due tipi sono uguali: due tipi vengono considerati uguali se e solo se essi sono lo stesso oggetto. Questo ha senso poiché il package `types` è organizzato in modo che non sia possibile creare istanze diverse dello stesso tipo. Per esempio, l'unico oggetto che rappresenta `int` è la costante `INT` di `types/Types.java`.

4.1.1 L'implementazione dei tipi non di riferimento di Kitten

Mostriamo adesso le sottoclassi di `types.Type` che implementano i tipi non di riferimento del linguaggio Kitten cioè `void` e i tipi primitivi.

Cominciamo dalla classe `VoidType.java` in Figura 4.4. La prima osservazione è che il suo costruttore è `protected`, in modo tale che non sia possibile costruire istanze di questo tipo all'esterno del package `types` (e delle inesistenti sottoclassi di `VoidType`). In questo modo garantiamo che l'unica istanza di questo tipo sia la costante `VOID` in Figura 4.3. Inoltre va osservato che non ammettiamo mai di assegnare `void` a un altro tipo. L'assegnamento *speciale* è invece pos-

```

public abstract class Type {
    protected Type() {}

    // delle costanti di uso frequente
    public final static BooleanType BOOLEAN = new BooleanType();
    public final static FloatType FLOAT = new FloatType();
    public final static IntType INT = new IntType();
    public final static NilType NIL = new NilType();
    public final static VoidType VOID = new VoidType();

    abstract public String toString();

    abstract public boolean canBeAssignedTo(Type other);

    public final boolean subtypeOf(Type other) { return canBeAssignedTo(other); }

    public boolean canBeAssignedToSpecial(Type other) {
        // i tipi primitivi ridefiniranno questo metodo
        return canBeAssignedTo(other);
    }

    public Type leastCommonSupertype(Type other) {
        // questo e' ok per i tipi primitivi. Classi e array ridefiniscono
        if (this.canBeAssignedTo(other)) return other;
        if (other.canBeAssignedTo(this)) return this;
        // se si arriva qui allora non c'e' un sovratipo comune
        return null;
    }

    public final boolean equals(Type other) { return this == other; }
}

```

Figura 4.3: La superclasse astratta dei tipi semantici di Kitten

sibile solo fra tipi void, in modo da permettere a un metodo che ritorna void di essere ridefinito, purché il tipo di ritorno sia mantenuto a void.

La Figura 4.5 mostra le classi `types/PrimitiveType.java` e `types/IntType.java`. La prima ridefinisce il metodo `canBeAssignedToSpecial()` in modo che la ridefinizione di un metodo che ritorna un tipo primitivo debba necessariamente restituire lo stesso tipo primitivo. Il motivo di questa scelta, apparentemente strana, è che se un metodo:

```
float m()
```

potesse essere ridefinito, in una sottoclasse, in

```
int m()
```

allora una chiamata virtuale del tipo `float f = o.m()` richiederebbe una conversione di tipo da `int` a `float` sulla base della classe, a tempo di esecuzione, dell'oggetto contenuto in `o`. Questo complica la generazione del codice, per cui impediamo al programmatore di fare una simile ridefinizione del tipo di ritorno del metodo `m()`. Si noti che la stessa scelta è fatta da Java.

Il metodo `canBeAssignedTo()` di `IntType.java` permette invece di assegnare un valore di tipo `int` a un altro `int` o a un `float`, previa conversione di tipo (Figura 4.5).

4.1.2 L'implementazione dei tipi riferimento di Kitten

Mostriamo adesso l'implementazione dei tipi riferimento di Kitten, cioè gli array, le classi e `nil`.

```
public class VoidType extends Type {
    protected VoidType() {}
    public String toString() { return "void"; }
    public boolean canBeAssignedTo(Type other) { return false; }
    public boolean canBeAssignedToSpecial(Type other) { return this == other; }
}
```

Figura 4.4: La classe `types/VoidType.java`.

```
public abstract class PrimitiveType extends Type {
    protected PrimitiveType() {}
    public boolean canBeAssignedToSpecial(Type other) { return this == other; }
}

public class IntType extends PrimitiveType {
    protected IntType() {}
    public String toString() { return "int"; }
    public boolean canBeAssignedTo(Type other) {
        return other == Type.INT || other == Type.FLOAT;
    }
}
```

Figura 4.5: La classe `types/PrimitiveType.java` e la sua sottoclasse `types/IntType.java`.

La classe in Figura 4.6 rappresenta il tipo degli array. L'invariante che non esistano istanze diverse dello stesso tipo è mantenuta rendendo `private` il costruttore di `ArrayType` e permettendo la creazione di tipi array solo tramite il metodo statico `mk()`, che usa una memoria per evitare di creare duplicati. L'assegnamento di un tipo array `this` a un altro tipo `other` è considerata legale solo se `other` è `Object` oppure se anche `other` è un tipo array e gli elementi di `this` possono a loro volta essere assegnati a quelli di `other`. Ma si noti l'uso di `canBeAssignedToSpecial()` per questa chiamata ricorsiva! Questo al fine di rispettare il vincolo sulla non primitività di t_1 della Sezione 1.7. In pratica, se gli elementi di `this` sono un tipo primitivo, quelli di `other` devono essere *lo stesso* tipo primitivo. Questo vincolo, anch'esso apparentemente strano, è giustificato dal fatto che il codice

```
int[] arr := [3,4,5];
int[] copy := arr
```

rende `arr` e `copy` *alias*, cioè riferimenti diversi allo stesso oggetto array. Mentre nel codice

```
int[] arr := [3,4,5];
float[] copy := arr
```

saremmo costretti a convertire ciascun elemento di `arr` da `int` a `float`. Dal momento che dobbiamo lasciare immutato l'array `arr`, la conversione è possibile solo a costo di creare un nuovo array di `float` che contiene i valori convertiti. Tale array verrebbe poi assegnato `copy`. Ma questo significa che `arr` e `copy` non sarebbero più *alias*! Detto altrimenti, la scelta del tipo degli elementi di `copy` determinerebbe la condivisione (o meno) fra `arr` e `copy`. Tale comportamento, nettamente inaspettato dal programmatore, è da considerarsi semanticamente pericoloso ed è quindi conveniente vietare tali assegnamenti. Va notato inoltre che il costo computazionale dell'assegnamento diventerebbe lineare nella lunghezza dell'array piuttosto che costante, come normalmente si richiede.

Ancora più complicato è il metodo `leastCommonSupertype()` di `ArrayType`, che deve determinare il minimo supertipo comune (*lcs*) fra il tipo array `this` e un altro tipo `other`. Le regole che portano alla definizione di *lcs* sono le seguenti:

```

public class ArrayType extends Type {
    private Type elementType;

    private ArrayType(Type elementType) { this.elementType = elementType; }

    public static ArrayType mk(Type elementType) {
        // usa una memoria per non ricreare tipi array gia' creati in passato
        ...
    }

    public String toString() { return "array of " + elementType; }

    public boolean canBeAssignedTo(Type other) {
        if (other instanceof ArrayType)
            return elementType.canBeAssignedToSpecial(((ArrayType)other).elementType);
        else return other == ClassType.OBJECT;
    }

    public Type leastCommonSupertype(Type other) {
        // l'lcs fra un array e una classe e' Object
        if (other instanceof ClassType) return ClassType.OBJECT;
        if (other instanceof ArrayType)
            if (elementType instanceof PrimitiveType)
                // fra un array di tipi primitivi e se stesso l'lcs e' l'array.
                if (this == other) return this;
                // fra due array di tipi primitivi diversi, l'lcs e' Object
                else return ClassType.OBJECT;
            else return mk(elementType.leastCommonSupertype
                (((ArrayType)other).elementType));
        if (other == Type.NIL) return this; // l'lcs fra un array e nil e' l'array
        return null; // non esiste alcun lcs
    }
}

```

Figura 4.6: La classe `types/ArrayType.java` che rappresenta i tipi array.

- se `other` è una classe allora `lcs` è `Object`. Si noti infatti che tutti gli array e tutte le classi sono sottotipi di `Object` (Sezione 1.7);
- se anche `other` è un tipo array allora:
 - se entrambi sono array di tipi primitivi e tali tipi primitivi sono gli stessi allora `lcs` è uguale a `this` (o equivalentemente a `other`);
 - altrimenti, se entrambi sono array di tipi primitivi allora `lcs` è `Object`; si noti che sarebbe errato definire in questo caso `lcs` come `array of Object`, poiché i tipi primitivi non sono sottotipi di `Object`;
 - altrimenti, se entrambi sono array di tipi non primitivi, allora `lcs` è il tipo array del minimo sovratipo comune fra i tipi degli elementi di `this` e `other`;
- se `other` è il tipo `NilType`, allora `lcs` è `this` poiché `NilType` è un sottotipo di qualsiasi tipo array (Sezione 1.7);
- altrimenti `lcs` non esiste.

Vediamo infine il tipo `ClassType`, che rappresenta i tipi classe come `Object`, `String` o `Led` in Figura 1.4. La Figura 4.7 riporta il codice di `types/ClassType.java`. Il costruttore è lasciato `private` e la costruzione del tipo classe è possibile solo tramite il metodo statico `mk()` che

```

public class ClassType extends Type {
    private static HashMap memory = new HashMap();
    // utili costanti
    public static final ClassType OBJECT = mk(Symbol.OBJECT);
    public static final ClassType STRING = mk(Symbol.STRING);
    private Symbol name;           // il nome di questa classe
    private ClassType superclass;  // la sua superclasse (se esiste)
    private ClassSignature signature; // la sua segnatura

    private ClassType(Symbol name) {
        // memorizziamo questo oggetto per un riciclo futuro
        memory.put(name,this);
        this.name = name;
        this.signature = new ClassSignature(this);
        ClassSignature superSig = signature.getExtendedSignature();
        if (superSig != null) this.superclass = superSig.getDefiningClass();
        // facciamo l'analisi semantica di questa classe
        signature.getAbstractSyntax().typeCheck(this);
    }

    public static ClassType mk(Symbol name) {
        ClassType result = (ClassType)memory.get(name);
        if (result != null) return result;
        else return new ClassType(name);
    }

    public String toString() { return name.toString(); }

    public boolean canBeAssignedTo(Type other) {
        return other instanceof ClassType && this.subclass((ClassType)other);
    }

    public boolean subclass(ClassType other) {
        return this == other ||
            (superclass != null && superclass.subclass(other));
    }

    public Type leastCommonSupertype(Type other) {
        if (other instanceof ArrayType) return OBJECT;
        if (other instanceof ClassType)
            for (ClassType cursor = this; cursor != null; cursor = cursor.superclass)
                if (other.canBeAssignedTo(cursor)) return cursor;
        if (other == Type.NIL) return this;
        return null; // non c'e' alcun lcs
    }
}

```

Figura 4.7: La classe `types/ClassType.java` che rappresenta il tipo delle classi Kitten.

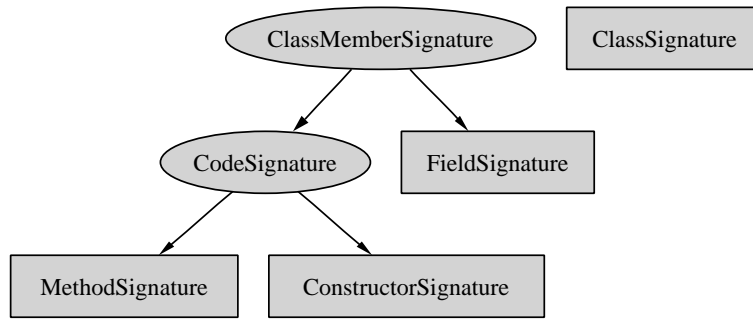


Figura 4.8: Le classi del package `types` usate per rappresentare le signature di una classe e dei suoi membri.

garantisce l'unicità dei tipi classe per ogni dato nome di classe, tramite una memoria che ricorda l'insieme dei tipi classe già creati. Si noti che un tipo classe ha informazione sulla *segnatura* della classe. Ovvero, esso contiene una descrizione dei suoi campi, costruttori e metodi. Vedremo in seguito queste segnature (Sezione 4.2). Il metodo `canBeAssignedTo()` ammette solo l'assegnamento dalla stessa classe o da una sottoclasse del tipo verso cui si assegna. Il test di sottoclasse è realizzato dal metodo `subclass()` che scorre verso l'alto la catena di estensione delle classi alla ricerca dell'ipotetica superclasse. Il metodo `leastCommonSupertype()` determina il minimo sovratipo comune *lcs* fra il tipo classe `this` e un altro tipo `other` secondo le regole seguenti:

- se `other` è un tipo array, allora *lcs* è `Object`, poiché tutte le classi e gli array sono sottotipi di `Object` (Sezione 1.7);
- se anche `other` è un tipo classe allora *lcs* è la più specifica sopraclasse di `this` che è anche sovraclasse di `other`. Si noti che abbiamo la garanzia che tale *lcs* esista poiché questa ricerca si ferma, nel peggiore dei casi, su `Object`;
- se `other` è il tipo `NilType`, allora *lcs* è `this`, poiché `NilType` è sempre un sottotipo dei tipi classe (Sezione 1.7);
- altrimenti *lcs* non esiste.

Due osservazioni sono essenziali in relazione alla classe `ClassType`:

1. la creazione della segnatura della classe da parte del suo costruttore `private` comporta l'accesso al file system, alla ricerca di un file che si chiami come la classe e che termini con `.kit`. Tale file deve essere soggetto ad analisi lessicale e sintattica (sezione 4.2). La segnatura potrebbe fare riferimento ad altri tipi classe. Conseguentemente, la creazione di un tipo classe può provocare la creazione di altri tipi classe, ricorsivamente, ma questo processo termina necessariamente poiché usiamo una memoria che evita la creazione di diversi oggetti `ClassType` per la stessa classe: la seconda volta che si tenta di creare lo stesso tipo classe, la creazione termina immediatamente restituendo lo stesso oggetto creato al primo tentativo;
2. ottenuta la segnatura di una classe, il costruttore di `ClassType` effettua l'analisi semantica della sintassi astratta della classe (metodo `typeCheck()` in Figura 4.7). Vedremo in Sezione 4.6 come questo sia effettuato. Essenzialmente, esso controlla la validità dei punti enumerati nell'introduzione a questo capitolo. Qui osserviamo solo che l'analisi semantica di una classe può richiedere, ricorsivamente, la creazione di tipi classe non ancora esistenti. Ancora una volta, questo processo deve terminare grazie all'uso di una memoria nella creazione dei tipi classe.

```

 $\sigma^\kappa[\_]$  : absyn.ClassMemberDeclaration  $\mapsto$  ClassMemberSignature

 $\sigma^\kappa[\text{FieldDeclaration}(type, name, next)] = \text{new FieldSignature}(\kappa, \tau[type], name)$ 
 $\sigma^\kappa[\text{ConstructorDeclaration}(formals, body, next)] = \text{new ConstructorSignature}(\kappa, \tau[formals])$ 
 $\sigma^\kappa[\text{MethodDeclaration}(returnType, name, formals, body, next)]$ 
    = new MethodSignature( $\kappa, \tau[returnType], \tau[formals], name$ )

```

dove $\tau[formals]$ è `null` se `formals = null` ed è altrimenti l'estensione ai parametri formali della funzione $\tau[_]$ della Figura 4.2:

```

 $\tau[\_]$  : absyn.FormalParameters  $\mapsto$  types.TypeList

 $\tau[\text{FormalParameters}(type, name, next)] = \begin{cases} \text{new TypeList}(\tau[type], \text{null}) & \text{se } next = \text{null} \\ \text{new TypeList}(\tau[type], \tau[next]) & \text{altrimenti.} \end{cases}$ 

```

Figura 4.9: La funzione $\sigma^\kappa[_]$ che associa alla sintassi astratta dei membri di una classe κ la loro segnatura.

4.2 Le signature di classi, campi, costruttori e metodi

Una *segnatura* è una specifica delle proprietà di tipo di un campo, costruttore, metodo o classe. Per esempio, la segnatura di un campo di una classe specifica il nome del campo e il suo tipo semantico di dichiarazione, nonché il tipo semantico della classe in cui il campo è definito. La segnatura di una classe è l'insieme delle signature dei campi, costruttori e metodi che essa definisce, più un riferimento alla segnatura della sua superclasse, se esiste.

La Figura 4.8 mostra le classi del package `types` usate per rappresentare le signature di campi, costruttori, metodi e classi Kitten. Prima di esaminare in dettaglio tali classi, vediamo come sia possibile costruire le signature a partire dalla sintassi astratta di una classe Kitten κ . In particolare, la Figura 4.9 mostra la funzione σ^κ che associa una segnatura alla sintassi astratta di un membro di κ (campo, costruttore o metodo). Costruiamo la segnatura per un tipo classe κ creando un oggetto di classe `ClassSignature` e accumulando al suo interno le signature dei suoi campi, costruttori e metodi, ottenute con la precedente funzione $\sigma^\kappa[_]$. Infine aggiungiamo un riferimento alla segnatura della superclasse di κ , costruita ricorsivamente in modo simile.

La Figura 4.10 mostra l'implementazione delle signature per i membri di una classe (campi, costruttori e metodi). La classe `types/ClassMemberSignature.java` descrive la segnatura di un membro di una classe. Essa contiene semplicemente un riferimento al tipo classe a cui il membro appartiene, inizializzato dal costruttore. La classe `FieldSignature` ha inoltre tipo e nome del campo descritto. La classe `CodeSignature` ha invece una lista di tipi, corrispondenti ai tipi dei parametri formali del costruttore o metodo che essa rappresenta. La classe `ConstructorSignature` è una estensione di `CodeSignature` che non aggiunge alcun campo, mentre `MethodSignature` specifica anche il nome e il tipo di ritorno del metodo.

La classe `ClassSignature` descrive la segnatura di una classe Kitten. Essa è mostrata in Figura 4.11. Anche la segnatura di una classe conosce il tipo classe di cui essa è la segnatura. Inoltre sa quale segnatura sta estendendo (quella della sua superclasse, se esiste). Nonché il parser usato per effettuare l'analisi sintattica del file Kitten che contiene la dichiarazione della classe e l'albero di sintassi astratta risultante dal suo parsing. La segnatura di una classe include inoltre tutte le signature dei suoi membri. In particolare, contiene una funzione (una `java.util.HashMap`) che lega i simboli di campo definiti nella classe alla loro segnatura, cioè a un oggetto di classe `FieldSignature`. Contiene anche un insieme (un `java.util.HashSet`) di `ConstructorSignature` che descrive i costruttori della classe. Nonché una funzione (`java.util.HashMap`) che lega i sim-

```

public abstract class ClassMemberSignature {
    private ClassType clazz;
    protected ClassMemberSignature(ClassType clazz) { this.clazz = clazz; }
    public ClassType getDefiningClass() { return clazz; }
}

public class FieldSignature extends ClassMemberSignature {
    private Type type; // il tipo del campo
    private Symbol name; // il nome del campo
    public FieldSignature(ClassType clazz, Type type, Symbol name) {
        super(clazz); this.type = type; this.name = name;
    }
    ...
}

public abstract class CodeSignature extends ClassMemberSignature {
    private TypeList parameters; // i tipi dei parametri
    protected CodeSignature(ClassType clazz, TypeList parameters) {
        super(clazz); this.parameters = parameters;
    }
}

public class ConstructorSignature extends CodeSignature {
    public ConstructorSignature(ClassType clazz, TypeList parameters) {
        super(clazz,parameters);
    }
    ...
}

public class MethodSignature extends CodeSignature {
    private Symbol name; // il nome del metodo
    private Type returnType; // il suo tipo di ritorno
    public MethodSignature
        (ClassType clazz, Type returnType, TypeList parameters, Symbol name) {
        super(clazz,parameters); this.name = name; this.returnType = returnType;
    }
    ...
}

```

Figura 4.10: Le classi che rappresentano le signature dei membri di una classe Kitten.

boli di metodo definiti nella classe a un insieme (`java.util.HashSet`) di `MethodSignature`. Si noti che serve un insieme poiché lo stesso simbolo potrebbe essere usato per più metodi, in caso di sovraccarico.

I membri della segnatura di una classe possono essere estesi tramite i metodi `addField()`, `addConstructor()` e `addMethod()`. Tali metodi sono richiamati da un metodo `addMember()` di `absyn/ClassMemberSignature.java` che aggiunge a una classe la segnatura di uno dei suoi membri, definita come in Figura 4.9. Tale metodo viene poi richiamato su tutta la lista dei membri di una classe dal metodo `addMembers()` di `absyn.ClassDefinition`. Come si vede in Figura 4.11, tale metodo viene richiamato al momento della costruzione della segnatura di una classe, subito dopo aver effettuato l'analisi lessicale e sintattica della classe e averne appunto ottenuto la sintassi astratta. Se non esistesse nessun file col nome della classe seguito da `.kit` o se tale file contenesse degli errori di sintassi, il metodo `parse()` del parser fallirebbe senza restituire alcuna sintassi astratta. Tale eccezione sarebbe allora intercettata da un gestore di eccezione (non mostrato in Figura 4.11) che fornisce alla classe una sintassi astratta minimale (superclasse `Object`, nessun campo, né costruttori, né metodi). In questo modo si evita di bloccare la compilazione di

```

public class ClassSignature {
    private ClassType clazz; // il tipo classe di cui questa e' la segnatura
    private ClassSignature extendedSignature; // la segnatura della sua superclasse
    private Parser parser; // il parser usato per leggere questa classe
    private ClassDefinition abstractSyntax; // la sua sintassi astratta
    private HashMap fields = new HashMap(); // le sue FieldSignature
    private HashSet constructors = new HashSet(); // le sue ConstructorSignature
    private HashMap methods = new HashMap(); // le sue MethodSignature

    ClassSignature(ClassType clazz) {
        this.clazz = clazz;
        // creiamo un parser per questa classe
        this.parser = new Parser(new Lexer(clazz.getName()));
        // effettuiamo il parsing del file sorgente della classe
        this.abstractSyntax = (ClassDefinition)parser.parse().value;
        // aggiungiamo a questa segnatura le segnature dei suoi membri
        abstractSyntax.addMembers(this);
        // proseguiamo ricorsivamente sulla superclasse, se esiste
        Symbol supName = abstractSyntax.getSuperclassName();
        if (supName != null) this.extendedSignature = ClassType.mk(supName).getSignature();
    }

    public void addField(Symbol name, FieldSignature sig) { ... }
    public void addMethod(Symbol name, MethodSignature sig) { ... }
    public void addConstructor(ConstructorSignature sig) { ... }
    public MethodSignature methodLookup(Symbol name, TypeList formals) { ... }
    public HashSet methodsLookup(Symbol name, TypeList formals) { ... }
    public HashSet constructorsLookup(TypeList formals) { ... }
    public FieldSignature fieldLookup(Symbol name) { ... }
    ...
}

```

Figura 4.11: La classe `types/ClassSignature.java` che rappresenta la segnatura di una classe `Kitten`.

un programma soltanto perché una delle sue classi ha un problema. Si va invece avanti con la compilazione finché si può, segnalando quanto più errori possibile al programmatore.

È possibile cercare un campo, costruttore o metodo a partire dalla segnatura di una classe. `Kitten` implementa l'ereditarietà per campi e metodi, per cui questa ricerca, nel caso di campi e metodi, inizia in una data segnatura e, se tale segnatura non definisce il campo o il metodo, prosegue ricorsivamente verso l'alto risalendo la catena delle estensioni, verso `Object`. I metodi di ricerca sono `fieldLookup()` e `methodLookup()`, che cercano rispettivamente un ben preciso campo, a partire dal suo nome, o un ben preciso metodo, a partire dal suo nome e dal tipo esatto dei suoi parametri formali. Esiste anche il metodo `constructorsLookup()` che restituisce l'insieme S di tutti i costruttori con tipi dei parametri formali compatibili con quelli specificati nella ricerca e con l'ulteriore vincolo che nessun costruttore in S abbia un altro costruttore in S con parametri formali più specifici dei suoi. Per esempio, nella segnatura della classe

```

class Ambiguous {
    constructor(int i, float d) {}
    constructor(float d, int i) {}
    /* constructor(int i1, int i2) {} */
}

```

il risultato di `constructorsLookup()` con per parametro una lista di due `IntType` è l'insieme delle `ConstructorSignature` per i due costruttori della classe. Entrambi sono infatti com-

patibili con due parametri di tipo `int`. Se si eliminasse il commento intorno al terzo costruttore, la stessa chiamata a `constructorsLookup()` restituirebbe un insieme formato da un'unica `ConstructorSignature`, quella per il terzo costruttore, che è più specifico delle altre due. Si può adesso comprendere a cosa ci servirà il metodo `constructorsLookup()`: di fronte a una invocazione del costruttore di `Ambiguous`, del tipo `new Ambiguous(3,4)`, il compilatore Kitten determina prima, tramite `constructorsLookup()`, l'insieme dei possibili costruttori candidati a essere chiamati in tale punto di programma. Se ce ne fosse più d'uno, la chiamata verrebbe considerata *ambigua*. Se non ce ne fosse nessuno, la chiamata verrebbe considerata *indefinita*. In entrambi i casi, essa verrebbe rifiutata in fase di analisi semantica (Figura 4.12, caso per `NewObject`). Simile è la funzione del metodo `methodsLookup()`, che però determina un insieme di metodi piuttosto che di costruttori.

Adesso che abbiamo compreso la struttura di un tipo Kitten e delle segnature di una classe, possiamo esaminare come sia effettuata l'analisi semantica del codice Kitten. Considereremo in ordine i tipi, le espressioni, i comandi e le classi Kitten.

4.3 L'analisi semantica dei tipi Kitten

Effettuare l'analisi semantica dei tipi Kitten significa annotare ciascuna espressione di tipo t che occorre nel programma sorgente con il tipo semantico $\tau[[t]]$ che tale espressioni di tipo rappresenta (Figura 4.2). Occorre inoltre segnalare un errore se per qualche tipo classe κ usato in t non esiste il file $\kappa.kit$ o tale file è corrotto o sintatticamente errato. L'analisi semantica delle espressioni di tipo è implementata tramite un metodo d'istanza di nome `typeCheck()` aggiunto alla classe `absyn/TypeExpression.java`:

```
private Type staticType;

public final Type typeCheck() { return staticType = typeCheck$0(); }

protected abstract Type typeCheck$0();
```

Ancora una volta, come nel caso del metodo `toDot()` della Sezione 3.4.2, usiamo un metodo `typeCheck()` pubblico e `final` che effettua il lavoro comune a tutte le sottoclassi. In questo caso, tale lavoro consiste nell'annotare in un campo `staticType` il tipo semantico inferito per l'espressione di tipo. Tale annotazione potrà essere utile in fase di generazione del codice. Lasciamo invece a un metodo `protected` ausiliario `typeCheck$0()` il compito di completare il lavoro con quanto è specifico a ciascuna sottoclasse. Per esempio, per implementare la definizione di $\tau[[\]]$ data in Figura 4.2, dentro `absyn/IntTypeExpression.java` ridefiniamo:

```
protected Type typeCheck$0() { return Type.INT; }
```

dentro `absyn/ArrayTypeExpression.java` ridefiniamo:

```
protected Type typeCheck$0() { return ArrayType.mk(elementsType.typeCheck()); }
```

e dentro `absyn/ClassTypeExpression.java` ridefiniamo:

```
protected Type typeCheck$0() { return ClassType.mk(name); }
```

Si noti che quest'ultima chiamata al costruttore `mk()` segnala un errore se non esiste alcun file per la classe di nome `name` o se tale file è sintatticamente o semanticamente errato, usando in tal caso una sintassi astratta minimale di default per tale classe (Sezione 4.2).

4.4 L'analisi semantica delle espressioni Kitten

L'analisi semantica delle espressioni Kitten consiste nell'annotare ciascuna espressione e che occorre nel programma sorgente con il suo tipo statico $\tau[[e]]$ a momento di compilazione (Sezione 1.7).

$$\rho : V \mapsto \text{types.Type} \quad \tau^\rho \llbracket _ \rrbracket : \text{absyn.Expression} \rightarrow \text{types.Type}$$

$$\tau^\rho \llbracket \text{Variable}(name) \rrbracket = \rho(name) \quad \text{purché } name \in V$$

$$\tau^\rho \llbracket \text{FieldAccess}(receiver, name) \rrbracket = field.getType()$$

purché $\tau^\rho \llbracket receiver \rrbracket \in \text{ClassType}$ abbia un campo con segnatura *field*

$$\tau^\rho \llbracket \text{ArrayAccess}(array, index) \rrbracket = \tau^\rho \llbracket array \rrbracket .getElementsType()$$

purché $\tau^\rho \llbracket array \rrbracket \in \text{ArrayType}$ e $\tau^\rho \llbracket size \rrbracket = \text{Type.INT}$

$$\tau^\rho \llbracket \text{False}() \rrbracket = \tau^\rho \llbracket \text{True}() \rrbracket = \text{Type.BOOLEAN}$$

$$\tau^\rho \llbracket \text{Nil}() \rrbracket = \text{Types.NIL}$$

$$\tau^\rho \llbracket \text{IntLiteral}(value) \rrbracket = \text{Types.INT}$$

$$\tau^\rho \llbracket \text{FloatLiteral}(value) \rrbracket = \text{Types.FLOAT}$$

$$\tau^\rho \llbracket \text{String}(value) \rrbracket = \text{ClassType.STRING}$$

$$\tau^\rho \llbracket \text{NewObject}(className, actuals) \rrbracket = \text{ClassType.mk}(className)$$

purché esista il costruttore più specifico di
 $\text{ClassType.mk}(className)$ compatibile con $\tau^\rho \llbracket actuals \rrbracket$

$$\tau^\rho \llbracket \text{NewArray}(elementsType, size) \rrbracket = \text{ArrayType.mk}(\tau \llbracket elementsType \rrbracket)$$

purché $\tau^\rho \llbracket size \rrbracket = \text{Type.INT}$

$$\tau^\rho \llbracket \text{MethodCallExpression}(receiver, name, actuals) \rrbracket = method.getReturnType()$$

purché in $\tau^\rho \llbracket receiver \rrbracket \in \text{ClassType}$ esista il metodo più specifico
di nome *name* compatibile con $\tau^\rho \llbracket actuals \rrbracket$ e segnatura *method*
e tale che $method.getReturnType() \neq \text{Type.VOID}$

$$\tau^\rho \llbracket \text{Not}(expression) \rrbracket = \text{Type.BOOLEAN} \quad \text{purché } \tau^\rho \llbracket expression \rrbracket = \text{Type.BOOLEAN}$$

$$\tau^\rho \llbracket \text{Minus}(expression) \rrbracket = \tau^\rho \llbracket expression \rrbracket \quad \text{purché } \tau^\rho \llbracket expression \rrbracket \leq \text{Type.FLOAT}$$

$$\tau^\rho \llbracket \text{Array}(elements) \rrbracket = \text{ArrayType.mk}(\text{minimo sovratipo comune } lcs \text{ fra i tipi } \tau^\rho \llbracket elements \rrbracket)$$

purché *lcs* esista e non sia Type.NIL

$$\tau^\rho \llbracket \text{Cast}(type, expression) \rrbracket = \tau \llbracket type \rrbracket \quad \text{purché } \tau \llbracket type \rrbracket \leq \tau^\rho \llbracket expression \rrbracket$$

$$\tau^\rho \llbracket \text{BooleanBinOp}(left, right) \rrbracket = \text{Type.BOOLEAN}$$

purché $\tau^\rho \llbracket left \rrbracket = \text{Type.BOOLEAN}$ e $\tau^\rho \llbracket right \rrbracket = \text{Type.BOOLEAN}$

$$\tau^\rho \llbracket \text{ArithmeticBinOp}(left, right) \rrbracket = \text{il minimo sovratipo comune } lcs \text{ fra } \tau^\rho \llbracket left \rrbracket \text{ e } \tau^\rho \llbracket right \rrbracket$$

purché $\tau^\rho \llbracket left \rrbracket \leq \text{Type.FLOAT}$ e $\tau^\rho \llbracket right \rrbracket \leq \text{Type.FLOAT}$

$$\tau^\rho \llbracket \text{NumericalComparisonBinOp}(left, right) \rrbracket = \text{Type.BOOLEAN}$$

purché $\tau^\rho \llbracket left \rrbracket \leq \text{Type.FLOAT}$ e $\tau^\rho \llbracket right \rrbracket \leq \text{Type.FLOAT}$

$$\tau^\rho \llbracket \text{Equal}(left, right) \rrbracket = \tau^\rho \llbracket \text{NotEqual}(left, right) \rrbracket = \text{Type.BOOLEAN}$$

purché $\tau^\rho \llbracket left \rrbracket \leq \tau^\rho \llbracket right \rrbracket$ oppure $\tau^\rho \llbracket right \rrbracket \leq \tau^\rho \llbracket left \rrbracket$

Figura 4.12: La funzione di analisi semantica $\tau^\rho \llbracket _ \rrbracket$ per le espressioni Kitten.

Essendo Kitten un linguaggio fortemente tipato, occorre definire $\tau \llbracket e \rrbracket$ in modo che, a tempo di esecuzione, il tipo dinamico di e (cioè il tipo del valore di e , Sezione 1.7) sia $\tau \llbracket e \rrbracket$ o un sottotipo di $\tau \llbracket e \rrbracket$. L'analisi semantica deve inoltre garantire che i tipi siano usati correttamente dentro e . Per esempio, deve rifiutare espressioni e del tipo $3+1$ dove 1 è una variabile dichiarata di tipo `Led` (Figura 1.4). Deve anche determinare il costruttore o metodo che deve essere chiamato a tempo di esecuzione dalle espressioni `new` o dalle invocazioni di metodo contenute in e . Per esempio, deve determinare che l'espressione `1.isOn()` chiama il metodo `isOn()` della Figura 1.4 o una delle ridefinizioni di tale metodo nelle sottoclassi di `Led` (se mai ne esistessero). Questo è essenziale sia per determinare il tipo statico dell'espressione `1.isOn()` (che sarà il tipo di ritorno di `isOn()` in Figura 1.4, cioè `boolean`) che per garantire, a tempo di compilazione, che tale chiamata di metodo non terminerà mai, a tempo di esecuzione, con un'eccezione causata dalla mancata identificazione di un metodo da invocare (questa garanzia è possibile per Kitten poiché esso non ammette il caricamento dinamico delle classi. Non è invece possibile per Java che lo ammette). Infine, tale controllo è utile per la futura generazione del codice oggetto, momento in cui sapremo già con quale codice (o insiemi di codici, nel caso di chiamate virtuali) legare questa invocazione di metodo. Un

discorso analogo si può fare per gli accessi ai campi delle classi, per i quali l'analisi semantica deve identificare la classe che definisce il campo a cui si fa accesso.

Gli esempi precedenti mostrano che la definizione di $\tau[e]$ richiede di conoscere il tipo di dichiarazione delle variabili in scope nel punto di programma in cui e occorre, al fine di determinare il tipo delle variabili contenute in e . Estendiamo quindi la segnatura di $\tau[_]$ in $\tau^\rho[_]$, dove ρ è un *ambiente* o *contesto*. Formalmente $\rho : V \mapsto \text{types.Type}$, dove V è l'insieme delle variabili che sono in scope nel punto di programma in cui e occorre.

La Figura 4.12 definisce $\tau^\rho[_]$. La prima osservazione da fare è che si tratta di una funzione parziale. Per esempio, $\tau^\rho[\text{Variable}(name)]$ non è definito se $name \notin V$, ovvero se la variabile $name$ non è in scope nel punto di programma in cui occorre. Le condizioni che in Figura 4.12 vengono espresse con *purché...* vanno intese come precondizioni per la definizione della funzione $\tau^\rho[_]$. Possiamo immaginare che, quando esse non sono soddisfatte, un messaggio di errore venga comunicato al programmatore (del tipo: *undefined variable name*) e che un tipo arbitrario venga assegnato a $\tau^\rho[_]$ (tipicamente, `Type.INT`), al fine di poter continuare l'analisi semantica senza fermarsi al primo errore semantico incontrato. Si noti che la scelta di tale tipo arbitrario per le situazioni di errore è critica, perché può generare una cascata di errori. Ciò è tipico di molti compilatori, incluso quello per Kitten, per cui gli errori più significativi sono in effetti i primi segnalati dall'analizzatore semantico.

Consideriamo adesso le regole più significative della definizione di $\tau^\rho[_]$ in Figura 4.12.

Variable(name). Abbiamo già osservato che l'ambiente ρ serve proprio a specificare il tipo di dichiarazione delle variabili in scope nel punto di programma in cui occorre l'espressione che stiamo analizzando. In questo caso, quindi, basta leggere il tipo di dichiarazione di $name$ per determinare il tipo statico di **Variable(name)**. Questo è in effetti l'unico caso in cui usiamo esplicitamente l'ambiente ρ . Negli altri casi, ci limiteremo a passarlo ricorsivamente alle componenti dell'espressione che stiamo analizzando.

FieldAccess(receiver, name). L'accesso al campo di nome $name$ dell'oggetto contenuto nell'espressione $receiver$ richiede di determinare il tipo statico di $receiver$. Questo viene fatto con la chiamata ricorsiva che determina $\kappa = \tau^\rho[receiver]$. La precondizione richiede che κ sia una classe, poiché in Kitten solo le classi hanno campi. L'ulteriore richiesta è che κ abbia effettivamente un campo di nome $name$, definito da κ stesso o ereditato da una superclasse di κ . Questo si può verificare con il metodo `fieldLookup()` a partire dalla segnatura di κ (Figura 4.11). Il risultato di tale metodo è la segnatura $field$ del campo a cui si sta facendo riferimento. Il tipo dell'espressione di accesso al campo è quindi il tipo di dichiarazione del campo, ottenibile come `field.getType()`.

ArrayAccess(array, index). L'accesso a un elemento di un array richiede di effettuare ricorsivamente l'analisi semantica dell'espressione $array$ che contiene l'array a cui si accede e dell'espressione $index$ che contiene l'indice in cui si accede nell'array. Si richiede come precondizione che $array$ abbia tipo array t e che $index$ abbia tipo `int`. Il tipo dell'accesso all'array è il tipo degli elementi di t , cioè `t.getElementType()`.

NewObject(className, actuals). Il tipo statico di questa espressione, che crea un oggetto di classe $className$, è il tipo classe κ di nome $className$: $\kappa = \text{ClassType.mk}(className)$. Occorre però controllare che non ci siano errori semantici nei parametri attuali $actuals$. Questo si ottiene richiamando ricorsivamente su di essi l'analisi semantica, cioè calcolando $l = \tau^\rho[actuals]$, che è l'estensione di $\tau^\rho[_]$ a una sequenza di espressioni:

$$\tau^\rho[\text{null}] = \text{null}$$

$$\tau^\rho[\text{ExpressionSeq}(head, tail)] = \begin{cases} \text{new TypeList}(\tau^\rho[head], \text{null}) \\ \text{new TypeList}(\tau^\rho[head], \tau^\rho[tail]) \end{cases}$$

Il calcolo di l serve anche a garantire che ci sia un costruttore più specifico fra quelli di κ che possono essere chiamati con parametri attuali di tipo statico l . Questo è ottenibile chiamando

il metodo `constructorsLookup(l)` sulla segnatura della classe di nome κ (Figura 4.11) e controllando che il risultato sia un insieme di un solo elemento.

MethodCallExpression(*receiver, name, actuals*). L'analisi semantica dell'invocazione di un metodo richiede in primo luogo di effettuare ricorsivamente l'analisi semantica del ricevitore e dei parametri attuali dell'invocazione, cioè di calcolare $\kappa = \tau^\rho[\textit{receiver}]$ e $l = \tau^\rho[\textit{actuals}]$ (quest'ultimo è l'estensione di $\tau^\rho[_]$ alle sequenze di espressioni, si veda sopra il caso di `NewObject`). Si richiede che κ sia un tipo classe, poiché solo le classi hanno metodi in Kitten. Inoltre fra i metodi definiti o ereditati da κ e che possono essere chiamati con parametri attuali di tipo statico l ne deve esistere uno che è più specifico di tutti gli altri. Questo si ottiene chiamando il metodo `methodsLookup(l)` sulla segnatura di κ (Figura 4.11) e verificando che il risultato sia un insieme di un solo elemento, la `MethodSignature` *method*. Il tipo statico dell'invocazione di metodo è quindi il tipo del valore ritornato dal metodo, cioè `method.getReturnType()`, e si richiede che esso non sia `void`, poiché un'espressione deve avere un valore associato a tempo di esecuzione.

Array(*elements*). La creazione di un array a partire da una enumerazione esplicita dei suoi elementi effettua in primo luogo l'analisi semantica di tali elementi, cioè calcola la lista di tipi $l = \tau^\rho[\textit{elements}]$. Quindi calcola il minimo tipo comune *lcs* fra i tipi in l , utilizzando ripetutamente il metodo `leastCommonSupertype()` della Figura 4.3. Il tipo statico della creazione di array è quindi il tipo array con elementi di tipo *lcs*. L'unico vincolo che si richiede è che il minimo sovratipo comune *lcs* esista e non sia `nil`. Questo significa che vengono rifiutate espressioni come `[3, new Led()]` poiché *lcs* non esiste, o come `[nil, nil]` poiché *lcs* sarebbe `nil`. Si noti che questo secondo vincolo non impedisce di creare array i cui elementi siano tutti inizializzati a `nil`. A tal fine, basta usare la sintassi Kitten `new type[size]`. Esso garantisce però di conoscere sempre il tipo, non `nil`, degli elementi dell'array.

Cast(*type, expression*). Quest'espressione effettua il cast di *expression* al tipo *type*. La sua analisi semantica effettua ricorsivamente l'analisi semantica di *type* ed *expression* e poi richiede che il tipo semantica di *type* sia un sottotipo (non necessariamente stretto) del tipo statico di *expression*. Questo vincolo accetta esclusivamente cast verso il basso, scartando per esempio espressioni come `3 as float` o come `student as Persona`. Il motivo per cui tali cast sono rifiutati è che sarebbero sempre veri: è sempre possibile usare un intero dove serve un valore in virgola mobile ed è sempre possibile usare uno studente dove serve una persona. Rifiutando tali cast si obbliga il programmatore a scrivere del codice più semplificato (`3` al posto di `3 as float`, `studente` al posto di `studente as Persona`).

ArithmeticBinOp(*left, right*). L'analisi semantica di un'operazione binaria aritmetica effettua ricorsivamente l'analisi semantica dei suoi due operandi, cioè calcola $\tau^\rho[\textit{left}]$ e $\tau^\rho[\textit{right}]$. Tali due espressioni devono avere un tipo statico che sia `int` o `float`. Il tipo statico del risultato dell'operazione è il minimo sovratipo comune fra $\tau^\rho[\textit{left}]$ e $\tau^\rho[\textit{right}]$. Questo significa per esempio che `3 + 4` ha tipo statico `int` e che `3 + 4.5` ha tipo statico `float`. Si noti che dando una regola per la classe astratta delle operazioni binarie aritmetiche non abbiamo bisogno di specificare esplicitamente alcuna regola per le sue quattro sottoclassi (Figura 3.6).

NumericalComparisonBinOp(*left, right*). Il ragionamento è simile a quello per le espressioni aritmetiche binarie, con l'unica differenza che il risultato di un confronto fra due espressioni è sempre un booleano.

Equal(*left, right*) e NotEqual(*left, right*). L'analisi semantica dell'uguaglianza e della disuguaglianza fra due espressioni richiede di effettuare ricorsivamente l'analisi semantica delle due espressioni confrontate. Richiede inoltre che $\tau^\rho[\textit{left}] \leq \tau^\rho[\textit{right}]$ oppure che $\tau^\rho[\textit{right}] \leq \tau^\rho[\textit{left}]$. Questo vincolo serve a rifiutare espressioni di uguaglianza che non potrebbero mai essere vero ed espressioni di disuguaglianza che sarebbero sempre false. Per esempio, se *p* è una variabile dichiarata di classe `Person`, sottoclasse diretta di `Object`, e *c* è una variabile dichiarata di classe `Car`, anch'essa sottoclasse diretta di `Object`, allora l'uguaglianza `p == c` sarebbe

sempre falsa, poiché non esisterà mai un oggetto che sia al contempo una `Person` e una `Car`. Per lo stesso motivo, la disuguaglianza `p <> c` sarebbe sempre vera. Rifiutando queste espressioni costringiamo il programmatore a eliminare dal suo programma dei test inutili.

4.4.1 L'implementazione dell'analisi semantica delle espressioni

L'implementazione delle regole di Figura 4.12 richiede in primo luogo di implementare l'ambiente ρ . Si potrebbe pensare di utilizzare una semplice `java.util.HashMap` che lega le variabili ai loro tipi di dichiarazione. Ma fra poco (Sezione 4.5) avremo bisogno di un'operazione di estensione *non distruttiva* sugli ambienti, tale cioè da lasciare il vecchio ambiente intatto dopo la sua applicazione. Questo rende l'uso di `java.util.HashMap` sconveniente, poiché tale struttura dati ha solo operazioni distruttive. Decidiamo quindi di usare una nostra struttura dati per rappresentare gli ambienti, cioè la classe `symbol/Table.java` e le sue due sottoclassi in Figura 4.13. L'interfaccia `symbol.Table` specifica semplicemente che un ambiente ha un'operazione `get(key)` che permette di leggere il valore di una variabile (`symbol.Symbol` `key`) e un'operazione `put(key, value)` che costruisce un nuovo ambiente in cui la variabile `key` è legata a `value`. Si noti che le variabili sono genericamente legate a degli `Object`, benché a noi servirebbero degli ambienti che legano le variabili a dei `types.Type`. Questo dà maggiore generalità a questi ambienti, che in futuro potrebbero essere usati per altri scopi, in cui alle variabili sono legate strutture dati diverse da `types.Type`. Si noti inoltre che il metodo `put()` restituisce un *nuovo* ambiente con aggiunto un nuovo legame, mentre il vecchio ambiente non è modificato ed è ancora utilizzabile. Questo al fine di implementare un'operazione `put()` non distruttiva, come volevamo.

Le due sottoclassi di `symbol.Table` sono `symbol.EmptyTable` e `symbol.NonEmptyTable`. La prima implementa un ambiente vuoto in cui non esiste alcun legame per le variabili. La seconda implementa un ambiente in cui c'è almeno un legame per una variabile. Questo ambiente è rappresentato come un albero binario di ricerca, in cui cioè le variabili che precedono la radice, in ordine lessicografico, vanno cercate nel sottoalbero di sinistra e quelle che la seguono vanno cercate nel sottoalbero destro. Questo è proprio quello che fa il suo metodo `get()` (Figura 4.13). Il metodo `put()` di `symbol.NonEmptyTable` costruisce un *nuovo* albero binario in cui la variabile è legata al valore passato come argomento, senza modificare l'albero originale. Esso implementa quindi un'inserzione non distruttiva. La Figura 4.14 mostra come l'inserzione venga effettuata. Al posto di ricreare integralmente l'albero binario, se ne condivide una gran parte, ricostruendo solo il cammino dalla radice dell'albero al nodo che è stato aggiunto o modificato nell'albero.

Gli ambienti sono contenuti in un *wrapper* che chiamiamo *type-checker*, implementato dalla classe `semantical/TypeChecker.java` mostrata nella Figura 4.15. L'esistenza di tale wrapper risulterà significativa nella Sezione 4.5. Per adesso infatti l'ambiente è tutto quello di cui abbiamo bisogno per effettuare l'analisi semantica delle espressioni, ma per i comandi avremo bisogno di ulteriori informazioni che aggiungeremo a questo wrapper. La classe in Figura 4.15 verrà significativamente riscritta nella Sezione 4.5, ma per adesso possiamo accontentarci della definizione contenuta in tale figura.

Implementati gli ambienti dentro dei *type-checker*, possiamo implementare le regole della Figura 4.12 tramite una discesa ricorsiva sulla sintassi astratta delle espressioni. Dentro la classe `absyn/Expression.java` aggiungiamo:

```
private Type staticType;
private TypeChecker checker;

public final Type typeCheck(TypeChecker checker) {
    return staticType = typeCheck$(this.checker = checker);
}

protected abstract Type typeCheck$(TypeChecker checker);
```

Ancora una volta, un metodo `public` e `final`, di nome `typeCheck()`, effettua le operazioni comuni a tutte le espressioni, che consistono nell'annotare il tipo statico inferito per l'espressione e il

```

public abstract class Table {
    public final static EmptyTable EMPTY = new EmptyTable();
    public abstract Object get(Symbol key);
    public abstract Table put(Symbol key, Object value);
}

class EmptyTable extends Table {
    EmptyTable() {}
    public Object get(Symbol key) { return null; }

    public Table put(Symbol key, Object value) {
        return new NonEmptyTable(key,value);
    }
}

class NonEmptyTable extends Table {
    private Symbol key;
    private Object value;
    private Table left, right;

    private NonEmptyTable(Symbol key, Object value, Table left, Table right) {
        this.key = key; this.value = value;
        this.left = left; this.right = right;
    }

    NonEmptyTable(Symbol key, Object value) { this(key,value,EMPTY,EMPTY); }

    public Object get(Symbol key) {
        int comp = this.key.compareTo(key);
        if (comp < 0) return left.get(key);
        else if (comp == 0) return value;
        else return right.get(key);
    }

    public Table put(Symbol key, Object value) {
        Table temp;
        int comp = this.key.compareTo(key);

        if (comp < 0) {
            temp = left.put(key,value);
            if (temp == left) return this;
            else return new NonEmptyTable(this.key,this.value,temp,right);
        }
        else if (comp == 0)
            if (value == this.value) return this;
            else return new NonEmptyTable(this.key,this.value,left,right);
        else {
            temp = right.put(key,value);
            if (temp == right) return this;
            else return new NonEmptyTable(this.key,this.value,left,temp);
        }
    }
}

```

Figura 4.13: Le classi del package `symbol` che implementano gli ambienti per l'analisi semantica di espressioni e comandi.

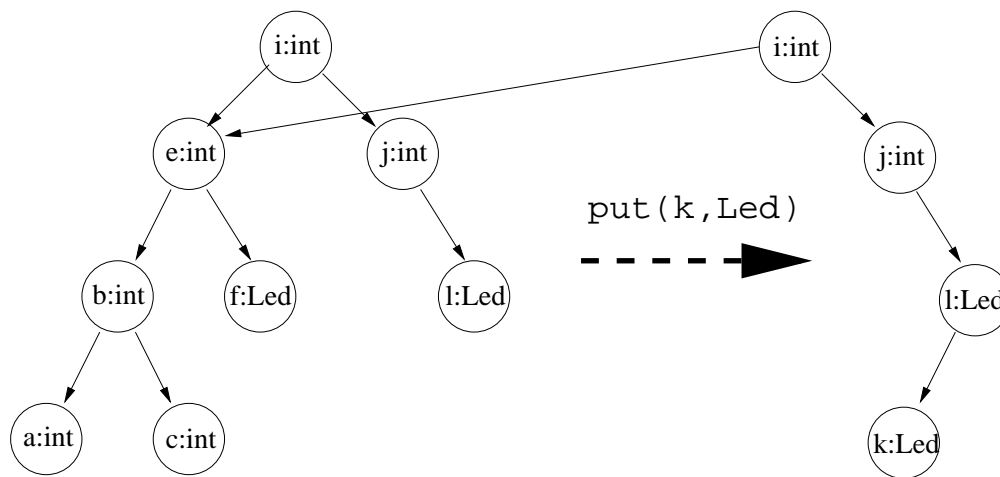


Figura 4.14: L'inserzione non distruttiva in un ambiente di un legame per una variabile.

type-checker usato per inferirlo. Un metodo ausiliario e `protected`, di nome `typeCheck$0()`, implementa le operazioni specifiche all'espressione, come specificate nella Figura 4.12.

Vediamo adesso alcuni esempi di implementazione delle regole in Figura 4.12 tramite il metodo `typeCheck$0()`. Dentro la classe `absyn/Variable.java` ridefiniamo:

```
protected Type typeCheck$0(TypeChecker checker) {
    Type result = checker.getVar(name);
    if (result == null) {
        error(checker, "undefined variable " + name);
        result = Type.INT;
    }
    return result;
}
```

Questa implementazione riflette la sua specifica nella Figura 4.12: si prova a cercare la variabile nell'ambiente; se non viene trovata, si dà un errore e si assume un tipo d'emergenza, tipicamente `int`. Il metodo `error()` è definito dentro `absyn/Absyn.java` come

```
protected void error(TypeChecker checker, String msg) { checker.error(pos,msg); }
```

Si noti come esso utilizzi il campo `pos` per indicare in che punto dare l'errore all'utente. Tale campo era il numero di caratteri passati dall'inizio del file a un punto significativo della parte di sintassi astratta in considerazione (Sezione 3.3).

Esaminiamo un altro esempio, quello di `absyn/FieldAccess.java`:

```
protected Type typeCheck$0(TypeChecker checker) {
    Type receiverType = receiver.typeCheck(checker);
    if (!(receiverType instanceof ClassType)) {
        error(checker, "class type required"); return Type.INT;
    }
    ClassType receiverClass = (ClassType)receiverType;
    if ((field = receiverClass.getSignature().fieldLookup(name)) == null) {
        error(checker, "unknown field " + name); return Type.INT;
    }
    return field.getType();
}
```

```

public class TypeChecker {
    private Table env;
    private ErrorMsg errorMsg;

    public TypeChecker(ErrorMsg errorMsg) {
        env = Table.EMPTY;
        this.errorMsg = errorMsg;
    }

    public TypeChecker putVar(Symbol var, Type type) {
        return new TypeChecker(env.put(var,type));
    }

    public Type getVar(Symbol var) { return (Type)env.get(var); }

    public void error(int pos, String msg) {
        errorMsg.error(pos,msg);
    }
}

```

Figura 4.15: Il type-checker usato per effettuare l'analisi semantica delle espressioni.

Consistentemente con la Figura 4.12, tale metodo effettua ricorsivamente l'analisi semantica di `receiver` e quindi impone che esso abbia un tipo classe. Infine cerca il campo di nome `name` dentro tale tipo classe e ne restituisce il tipo di dichiarazione.

Un altro esempio è quello di `absyn/ArrayAccess.java`:

```

protected Type typeCheck$0(TypeChecker checker) {
    Type arrayType = array.typeCheck(checker);
    index.checkItIsInt(checker);
    if (!(arrayType instanceof ArrayType)) {
        error(checker,"array type required"); return Type.INT;
    }
    return ((ArrayType)arrayType).getElementsType();
}

```

Consistentemente con la Figura 4.12, tale metodo effettua ricorsivamente l'analisi semantica di `array` e `index`. Per `index` usa il metodo ausiliario `checkItIsInt()` che è definito dentro `absyn/Expression.java` come:

```

protected void checkItIsInt(TypeChecker checker) {
    if (typeCheck(checker) != Type.INT) error(checker,"integer expected");
}

```

Come ultimo esempio, consideriamo la definizione del metodo `typeCheck$0()` dentro la classe `absyn/ArithmeticBinOp.java`:

```

protected Type typeCheck$0(TypeChecker checker) {
    Type leftType = getLeft().typeCheck(checker);
    Type rightType = getRight().typeCheck(checker);
    if (leftType.canBeAssignedTo(Type.FLOAT) && rightType.canBeAssignedTo(Type.FLOAT))
        return leftType.leastCommonSupertype(rightType);
    else { error(checker,"numerical argument required"); return Type.INT; }
}

```

Consistentemente con la Figura 4.12, tale metodo effettua ricorsivamente l'analisi semantica di `left` e `right` e impone che abbiano un tipo statico che sia `float` o un sottotipo di `float`. Il tipo statico dell'operazione binaria è il minimo sovratipo comune fra i tipi statici di `left` e `right`.

$$\rho : V \mapsto \text{types.Type} \quad \tau^\rho[-] : \text{absyn.Command} \rightarrow \text{absyn.Command}$$

$\tau^\rho[\text{Skip}()] = \rho$
 $\tau^\rho[\text{Break}()] = \tau^\rho[\text{Continue}()] = \rho$
 purché il comando occorra dentro a un ciclo
 $\tau^\rho[\text{IfThenElse}(condition, then, _else)] = \rho$
 purché $\tau^\rho[condition] = \text{Type.BOOLEAN}$,
 $\tau^\rho[then] = \rho'$ e, se $_else \neq \text{null}$, anche $\tau^\rho[_else] = \rho''$
 $\tau^\rho[\text{Return}(expression)] = \rho$
 purché $expression = \text{null}$ e il comando occorre
 dentro un metodo che ritorna null o dentro un costruttore
 oppure $expression \neq \text{null}$ e il comando occorre
 dentro un metodo che ritorna $t \geq \tau^\rho[expression]$
 $\tau^\rho[\text{Assignment}(lvalue, rvalue)] = \rho$
 purché $\tau^\rho[rvalue] \leq \tau^\rho[lvalue]$
 $\tau^\rho[\text{For}(initialisation, condition, update, body)] = \rho$
 purché $\tau^\rho[initialisation] = \rho'$,
 $\tau^{\rho'}[condition] = \text{Type.BOOLEAN}$, $\tau^{\rho'}[update] = \rho''$ e $\tau^{\rho'}[body] = \rho'''$
 $\tau^\rho[\text{While}(condition, body)] = \rho$
 purché $\tau^\rho[condition] = \text{Type.BOOLEAN}$ e $\tau^\rho[body] = \rho'$
 $\tau^\rho[\text{LocalDeclaration}(type, name, initialiser)] = \rho[name \mapsto \tau[type]]$
 purché, se $initialiser \neq \text{null}$, allora $\tau^\rho[initialiser] \leq \tau[type]$
 $\tau^\rho[\text{LocalScope}(body)] = \rho$
 purché $\tau^\rho[body] = \rho'$
 $\tau^\rho[\text{MethodCallCommand}(receiver, name, actuals)] = \rho$
 purché in $\tau^\rho[receiver] \in \text{ClassType}$ esista il metodo più specifico
 di nome $name$ compatibile con $\tau^\rho[actuals]$

Figura 4.16: La funzione di analisi semantica $\tau^\rho[-]$ per i comandi Kitten.

4.5 L'analisi semantica dei comandi Kitten

La Figura 4.16 mostra le regole di analisi semantica per i comandi Kitten. Si noti che la funzione $\tau^\rho[-]$ associa adesso un ambiente a ogni comando, mentre essa associava un tipo a ogni espressione (Figura 4.12). Il senso della notazione $\tau^\rho[c] = \rho'$ è che il comando c eseguito a partire da un ambiente ρ porta in un ambiente ρ' . Normalmente, $\rho' = \rho$ e la Figura 4.16 esprime esclusivamente delle precondizioni per la correttezza dell'analisi semantica. C'è solo un caso in cui $\rho' \neq \rho$ ed è la dichiarazione di una variabile (una `LocalDeclaration` in Figura 4.16). In tal caso, infatti, l'ambiente viene esteso con una nuova variabile locale.

Esaminiamo adesso alcune regole della Figura 4.16:

Break() e Continue(). L'analisi semantica di questi comandi richiede semplicemente di controllare che essi occorranza nel corpo di un ciclo (`for` o `while`).

IfThenElse(condition, then, _else). Il condizionale richiede che la condizione sia un'espressione di tipo booleano. Inoltre, effettua ricorsivamente l'analisi semantica di *then* e, se esiste, anche di *_else*. Si noti che la scelta di avere ρ come risultato dell'analisi semantica del condizionale implica che eventuali variabili locali dichiarate all'interno del ramo *then* o del ramo *_else* del comando non sono più in scope alla fine del condizionale.

Return(*expression*). Il comando di ritorno da metodo o costruttore richiede di effettuare ricorsivamente l'analisi semantica dell'espressione ritornata, se esiste. Nel caso in cui essa non sia `null`, allora questo comando deve occorrere dentro un metodo che ritorna il tipo statico di *expression* o un suo sovratipo. Altrimenti questo comando deve occorrere dentro un metodo che ritorna `void` o dentro un costruttore.

Assignment(*lvalue*, *rvalue*). L'analisi semantica dell'assegnamento del valore di un'espressione a un `lvalue` consiste nel controllare che il tipo statico dell'espressione sia lo stesso o un sottotipo del tipo statico del `lvalue`.

For(*initialisation*, *condition*, *update*, *body*). L'analisi semantica del ciclo `for` comincia analizzando ricorsivamente il comando *initialisation*. Il risultato di questa analisi è un ambiente ρ' . Inoltre si impone che la condizione abbia tipo booleano. Si noti che l'ambiente ρ' viene usato per effettuare l'analisi semantica di *initialisation*, *update* e *body*. Questo al fine di permettere al programmatore di dichiarare variabili locali dentro *initialisation* e di usarle nelle altre componenti del `for`, come in

```
for (int i := 0; i < 5; i := i + 1) "".concat(i).output()
```

Se si fosse usato ρ per l'analisi di *condition*, *update* e *body*, la variabile `i` sarebbe risultata indefinita o avrebbe fatto riferimento a un'altra variabile, definita esternamente al ciclo.

LocalDeclaration(*type*, *name*, *initialiser*). L'analisi semantica della dichiarazione di una variabile locale estende l'ambiente legando la variabile *name* al tipo semantico di *type*. Nel caso in cui *initialiser* non fosse `null`, si effettua ricorsivamente la sua analisi semantica e si impone che il tipo statico di *initialiser* sia lo stesso o un sottotipo del tipo semantico di *type*.

LocalScope(*body*). L'analisi semantica della creazione di uno scope locale effettua ricorsivamente l'analisi semantica del corpo dello scope. Definendo ρ come risultato di questa analisi semantica, facciamo in modo che eventuali variabili locali dichiarate all'interno del corpo non siano più visibili all'esterno dello scope. Questo significa, per esempio, che in un comando come `{ int a; a := 5 }` la variabile `a` non è più visibile dopo la parentesi graffa di chiusura.

MethodCallCommand(*receiver*, *name*, *actuals*). L'analisi semantica del comando di invocazione di metodo è estremamente simile a quella dell'espressioni di invocazione di metodo in Figura 4.12. L'unica differenza è che qui non si impone alcun vincolo sul tipo di ritorno del metodo, che può quindi ritornare anche `void`.

4.5.1 L'implementazione dell'analisi semantica dei comandi

Abbiamo già visto nella Sezione 4.5.1 come implementare gli ambienti ρ dentro un type checker. Dal momento che l'analisi semantica di un comando restituisce un ambiente, implementiamo il metodo di analisi semantica dentro `absyn/Command.java` come

```
private TypeChecker checker;

public final TypeChecker typeCheck(TypeChecker checker) {
    this.checker = checker = typeCheck$(checker);
    if (next != null) return next.typeCheck(checker);
    else return checker;
}

protected abstract TypeChecker typeCheck$(TypeChecker checker);
```

Il metodo `public` e `final` di nome `typeCheck()` effettua la parte di analisi semantica comune a tutti i comandi, che consiste nel chiamare il metodo ausiliario `typeCheck$0()`, annotare il type-checker risultante dall'analisi e richiamarsi ricorsivamente sul comando seguente, se esiste. In questo modo si implementa l'analisi semantica delle sequenze di comandi come $\tau^p[[c_1; c_2]] = \tau^p[[c_1]][[c_2]]$.

Il metodo `typeCheck$0()` effettua la parte di analisi semantica specifica a ciascun comando, secondo le regole della Figura 4.16. Per esempio, dentro `absyn/IfThenElse.java` lo ridefiniamo come

```
protected TypeChecker typeCheck$0(TypeChecker checker) {
    condition.checkItIsBoolean(checker);
    then.typeCheck(checker);
    if (_else != null) _else.typeCheck(checker);
    return checker;
}
```

Invece dentro `absyn/For.java` lo ridefiniamo come

```
protected TypeChecker typeCheck$0(TypeChecker checker) {
    TypeChecker initChecker = initialisation.typeCheck(checker);
    condition.checkItIsBoolean(initChecker);
    update.typeCheck(initChecker);
    body.typeCheck(initChecker.cloneIntoLoopMode());
    return checker;
}
```

Si noti in quest'ultimo esempio come l'ambiente (in effetti, il type-checker) risultante dall'analisi di `initialisation` sia poi usato per effettuare l'analisi semantica di `condition`, `update` e `body`, conformemente alla Figura 4.16. L'uso del metodo `cloneIntoLoopMode()` sul type-checker serve a indicare che l'analisi del corpo del `for` deve essere fatta con un type-checker che sa di essere dentro a un ciclo. Questo è implementato come in Figura 4.17, che è una estensione del codice in Figura 4.15. Un campo booleano `loop` dice se il type-checker è usato per analizzare il corpo di un ciclo. Come abbiamo visto sopra, tale flag è attivato quando si analizza il corpo di un `for` (e di un `while`). Esso è poi letto per effettuare l'analisi semantica di `break` e `continue`. Per esempio, dentro `absyn/Break.java` definiamo:

```
protected TypeChecker typeCheck$0(TypeChecker checker) {
    if (!checker.isInLoopMode()) error(checker, "illegal break statement");
    return checker;
}
```

La Figura 4.17 mostra che il type-checker è stato potenziato in altre direzioni. Oltre al campo `loop`, esso conosce qual è il tipo di ritorno che ci si aspetta dal metodo che si sta analizzando. Tale tipo deve essere fornito al momento della costruzione del type-checker tramite l'unico costruttore pubblico in Figura 4.17. Esso è utile per implementare l'analisi semantica dei `return`: dentro `absyn/Return.java` definiamo

```
protected TypeChecker typeCheck$0(TypeChecker checker) {
    Type expectedReturnType = checker.getReturnType();
    if (returned == null && expectedReturnType != Type.VOID)
        error(checker, "missing return value");
    if (returned != null &&
        !returned.typeCheck(checker).canBeAssignedTo(expectedReturnType))
        error(checker, "illegal return type: " + expectedReturnType + " expected");
    return checker;
}
```

```

public class TypeChecker {
    private Type returnType;    private Table env;           private int varNum;
    private boolean loop;      private ErrorMsg errorMsg;

    private TypeChecker(Type returnType, Table env, int varNum, boolean loop,
                        ErrorMsg errorMsg) {
        this.returnType = returnType; this.env = env;
        this.varNum = varNum; this.loop = loop; this.errorMsg = errorMsg; }

    public TypeChecker(Type returnType, ErrorMsg errorMsg) {
        this(returnType, Table.EMPTY, 0, false, errorMsg); }

    public TypeChecker cloneIntoLoopMode() {
        return new TypeChecker(returnType, env, varNum, true, errorMsg); }

    public boolean isInLoopMode() { return loop; }

    public TypeChecker setReturnType(Type returnType) {
        return new TypeChecker(returnType, env, varNum, loop, errorMsg); }

    public Type getReturnType() { return returnType; }

    public TypeChecker putVar(Symbol var, Type type) {
        return new TypeChecker
            (returnType, env.put(var, new TypeAndNumber(type, varNum)),
             varNum + 1, loop, errorMsg); }

    public Type getVar(Symbol var) {
        TypeAndNumber tan = (TypeAndNumber)env.get(var);
        if (tan != null) return tan.getType(); else return null; }

    public int getVarNum(Symbol var) {
        TypeAndNumber tan = (TypeAndNumber)env.get(var);
        if (tan != null) return tan.getNumber(); else return -1; }
    ...
}

```

Figura 4.17: La classe `semantical/TypeChecker.java` che implementa un type-checker

conformemente alla Figura 4.16.

Infine, il type-checker in Figura 4.17 associa alle variabile dell'ambiente non solo il loro tipo di dichiarazione, ma anche un numero progressivo, che sarà utile in fase di generazione del codice (Capitolo ??).

4.6 L'analisi semantica delle classi Kitten

Fare l'analisi semantica di una classe Kitten significa effettuare l'analisi semantica dei suoi membri, cioè campi, costruttori e metodi. Nulla va controllato per quanto riguarda i campi. Per quanto riguarda costruttori e metodi, invece, occorre effettuare l'analisi semantica del loro corpo. Essendo il loro corpo un comando, possiamo usare a tal fine le regole della Figura 4.16, cominciando l'analisi da un ambiente iniziale in cui i parametri del costruttore o del metodo sono legati al loro tipo di dichiarazione (incluso il parametro implicito `this`). A tal fine, definiamo una funzione che

aggiunge a un ambiente una lista di variabili dichiarate come parametri formali:

$$\begin{aligned} \rho + \text{null} &= \rho \\ \rho + \text{FormalParameters}(type, name, next) &= (\rho + next)[name \mapsto \tau[type]] \end{aligned}$$

L'analisi semantica di un costruttore o metodo con parametri formali *formals* e dichiarato in una classe il cui tipo semantico è κ viene quindi effettuata a partire da un ambiente iniziale

$$\bar{\rho} = [\text{this} \mapsto \kappa] + \text{formals}$$

Se *body* è il corpo del costruttore o metodo, si tratterà semplicemente di calcolare $\tau^{\bar{\rho}}[\text{body}]$.

4.6.1 L'implementazione dell'analisi semantica delle classi

Il metodo che effettua l'analisi semantica dei membri di una classe si chiama ancora `typeCheck()`. Esso è definito dentro `absyn/ClassMemberDeclaration.java` come

```
final void typeCheck(ClassType currentClass) {
    typeCheck$0(currentClass);
    if (next != null) next.typeCheck(currentClass);
}
```

```
protected abstract void typeCheck$0(ClassType currentClass);
```

ovvero tramite il solito metodo `final` che richiama, su tutta la lista dei membri della classe, il metodo ausiliario `typeCheck$0()` che effettua l'analisi specifica a ciascun membro. Abbiamo detto che l'analisi semantica dei campi non richiede nessun controllo: dentro `absyn/FieldDeclaration.java` definiamo

```
protected void typeCheck$0(ClassType currentClass) {}
```

Invece, per i costruttori definiamo

```
protected void typeCheck$0(ClassType currentClass) {
    TypeChecker = new TypeChecker
        (Type.VOID, currentClass.getSignature().getParser().getErrMsg());
    checker = checker.putVar(Symbol.THIS, currentClass);
    if (getFormals() != null) checker = getFormals().typeCheck(checker);
    getBody().typeCheck(checker);
}
```

Come si vede, si comincia col costruire un type-checker con ambiente vuoto e che si aspetta come tipo di ritorno `Type.VOID`. Quindi si aggiunge al suo interno la variabile `this` legata al tipo semantico della classe e i parametri formali legati al proprio tipo di dichiarazione, secondo la definizione precedente di $\bar{\rho}$. Quindi si effettua il type-checking del corpo del costruttore. Il ragionamento è simile nel caso della dichiarazione di un metodo, ma si usa il tipo di ritorno del metodo al posto di `Type.VOID`. Inoltre, occorre controllare che, se il metodo ridefinisce un metodo di una super-classe, allora la ridefinizione del tipo di ritorno soddisfa il test `canBeAssignedToSpecial()` sui tipi semantici visto in Sezione 4.1.

4.7 Identificazione del codice morto

Esiste ancora un ulteriore controllo affidato all'analisi semantica. Esso si occupa di rifiutare un metodo del tipo:

$$\delta : \text{absyn.Command} \mapsto \{true, false\}$$

$$\begin{aligned} \delta(\text{Break}()) &= true \\ \delta(\text{Continue}()) &= true \\ \delta(\text{Return}(returned)) &= true \\ \delta(\text{IfThenElse}(condition, then, _else)) &= \delta(then) \wedge _else \neq \text{null} \wedge \delta(_else) \\ \delta(\text{For}(initialisation, condition, update, body)) &= false \\ \text{purch\`e } \delta(initialisation) &= false \text{ e } \delta(update) = b_1 \text{ e } \delta(body) = b_2 \\ \delta(\text{While}(condition, body)) &= false \\ \text{purch\`e } \delta(body) &= b \\ \delta(\text{LocalDeclaration}(type, name, initialiser)) &= false \\ \delta(\text{LocalScope}(body)) &= \delta(body) \\ \delta(\text{Skip}()) &= false \\ \delta(\text{MethodCallCommand}(receiver, name, actuals)) &= false \\ \delta(\text{Assignment}(lvalue, rvalue)) &= false \end{aligned}$$

Figura 4.18: La funzione δ che determina se un comando, se termina, allora termina sicuramente con `return`, `break` o `continue`.

```
method int fib(int i) {
  if (i < 2) then return 1
  else return this.fib(i - 1) + this.fib(i - 2);

  "ciao".output()
}
```

Il motivo è che la linea `ciao.output()` non può mai essere raggiunta. Si parla in tal caso di *codice morto*. Evitare la compilazione di programmi che contengono codice morto può sembrare eccessivamente restrittivo, ma è invece spesso importante poiché costringe il programmatore a ragionare sulla struttura di controllo del codice. Molto spesso, infatti, la presenza di codice morto è un sintomo di un bug in un programma.

Un problema imparentato a quello del codice morto è quello del codice che non termina necessariamente con un `return`. Questo è importante nel caso di metodi che non ritornano `void`, come per esempio

```
method int fib(int i)
  if (i < 2) then return 1
```

Poiché non è specificato cosa deve essere ritornato nel caso in cui il parametro `i` fosse maggiore o uguale a 2, il precedente metodo deve essere rifiutato come incorretto e non compilato in codice eseguibile. In altre parole, per i metodi che non ritornano `void` pretendiamo che qualsiasi percorso che porta a concludere l'esecuzione del metodo termini con un'istruzione `return`. Si noti che abbiamo già controllato che il tipo dell'espressione ritornato sia compatibile con il tipo di ritorno del metodo (Sezione 4.5).

Il modo in cui risolviamo entrambi questi problemi è tramite una funzione $\delta : \text{absyn.Command} \mapsto \{true, false\}$. Il suo significato è il seguente: $\delta(c)$ è vero se e solo se ogni esecuzione del comando c , se termina, allora termina con un comando fra `return`, `break` e `continue`. Per esempio, è ovvio che $\delta(\text{Break}()) = true$ e che $\delta(\text{Skip}()) = false$. La definizione completa di questa funzione è data in Figura 4.18.

Discutiamo alcuni dei casi in Figura 4.18.

IfThenElse(*condition*, *then*, *_else*). Il condizionale termina sicuramente con un **return**, **break** o **continue**, se questo è vero per *entrambi* i suoi rami. Inoltre, deve essere presente il ramo **else**. Si noti che questa condizione è sufficiente ma non necessaria. Per esempio, sarebbe sensato definire $\delta = true$ per il comando `if (a = a) then return`. Le regole in Figura 4.18 vanno intese come regole corrette ma non complete per garantire che un comando termini con un **return**, **break** o **continue**. La determinazione corretta e completa di tale proprietà è purtroppo impossibile, dal momento che si tratta di una proprietà indecibile dei programmi. Si noti che la stessa approssimazione viene usata dai compilatori Java.

For(*initialisation*, *condition*, *update*, *body*) e While(*condition*, *body*). Dal momento che non è decidibile se un ciclo verrà eseguito almeno una volta, decidiamo che non è mai possibile dire se un ciclo termina sicuramente con un **return**, **break** o **continue**, neppure quando il suo corpo è uno di questi tre comandi!

La funzione δ della Figura 4.18 è implementata dentro `absyn/Command.java` da un metodo `checkForDeadcode()` che scorre una sequenza di comandi legati a lista e richiama il metodo ausiliario `checkForDeadcode$0()` che implementa le regole in Figura 4.18:

```
public final boolean checkForDeadcode(TypeChecker checker) {
    boolean stopping = checkForDeadcode$0(checker);
    if (next != null) {
        if (stopping)
            error(checker, "unreachable code after this statement");
        return next.checkForDeadcode(checker);
    }
    else return stopping;
}

protected abstract boolean checkForDeadcode$0(TypeChecker checker);
```

Si noti come il valore di ritorno di `checkForDeadcode$0()` (cioè della funzione δ in Figura 4.18) sia usato dentro `checkForDeadcode()` per determinare il codice morto: se in una sequenza di comandi $c_1;c_2$ il primo comando c_1 , se termina, termina sicuramente con **return**, **break** o **continue** (cioè, nel codice precedente, se `stopping` è vero), allora c_2 è sicuramente codice morto.

Il metodo `checkForDeadcode()` viene richiamato alla fine dell'analisi semantica di costruttori e metodi, aggiungendo la sua chiamata al codice riportato in Sezione 4.6.1. Nel caso di metodi che non ritornano `void`, si impone inoltre che il suo valore di ritorno, per il corpo del metodo, sia *true*. Questo significa che il corpo del metodo, se termina, termina sicuramente con un comando **return**, **break** o **continue**. Dal momento che abbiamo già controllato che **break** e **continue** occorrono sempre dentro un ciclo (Sezione 4.5), concludiamo che essi non possono mai essere il comando con cui termina il corpo di un metodo. In altre parole, Questo è impone che il corpo di un metodo che non ritorna `void`, se termina, allora termina con un'istruzione **return**. Abbiamo per esempio reso illegale il secondo dei condizionali visti all'inizio di questa sezione.