

Capitolo 5

Analisi Semantica



Le analisi lessicale e sintattica dei Capitoli 2 e 3 garantiscono che il programma sorgente soddisfi le regole di sintassi specificate, rispettivamente, da un insieme di token e da una grammatica. L'analisi sintattica ha inoltre costruito un albero di sintassi astratta che fornisce una visione strutturata 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 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, ovvero quello della discesa ricorsiva sull'albero di sintassi astratta del codice sorgente, alla ricerca di errori *semantici* nel codice. Questa *analisi semantica* è l'oggetto di questo capitolo.

Più in dettaglio, i compiti di un'analisi semantica sono quelli di

1. costruire una rappresentazione (una struttura dati) che descrive i tipi usati dal programma (tipi primitivi ma anche array e classi);
2. identificare usi di espressioni incompatibili con i loro tipi statici (*errori di tipo*);
3. identificare occorrenze di variabili usate ma non dichiarate;

4. garantire che un metodo non `void` termini sempre con un comando `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`, sulla base del tipo statico dei parametri attuali, il costruttore di *Classe* che deve essere chiamato in tale punto a tempo di esecuzione;
9. identificare per ogni invocazione di metodo, sulla base del tipo statico dei parametri attuali, la dichiarazione del metodo che deve essere chiamato (o una cui ridefinizione deve essere chiamata) in tal punto a tempo di esecuzione.

Potremmo quindi dire che l'analisi semantica si occupa di costruire una rappresentazione dei tipi usati dal programma (punto 1) che viene poi usata per garantire condizioni di correttezza elementari, senza le quali non ha neppure senso compilare il programma in codice oggetto (*verifica del codice*, punti 2–5) e per 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 varia comunque molto da linguaggio a linguaggio. Altre verifiche effettuate da Java *ma non da Kitten* sono per esempio:

10. garantire che i comandi `break` e `continue` occorrono solo dentro un costrutto iterativo o, per il solo `break`, dentro un comando `switch`;
11. garantire che l'uso di una variabile locale trovi la variabile inizializzata, indipendentemente dal percorso di esecuzione che ha portato al punto di utilizzo della variabile¹.

5.1 I tipi Kitten

Il concetto di *tipo* (Sezione 1.8) è al centro dell'analisi semantica (punti 1,2,4,6,8,9 della precedente enumerazione). Va subito notato che per *tipo* non intendiamo qui la sintassi astratta di una *espressione* di tipo, come nella Sezione 3.6.1. In quel contesto avevamo bisogno di un modo per rappresentare la *struttura sintattica* di una parte di codice che rappresentava un tipo Kitten. Si

¹In Kitten una variabile va inizializzata al momento della sua dichiarazione, per cui questo controllo è inutile.

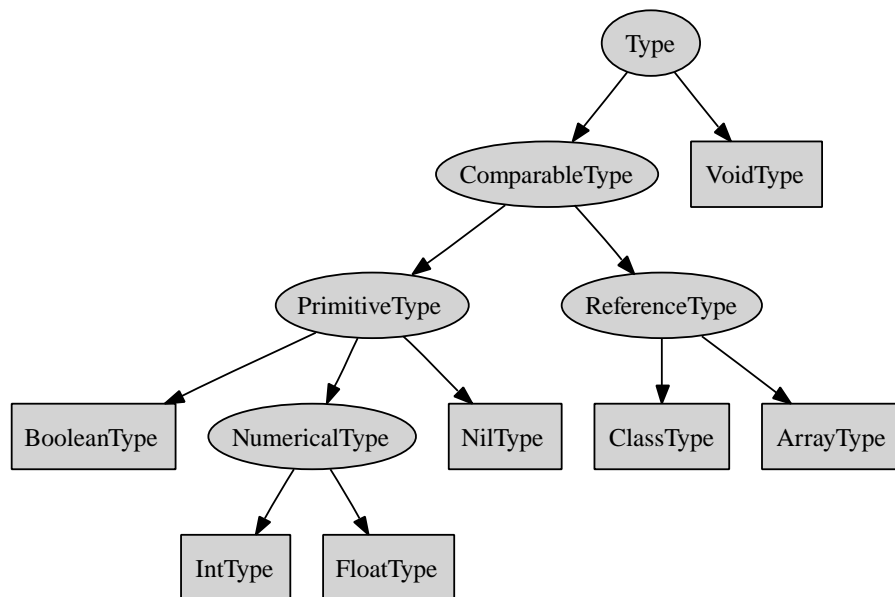


Figura 5.1: Le classi del package `types` che rappresentano i tipi semantici di Kitten.

tratta invece adesso di rappresentare la *struttura semantica* dei tipi 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 supertipo comune fra due o più tipi, se esso esiste (Sezione 1.8), o quali sono i campi o i costruttori o metodi di un tipo classe. Per apprezzare 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 5.1 presenta la gerarchia di tali classi. I tipi sono in primo luogo divisi fra *confrontabili* e *void*. I tipi confrontabili sono quelli per i cui valori è definito almeno l'operatore di confronto `=`. Essi sono a loro volta divisi fra tipi *primitivi* e *riferimento* (Sezione 1.8). I tipi *numerici* sono quei tipi primitivi che rappresentano numeri e per cui sono definite le usuali operazioni di confronto, come il `<`, oltre a `=`. Si noti che non esiste un tipo specifico per le stringhe, che sono invece considerate come un caso di `ClassType`.

La Figura 5.2 mostra l'implementazione della superclasse `types/Type.java`. Essa definisce in primo luogo delle costanti per dei tipi di uso comune. Le sue sottoclassi dovranno istanziare il metodo `canBeAssignedTo()` che determina se un tipo può essere assegnato a un altro, seguendo le regole che nella Sezione 1.8 hanno portato alla definizione della relazione \leq sui tipi. Il metodo `canBeAssignedToSpecial()` è per default un sinonimo di `canBeAssignedTo()`, ma viene ridefinito in `types/PrimitiveType.java` e `types/Void.java` come segue:

```
public boolean canBeAssignedToSpecial(Type other) {
```

```

public abstract class 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();

    protected Type() {}
    public abstract boolean canBeAssignedTo(Type other);
    public boolean canBeAssignedToSpecial(Type other) {
        return canBeAssignedTo(other); // i tipi primitivi lo ridefiniscono
    }
    public Type leastCommonSupertype(Type other) {
        // questo e' ok per i tipi primitivi. Classi e array lo ridefiniscono
        if (this.canBeAssignedTo(other)) return other;
        else if (other.canBeAssignedTo(this)) return this;
        else return null; // non esiste
    }
    public static final ClassType getObjectType() { ... ritorna il tipo per Object }
}

```

Figura 5.2: La superclasse astratta dei tipi semantici di Kitten

```

    return this == other;
}

```

in modo che i tipi primitivi e void siano *sottotipo speciale* solo di se stessi. Questo metodo è utile all'interno della classe `ArrayType`, che vedremo fra un attimo, per implementare la relazione di sottotipaggio \leq che come sappiamo non è monotona sugli array di tipi primitivi (Sezione 1.8). Esso è usato anche per 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 supertipo comune fra due tipi. Tale supertipo potrebbe non esistere: fra `int` e `boolean` non c'è alcun supertipo comune. La definizione fornita dentro `types/Types.java` funziona per tutti i tipi primitivi, ma come vedremo deve essere ridefinita per i tipi riferimento. Si noti che il costruttore di `types/Type.java` è `protected`. Anche i costruttori delle altre classi che implementano i tipi semantici sono `protected` o `private`. Quindi l'unico modo per ottenere degli oggetti della gerarchia in Figura 5.1 sarà tramite le costanti definite in Figura 5.2 o tramite dei costruttori con memoria che definiremo dentro le classi per i tipi riferimento. Questo implica che *esiste al più un oggetto per un dato tipo semantico* e l'uguaglianza fra tipi può essere controllata con semplici confronti Java `==`. Il metodo `getObjectType()` ritorna il tipo della superclasse `Object` di tutte le classi ed array.

Si consideri la classe `types/IntType.java` in Figura 5.3. Come si vede, ammettiamo che il tipo `int` sia assegnato a `int` stesso ma anche a `float`, poiché $\text{int} \leq \text{float}$ (Sezione 1.8).

```
public class IntType extends NumericalType {
    protected IntType() {}
    public boolean canBeAssignedTo(Type other) {
        return other == Type.INT || other == Type.FLOAT;
    }
}

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

Figura 5.3: Le classi `types/IntType.java` e `types/VoidType.java` che implementano rispettivamente i tipi `int` e `void`.

La classe `types/VoidType.java` è simile, ma non ammettiamo l'assegnamento verso nessun tipo, neppure `void`. L'assegnamento speciale è invece possibile ma solo verso `void` stesso, come abbiamo già detto.



La scelta di imporre l'uguaglianza nella relazione di sottotipo speciale per i tipi primitivi ha l'effetto che, nel controllo di compatibilità del tipo di ritorno della ridefinizione di un metodo, un tipo primitivo può essere solo sottotipo di se stesso. Si osservi che se `float m()` potesse essere ridefinito, in una sottoclasse, in `int m()` allora una chiamata virtuale del tipo `float f = o.m()` richiederebbe, o meno, una conversione di tipo da `int` a `float` sulla base della classe, a tempo di esecuzione, dell'oggetto contenuto in `o`, il che complica la generazione del codice. Quindi impediamo al programmatore di fare una simile ridefinizione del tipo di ritorno del metodo `m()`. Questo stesso vincolo è imposto nel linguaggio Java.

La classe `types/ArrayType.java` in Figura 5.4 implementa i tipi array. L'invariante che non esistano istanze diverse dello stesso tipo è mantenuta rendendone `private` il costruttore 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 imporre il vincolo della Sezione 1.8 che richiede che se gli elementi di `this` sono un tipo primitivo allora quelli di `other` devono essere *lo stesso* tipo primitivo.

```

public class ArrayType extends ReferenceType {
    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 boolean canBeAssignedTo(Type other) {
        if (other instanceof ArrayType)
            return elementType.canBeAssignedToSpecial(((ArrayType)other).elementType);
        else return other == getObjectType();
    }
    public Type leastCommonSupertype(Type other) {
        // l'lcs fra un array e una classe e' Object
        if (other instanceof ClassType) return getObjectType();
        else 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 getObjectType();
            else {
                Type lcs = elementType.leastCommonSupertype(((ArrayType)other).elementType);
                if (lcs == null) return getObjectType();
                else return mk(lcs);
            }
        else if (other == Type.NIL) return this; // fra un array e nil e' l'array
        else return null; // non esiste alcun lcs
    }
}

```

Figura 5.4: La classe `types/ArrayType.java` che rappresenta i tipi array.



Questo vincolo, apparentemente strano, è giustificato dal fatto che se `arr` è un array di interi allora il comando `int[] copy := arr` rende `arr` e `copy` *alias*, cioè riferimenti diversi allo stesso oggetto array. Mentre il comando `float[] copy := arr` ci impone di 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 a `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.

Il metodo `leastCommonSupertype()` di `ArrayType` 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:

- 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 1.8);
- se anche `other` è un tipo array allora:
 - se entrambi sono array dello stesso tipo primitivo allora *lcs* è uguale a `this` (o equivalentemente a `other`);
 - se entrambi sono array di tipi primitivi diversi 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`;
 - se entrambi sono array di tipi non primitivi allora *lcs* è il tipo array del minimo supertipo 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 1.8);
- 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 5.5 ne riporta il codice. Il costruttore è lasciato `private` e la costruzione di tipi classe è possibile solo tramite il metodo statico `mk()` che ne garantisce l'unicità. Il costruttore si occupa di creare un analizzatore lessicale per il file sorgente della classe, interfacciarlo con un analizzatore sintattico ed effettuare il parsing sintattico della classe. La sintassi astratta così costruita è memorizzata nel tipo classe. La costruzione prosegue con la superclasse, di cui il nuovo oggetto diventa una sottoclasse. 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 eccezioni (non mostrato in Figura 5.5) 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 un programma soltanto perché una delle sue classi contiene un errore: si va avanti con la compilazione finché si può, segnalando quanti più errori possibile al programmatore.

Un tipo classe ha informazione sulla *segnatura* della classe, cioè sui suoi campi, costruttori e metodi. Questa informazione è estratta dalla sua sintassi astratta al momento della costruzione di un tipo classe tramite il metodo `addMembers()` a discesa ricorsiva (Figura 5.5). Si noti che se i campi, costruttori o metodi fanno riferimento alla stessa classe che stiamo creando non entriamo in loop poiché abbiamo avuto cura di registrare il tipo classe che stiamo creando nella memoria di `mk()` prima di chiamare `addMembers()`.

La segnatura della classe può essere consultata in seguito tramite dei metodi di ricerca (*lookup*). La differenza fra i metodi `constructorLookup()` e `constructorsLookup()` è che il

```

public class ClassType extends ReferenceType {
    private Symbol name;           // il nome di questa classe
    private ClassType superclass;  // la sua superclasse (se esiste)
    private ClassTypeList subclasses; // le sue sottoclassi (se esistono)
    private Parser parser;        // il parser usato per questa classe
    private ClassDefinition abstractSyntax; // la sintassi astratta della classe

    private ClassType(Symbol name) {
        ... salva this nella memoria usata da mk()
        this.name = name; parser = new Parser(new Lexer(name));
        (abstractSyntax = (ClassDefinition)parser.parse().value).addMembers(this);
        if (name != Symbol.OBJECT) {
            superclass = mk(abstractSyntax.getSuperclassName());
            superclass.subclasses = new ClassTypeList(this, superclass.subclasses);
        }
    }

    public static ClassType mk(Symbol name) {
        ... restituisci l'eventuale classe di nome name contenuta nella memoria
        ... altrimenti restituisci new ClassType(name)
    }

    public FieldSignature fieldLookup(Symbol name) { ... }
    public ConstructorSignature constructorLookup(TypeList formals) { ... }
    public HashSet constructorsLookup(TypeList formals) { ... }
    public MethodSignature methodLookup(Symbol name, TypeList formals) { ... }
    public HashSet methodsLookup(Symbol name, TypeList formals) { ... }
    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 getObjectType();
        else if (other instanceof ClassType)
            for (ClassType cursor = this; cursor != null; cursor = cursor.superclass)
                if (other.canBeAssignedTo(cursor)) return cursor;
        else if (other == Type.NIL) return this; else return null;
    }
}

```

Figura 5.5: La classe `types/ClassType.java` che implementa il tipo classe di Kitten.

primo cerca il costruttore con parametri formali esattamente identici a quelli indicati, mentre il secondo fornisce l'insieme S di *tutti* i costruttori con parametri formali aventi tipi compatibili con quelli indicati. È garantito il vincolo che nessun costruttore in S ha 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 dei due costruttori della classe non commentati. Entrambi sono infatti compatibili con due parametri di tipo `int`. Se si eliminasse il commento intorno al terzo costruttore, la stessa chiamata a `constructorsLookup()` restituirebbe un insieme formato dal solo terzo costruttore,

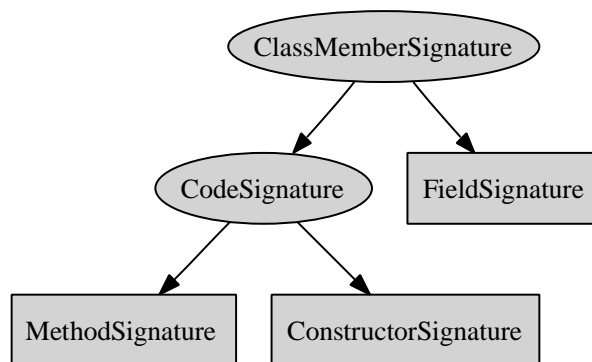


Figura 5.6: Le classi del package `types` che rappresentano le signature dei membri di una classe.

che è più specifico degli altri 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, 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 (lo vedremo quando commenteremo la Figura 5.11). Lo stesso discorso si può fare per `methodLookup()` e `methodsLookup()`, con l'unica differenza che, dal momento che Kitten implementa l'ereditarietà per campi e metodi, la loro ricerca inizia in una data segnatura e, se tale segnatura non definisce esplicitamente il campo o il metodo, la ricerca prosegue ricorsivamente verso l'alto risalendo la catena delle estensioni, verso `Object`.

Il metodo `canBeAssignedTo()` permette l'assegnamento verso la stessa classe o una sua superclasse. Il test di sottoclasse è realizzato dal metodo `subclass()` che scorre verso l'alto a partire da `this` la catena di estensione delle classi alla ricerca dell'ipotetica superclasse. Il metodo `leastCommonSupertype()` determina il minimo supertipo 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 1.8);
- se anche `other` è un tipo classe allora *lcs* è la più specifica superclasse di `this` che è anche superclasse 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 1.8);
- altrimenti *lcs* non esiste.

La Figura 5.5 mostra che il tipo di ritorno dei metodi di ricerca in una classe è un oggetto (o un insieme di oggetti) di tipo `FieldSignature`, `ConstructorSignature` o `MethodSignature`.

```

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 5.7: Le classi che implementano le signature dei membri di una classe Kitten.

Tali classi implementano le *signature* di campi, costruttori e metodi, rispettivamente, cioè una specifica delle loro proprietà di tipo. Per esempio, la signature di un campo di una classe specifica il nome del campo, il suo tipo semantico di dichiarazione e il tipo semantico della classe in cui il campo è definito.

La Figura 5.6 mostra la gerarchia delle classi del package `types` che rappresentano le signature di campi, costruttori e metodi Kitten. La Figura 5.7 ne mostra l'implementazione. La superclasse comune `types/ClassMemberSignature.java` (non mostrata in Figura 5.7) descrive la signature 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 in più il 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

```

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 5.8: La funzione di analisi semantica $\tau[[-]]$ per le espressioni di tipo Kitten.

campo, mentre `MethodSignature` specifica anche il nome e il tipo di ritorno del metodo.

5.2 L'analisi semantica delle espressioni di tipo Kitten

Effettuare l'analisi semantica dei tipi Kitten significa costruire il tipo semantico $\tau[[t]]$ rappresentato da ogni espressione sintattica di tipo t che occorre nel programma e annotare tale tipo dentro t stessa. La costruzione di $\tau[[t]]$ è formalizzata nella Figura 5.8. Le espressioni di tipo che rappresentano i tipi primitivi 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. Le espressioni di tipo che rappresentano un tipo classe vengono mappate nell'oggetto di tipo `types.ClassType` per il nome della classe.

La funzione $\tau[[-]]$ per le espressioni di tipo è implementata tramite un metodo d'istanza di nome `typeCheck()` aggiunto alla classe di sintassi astratta `absyn/TypeExpression.java`:

```

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

```

Il metodo `typeCheck()`, pubblico e `final`, annota nel 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 5.8, dentro `absyn/IntTypeExpression.java` ridefiniamo:

```

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

```

dentro `absyn/ArrayTypeExpression.java`:

```

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

```

$\tau^\kappa[_] : \text{absyn.ClassMemberDeclaration} \mapsto \text{ClassMemberSignature}$

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

dove $\tau[\text{formals}]$ è l'estensione ai parametri formali della funzione $\tau[_]$ della Figura 5.8:

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

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

Figura 5.9: La funzione $\tau^\kappa[_]$ che associa alla sintassi astratta dei membri di una classe κ la loro segnatura.

e dentro `absyn/ClassTypeExpression.java`:

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

Possiamo adesso mostrare in Figura 5.9 una funzione $\tau^\kappa[_]$ che costruisce le segnature dei membri di una classe κ a partire dalla loro sintassi astratta. Questa funzione è implementata dal metodo `addMembers()` usato al momento della costruzione di un tipo classe per arricchirlo con le segnature dei suoi membri (Figura 5.5).

5.3 L'analisi semantica delle espressioni Kitten

L'analisi semantica delle espressioni Kitten consiste nell'annotare a tempo di compilazione ciascuna espressione e che occorre nel programma sorgente con il suo tipo statico t_e (Sezione 1.8). Essendo Kitten un linguaggio fortemente tipato, occorre definire t_e in modo che, a tempo di esecuzione, il tipo dinamico di e (cioè il tipo del valore di e , Sezione 1.8) sia t_e o un sottotipo di t_e . L'analisi semantica deve inoltre garantire che i tipi siano usati correttamente dentro e . Deve rifiutare per esempio espressioni 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

$$\begin{array}{c}
\frac{\rho(\text{name}) \text{ è definito}}{\rho \vdash \text{Variable}(\text{name}) : \rho(\text{name})} \quad \frac{\rho \vdash \text{receiver} : \kappa \quad \kappa \in \text{ClassType} \quad \text{field} = \kappa.\text{fieldLookup}(\text{name}) \quad \text{field} \neq \text{null}}{\rho \vdash \text{FieldAccess}(\text{receiver}, \text{name}) : \text{field.getType}()} \\
\frac{\rho \vdash \text{array} : t \quad t \in \text{ArrayType} \quad \rho \vdash \text{index} : \text{Type.INT}}{\rho \vdash \text{ArrayAccess}(\text{array}, \text{index}) : t.\text{getElementsType}()} \\
\frac{}{\rho \vdash \text{False}() : \text{Type.BOOLEAN}} \quad \frac{}{\rho \vdash \text{True}() : \text{Type.BOOLEAN}} \quad \frac{}{\rho \vdash \text{Nil}() : \text{Type.NIL}} \\
\frac{}{\rho \vdash \text{IntLiteral}() : \text{Type.INT}} \quad \frac{}{\rho \vdash \text{FloatLiteral}() : \text{Type.FLOAT}} \\
\frac{}{\rho \vdash \text{StringLiteral}(\text{value}) : \text{ClassType.mk}(\text{Symbol.STRING})}
\end{array}$$

Figura 5.10: Le regole per l'analisi semantica dei leftvalue e dei letterali Kitten.

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 in vista della generazione del codice intermedio (Capitolo 6), 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.

Effettueremo il controllo semantico di un'espressione e tramite un giudizio $\vdash e : t_e$ definito a discesa ricorsiva sulla sintassi astratta delle espressioni. Gli esempi precedenti mostrano però che a tal fine avremo bisogno 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 il nostro giudizio in $\rho \vdash e : t_e$, 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 definizione di questo giudizio è in Figura 5.10 per quanto riguarda i leftvalue e i letterali Kitten e in Figura 5.11 per le restanti espressioni. Si noti subito che il giudizio $\rho \vdash e : t_e$ non è sempre definito. Si deve immaginare che, quando esso non è definito, un messaggio di errore viene comunicato al programmatore.

Consideriamo adesso le regole più significative delle Figure 5.10 e 5.11.

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 nel-

$$\begin{array}{c}
\frac{\text{ClassType.mk}(\text{className}) = \kappa \quad \rho \vdash \text{actuals} : \vec{\tau} \quad \kappa.\text{constructorsLookup}(\vec{\tau}) = \{\text{constructor}\}}{\rho \vdash \text{NewObject}(\text{className}, \text{actuals}) : \kappa} \\
\\
\frac{\rho \vdash \text{elementType} : t \quad \rho \vdash \text{size} : \text{Type.INT}}{\rho \vdash \text{NewArray}(\text{elementType}, \text{size}) : \text{ArrayType.mk}(t)} \\
\\
\frac{\rho \vdash \text{receiver} : \kappa \quad \kappa \in \text{ClassType} \quad \rho \vdash \text{actuals} : \vec{\tau} \quad \kappa.\text{methodsLookup}(\text{name}, \vec{\tau}) = \{\text{method}\} \quad r = \text{method.getReturnType()} \quad r \neq \text{Type.VOID}}{\rho \vdash \text{MethodCallExpression}(\text{receiver}, \text{name}, \text{actuals}) : r} \\
\\
\frac{\rho \vdash \text{expression} : \text{Type.BOOLEAN}}{\rho \vdash \text{Not}(\text{expression}) : \text{Type.BOOLEAN}} \quad \frac{\rho \vdash \text{expression} : t \quad t \leq \text{Type.FLOAT}}{\rho \vdash \text{Minus}(\text{expression}) : t} \\
\\
\frac{\text{intoType} = \tau[\text{type}] \quad \rho \vdash \text{expression} : \text{fromType} \quad \text{intoType} < \text{fromType}}{\rho \vdash \text{Cast}(\text{type}, \text{expression}) : \text{intoType}} \\
\\
\frac{\rho \vdash \text{left} : \text{Type.BOOLEAN} \quad \rho \vdash \text{right} : \text{Type.BOOLEAN}}{\rho \vdash \text{BooleanBinOp}(\text{left}, \text{right}) : \text{Type.BOOLEAN}} \\
\\
\frac{\rho \vdash \text{left} : t_l \quad \rho \vdash \text{right} : t_r \quad t_l \leq \text{Type.FLOAT} \quad t_r \leq \text{Type.FLOAT}}{\rho \vdash \text{ArithmeticBinOp}(\text{left}, \text{right}) : t_l.\text{leastCommonSupertype}(t_r)} \\
\\
\frac{\rho \vdash \text{left} : t_l \quad \rho \vdash \text{right} : t_r \quad t_l \leq \text{Type.FLOAT} \quad t_r \leq \text{Type.FLOAT}}{\rho \vdash \text{NumericalComparisonBinOp}(\text{left}, \text{right}) : \text{Type.BOOLEAN}} \\
\\
\frac{\rho \vdash \text{left} : t_l \quad \rho \vdash \text{right} : t_r \quad (t_l \leq t_r \text{ oppure } t_r \leq t_l)}{\rho \vdash \text{Equal}(\text{left}, \text{right}) : \text{Type.BOOLEAN}} \\
\\
\frac{\rho \vdash \text{left} : t_l \quad \rho \vdash \text{right} : t_r \quad (t_l \leq t_r \text{ oppure } t_r \leq t_l)}{\rho \vdash \text{NotEqual}(\text{left}, \text{right}) : \text{Type.BOOLEAN}}
\end{array}$$

Figura 5.11: Le regole per l'analisi semantica delle restanti espressioni Kitten.

l'espressione *receiver* richiede in primo luogo di determinare il tipo statico κ di *receiver*. La preconditione richiede che κ sia un tipo 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 5.5). 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 preconditione 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.getElementsType()`.

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 verificando il giudizio $\rho \vdash \text{actuals} : \vec{\tau}$, che è l'estensione del giudizio $\rho \vdash e : t_e$ a sequenze di espressioni:

$$\rho \vdash \text{null} : \text{null}$$

$$\frac{\rho \vdash \text{head} : h \quad \rho \vdash \text{tail} : \vec{\tau}}{\rho \vdash \text{ExpressionSeq}(\text{head}, \text{tail}) : \text{new TypeList}(h, \vec{\tau})}$$

Occorre anche garantire che fra i costruttori di κ che possono essere chiamati con parametri attuali di tipo $\vec{\tau}$ ce ne sia uno più specifico degli altri. Questo si verifica chiamando il metodo `constructorsLookup($\vec{\tau}$)` sulla classe κ (Figura 5.5) 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 verificare i giudizi $\rho \vdash \text{receiver} : \kappa$ e $\rho \vdash \text{actuals} : \vec{\tau}$ (quest'ultimo è l'estensione di $\rho \vdash e : t_e$ a 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 $\vec{\tau}$ ne deve esistere uno che è più specifico di tutti gli altri. Questo si ottiene chiamando il metodo `methodsLookup($\vec{\tau}$)` sulla classe κ (Figura 5.5) 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()`. Si richiede che tale tipo non sia `void` poiché un'espressione deve avere un valore associato a tempo di esecuzione.

Cast(*type*, *expression*). Quest'espressione effettua il cast di *expression* verso il tipo *type*. La sua analisi semantica effettua ricorsivamente l'analisi semantica di *type* ed *expression* e poi richiede che il tipo semantico di *type* sia un sottotipo stretto del tipo semantico di *expression*. Questo vincolo accetta quindi esclusivamente cast verso il basso scartando per esempio espressioni come `3 as Persona`, `3 as float`, `3 as int` o `studente as Persona`. Il motivo per cui tali cast sono rifiutati è che sarebbero impossibili (come nell'esempio `3 as Persona`) oppure sempre veri: è sempre possibile usare un intero dove serve un valore in virgola mobile o un intero; è sempre possibile usare uno studente dove serve una persona. Rifiutando questi ultimi cast si obbliga il programmatore a scrivere del codice più semplificato (`3` al posto di `3 as float` e di `3 as int`, `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 verifica i giudizi $\rho \vdash \text{left} : t_l$ e $\rho \vdash \text{right} : t_r$. Tali due espressioni devono avere un tipo statico che sia `int` o `float`. Il tipo

statico del risultato dell'operazione è il minimo supertipo comune fra t_l e t_r . Questo significa per esempio che $3 + 4$ ha tipo statico `int` e $3 + 4.5$ 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 sottoclassi (Figura 3.26).

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 effettuarne ricorsivamente l'analisi semantica e impone che il tipo di una delle due espressioni sia un sottotipo (non stretto) di quello dell'altra. Questo vincolo serve a rifiutare espressioni di uguaglianza che non potrebbero mai essere vere ed espressioni di disuguaglianza che sarebbero sempre false. Per esempio, se `p` è una variabile di classe `Persona`, sottoclasse diretta di `Object`, e `c` è una variabile di classe `Automobile`, anch'essa sottoclasse diretta di `Object`, allora l'uguaglianza `p = c` è sempre falsa, poiché non esisterà mai un oggetto che sia al contempo una `Persona` e un'`Automobile`. Per lo stesso motivo, la disuguaglianza `p != c` è sempre vera. Rifiutando queste espressioni costringiamo il programmatore a eliminare dal suo programma dei test inutili.

5.3.1 L'implementazione dell'analisi semantica delle espressioni

L'implementazione delle regole nelle Figure 5.10 e 5.11 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 5.4) avremo bisogno di un'operazione di estensione *non distruttiva* sugli ambienti, tale cioè da lasciare il vecchio ambiente intatto dopo la sua estensione. 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 5.12. L'interfaccia `symbol.Table` specifica semplicemente che un ambiente ha un'operazione `get(key)` che permette di leggere il valore di una variabile `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: il vecchio ambiente non è modificato ed è ancora utilizzabile. Questo al fine di implementare un'operazione `put()` non distruttiva, come volevamo.

Le 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,

```
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 {
    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 5.12: Le classi del package `symbol` che implementano gli ambienti.

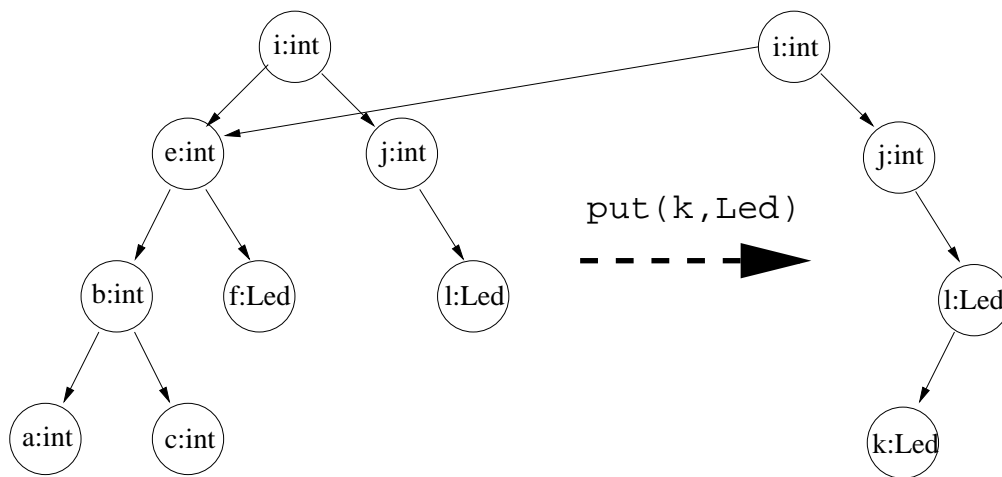


Figura 5.13: L'inserzione non distruttiva in un ambiente di un legame per una variabile.

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 metodo `get()` (Figura 5.12). Il metodo `put()` invece 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 5.13 mostra come è effettuata l'inserzione. 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.

Gli ambienti sono contenuti dentro un *type-checker*, il quale è implementato dalla classe `semantical/TypeChecker.java` in Figura 5.14. Per adesso l'ambiente è tutto quello di cui abbiamo bisogno per effettuare l'analisi semantica delle espressioni, ma per i comandi aggiungeremo al *type-checker* ulteriori informazioni (Sezione 5.4).

Possiamo a questo punto implementare le regole delle Figure 5.10 e 5.11 tramite una discesa ricorsiva sulla sintassi astratta delle espressioni. In `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);
```

Il metodo `public` e `final`, di nome `typeCheck()`, effettua le operazioni comuni a tutte le espressioni, cioè l'annotazione del tipo statico inferito per l'espressione e del *type-checker* usato

```

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

    private TypeChecker(Table env, ErrorMsg errorMsg) {
        this.env = env; this.errorMsg = errorMsg;
    }
    public TypeChecker(ErrorMsg errorMsg) {
        this(Table.EMPTY,errorMsg);
    }
    public TypeChecker putVar(Symbol var, Type type) {
        return new TypeChecker(env.put(var,type),errorMsg);
    }
    public Type getVar(Symbol var) { return (Type)env.get(var); }
    public void error(int pos, String msg) { errorMsg.error(pos,msg); }
}

```

Figura 5.14: Il type-checker usato per effettuare l'analisi semantica delle espressioni Kitten.

per inferirlo. Un metodo ausiliario e `protected`, di nome `typeCheck$0()`, implementa le operazioni specifiche alla singola espressione, come specificate nelle Figure 5.10 e 5.11.

Vediamo alcuni esempi di implementazione del metodo `typeCheck$0()`. Dentro la classe `absyn/Variable.java` definiamo:

```

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

```

Questa implementazione riflette la specifica in Figura 5.10: si cerca la variabile nell'ambiente; se non esiste si dà un errore altrimenti se ne restituisce il tipo. Il metodo `error()` è definito dentro `absyn/Expression.java` come

```

protected Type error(String msg) {
    error(checker,msg);
    return Type.INT;
}

```

Esso stampa il messaggio di errore tramite il type-checker in utilizzo per l'espressione e ritorna il tipo di emergenza `int`. Questo permette di continuare il type-checking anche in presenza di un errore, benché possa causare degli errori di tipo a cascata. Il metodo `error()` a due argomenti è poi definito dentro `absyn/Absyn.java` come

```

protected void error(TypeChecker checker, String msg) {

```

```

    checker.error(pos,msg);
}

```

Esso usa il campo `pos` della sintassi astratta 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 3.6).

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

```

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

```

Consistentemente con la Figura 5.10, 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.

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

```

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

```

Consistentemente con la Figura 5.10, tale metodo effettua ricorsivamente l'analisi semantica di `array` e `index`. Per `index` usa il metodo ausiliario `mustBeInt()` che è definito dentro `absyn/Expression.java` come:

```

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

```

Consideriamo infine la definizione di `typeCheck$0()` in `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))

```

$$\begin{array}{c}
\frac{}{\rho \vdash \text{Skip}() : \rho} \quad \frac{\rho \vdash \text{condition} : \text{Type.BOOLEAN} \quad \rho \vdash \text{then} : \rho' \quad \rho \vdash \text{else} : \rho''}{\rho \vdash \text{IfThenElse}(\text{condition}, \text{then}, \text{else}) : \rho} \\
\frac{\text{expression} \neq \text{null} \quad \rho \vdash \text{expression} : t \quad \text{il comando occorre in un metodo che ritorna } r \quad t \leq r}{\rho \vdash \text{Return}(\text{expression}) : \rho} \\
\frac{\text{il comando occorre in un costruttore o in un metodo che ritorna void}}{\rho \vdash \text{Return}(\text{null}) : \rho} \\
\frac{\rho \vdash \text{lvalue} : t_l \quad \rho \vdash \text{rvalue} : t_r \quad t_l \leq t_r}{\rho \vdash \text{Assignment}(\text{lvalue}, \text{rvalue}) : \rho} \\
\frac{\rho \vdash \text{initialisation} : \rho' \quad \rho' \vdash \text{condition} : \text{Type.BOOLEAN} \quad \rho' \vdash \text{update} : \rho'' \quad \rho' \vdash \text{body} : \rho'''}{\rho \vdash \text{For}(\text{initialisation}, \text{condition}, \text{update}, \text{body}) : \rho} \\
\frac{\rho \vdash \text{condition} : \text{Type.BOOLEAN} \quad \rho \vdash \text{body} : \rho'}{\rho \vdash \text{While}(\text{condition}, \text{body}) : \rho} \\
\frac{t = \tau[\![\text{type}]\!] \quad \rho \vdash \text{initialiser} : i \quad i \leq t}{\rho \vdash \text{LocalDeclaration}(\text{type}, \text{name}, \text{initialiser}) : \rho[\text{name} \mapsto t]} \\
\frac{\rho \vdash \text{body} : \rho'}{\rho \vdash \text{LocalScope}(\text{body}) : \rho} \quad \frac{\rho \vdash \text{receiver} : \kappa \quad \kappa \in \text{ClassType} \quad \rho \vdash \text{actuals} : \vec{\tau} \quad \kappa.\text{methodsLookup}(\text{name}, \vec{\tau}) = \{\text{method}\}}{\rho \vdash \text{MethodCallCommand}(\text{receiver}, \text{name}, \text{actuals}) : \rho} \\
\frac{\rho \vdash c_1 : \rho' \quad \rho' \vdash c_2 : \rho''}{\rho \vdash c_1; c_2 : \rho''}
\end{array}$$

Figura 5.15: Le regole per l'analisi semantica dei comandi Kitten.

```

    return leftType.leastCommonSupertype(rightType);
    else return error("numerical argument required");
}

```

Consistentemente con la Figura 5.11, esso 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 supertipo comune fra i tipi statici di `left` e `right`.

5.4 L'analisi semantica dei comandi Kitten

La Figura 5.15 mostra le regole di analisi semantica per i comandi Kitten. Questa volta usiamo un giudizio $\rho \vdash c : \rho'$ il cui significato è che il comando c eseguito a partire da un ambiente ρ porta in un ambiente ρ' . Questo perché i comandi non hanno un valore ma possono modificare l'ambiente e le uniche modifiche visibili al livello dei tipi sono quelle dell'insieme e del tipo delle variabili in scope. In particolare, è la dichiarazione di una variabile (la `LocalDeclaration` in Figura 5.15) che estende l'ambiente con una nuova variabile locale, che sostituisce eventualmente una variabile già in scope e con lo stesso nome.

Esaminiamo adesso alcune regole della Figura 5.15:

IfThenElse(*condition*, *then*, *else*). Il condizionale richiede che la condizione sia un'espressione di tipo booleano ed effettua ricorsivamente l'analisi semantica di *then* ed *else*. La scelta di lasciare ρ immutato come risultato dell'analisi semantica del condizionale implica che eventuali variabili locali dichiarate all'interno del ramo *then* o del ramo *else* 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 supertipo. 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' , eventualmente arricchito, rispetto a ρ , con una dichiarazione di una variabile locale al ciclo. Si impone poi che *condition* abbia tipo booleano. L'ambiente ρ' viene usato per effettuare l'analisi semantica di *initialisation*, *update* e *body*, al fine di permettere al programmatore di dichiarare una variabile locale dentro *initialisation* e di usarla 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*. Si effettua anche ricorsivamente l'analisi semantica di *initialiser* e si impone che il suo tipo statico 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. Per esempio, nel comando `{ 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 5.11. L'unica differenza è che qui non imponiamo alcun vincolo sul tipo di ritorno del metodo, che può quindi anche essere `void`.

$c_1; c_2$. L'analisi semantica della sequenza di comandi si richiama ricorsivamente sui due comandi, usando l'ambiente risultante dall'analisi semantica del primo per effettuare l'analisi semantica del secondo. In questo modo eventuali variabili locali dichiarate in c_1 possono essere usate da c_2 .

5.4.1 L'implementazione dell'analisi semantica dei comandi

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) {
    checker = typeCheck$(this.checker = 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$()`, annotare il type-checker risultante dall'analisi e richiamarsi ricorsivamente sul comando seguente, se esiste. In questo modo si implementa l'analisi semantica della sequenza di comandi.

Il metodo `typeCheck$()` effettua la parte di analisi semantica specifica a ciascun comando, secondo le regole della Figura 5.15. Per esempio, dentro `absyn/IfThenElse.java` lo definiamo come

```
protected TypeChecker typeCheck$(TypeChecker checker) {
    condition.mustBeBoolean(checker);
    then.typeCheck(checker);
    else.typeCheck(checker);
    return checker;
}
```

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

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

```

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

    private TypeChecker(Type returnType, Table env, int varNum, ErrorMsg errorMsg) {
        this.returnType = returnType; this.env = env;
        this.varNum = varNum; this.errorMsg = errorMsg;
    }
    public TypeChecker(Type returnType, ErrorMsg errorMsg) {
        this(returnType, Table.EMPTY, 0, errorMsg);
    }
    public TypeChecker setReturnType(Type returnType) {
        return new TypeChecker(returnType, env, varNum, 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, 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 5.16: La classe `semantical/TypeChecker.java` che implementa un type-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 5.15.

La Figura 5.16 mostra una revisione del type-checker della Figura 5.14. Adesso esso conosce il tipo di ritorno del metodo che si sta analizzando, fornito al momento della costruzione del type-checker tramite l'unico costruttore pubblico in Figura 5.16 e usato poi per implementare l'analisi semantica dei comandi `return`: in `absyn/Return.java` definiamo infatti:

```

protected TypeChecker typeCheck$0(TypeChecker checker) {
    Type expectedReturnType = checker.getReturnType();
    if (returned == null && expectedReturnType != Type.VOID)
        error("missing return value");
    if (returned != null &&

```

```

        !returned.typeCheck(checker).canBeAssignedTo(expectedReturnType))
        error("illegal return type: " + expectedReturnType + " expected");
    return checker;
}

```

conformemente alla Figura 5.15.

Si noti che il type-checker in Figura 5.16 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 6).

5.5 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 5.15, 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:

$$\rho + \text{null} = \rho$$

$$\rho + \text{FormalParameters}(type, name, next) = (\rho + next)[name \mapsto \tau[[type]]]$$

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 di verificare che il giudizio $\bar{\rho} : body : \rho'$ sia valido per un qualche ρ' .

Anche il metodo che fa l'analisi semantica dei membri di una classe si chiama `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 la classe di sintassi astratta `absyn/FieldDeclaration.java` definiamo quindi:

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

In `absyn/ConstructorDeclaration.java` definiamo invece:

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

Questo metodo comincia col costruire un type-checker con ambiente vuoto e che si aspetta come tipo di ritorno `Type.VOID`. Quindi aggiunge la variabile `this` legata al tipo semantico della classe e i parametri formali legati al loro tipo di dichiarazione, rispecchiando la definizione precedente di $\bar{\rho}$. Infine effettua il type-checking del corpo del costruttore e controlla che al suo interno non ci sia del codice morto (Sezione 4.3). 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` e si controlla che se il metodo ne ridefinisce un altro di una superclasse allora la ridefinizione del tipo di ritorno soddisfi il test `canBeAssignedToSpecial()` visto in Sezione 5.1. Se inoltre il metodo non ritorna `void`, si impone che il valore di ritorno del metodo `checkForDeadcode()` sia `true`, in modo da garantire che il metodo termini sempre con un comando `return`.



L'analisi semantica di Kitten descritta in questo capitolo è un po' semplificata rispetto alla realtà. In particolare non abbiamo considerato come dall'analisi della classe di partenza (Sezione 5.5) si arrivi a quella delle altre classi a cui essa fa riferimento. Questo è ottenuto facendo in modo che le regole delle Figure 5.10, 5.11 e 5.15, quando hanno bisogno di ottenere il tipo semantico delle espressioni di tipo, richiama ricorsivamente il type-checking su tutte le classi che vi occorrono. Al fine di evitare cicli, si usa un flag `typeChecked` all'interno di `types.ClassType`.

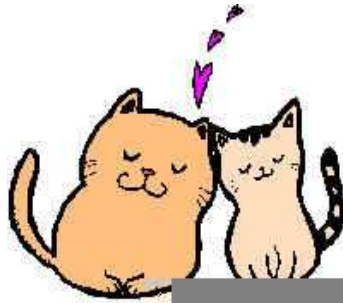
Esercizio 22. Si aggiunga alle espressioni la sintassi astratta di un'espressione condizionale $exp_1 ? exp_2 : exp_3$ che restituisce il valore di exp_2 se exp_1 è vera e il valore di exp_3 altrimenti. Si dia la sua regola di type-checking.

Esercizio 23. Si aggiunga ai comandi la sintassi astratta di un comando `switch`. Non ci si limiti a espressioni costanti nei vari casi. Si dia la regola di type-checking per tale comando.

Esercizio 24. Si aggiunga ai comandi la sintassi astratta dei comandi `break` e `continue`. Si diano le loro regole di type-checking, che devono garantire che tali comandi occorran solo dentro un ciclo. Come modifichereste il type-checker in Figura 5.16 in modo da implementare tali controlli?

Capitolo 6

Generazione del Bytecode Kitten



L'analisi semantica del Capitolo 5 ha garantito che il codice Kitten non contiene alcun errore semantico. Ha inoltre annotato l'albero di sintassi astratta con informazioni relative al tipo statico delle espressioni che vi occorrono; gli accessi a campi, costruttori e metodi con la dichiarazione di campo, costruttore o metodo a cui fanno riferimento. Siamo ora nelle condizioni di generare del codice *intermedio*, cioè indipendente dall'architettura verso la quale stiamo compilando, ma pensato piuttosto per essere facilmente sintetizzabile a partire dall'albero di sintassi astratta e facilmente ottimizzabile. Esso verrà poi traslato in codice oggetto, specifico all'architettura verso cui compiliamo. Il codice intermedio che useremo è il *bytecode Kitten*, che può essere visto come una versione semplificata ed esplicitamente tipata del *Java bytecode*.

6.1 Il bytecode Kitten

Il bytecode Kitten è un linguaggio di programmazione pensato per essere eseguito da una macchina astratta che ha a disposizione:

1. un insieme di *variabili locali*, potenzialmente illimitato, che possono contenere valori primitivi o riferimenti ad oggetti o array;
2. uno stack di variabili temporanee, detto *stack degli operandi*, potenzialmente illimitato, che può contenere valori primitivi o riferimenti a oggetti o array;

```

Led():
    return void

on():
    load 0 of type Led
    const true
    putfield Led.state
    return void

off():
    load 0 of type Led
    const false
    putfield Led.state
    return void

isOn():
    load 0 of type Led
    getfield Led.state
    return boolean

isOff():
    load 0 of type Led
    getfield Led.state
    neg boolean
    return boolean

```

Figura 6.1: La compilazione in bytecode Kitten dei metodi della classe Led in Figura 1.4.

3. uno *stack di attivazione*, formato da un numero potenzialmente illimitato di *frame di attivazione* di metodi. Ciascun frame di attivazione contiene le variabili locali e lo stack di attivazione di un metodo;
4. una *memoria o heap*, che contiene oggetti e array allocati dinamicamente dal programma in esecuzione.

La maggior parte delle istruzioni del bytecode Kitten operano sulle variabili locali e sullo stack degli operandi. Un numero limitato (invocazione e ritorno da metodo) operano anche sullo stack di attivazione. Le sole operazioni che operano sulla memoria sono quelle di creazione di oggetto o array e di accesso a campi o array.

Si consideri la Figura 6.1. Essa mostra la compilazione in bytecode Kitten dei metodi della classe Led in Figura 1.4. All'inizio dell'esecuzione di un metodo o costruttore, lo stack degli operandi è vuoto e le variabili locali contengono i parametri attuali del metodo o costruttore. In particolare, la variabile locale numero 0 contiene sempre il riferimento all'oggetto corrente, cioè quello che nel codice sorgente sarebbe stato `this`, che è un parametro implicito in tutte le chiamate di metodo o costruttore. La variabile locale 1 contiene il primo parametro attuale esplicito, la variabile locale 2 il secondo parametro attuale esplicito, e così via. Si noti comunque che le variabili locali possono essere usate anche per contenere vere e proprie variabili locali ai metodi e non solo per contenere i parametri attuali. Nell'esempio in Figura 6.1, solo la variabile locale 0 è utilizzata, dal momento che nessun metodo richiede dei parametri espliciti né variabili locali. L'istruzione `load 0 of type Led` indica di copiare il riferimento all'oggetto corrente in cima allo stack degli operandi. L'istruzione `const` serve invece a caricare in cima allo stack degli operandi una costante. Nella Figura 6.1 si tratta di una costante booleana. Le istruzioni `getfield` e `putfield` servono, rispettivamente, a leggere e a scrivere un campo di un oggetto.

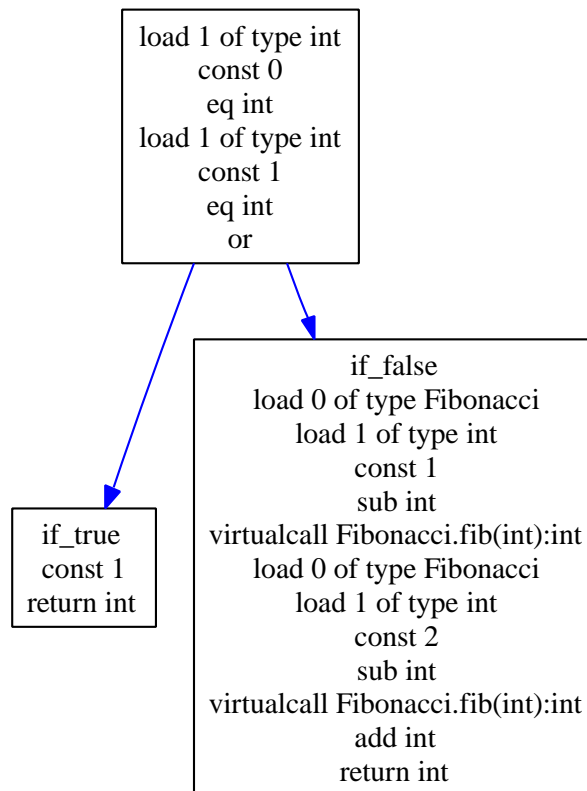


Figura 6.2: La compilazione in bytecode Kitten del metodo `fib()` in Figura 1.2.

L'istruzione `neg` nega il valore che sta in cima allo stack degli operandi. L'istruzione `return` termina l'esecuzione di un metodo o costruttore ritornando possibilmente un valore al chiamante.

Il bytecode in Figura 6.1 ha una struttura di controllo particolarmente semplice, dal momento che non prevede condizionali né cicli. La Figura 6.2 mostra un esempio più complesso: la compilazione in bytecode Kitten del metodo `fib()` in Figura 1.2. La presenza di un comando condizionale in Figura 1.2 diventa un'alternativa di controllo nel bytecode Kitten in Figura 6.2: il risultato dell'istruzione `or` determina l'instradamento del controllo verso il ramo `if_true` o verso quello `if_false`.

L'esempio precedente mostra che il bytecode Kitten è in effetti un grafo di *blocchi di codice* che contengono codice sequenziale. Un ulteriore esempio è la compilazione del ciclo:

```
for (int i := 0; i < 5; i := i + 1) {}
```

mostrata in Figura 6.3. Questa volta l'instradamento del controllo dipende dal risultato di un confronto. In particolare, il confronto fra la variabile locale 1, che contiene la variabile `i` del ciclo, e la costante intera 5 determina l'instradamento del codice verso il ramo `if_cmp1t` (*IF the CoMPArison is Less Than*) o verso quello `if_compge` (*IF the CoMPArison is Greater than or Equal*).

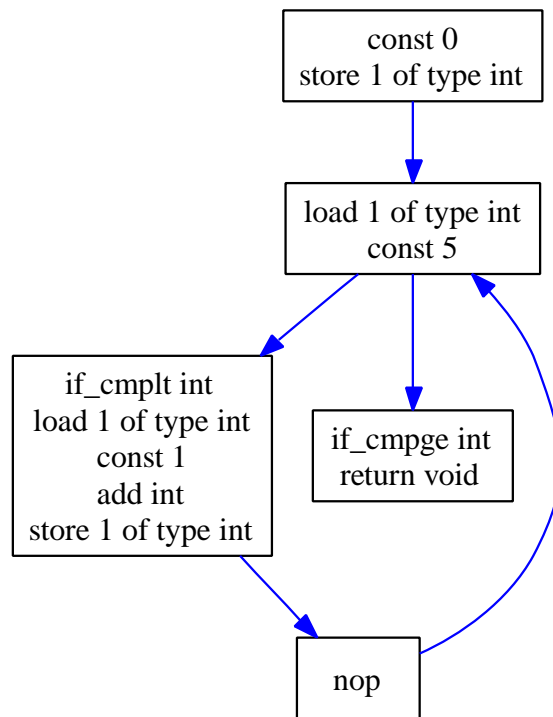


Figura 6.3: La compilazione in bytecode Kitten di un ciclo for.

6.1.1 Le istruzioni sequenziali

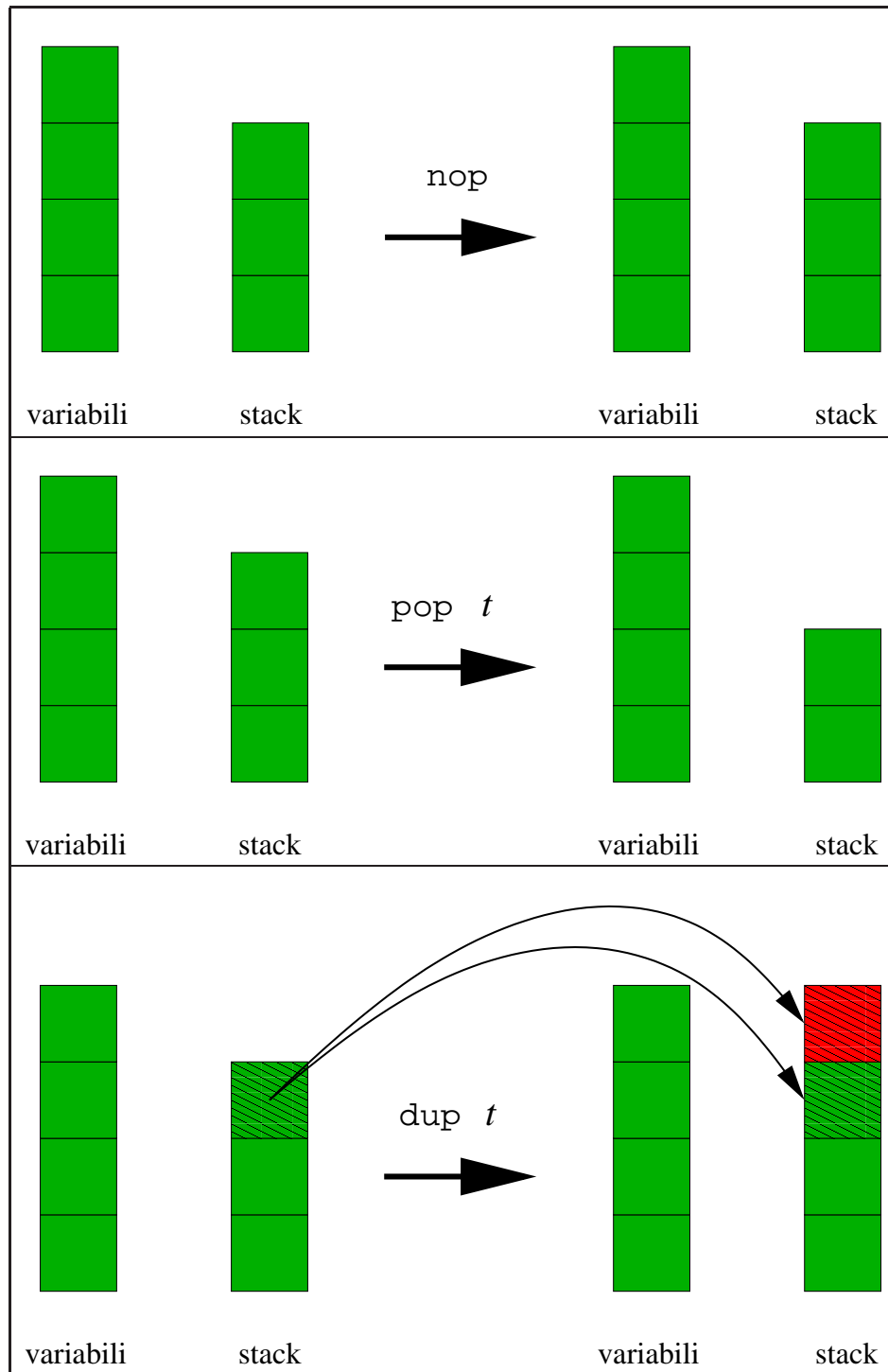
Esaminiamo adesso il set di istruzioni messe a disposizione dal bytecode Kitten. Per ognuna di esse mostriamo il suo effetto sulle variabili locali, sullo stack degli operandi e sulla memoria o heap. Dal momento che poche istruzioni operano sullo heap, lo indicheremo solo per quelle poche istruzioni per cui esso è effettivamente significativo. Le istruzioni del bytecode Kitten sono *tipate*, nel senso che è specificato il tipo degli operandi su cui possono operare. Esse non effettuano mai una promozione di tipo, per cui quando nella loro descrizione useremo il termine *sottotipo*, esso va inteso nel senso dell'operazione `canBeAssignedToSpecial()` della Sezione 5.1.

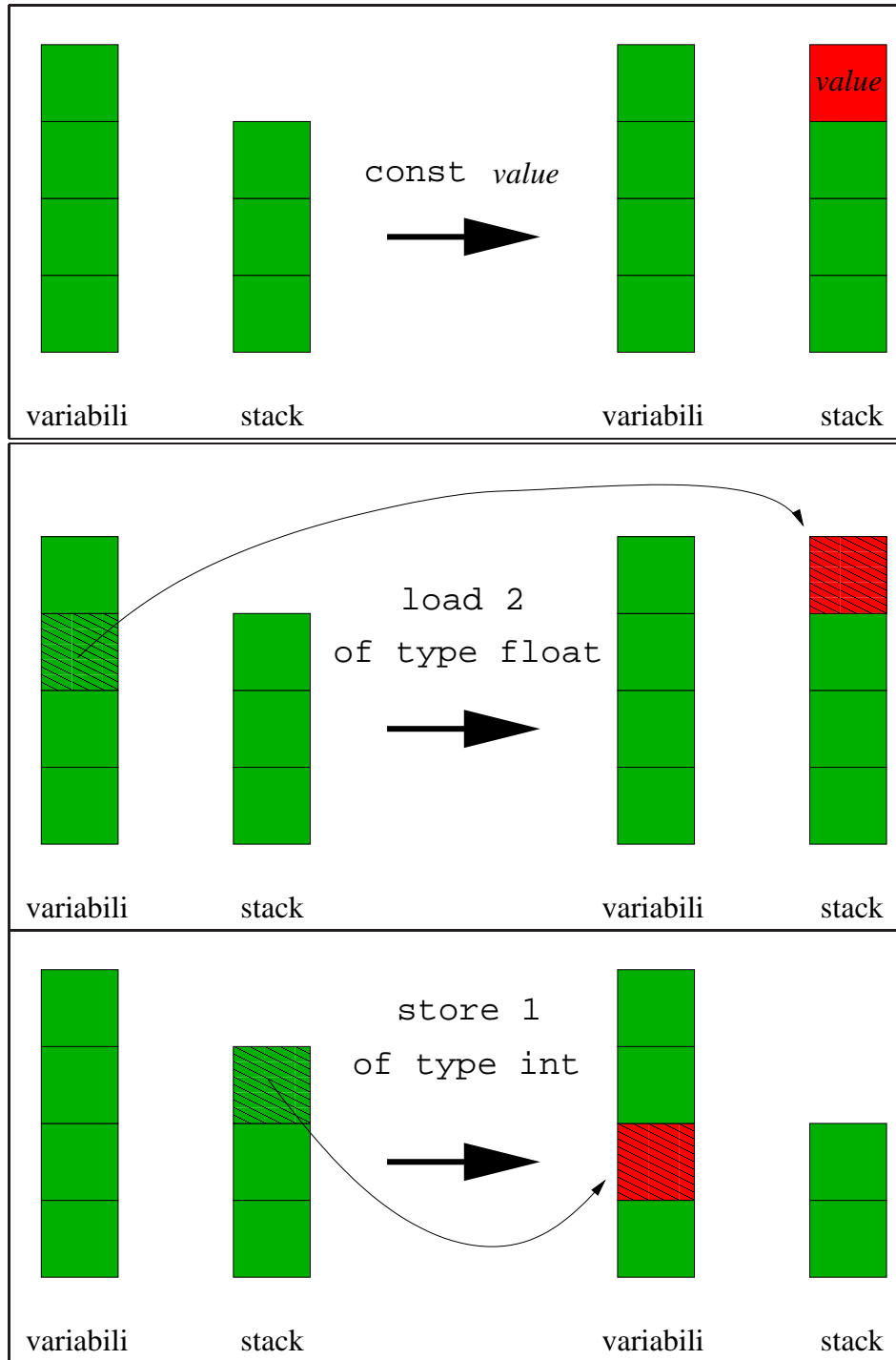
nop. Questa istruzione non modifica in nulla lo stato della macchina astratta. L'effetto della sua esecuzione può quindi essere rappresentato come in Figura 6.4.

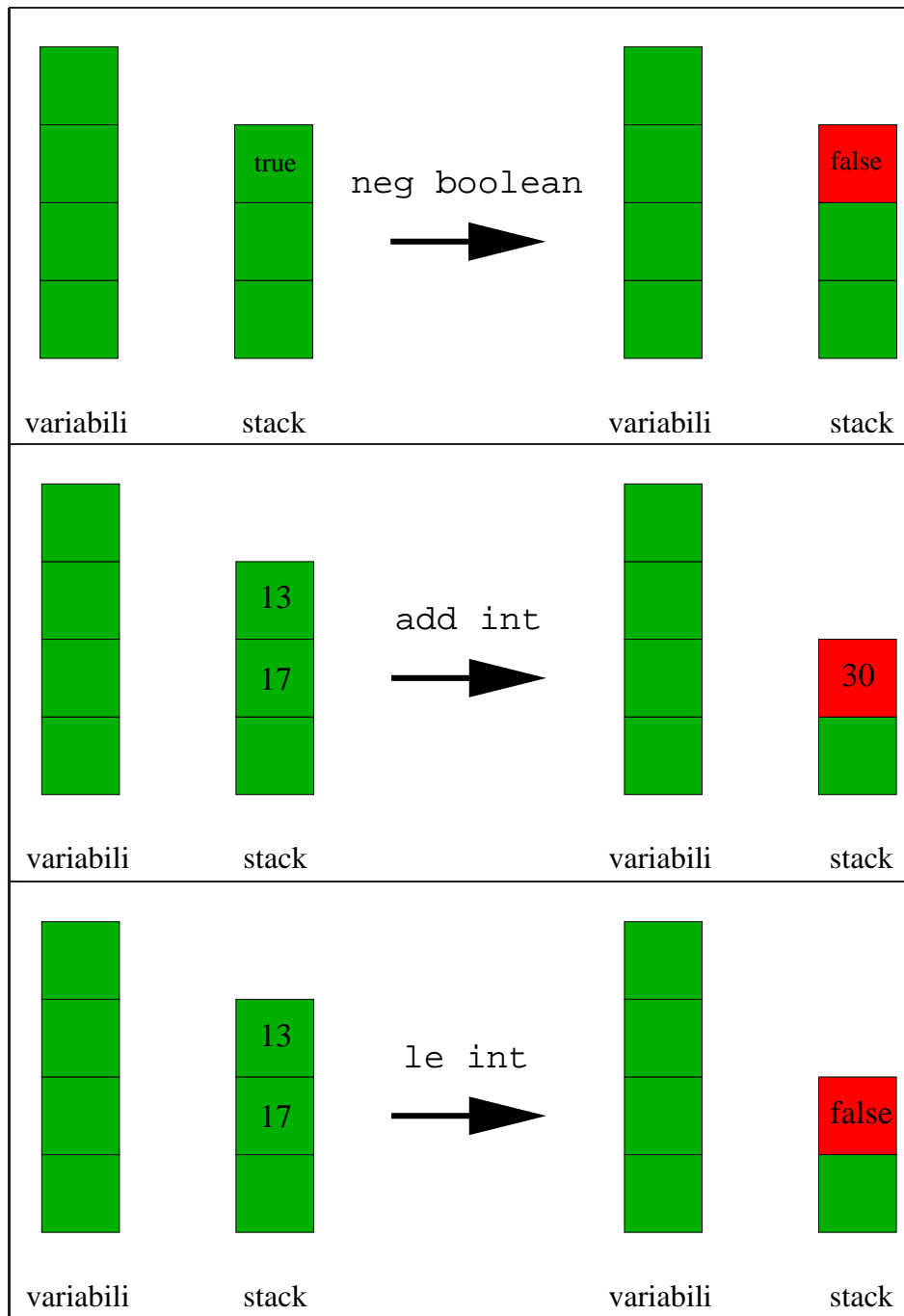
pop *t*. Rimuove la cima dello stack degli operandi, che deve avere tipo *t* (Figura 6.4).

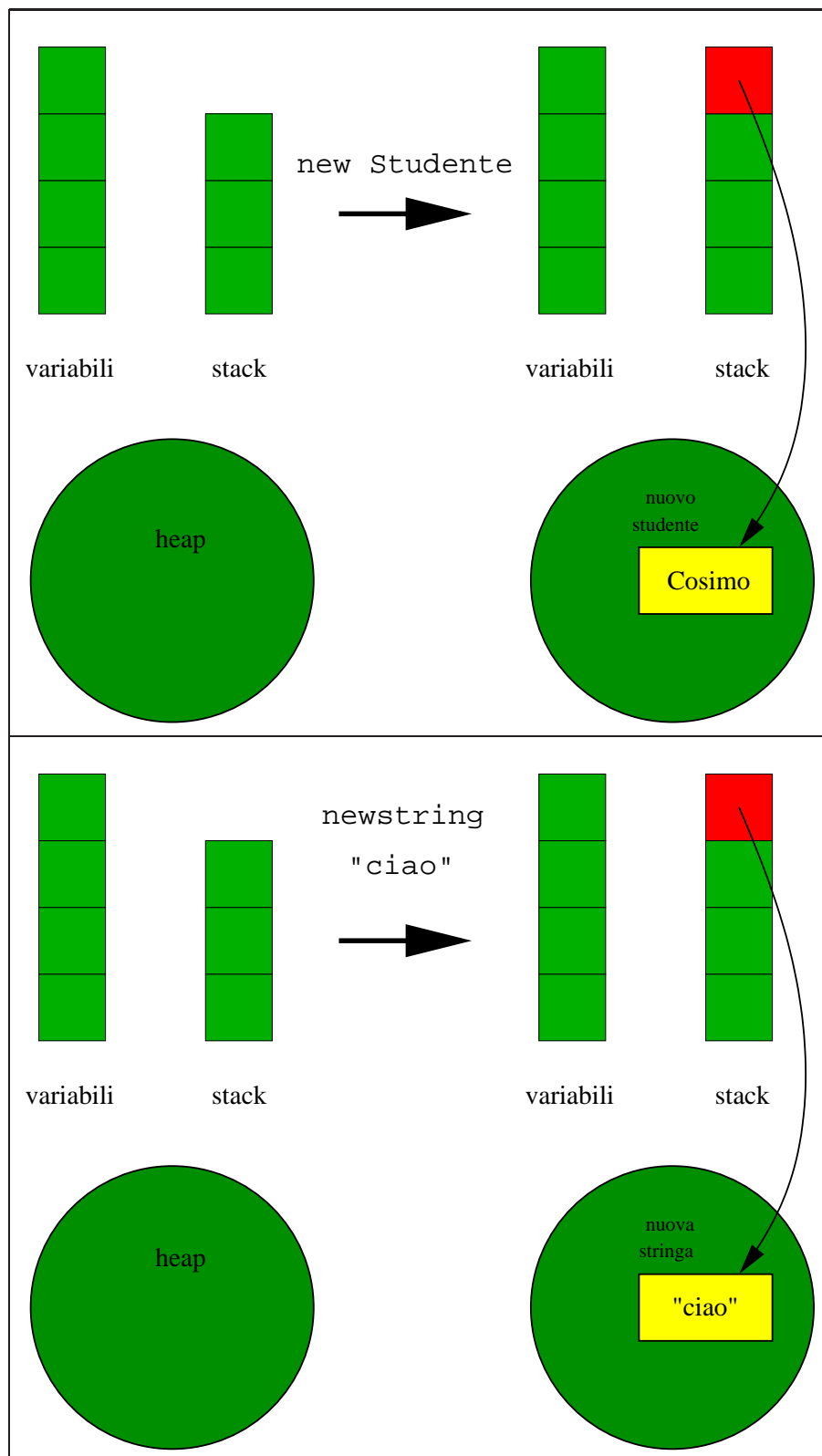
dup *t*. Duplica il valore in cima allo stack (Figura 6.4) che deve avere tipo *t*. Si noti che se tale valore fosse un riferimento a un oggetto o a un array allora verrebbe duplicato il riferimento, non l'oggetto o l'array.

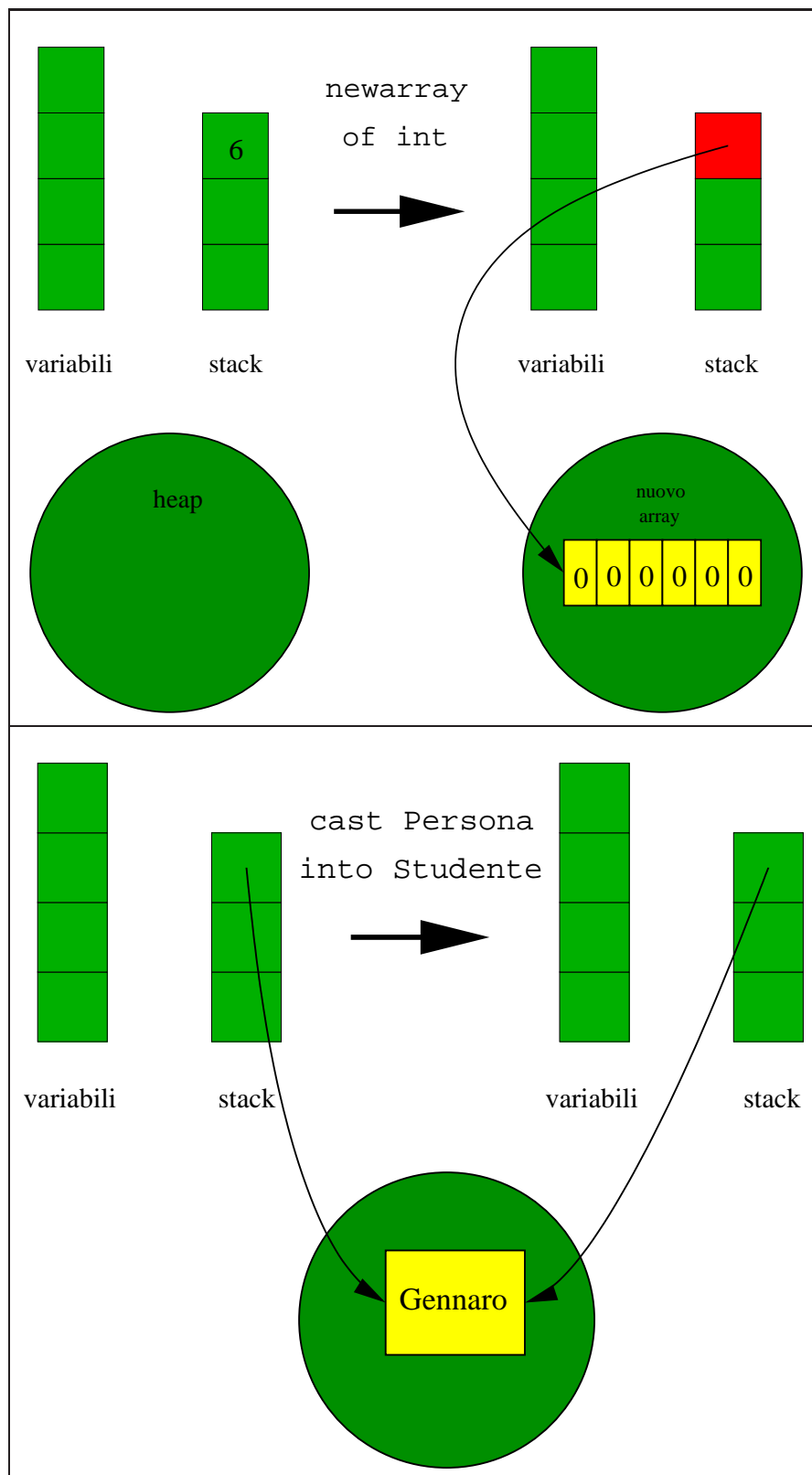
const *value*. Carica in cima allo stack un valore costante (Figura 6.5). È possibile caricare valori booleani, interi, float e la costante `nil`.

Figura 6.4: Le istruzioni `nop`, `pop` e `dup` del bytecode Kitten.

Figura 6.5: Le istruzioni `const`, `load` e `store` del bytecode Kitten.

Figura 6.6: Le istruzioni `neg`, `add` e `le` del bytecode Kitten.

Figura 6.7: Le istruzioni `new` e `newstring` del bytecode Kitten.

Figura 6.8: Le istruzioni `newarray` e `cast` del bytecode Kitten.

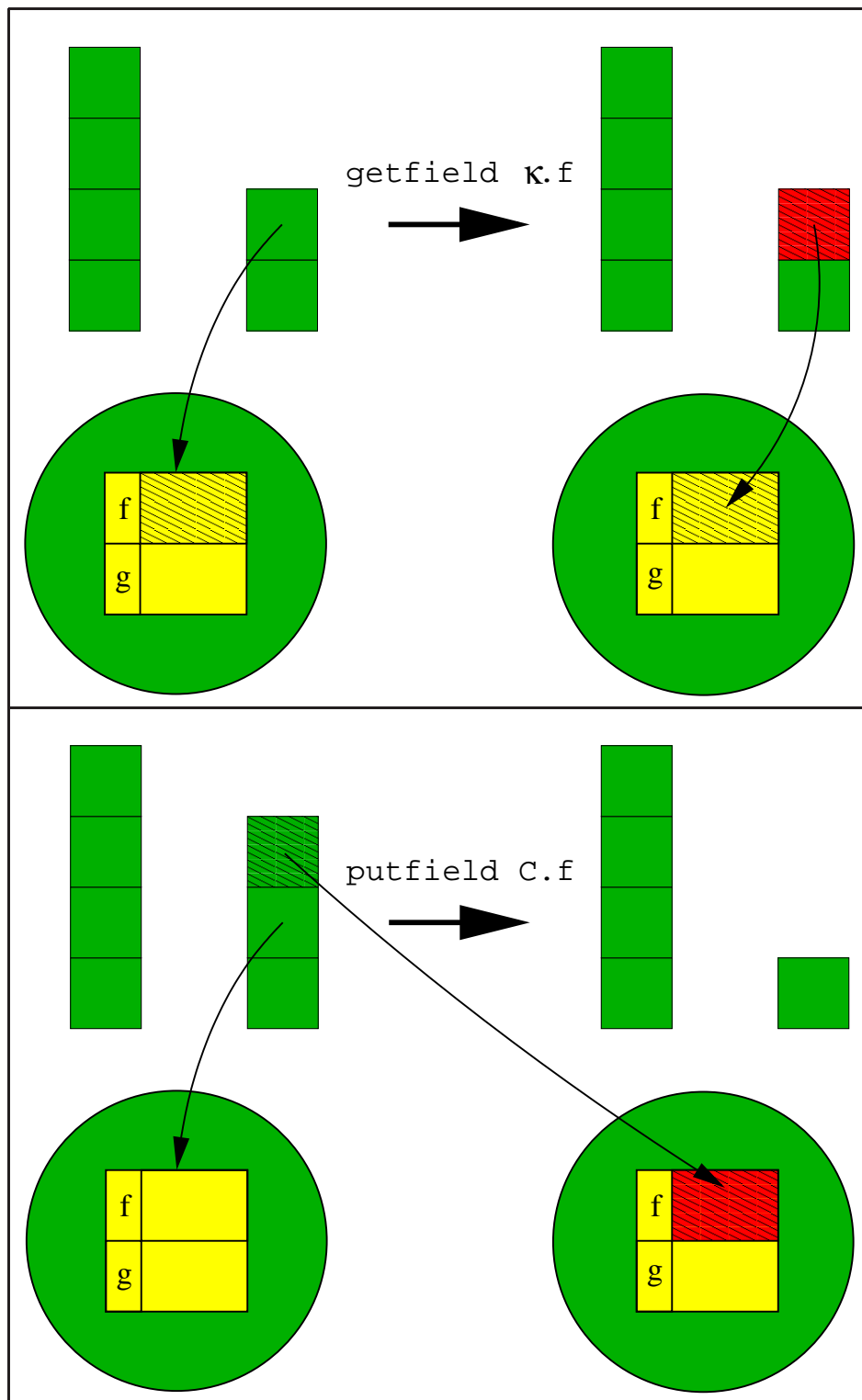


Figura 6.9: Le istruzioni getfield e putfield del bytecode Kitten.

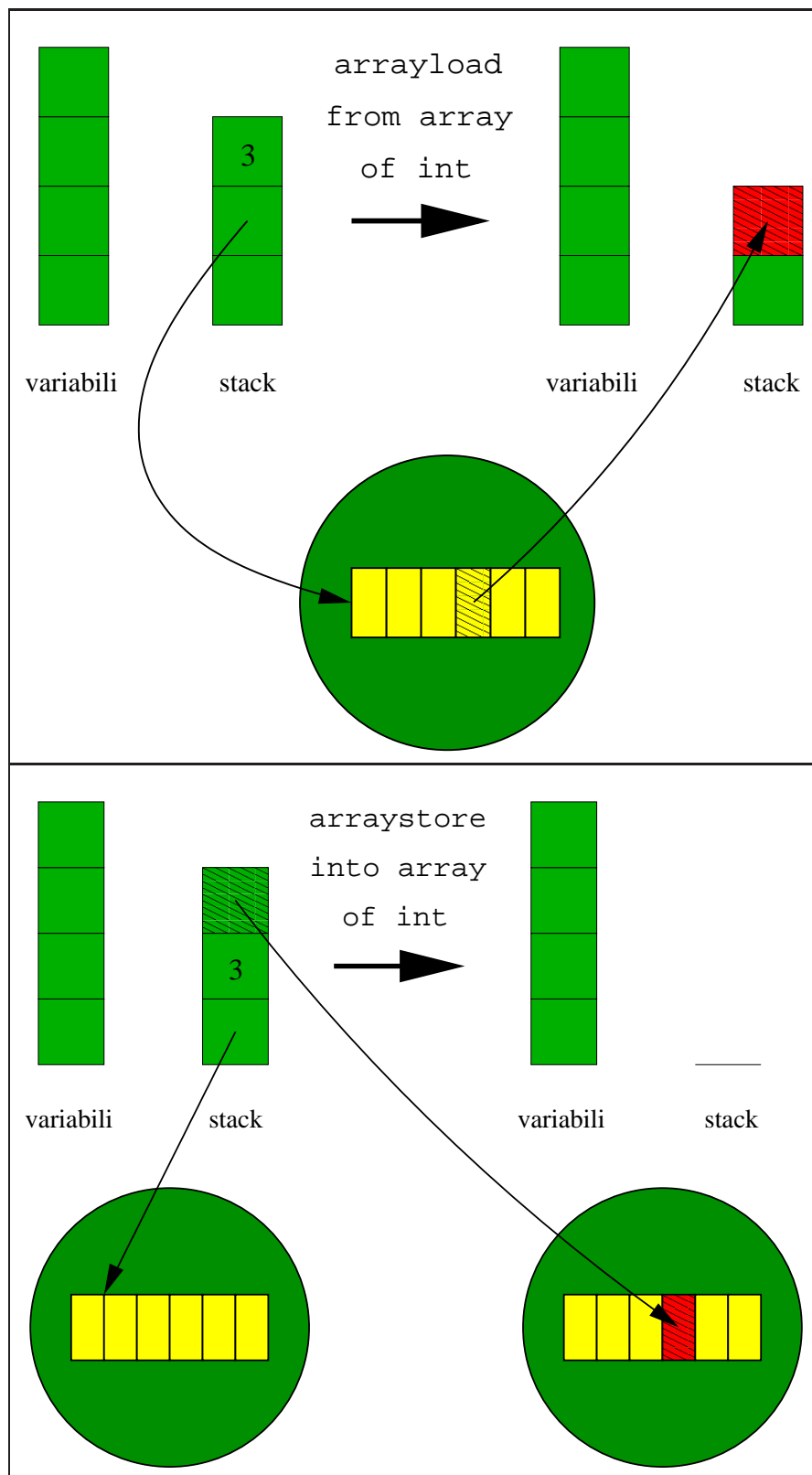


Figura 6.10: Le istruzioni arrayload e arraystore del bytecode Kitten.

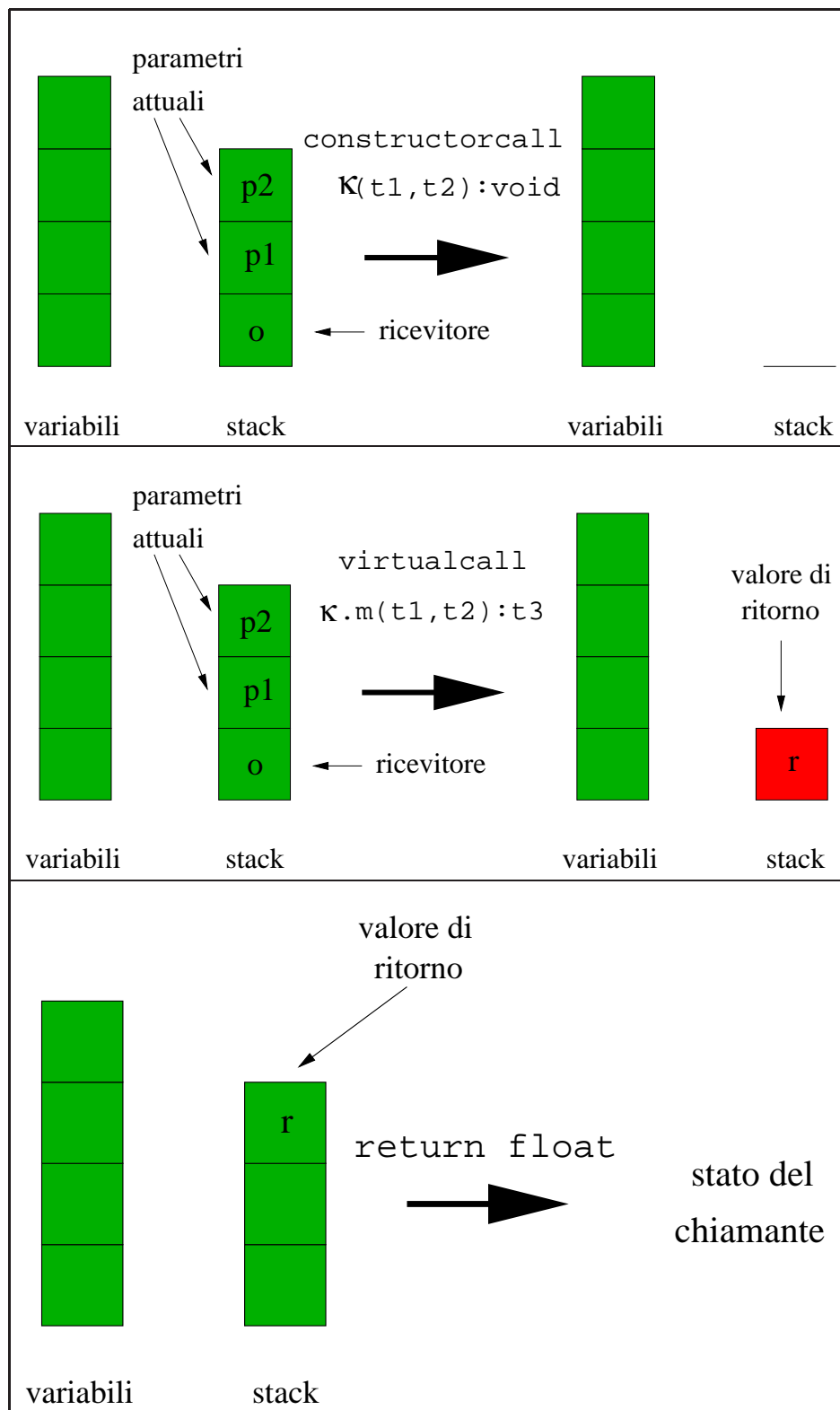


Figura 6.11: Le istruzioni `constructorcall`, `virtualcall` e `return` del bytecode Kitten.

load l of type t . Carica in cima allo stack degli operandi una copia del valore della variabile locale numero l , che deve contenere un valore di tipo t (Figura 6.5).

store l of type t . Sposta dentro la variabile locale numero l il valore che si trova in cima allo stack degli operandi. La cima di tale stack deve contenere un valore del tipo t e viene rimossa dall'operazione (Figura 6.5).

neg t . Nega il valore in cima allo stack degli operandi (Figura 6.6). Tale valore deve essere di tipo t . È possibile che t sia `boolean`, `int` o `float`. Si noti che il valore in cima allo stack che viene usato per calcolare l'operazione `neg` scompare dallo stack e viene sostituito dal risultato dell'operazione.

add t . Aggiunge i due valori in cima allo stack degli operandi (Figura 6.6). Tali valori devono essere entrambi di tipo t . È possibile che t sia `int` o `float`. Similmente ad `add`, esistono anche le istruzioni `sub`, `mul` e `div`. Esistono anche le istruzioni `and` e `or` che però operano su due valori di tipo `boolean`. I due valori in cima allo stack che sono usati per calcolare l'operazione binaria scompaiono dallo stack e vengono sostituiti col risultato dell'operazione. Si noti che questa operazione non effettua alcuna promozione di tipo, per cui è vietato aggiungere un intero con un numero in virgola mobile usando $t = \text{float}$.

le t . Controlla che il valore sotto la cima dello stack degli operandi sia minore o uguale al valore in cima allo stesso stack e sostituisce tali due valori con il risultato booleano del confronto. (Figura 6.6). I due valori devono essere di tipo t pari a `int` o `float`. Esistono anche le istruzioni `lt`, `ge` e `gt`. Infine esistono anche le istruzioni `eq` ed `ne`, che possono operare su valori di tipo t arbitrario, anche riferimento.

new κ . Crea un nuovo oggetto di classe κ . Un riferimento a tale oggetto viene posto in cima allo stack degli operandi (Figura 6.7). Si noti che non viene chiamato alcun costruttore per l'oggetto appena creato. Esso dovrà essere chiamato successivamente con un'esplicita istruzione `constructorcall` (si veda dopo).

newstring s . Crea un nuovo oggetto stringa che rappresenta s e pone in cima allo stack un riferimento all'oggetto, che è già inizializzato (Figura 6.7).

newarray of t . Crea un array i cui elementi hanno tipo t . La lunghezza dell'array è specificata in cima allo stack degli operandi ed è sostituita con un riferimento all'array appena creato (Figura 6.8).

cast t_1 into t_2 . Effettua il cast del valore che sta in cima allo stack, che deve avere tipo t_1 , nel tipo t_2 . Questo bytecode può essere usato per fare cast verso il basso di tipi riferimento (nel qual caso un cast errato interrompe il programma) o per effettuare conversioni di tipo da `int` a `float` o viceversa (Figura 6.8).

getfield $\kappa.f$. Legge il campo f dell'oggetto il cui riferimento è in cima allo stack degli operandi. Tale riferimento viene rimosso e al suo posto viene messo il valore letto dal campo

(Figura 6.9). L'oggetto in cima allo stack deve essere di tipo κ o di una sottoclasse di κ . Se tale oggetto è `nil` il programma viene interrotto.

putfield $\kappa.f$. Scrive il valore in cima allo stack degli operandi dentro il campo f dell'oggetto il cui riferimento sta subito sotto la cima dello stack. I primi due elementi dello stack vengono rimossi (Figura 6.9). Il valore in cima allo stack deve essere del tipo del campo dentro cui si sta scrivendo o di un suo sottotipo. L'oggetto sotto la cima dello stack deve essere di tipo κ o di una sottoclasse di κ . Se tale oggetto è `nil` il programma viene interrotto.

arrayload from array of t . Copia in cima allo stack degli operandi il valore di un elemento di un array. L'indice dell'elemento è in cima allo stack. Subito sotto è presente il riferimento all'array (Figura 6.10) i cui elementi hanno tipo t o sottotipo di t . Se il riferimento all'array è `nil` o se l'indice è fuori dagli estremi dell'array, il programma viene interrotto. I primi due elementi in cima allo stack vengono rimossi dall'operazione e sostituiti con il valore letto dall'array.

arraystore into array of t . Scrive dentro a un array il valore che sta in cima allo stack. Sotto la cima dello stack c è l'indice dell'elemento dell'array che deve essere scritto. Ancora sotto c è il riferimento all'array che si sta modificando (Figura 6.10). Gli elementi dell'array che si sta modificando devono essere di tipo t o di un sottotipo di t . Se il riferimento all'array è `nil` o se l'indice è fuori dagli estremi dell'array, il programma viene interrotto. I primi tre elementi in cima allo stack vengono rimossi dall'operazione.

6.1.2 Le istruzioni di chiamata e ritorno da metodo

La Figura 6.11 mostra le istruzioni usate per chiamare un costruttore o metodo e per ritornare il controllo al chiamante. Esse operano come segue:

constructorcall $\kappa(\vec{t}) : \text{void}$. Chiama il costruttore della classe κ i cui parametri formali hanno tipo \vec{t} . I parametri attuali e l'oggetto che si sta inizializzando (cioè il *ricevitore* dal punto di vista del chiamante e il parametro implicito `this` di Kitten dal punto di vista del chiamato) sono passati tramite lo stack degli operandi e vengono rimossi alla fine della chiamata. Questo è mostrato in Figura 6.11, dal punto di vista del chiamante. La classe del ricevitore deve essere κ . Se il ricevitore è `nil` l'esecuzione del programma termina.

virtualcall $\kappa.m(\vec{t}) : t'$. Chiama il metodo di nome m e parametri formali di tipo \vec{t} cercandolo a partire dalla classe del ricevitore e risalendo nella catena delle superclassi. Il ricevitore e i parametri attuali della chiamata si trovano sullo stack al momento della chiamata e vengono rimossi alla fine della chiamata e sostituiti con il valore di ritorno del metodo, nel caso in cui t' non è `void`. Questo è mostrato in Figura 6.11 dal punto di vista del chiamante. La classe del ricevitore deve essere κ o una sottoclasse di κ . Se il ricevitore è `nil` l'esecuzione del programma termina.

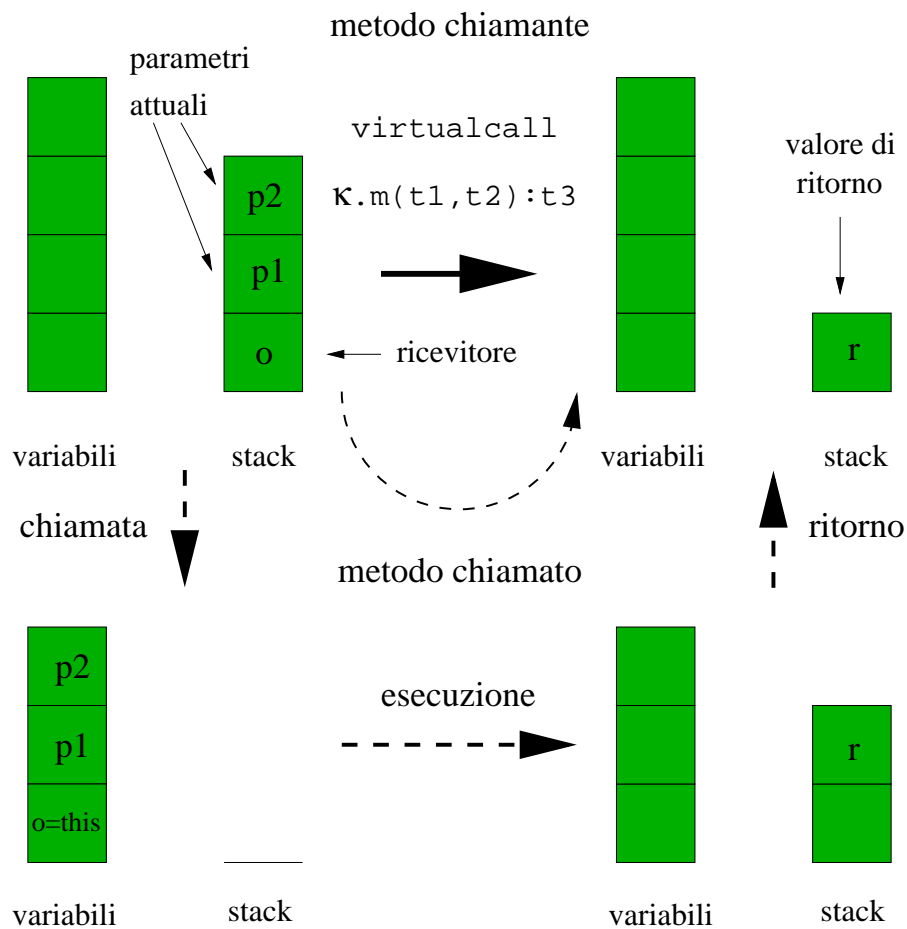


Figura 6.12: Il meccanismo di chiamata e ritorno da metodo.

return t . Termina l'esecuzione del metodo corrente, ritornando il controllo al chiamante, insieme a un eventuale valore di ritorno, che è la cima dello stack degli operandi (Figura 6.11) e deve avere tipo t .

Il funzionamento complessivo del meccanismo di chiamata e ritorno da costruttore o metodo è mostrato nella Figura 6.12. Il metodo chiamante prepara sullo stack degli operandi i parametri della chiamata, incluso il ricevitore della chiamata, indicato come o in Figura 6.12. Il metodo chiamato è esplicito nel caso della chiamata a un costruttore, mentre per le chiamate virtuali ai metodi è identificato sulla base della classe dell'oggetto a cui o fa riferimento. In entrambi i casi, esso inizia la sua esecuzione in un frame di attivazione nuovo, in cui le variabili locali contengono i parametri della chiamata e lo stack degli operandi è vuoto. Quando l'esecuzione del chiamato termina, se il metodo non ritorna `void` allora la cima dello stack degli operandi del chiamato contiene il valore di ritorno, r in Figura 6.12. La terminazione del metodo riabilita il frame di attivazione del chiamato, in cui però lo stack degli operandi è stato privato dei parametri e arricchito con il valore di ritorno r del metodo.

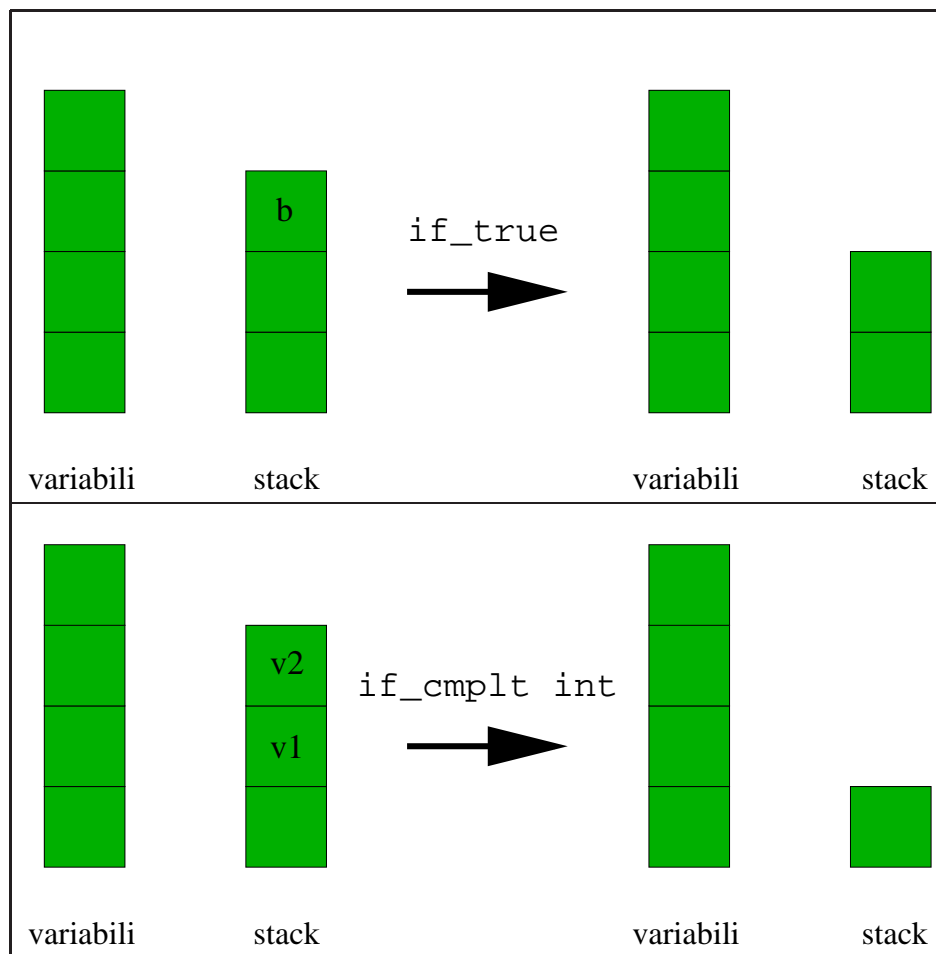


Figura 6.13: Le istruzioni `if_true` ed `if_cmplt` del bytecode Kitten.

6.1.3 Le istruzioni di diramazione

Le istruzioni di diramazione del bytecode Kitten sono sempre accoppiate all'inizio di due blocchi di codice con lo stesso predecessore. Esse indicano sotto quale condizione il controllo del programma deve essere instradato verso uno dei due blocchi. Ne sono esempi le istruzioni `if_true` ed `if_false` in Figura 6.1 e le istruzioni `if_cmplt int` e `if_cmpge int` in Figura 6.2. Quando la condizione espressa dall'istruzione condizionale è vera, essa viene eseguita, il che normalmente comporta l'eliminazione di alcuni valori dallo stack degli operandi.

Vediamo in dettaglio l'insieme delle istruzioni di diramazione del bytecode Kitten.

if_true. La condizione espressa da questa istruzione è che la cima dello stack degli operandi, che deve essere un booleano, sia il valore `true`. In tal caso il valore viene eliminato dallo stack (Figura 6.13). Esiste anche l'istruzione simmetrica `if_false`.

if_cmplt t. La condizione espressa da questa istruzione è che l'elemento che sta sotto la cima dello stack degli operandi sia minore dell'elemento che sta in cima allo stack. Entrambi

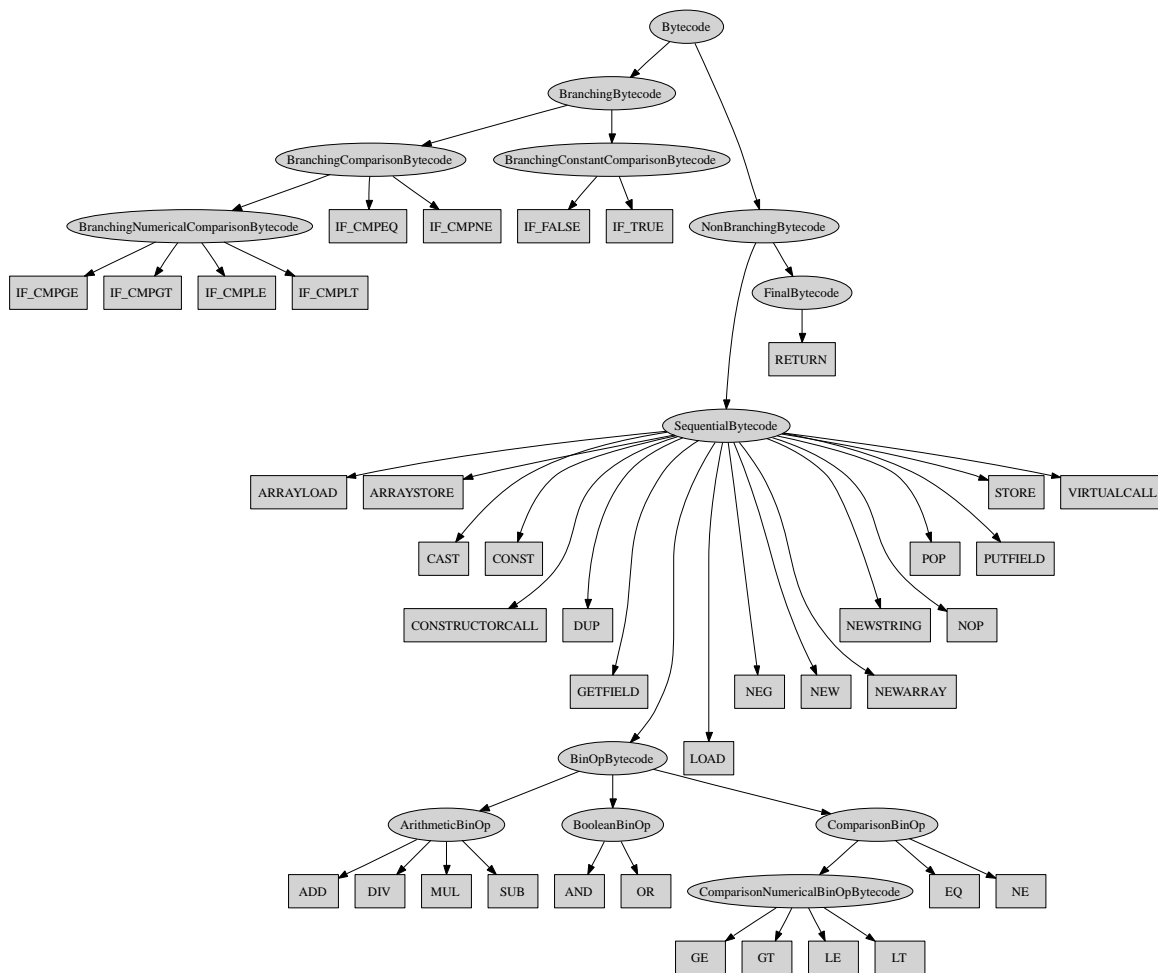


Figura 6.14: La gerarchia delle classi del package bytecode che rappresentano le istruzioni del bytecode Kitten. Le classi ovali sono classi astratte, quelle rettangolari sono classi concrete.

gli elementi devono avere tipo t e vengono rimossi dallo stack (Figura 6.13). Il tipo t può essere `int` o `float`. Esistono anche le istruzioni `if_cmpne`, `if_cmpgt` ed `if_cmpge`. Esistono inoltre le istruzioni `if_cmpeq` ed `if_cmpne` la cui condizione, rispettivamente, è l'uguaglianza e la disuguaglianza dei due elementi in cima allo stack degli operandi. Queste ultime due istruzioni possono operare su tipi t arbitrari, anche riferimento.

6.1.4 L'implementazione del bytecode Kitten

Le istruzioni del bytecode Kitten che abbiamo descritto nelle sezioni precedenti sono implementate nel package bytecode come istanze della classe `bytecode/Bytecode.java`. La gerarchia completa è mostrata in Figura 6.14. Le istruzioni vengono prima di tutto divise nelle due classi astratte `NonBranchingBytecode` e `BranchingBytecode`. La prima implementa le istruzioni

sequenziali delle Sezioni 6.1.1 e 6.1.2. La seconda implementa le istruzioni di diramazione della Sezione 6.1.3.

La creazione di un bytecode avviene tramite il suo costruttore, che richiede di specificare i tipi semantici su cui opera il bytecode. Per esempio, un'istruzione `arrayload from array of int` si crea con l'espressione Java

```
new ARRAYLOAD(Type.INT)
```

La classe `bytecode/BytecodeList.java` implementa poi una lista di bytecode che può essere inserita all'interno di un blocco di codice (come in Figura 6.2). La struttura dati che implementa tale blocco è la classe `translate/CodeBlock.java` il cui costruttore chiede di specificare la lista di bytecode contenuta nel blocco e la lista dei successori del blocco (eventualmente vuota).

Un metodo importante della classe dei bytecode sequenziali è `followedBy()`: esso richiede di specificare un blocco di codice e restituisce un blocco ottenuto aggiungendo il bytecode in testa al codice interno al blocco di codice. Per esempio, se il blocco di codice *b* contiene

```
const 1
return int
```

allora `new IF_TRUE().followedBy(b)` è un blocco di codice che contiene

```
if_true
const 1
return int
```

6.2 La generazione del bytecode Kitten per le espressioni

Mostriamo in questa sezione come tradurre la sintassi astratta di un'espressione Kitten in del bytecode Kitten.

Abbiamo visto che un programma scritto in bytecode Kitten è un insieme di blocchi all'interno dei quali si trova del codice, come mostrato in Figura 6.3. Il bytecode che genereremo per le espressioni sarà in effetti molto semplice, al punto che una sequenza di blocchi sarà sempre sufficiente per tutte le espressioni. Si noti comunque che questo non sarebbe più vero se Kitten ammettesse espressioni più complesse, come per esempio un'espressione condizionale come `exp ? exp : exp` (si veda l'Esercizio 25).

Ci sono tre contesti in cui un'espressione Kitten può trovarsi:

1. un contesto in cui di un'espressione serve il valore, come nel caso in cui essa occorre come lato destro di un assegnamento;
2. un contesto in cui di un'espressione serve sapere se è vera o falsa per decidere come intradare l'esecuzione del programma, come nel caso in cui essa occorre come test di un condizionale;
3. un contesto in cui il valore di un'espressione deve essere modificato, come nel caso in cui essa occorre alla sinistra di un assegnamento.

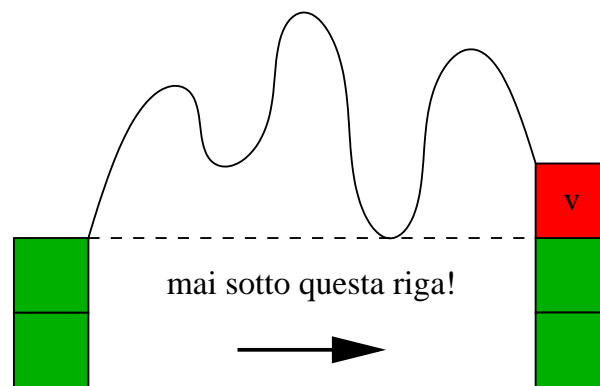


Figura 6.15: L'esecuzione del bytecode Kitten generato per un'espressione deve lasciare il valore dell'espressione sullo stack degli operandi e non deve modificare lo stack iniziale.

Compileremo un'espressione in tre modi diversi, sulla base del contesto in cui essa occorre. Tali modi sono detti rispettivamente *compilazione attiva*, *compilazione condizionale* e *compilazione passiva* dell'espressione. Descriviamo adesso in ordine questi tre tipi di compilazione delle espressioni.

6.2.1 La compilazione attiva delle espressioni

Quando di un'espressione ci interessa il valore, allora l'esecuzione del codice che vogliamo generare deve essere tale da:

1. lasciare intatti i valori iniziali sullo stack degli operandi;
2. aggiungere in cima allo stack degli operandi il valore dell'espressione.

Questi due principi sono mostrati in Figura 6.15. Il vincolo 1 è importante poiché esso ci permette di valutare in sequenza delle espressioni e ritrovarci alla fine i loro valori sullo stack. Questo è mostrato nella Figura 6.16, che mostra l'esecuzione del codice che genereremo per l'and logico di due espressioni e_1 ed e_2 : prima generiamo del codice che valuta e_1 e ne lascia il valore sullo stack, poi del codice che valuta e_2 e ne lascia il valore sullo stack. Grazie al precedente vincolo 1, siamo certi che a questo punto il valore di e_1 è ancora nello stack, sotto la cima. Possiamo quindi aggiungere un bytecode and per ottenere il risultato cercato.

Se β è un blocco di codice, allora con la notazione $\boxed{ins} \rightarrow \beta$ rappresentiamo un blocco di codice al cui interno si trova l'istruzione (o le istruzioni) ins e che ha β come successore. La Figura 6.17 usa tale notazione per definire le regole per la generazione del codice per le espressioni Kitten. Esse sono formalizzate tramite una funzione $\gamma[_]$ che associa alla sintassi astratta delle espressioni il bytecode Kitten che ne calcola il valore e lo lascia in cima allo stack degli operandi. Tale funzione richiede in primo luogo di specificare l'espressione e di cui si vuole generare il bytecode. La notazione $\gamma[e]$ è però ancora una funzione da `translate.CodeBlock`

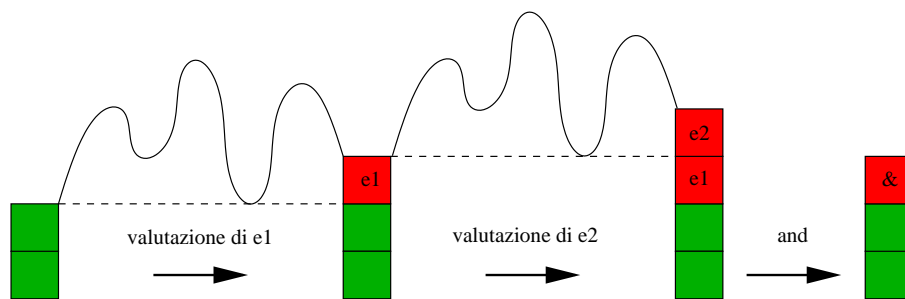


Figura 6.16: L'esecuzione del bytecode Kitten per l'and logico di due espressioni e_1 ed e_2 .

in `translate.CodeBlock`, cioè la classe usata per rappresentare un blocco di bytecode (Sezione 6.1.4). In particolare, quello che occorre ancora specificare è il bytecode β che deve essere eseguito *dopo* la valutazione di e . Il bytecode $\gamma[[e]](\beta)$ sarà quindi il bytecode che *prima* valuta l'espressione e , lasciandone il valore sullo stack degli operandi, e *dopo* esegue il codice β . Per esempio, la Figura 6.17 implica che

$$\gamma[[\text{IntLiteral}(3)]](\boxed{\text{return int}}) = \boxed{\text{const 3}} \rightarrow \boxed{\text{return int}}$$

Questo modo di generare il codice si chiama *compilazione con continuazioni* e β è detta la *continuazione* della compilazione di e . La compilazione per continuazioni è molto elegante poiché permette di semplificare la fusione fra il codice generato per due parti sequenziali di un programma.

La Figura 6.17 usa la funzione $\gamma^\tau[[e]]$ che rispetto a $\gamma[[e]]$ effettua in più, se necessario, la promozione a τ del valore dell'espressione e . In Kitten essa è utile ogni qual volta si usa un valore intero in un punto in cui si richiedeva un valore in virgola mobile come per esempio nell'espressione $3 + 4.5$, in cui occorre convertire il valore intero 3 in un `float` prima di sommarlo con il valore 4.5. Quando potrebbe essere necessaria una promozione di tipo del valore di un'espressione, la Figura 6.17 compila l'espressione tramite $\gamma^\tau[[_]]$ piuttosto che tramite $\gamma[[_]]$. Lo stesso fenomeno lo incontreremo fra poco con i comandi, in un assegnamento del tipo:

```
float f := 13
```

dove l'intero 13 deve essere convertito in `float` prima dell'assegnamento. Tale funzione di conversione è definita a partire da $\gamma[[e]]$:

$$\gamma^\tau[[e]](\beta) = \begin{cases} \gamma[[e]](\boxed{\text{cast from int into float}} \rightarrow \beta) & \text{se } \tau \text{ è float ed } e \text{ ha tipo statico int} \\ \gamma[[e]](\beta) & \text{altrimenti.} \end{cases} \quad (6.1)$$

Per esempio,

$$\begin{aligned} \gamma^{\text{int}}[[\text{IntLiteral}(3)]](\boxed{\text{return int}}) &= \gamma[[\text{IntLiteral}(3)]](\boxed{\text{return int}}) \\ &= \boxed{\text{const 3}} \rightarrow \boxed{\text{return int}} \end{aligned}$$

$$\begin{aligned}
\gamma[_]: \text{absyn.Expression} &\mapsto (\text{translate.CodeBlock} \mapsto \text{translate.CodeBlock}) \\
\gamma[\text{Variable}(name)](\beta) &= \boxed{\text{load num of type } \tau} \rightarrow \beta \\
&\text{dove } num \text{ è il numero progressivo della variabile } name \text{ nel metodo corrente} \\
\gamma[\text{FieldAccess}(receiver, name)](\beta) &= \gamma[receiver] \left(\boxed{\text{getfield field}} \rightarrow \beta \right) \\
&\text{dove } field \text{ è il campo identificato dall'analisi semantica (Figura 5.10)} \\
\gamma[\text{ArrayAccess}(array, index)](\beta) &= \gamma[array] \left(\gamma[index] \left(\boxed{\text{arrayload from array of } \tau} \rightarrow \beta \right) \right) \\
\gamma[\text{True}()](\beta) &= \boxed{\text{const true}} \rightarrow \beta \quad \gamma[\text{False}()](\beta) = \boxed{\text{const false}} \rightarrow \beta \\
\gamma[\text{IntLiteral}(value)](\beta) &= \gamma[\text{FloatLiteral}(value)](\beta) = \boxed{\text{const value}} \rightarrow \beta \\
\gamma[\text{String}(value)](\beta) &= \boxed{\text{newstring value}} \rightarrow \beta \quad \gamma[\text{Nil}()](\beta) = \boxed{\text{const nil}} \rightarrow \beta \\
\gamma[\text{NewObject}(className, actuals)](\beta) &= \boxed{\text{new } \kappa \text{ dup } \kappa} \rightarrow \gamma^{\vec{r}}[actuals] \left(\boxed{\text{constructorcall con}} \rightarrow \beta \right) \\
&\text{dove } con = \kappa.\kappa(\vec{r}): \text{void} \text{ è il costruttore identificato dall'analisi semantica (Figura 5.11)} \\
\gamma[\text{NewArray}(elementType, size)](\beta) &= \gamma[size] \left(\boxed{\text{newarray of } \tau.\text{getElementType}()} \rightarrow \beta \right) \\
\gamma[\text{MethodCallExpression}(receiver, name, actuals)] &= \gamma[receiver] \left(\gamma^{\vec{r}}[actuals] \left(\boxed{\text{virtualcall method}} \rightarrow \beta \right) \right) \\
&\text{dove } method = \kappa.m(\vec{r}): t' \text{ è il metodo identificato dall'analisi semantica (Figura 5.11)} \\
\gamma[\text{Not}(expression)](\beta) &= \gamma[\text{Minus}(expression)](\beta) = \gamma[expression] \left(\boxed{\text{neg } \tau} \rightarrow \beta \right) \\
\gamma[\text{Cast}(type, expression)](\beta) &= \gamma[expression] \left(\boxed{\text{cast from } \tau' \text{ into } \tau} \rightarrow \beta \right) \text{ con } \tau' \text{ è tipo statico di } expression \\
\gamma[\text{And}(left, right)](\beta) &= \gamma[left] \left(\gamma[right] \left(\boxed{\text{and}} \rightarrow \beta \right) \right) \\
\gamma[\text{Addition}(left, right)](\beta) &= \gamma^{\tau}[left] \left(\gamma^{\tau}[right] \left(\boxed{\text{add } \tau} \rightarrow \beta \right) \right) \\
\gamma[\text{LessThanOrEqual}(left, right)](\beta) &= \gamma^{\ell}[left] \left(\gamma^{\ell}[right] \left(\boxed{\text{le } \ell} \rightarrow \beta \right) \right) \text{ con } \ell \text{ minimo supertipo comune fra il tipo statico di } left \text{ e di } right
\end{aligned}$$

Figura 6.17: La funzione $\gamma[_]$ che genera il bytecode Kitten che valuta le espressioni. Il tipo τ è il tipo statico assegnato all'espressione durante la sua analisi semantica.

mentre

$$\begin{aligned}
&\gamma^{\text{float}}[\text{IntLiteral}(3)](\boxed{\text{return float}}) \\
&= \gamma[\text{IntLiteral}(3)](\boxed{\text{cast from int into float}} \rightarrow \boxed{\text{return float}}) \\
&= \boxed{\text{const 3}} \rightarrow \boxed{\text{cast from int into float}} \rightarrow \boxed{\text{return float.}}
\end{aligned}$$

L'esempio precedente sarebbe quello di un'istruzione `return 3` che occorre all'interno di un metodo il cui tipo di ritorno è `float`. La notazione $\gamma^{\tau}[_]$ viene infine estesa a sequenze di espressioni e di tipi (di uguale lunghezza), ottenendo la notazione $\gamma^{\vec{r}}[_]$, definita come segue:

$$\begin{aligned}
\gamma^{\epsilon}[\text{null}] &= \beta \\
\gamma^{\tau::\vec{r}}[\text{ExpressionSeq}(head, tail)](\beta) &= \gamma^{\tau}[\text{head}] \left(\gamma^{\vec{r}}[\text{tail}] \right) (\beta) .
\end{aligned}$$

Per esempio:

$$\begin{aligned}
 & \gamma^{\text{float}::\text{float}} \left[\left[\begin{array}{l} \text{ExpressionSeq}(\text{FloatLiteral}(3.4), \\ \text{ExpressionSeq}(\text{IntLiteral}(4), \text{null})) \end{array} \right] \right] (\text{add float}) \\
 &= \gamma^{\text{float}} \llbracket \text{FloatLiteral}(3.4) \rrbracket (\gamma^{\text{float}} \llbracket \text{IntLiteral}(4) \rrbracket (\text{add float})) \\
 &= \gamma^{\text{float}} \llbracket \text{FloatLiteral}(3.4) \rrbracket (\text{const } 4 \rightarrow \text{cast from int to float} \rightarrow \text{add float}) \\
 &= \text{const } 3.4 \rightarrow \text{const } 4 \rightarrow \text{cast from int to float} \rightarrow \text{add float.}
 \end{aligned}$$

Commentiamo adesso le regole di generazione del bytecode Kitten in Figura 6.17.

Variable(name). Per caricare sullo stack il valore di una variabile locale, usiamo il bytecode `load` della Figura 6.5. Il numero della variabile locale è già stato determinato in fase di analisi semantica e annotato dentro all'ambiente del punto di programma in cui ci troviamo, insieme al tipo della variabile (Figura 5.16).

FieldAccess(receiver, name). Per accedere a un campo dell'oggetto o contenuto in $receiver$, generiamo inizialmente il bytecode che lascia sullo stack degli operandi il riferimento ad o . Questo è ottenuto richiamando ricorsivamente la generazione del bytecode per $receiver$. Come continuazione, gli passiamo un blocco che contiene un bytecode `getField` (Figura 6.9) e che è legato alla continuazione β . L'effetto globale è quindi quello di valutare $receiver$, leggere il valore del campo di nome $name$ e quindi continuare con la continuazione β . Si noti che il campo da leggere è già stato identificato in fase di analisi semantica ($field$ in Figura 5.10).

ArrayAccess(array, index). Leggere un elemento di un array richiede in primo luogo di valutare l'espressione che contiene il riferimento all'array. Questo è ottenuto compilando ricorsivamente $array$. Come continuazione gli diamo la compilazione di $index$, seguita dal bytecode `arrayload` e dalla continuazione β . Si noti che il tipo statico τ dell'array è già stato calcolato in fase di analisi semantica. Il bytecode `arrayload` consumerà dallo stack il riferimento all'array e l'indice da cui leggere e li sostituirà con il valore dell'elemento letto (Figura 6.10).

True(), False(), IntLiteral(value), FloatLiteral(value), String(value), Nil(). Dal momento che queste classi di sintassi astratta rappresentano delle costanti, usiamo il bytecode `const` della Figura 6.5 e `newstring` della Figura 6.7 per caricare tali costanti in cima allo stack.

NewObject(className, actuals). Questo nodo di sintassi astratta per la creazione di un oggetto di classe $className$ è stato annotato durante l'analisi semantica con il costruttore $con = \kappa.\kappa(\vec{r})$: `void` della classe κ che è il corrispondente semantico di $className$. Tale costruttore è il più specifico fra quelli che possono essere chiamati da questa espressione sulla base del tipo statico dei parametri attuali (Figura 5.11). Il codice che generiamo inizia con un bytecode `new κ` che crea un nuovo oggetto o di classe κ e ne pone in cima allo stack

un riferimento (Figura 6.7). Tale riferimento viene quindi duplicato dal bytecode `dup κ` (Figura 6.4). Segue la compilazione dei parametri attuali del costruttore. A questo punto sullo stack troviamo due copie di un riferimento ad o sormontate dai valori dei parametri attuali. Con il bytecode `constructorcall` otteniamo quindi di chiamare il costruttore legando `this` ad o e i parametri attuali ai parametri formali (Figura 6.11). Per esempio, la compilazione di

```
NewObject(className,
            ExpressionSeq(IntLiteral(3), ExpressionSeq(IntLiteral(4), null)))
```

è

```
new  $\kappa$ 
dup  $\kappa$ 
const 3
const 4
constructorcall con
```

seguita dalla continuazione β (nell'ipotesi che non serva promozione di tipo nel passaggio dei parametri interi al costruttore). Dalla Figura 6.11 sappiamo che il bytecode `constructorcall` rimuove dallo stack degli operandi sia i parametri attuali che o . Questo è il motivo per cui usiamo il bytecode `dup`: senza di esso il riferimento ad o andrebbe perso dallo stack e avremmo ottenuto di inizializzare un oggetto che subito dopo diventava irraggiungibile.

`NewArray(elementType, size)`. Il bytecode generato per la creazione di un array inizia con la compilazione dell'espressione *size* che lascia in cima allo stack la dimensione richiesta per l'array. Basta quindi proseguire il codice con un bytecode `newarray` che consuma tale dimensione e la sostituisce con un riferimento a un nuovo array (Figura 6.8). Segue la continuazione β . Si ricordi che il tipo statico τ di questa espressione è il tipo dell'array che stiamo creando (Figura 5.11).

`MethodCallExpression(receiver, name, actuals)`. L'invocazione di un metodo è compilata in modo molto simile all'invocazione di un costruttore per un nodo `NewObject` di sintassi astratta (si veda sopra). La differenza è che si usa il bytecode `virtualcall` invece di `constructorcall` (Figura 6.11). Inoltre il riferimento all'oggetto ricevitore della chiamata è il valore lasciato sullo stack dal bytecode generato per *receiver*, piuttosto che un nuovo oggetto come per `NewObject`. Si ricordi che l'analisi semantica ha garantito che il metodo invocato ha un tipo di ritorno diverso da `void` (Figura 5.11). Siamo quindi sicuri che il bytecode `virtualcall` lascia sullo stack un valore di ritorno (Figura 6.11), che è il valore di questa espressione d'invocazione di un metodo.

`Not(expression)` e `Minus(expression)`. Entrambe queste espressioni sono compilate in del bytecode che inizia con la compilazione ricorsiva di *expression* e continua con il bytecode `neg` (Figura 6.6) e con la continuazione β . Si noti comunque che il tipo τ su cui opera `neg` è diverso: esso è `boolean` per `Not` ed è `int` oppure `float` per `Minus` (Figura 5.11).

Cast(*type, expression*). La compilazione di un cast verso il basso è essenzialmente la compilazione dell'espressione di cui si sta facendo il cast, seguita dalla continuazione β . In più inseriamo un bytecode cast che effettua il cast o la conversione di tipo da float ad int (Figura 6.8) nel caso in cui il cast sia in effetti una richiesta di arrotondamento di un valore a virgola mobile (3.14 as int). Si noti la differenza fra queste due situazioni: la conversione da float ad int modifica la rappresentazione binaria del valore in cima allo stack ma non può mai fallire (sappiamo con certezza che in cima allo stack c'è un float). La verifica di tipo non effettua invece alcuna modifica sul valore in cima allo stack, ma può fallire bloccando l'esecuzione del programma.

BinOp(*left, right*). La compilazione di un'operazione binaria è il codice formato dalla compilazione di *left* seguita dalla compilazione di *right* seguita da un bytecode che implementa l'operazione binaria opportuna e infine dalla continuazione β . Gli esempi mostrati in Figura 6.17 presentano tutte le tipologie di espressioni binarie. Quelle logiche usano un bytecode and od or per il quale non serve specificare il tipo degli operandi (è sempre boolean). Quelle aritmetiche possono invece operare sia su int che su float e i loro operandi potrebbero richiedere una promozione di tipo, per cui usiamo per essi $\gamma^r[_]$ piuttosto che $\gamma[_]$. Le operazioni binarie di confronto possono operare su tipo arbitrari e possono anch'esse richiedere una conversione di tipo per gli operandi.

Consideriamo adesso l'implementazione delle regole di compilazione in Figura 6.17. Un blocco di codice lo implementiamo come un oggetto di classe `translate.CodeBlock` contenente una lista di bytecode Kitten ed eventualmente legato ad altri blocchi successivi. L'implementazione della funzione γ è ottenuta tramite i seguenti due metodi aggiunti ad `absyn/Expression.java`:

```
protected abstract CodeBlock translate(CodeBlock continuation);

public final CodeBlock translateAs(Type type, CodeBlock continuation) {
    if (staticType == Type.INT && type == Type.FLOAT)
        continuation = new CAST(Type.INT, Type.FLOAT).followedBy(continuation);
    else
        return translate(continuation);
}
```

Il primo implementa $\gamma[_]$ ed è lasciato `abstract`. Esso verrà istanziato nelle sottoclassi di `absyn.Expression` con l'implementazione delle regole in Figura 6.17. Il secondo implementa la funzione $\gamma^r[_]$ dell'Equazione 6.1. Si noti l'uso di `followedBy()` per aggiungere un bytecode in cima a un blocco di codice.

Mostriamo alcuni esempi di istanziazione del metodo `translate()`. In `absyn/True.java` definiamo

```
public final CodeBlock translate(CodeBlock continuation) {
    return new CONST(true).followedBy(continuation);
}
```

che rispecchia fedelmente quanto riportato in Figura 6.17.

In `absyn/Variable.java` definiamo

```
public CodeBlock translate(CodeBlock continuation) {
    return new LOAD(getVarNum(),getStaticType()).followedBy(continuation);
}
```

Il numero della variabile era stato annotato in fase di analisi semantica (Sezione 5.3.1). Utilizziamo anche il tipo τ annotato per questa espressione, accessibile tramite `getStaticType()`. Ancora una volta, questa implementazione riflette la definizione in Figura 6.17.

Dentro `absyn/BinOp.java` definiamo

```
public final CodeBlock translate(CodeBlock continuation) {
    Type ell = getLeft().getStaticType()
        .leastCommonSupertype(getRight().getStaticType());
    return getLeft().translateAs
        (ell,getRight().translateAs
        (ell,operator(ell).followedBy(continuation)));
}
```

```
protected abstract BinOpBytecode operator(Type type);
```

L'idea è di calcolare il minimo sovratipo comune ℓ fra i tipi statici dei due operandi, compilarli entrambi con $\gamma^\ell[[_]]$ e farli quindi seguire da un bytecode binario specifico all'operazione binaria che si sta compilando. Tale bytecode è fornito dal metodo ausiliario `operator()` che è per esempio definito dentro `absyn/Addition.java` come

```
protected BinOpBytecode operator(Type type) {
    return new ADD((NumericalType)type);
}
```

Si noti che questo modo di procedere generalizza le tre ultime regole in Figura 6.17.

6.2.2 La compilazione condizionale delle espressioni booleane

Abbiamo descritto come un'espressione viene tradotta in del bytecode Kitten che ne calcola il valore e lo lascia in cima allo stack degli operandi. Tale codice è adeguato se quello a cui siamo interessati è il valore dell'espressione. Per esempio, di un parametro passato a un metodo abbiamo bisogno del valore, così come del lato destro di un assegnamento. Ci sono casi però in cui quello che ci interessa è di instradare l'esecuzione di un programma in due direzioni diverse sulla base del valore di un'espressione booleana. Per esempio, nel comando `if (exp) then com1 else com2` siamo interessati ad eseguire `com1` se il valore di `exp` è `true` e ad eseguire `com2` se tale valore è invece `false`. Occorre quindi definire un altro modo di generare il bytecode per le espressioni, alternativo a quello della Figura 6.17 e che chiameremo *compilazione condizionale* delle espressioni. Va comunque detto che ricicleremo in larghissima misura le

definizioni in tale figura. Va ricordato inoltre che la compilazione condizionale ha senso solo per le espressioni che hanno tipo `boolean`, dal momento che l'analisi semantica ci garantisce che esse sono le uniche che possono essere usate nei test dei condizionali e dei cicli (Figura 5.11).

Definiamo quindi una funzione

$$\begin{aligned} \gamma^{test} \llbracket _ \rrbracket : \text{absyn.Expression} &\rightarrow \text{translate.CodeBlock} \\ &\mapsto \text{translate.CodeBlock} \mapsto \text{translate.CodeBlock} \end{aligned}$$

che compila un'espressione in maniera condizionale. In particolare, $\gamma^{test} \llbracket exp \rrbracket (\beta_{true}) (\beta_{false})$ è la compilazione condizionale dell'espressione exp : se l'espressione contiene `true` l'esecuzione viene instradata verso la continuazione β_{true} ; altrimenti verso la continuazione β_{false} . La sua definizione sfrutta quella in Figura 6.17:

$$\gamma^{test} \llbracket exp \rrbracket (\beta_{true}) (\beta_{false}) = \gamma \llbracket exp \rrbracket \left(\boxed{\text{nop}} \langle \begin{array}{l} \boxed{\text{if_true}} \rightarrow \beta_{true} \\ \boxed{\text{if_false}} \rightarrow \beta_{false} \end{array} \right) \quad (6.2)$$

Per esempio, supponendo che la variabile `i` sia allocata nella variabile locale numero 1 e che abbia tipo `int`, allora la compilazione condizionale di `i < 5` è

$$\boxed{\begin{array}{l} \text{load 1 of type int} \\ \text{const 5} \\ \text{lt int} \end{array}} \rightarrow \boxed{\text{nop}} \langle \begin{array}{l} \boxed{\text{if_true}} \rightarrow \beta_{true} \\ \boxed{\text{if_false}} \rightarrow \beta_{false} \end{array}$$

La funzione $\gamma^{test} \llbracket _ \rrbracket$ è implementata aggiungendo ad `absyn.Expression` il metodo:

```
public CodeBlock translateAsTest(CodeBlock yes, CodeBlock no) {
    return translate(new CodeBlock(new IF_TRUE(), yes, no));
}
```

Il costruttore utilizzato per questo `CodeBlock` costruisce un blocco con codice `nop` e legato alle continuazioni `yes` e `no` tramite, rispettivamente, il bytecode condizionale `if_true` e il suo opposto.

La definizione di $\gamma^{test} \llbracket _ \rrbracket$ che abbiamo appena visto funziona per qualsiasi espressione condizionale. Genera però del codice particolarmente ridondante. Per esempio, la compilazione condizionale di `i < 5` che abbiamo ottenuto sopra è molto meno ottimizzata di quella in Figura 6.3, che non usa né l'istruzione `nop` né la `lt int` e usa invece i bytecode condizionali `if_cmlt int` ed `if_cmpge int` al posto di `if_true` ed `if_false`. Il problema della `nop` non deve preoccuparci: una volta generato il bytecode per una classe Kitten, elimineremo tutte le `nop` dal codice. Per usare invece dei bytecode condizionali specializzati, possiamo aggiungere delle definizioni specifiche per la funzione $\gamma^{test} \llbracket _ \rrbracket$, che ridefiniscono la precedente definizione generale su dei casi particolari molto frequenti. Per esempio definiamo

$$\gamma^{test} \llbracket \text{LessThan}(left, right) \rrbracket (\beta_{true}) (\beta_{false}) = \gamma^{\ell} \llbracket left \rrbracket \left(\gamma^{\ell} \llbracket right \rrbracket \left(\boxed{\text{nop}} \langle \begin{array}{l} \boxed{\text{if_cmlt}} \rightarrow \beta_{true} \\ \boxed{\text{if_cmpge}} \rightarrow \beta_{false} \end{array} \right) \right)$$

dove ℓ è il minimo sovratipo comune del tipo statico di `left` e `right`. Dal punto di vista implementativo, queste ridefinizioni diventano delle ridefinizioni del metodo `translateAsTest()` in alcune sottoclassi di `absyn.Expression`.

<i>lvalue</i>	<i>bytecode</i>
Variable(<i>name</i>)	$\gamma^{\tau} \llbracket rvalue \rrbracket \left(\boxed{\text{store num of type } \tau} \rightarrow \beta \right)$
FieldAccess(<i>receiver, name</i>)	$\gamma \llbracket receiver \rrbracket \left(\gamma^{\tau} \llbracket rvalue \rrbracket \left(\boxed{\text{putfield field}} \rightarrow \beta \right) \right)$
ArrayAccess(<i>array, index</i>)	$\gamma \llbracket array \rrbracket \left(\gamma \llbracket index \rrbracket \left(\gamma^{\tau} \llbracket rvalue \rrbracket \left(\boxed{\begin{array}{c} \text{arraystore into} \\ \text{array of } \tau \end{array}} \rightarrow \beta \right) \right) \right)$

Figura 6.18: La compilazione passiva di un leftvalue di tipo statico τ .

6.2.3 La compilazione passiva dei leftvalue

I leftvalue sono un caso particolare di espressioni (Sezione 3.2.3). Abbiamo quindi già specificato per essi una modalità di compilazione che lascia il loro valore in cima allo stack degli operandi (Sezione 6.2.1 e Figura 6.17), che usiamo quando del leftvalue ci interessa il valore, come per $a[6]$ in $v := a[6]$, e un'altra modalità che instrada l'esecuzione verso due direzioni diverse sulla base del valore booleano che essi contengono (Sezione 6.2.2 ed Equazione 6.2), che usiamo quando il leftvalue è usato come test booleano, per esempio per $a[8 + v]$ in $\text{if } (a[8 + v]) \text{ then...else...}$. A differenza delle altre espressioni, i leftvalue possono però essere usati anche alla sinistra di un assegnamento, come v in $v := b + c$ oppure $a[5]$ in $a[5] := b * c$. In questi casi non siamo interessati al valore del leftvalue, né a instradare l'esecuzione su due continuazioni diverse sulla base del valore booleano del leftvalue. Vogliamo invece *modificare* il valore del leftvalue. Conseguentemente, dobbiamo definire una terza modalità di compilazione per i leftvalue, che chiameremo *passiva* poiché il leftvalue subisce un assegnamento.

Si consideri un assegnamento del tipo $lvalue := rvalue$. Vogliamo generare il codice che effettua l'assegnamento e poi continua con una continuazione β . Sia τ il tipo statico di $lvalue$. Il bytecode che generiamo è mostrato in Figura 6.18. Essa mostra che la compilazione passiva di un leftvalue è sempre della forma

$$\gamma^{before} \llbracket lvalue \rrbracket (\gamma^{\tau} \llbracket rvalue \rrbracket (\gamma^{after} \llbracket lvalue \rrbracket (\beta)))$$

dove $\gamma^{before} \llbracket - \rrbracket, \gamma^{after} \llbracket - \rrbracket : \text{absyn.Lvalue} \mapsto \text{absyn.CodeBlock} \mapsto \text{absyn.CodeBlock}$ sono due funzioni che aggiungono del codice, rispettivamente, prima e dopo la compilazione di $rvalue$. Si noti che quest'ultimo è compilato rispetto al tipo τ di $lvalue$, in modo da effettuare una promozione di tipo quando $rvalue$ ha tipo `int` e lo si sta assegnando a un $lvalue$ di tipo `float` (Sezione 6.2). Le funzioni γ^{before} e γ^{after} sono implementate aggiungendo a `absyn/Lvalue.java` i due metodi

```
public abstract CodeBlock translateBeforeAssignment(CodeBlock continuation);
```

```
public abstract CodeBlock translateAfterAssignment(CodeBlock continuation);
```

che vengono istanziati nelle sottoclassi in modo da rispettare la Figura 6.18. Per esempio, dentro `absyn/Variable.java` sono ridefiniti come

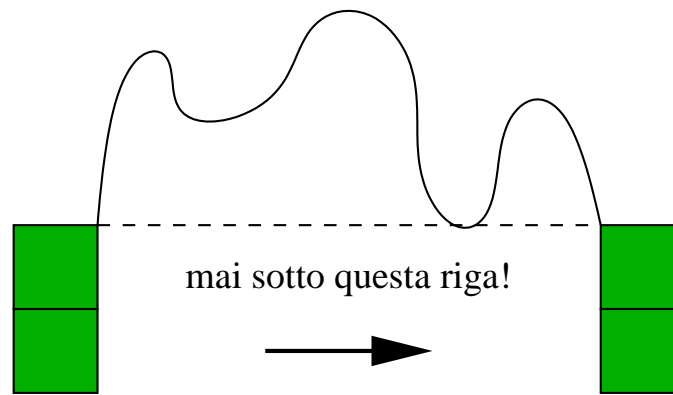


Figura 6.19: L'esecuzione del bytecode Kitten generato per un comando non deve modificare lo stack degli operandi iniziale.

```
public CodeBlock translateBeforeAssignment(CodeBlock continuation) {
    return continuation;
}

public CodeBlock translateAfterAssignment(CodeBlock continuation) {
    return new STORE(getVarNum(),getStaticType()).followedBy(continuation);
}
```

Dentro `absyn/ArrayAccess.java` sono ridefiniti come

```
public CodeBlock translateBeforeAssignment(CodeBlock continuation) {
    return array.translate(index.translate(continuation));
}

public CodeBlock translateAfterAssignment(CodeBlock continuation) {
    return new ARRAYSTORE(getStaticType()).followedBy(continuation);
}
```

Questi due metodi sono usati per compilare il comando di assegnamento, come vedremo nella prossima sezione.

6.3 La generazione del bytecode Kitten per i comandi

La generazione del bytecode per un comando Kitten è formalizzata tramite una funzione $\gamma[[_]] : \text{absyn.Command} \mapsto \text{translate.CodeBlock} \mapsto \text{translate.CodeBlock}$. Dato un comando com e una continuazione β , il codice $\gamma[[com]](\beta)$ dovrà essere del bytecode Kitten che esegue il comando com e poi continua eseguendo la continuazione β . Il codice generato per eseguire i comandi deve essere tale da lasciare intatti i valori iniziali sullo stack degli operandi. Si tratta

$$\begin{aligned}
\gamma[_] : \text{absyn.Command} &\mapsto (\text{translate.CodeBlock} \mapsto \text{translate.CodeBlock}) \\
\gamma[\text{Skip}()](\beta) &= \beta \quad \gamma[\text{LocalScope}(body)](\beta) = \gamma[body](\beta) \\
\gamma[\text{Return}(returned)](\beta) &= \begin{cases} \boxed{\text{return void}} & \text{se } returned = \text{null} \\ \gamma[returned](\boxed{\text{return } \tau}) & \text{se } returned \neq \text{null e ha tipo statico } \tau \end{cases} \\
\gamma[\text{IfThenElse}(condition, then, else)](\beta) &= \gamma^{test}[condition](\gamma[then](\beta))(\gamma[else](\beta)) \\
\gamma[\text{LocalDeclaration}(type, name, initialiser)](\beta) &= \gamma^\tau[initialiser](\boxed{\text{store num of type } \tau} \rightarrow \beta) \\
&\text{dove } \tau \text{ è il tipo semantico di } type \text{ e } num \text{ è il numero progressivo della variabile } name \\
\gamma[\text{MethodCallCommand}(receiver, name, actuals)](\beta) &= \begin{cases} \gamma[receiver](\gamma^\tau[actuals](\boxed{\text{virtualcall method}} \rightarrow \beta)) \\ \text{se } r' = \text{void} \\ \gamma[receiver](\gamma^\tau[actuals](\boxed{\text{virtualcall method}} \\ \text{pop } r' \rightarrow \beta)) \\ \text{altrimenti} \end{cases} \\
&\text{dove } method = \kappa.m(\vec{r}) : r' \text{ è il metodo identificato dall'analisi semantica (Figura 5.11)} \\
\gamma[\text{Assignment}(lvalue, rvalue)](\beta) &= \gamma^{before}[lvalue](\gamma^\tau[rvalue](\gamma^{after}[lvalue](\beta))) \\
&\text{dove } \tau \text{ è il tipo statico di } lvalue \\
\gamma[\text{While}(condition, body)](\beta) &= \underbrace{\boxed{\text{nop}}}_{pivot} \rightarrow \gamma^{test}[condition](\gamma[body](\gamma[pivot](\beta))) \\
\gamma[\text{For}(initialiser, condition, update, body)](\beta) &= \gamma[initialiser](\underbrace{\boxed{\text{nop}}}_{pivot} \rightarrow \gamma^{test}[condition](\gamma[body](\gamma[update](\gamma[pivot](\beta))))(\beta))
\end{aligned}$$

Figura 6.20: La funzione $\gamma[_]$ che genera il bytecode Kitten che esegue i comandi.

esattamente dello stesso vincolo imposto al codice generato per le espressioni nella Sezione 6.2. In tal caso si chiedeva però anche che il valore dell'espressione fosse aggiunto in cima allo stack degli operandi. Dal momento che i comandi non calcolano alcun valore, non esiste per essi tale secondo vincolo. Il comportamento del bytecode generato per i comandi sarà quindi come mostrato in Figura 6.19.

La Figura 6.20 mostra il codice generato per i comandi Kitten. Commentiamo tali regole di compilazione.

Skip(). Questo comando non genera alcun bytecode e quindi la sua compilazione restituisce la continuazione β .

LocalScope(body). L'esecuzione di uno scope locale consiste nell'esecuzione del suo corpo. Conseguentemente, la sua compilazione è, ricorsivamente, la compilazione del suo corpo.

Return(returned). L'istruzione di ritorno da metodo viene tradotta in un bytecode return per il tipo del valore ritornato, se esiste. In tal caso occorre prima compilare l'espressione il cui valore va ritornato. Si noti che la continuazione β è scartata poiché l'esecuzione di un metodo termina col ritorno al chiamante.

IfThenElse(*condition, then, else*). La compilazione del condizionale comincia con la compilazione come test della sua guardia (Sezione 6.2.2). Le due continuazioni della guardia sono, rispettivamente, la compilazione del ramo *then* e del ramo *else* del condizionale, seguite dalla continuazione β del condizionale.

LocalDeclaration(*type, name, initialiser*). La compilazione della dichiarazione di una variabile locale, con inizializzazione, è del codice che valuta l'inizializzatore e ne lascia il valore in cima allo stack, da cui è poi rimosso e scritto dentro alla variabile tramite un bytecode *store*. Si noti che il numero *num* della variabile è stato assegnato al momento dell'analisi semantica.

MethodCallCommand(*receiver, name, actuals*). La compilazione del comando di invocazione di metodo è quasi identica a quella che abbiamo visto per l'espressione di invocazione di metodo (Figura 6.17). La differenza è che qui è possibile invocare anche un metodo che ritorna *void*. Inoltre, dal momento che non dobbiamo modificare lo stack degli operandi (Figura 6.19), rimuoviamo il valore di ritorno di un metodo non *void* tramite un bytecode *pop*.

Assignment(*lvalue, rvalue*). La compilazione di un assegnamento di *rvalue* ad *lvalue* è ottenuta come in Figura 6.18.

While(*condition, body*). Il codice generato per un ciclo *while* è la compilazione condizionale della sua guardia (Sezione 6.2.2), usando come due continuazioni quella stessa del *while*, per il caso in cui la guardia è falsa, e la compilazione del corpo per il caso in cui la guardia è vera. Si noti che la continuazione fornita alla compilazione del corpo è un blocco *pivot* che continua con la compilazione condizionale della guardia stessa, in modo che dopo l'esecuzione del corpo del *while* si passi a valutare di nuovo la guardia del ciclo.

For(*initialiser, condition, update, body*). Il codice generato per un ciclo *for* comincia con il codice che esegue il comando di inizializzazione, seguito da un blocco *pivot* legato alla compilazione condizionale della guardia del *for* (Sezione 6.2.2). Le due continuazioni passate a tale compilazione condizionale sono la continuazione β del *for*, per il caso in cui la guardia è falsa, e la compilazione dell'*update* e del corpo del ciclo per il caso in cui la guardia è vera. Si noti che la continuazione usata per la compilazione del corpo è il *pivot*, in modo che dopo l'esecuzione del corpo del *for* si torni a valutare la guardia del ciclo.

L'implementazione della generazione del codice per i comandi è ottenuta aggiungendo i seguenti metodi ad `absyn/Command.java`:

```
public final CodeBlock translate(CodeBlock continuation) {
    if (next != null) continuation = next.translate(continuation);
    return translate$0(continuation);
}

protected abstract CodeBlock translate$0(CodeBlock continuation);
```

Il primo si occupa del lavoro comune a tutti i comandi, che consiste nel compilare il comando che potrebbe seguire ottenendo la continuazione da passare al metodo `translate$0()`. Quest'ultimo si occupa del lavoro specifico ad ogni comando. Esso implementa le regole in Figura 6.20.

Vediamo alcuni esempi di definizione di `translate$0()` in alcune delle sottoclassi della classe `absyn/Command.java`. Dentro `absyn/LocalScope.java` definiamo

```
protected CodeBlock translate$0(CodeBlock continuation) {
    return body.translate(continuation);
}
```

consistentemente con la Figura 6.20. In `absyn/IfThenElse.java` definiamo

```
protected CodeBlock translate$0(CodeBlock continuation) {
    return condition.translateAsTest
        (then.translate(continuation),else.translate(continuation));
}
```

ancora una volta questo rispecchia la formalizzazione in Figura 6.20.

L'implementazione delle regole per il `while` e il `for` richiedono di creare *prima* il *pivot* in modo da poterlo passare come continuazione, rispettivamente, alla compilazione del *body* o dell'*update* del ciclo. Alla fine si lega il blocco *pivot* con il suo successore, chiudendo il ciclo. Ecco per esempio il generatore di codice inserito dentro `absyn/For.java`:

```
protected CodeBlock translate$0(CodeBlock continuation) {
    CodeBlock pivot = new CodeBlock();
    CodeBlock test = condition.translateAsTest
        (body.translate(update.translate(pivot)),continuation);
    pivot.linkTo(test);
    return initialisation.translate(test);
}
```



Si noti che il blocco *pivot* va creato prima di usarlo come continuazione per la compilazione di *update*, nel caso del comando `for`. Sarebbe sbagliato dichiarare la variabile *pivot* e creare il blocco *pivot* subito prima della chiamata a `linkTo()`: l'*update* si troverebbe con una continuazione pari a `null`!

La generazione del bytecode Kitten per un metodo o costruttore è semplicemente la generazione del bytecode Kitten per il loro corpo, che essendo un comando segue le regole in Figura 6.20. Come continuazione di tale compilazione si usa il blocco

$$\bar{\beta} = \boxed{\text{return void}}.$$

In questo modo abbiamo la garanzia che, nel bytecode che viene generato, ogni percorso di esecuzione all'interno di un metodo che ritorna `void` o all'interno di un costruttore termina sempre con un'istruzione `return void`, anche nei casi in cui il comando `return` è stato lasciato

sottointeso dal programmatore. Si noti che nel caso in cui fossimo dentro un metodo che non ritorna `void` allora tale continuazione verrebbe sistematicamente scartata dalla regola per il comando Kitten `return` in Figura 6.20, dal momento abbiamo la garanzia che, in tal caso, ogni percorso di esecuzione all'interno del metodo termina già con un comando `return` esplicito (Sezione 4.3).

Esercizio 25. Si parta dalla sintassi astratta dell'espressione condizionale definita nell'Esercizio 22 e si scriva la sua funzione γ di compilazione, implementandola poi in Java.

Esercizio 26. Si definisca la sintassi astratta di un comando `do...while` e si dia quindi la sua funzione γ di compilazione, implementandola poi in Java.

Esercizio 27. Si parta dalla sintassi astratta del comando `switch` definito nell'Esercizio 23 e si definisca la sua funzione γ di compilazione, implementandola poi in Java.

Esercizio 28. Quali problemi vedete per definire la compilazione dei comandi `break` e `continue` dell'Esercizio 24? Come pensate di poter modificare lo schema di compilazione per continue in modo da poter compilare tali due comandi?

Bibliografia

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles Techniques and Tools*. Addison Wesley Publishing Company, 1986.
- [2] A. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, 2002.
- [3] A. Goldberg and D. Robson. *Smalltalk 80: The Language*. Addison-Wesley Professional, 1989.
- [4] J. Gosling, B. Joy, Guy Steel, and G. Bracha. *The Java™ Language Specification*. Addison-Wesley, third edition, 2005.
- [5] S. C. Johnson. Yacc - Yet Another Compiler Compiler. Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.
- [6] M. E. Lesk. Lex - A Lexical Analyzer Generator. Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.