

Progetto MiniJava

Corso di Linguaggi di Programmazione

Luca Moschella, Francesca Possenti, Federica Spini e Sara Tramonte

Differenze rispetto Java

La differenza principale rispetto a Java è nel dispatching dinamico dei metodi.

In Java, quando viene chiamato un metodo su un oggetto, non viene eseguito il primo metodo compatibile con il tipo degli argomenti passati che viene trovato, ma il metodo *più specifico* per quegli argomenti.

Nel seguente esempio:

```
class A extends Object
{
    int m(A x)
    {
        return 1;
    }
}

class B extends A
{
    int m(B x)
    {
        return 0;
    }
}

class C extends B
{
    int m( Object x )
    {
        return -1;
    }
}

class esempio extends Object
{
    int main()
    {
        C c = new C();
        return c.m( c );
    }
}
```

- In **Java**: (con piccole modifiche sintattiche: in java il main non può tornare un valore), si ottiene il valore **0**; infatti il metodo **m** più specifico è quello nella classe B.
- In **miniJava**: si ottiene il valore **-1**, in quanto il primo metodo **m** compatibile che si incontra risalendo la gerarchia è quello nella classe C.

Si noti che ciò non ha nulla a che vedere con l'Override, in quanto l'Override impone che i tipi dei parametri siano identici, e non soltanto compatibili. Si tratta infatti di Overloading: in questo esempio, in particolare, avviene nella gerarchia delle classi e non in una singola classe.

Inoltre, in Java, se non è possibile stabilire qual'è il metodo più specifico viene lanciata un'eccezione che comunica l'ambiguità della chiamata al metodo:

```
class esempio extends Object
{
    int m( Object x , esempio x2 )
    {
        return 1;
    }

    int m(  esempio x2, Object x )
    {
        return 1;
    }

    public static void main(String[] args)
    {

        System.out.println(new esempio().m(null, null));
    }
}

<terminated> esempio [Java Application] C:\Program Files\Java\jre1.8.0_91\bin\javaw.exe (01 giu 2016, 21:39:22)
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
    The method m(Object, esempio) is ambiguous for the type esempio

    at esempio.main(esempio.java:16)
```

Tale tipologia di errore non viene ovviamente gestita in miniJava in quanto non può presentarsi (il metodo chiamato sarebbe il primo compatibile trovato, quindi dipendente dall'ordine di apparizione nel codice).

Differenze rispetto le dispense

Durante la realizzazione del progetto abbiamo scelto di seguire le dispense, apportando modifiche al comportamento descritto e aggiungendo alcune funzionalità (specialmente durante il controllo dei tipi).

Comportamento

In questa sezione riportiamo le, poche, differenze nel comportamento rispetto le regole operazionali riportate nelle dispense.

- Il dispatching dei metodi descritto nelle dispense è stato leggermente rivisto, per farlo aderire maggiormente al comportamento di Java in caso di Override. Nel miniJava descritto nelle dispense viene tornato il primo metodo trovato che abbia argomenti compatibili con quelli passati, risalendo la gerarchia (partendo dal tipo dinamico, il tipo dell'oggetto). Tale comportamento può risultare scorretto. Nel seguente esempio:

```

class A extends Object
{
    public int m(Object o)
    {
        return 1;
    }
}
class B extends A
{
    @Override
    public int m(Object o)
    {
        return 2;
    }
}
class C extends B
{
    //NON IN OVERRIDE
    public int m( A x)
    {
        return 3;
    }
}

class esempio extends Object
{
    public static void main(String[] args)
    {
        B b = new C();
        System.out.println(b.m(null));
    }
}

```

<terminated> esempio [Java Application] C:\Program Files\Java\jre
2

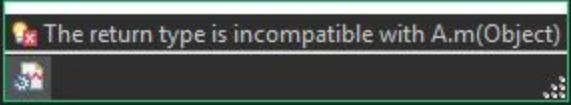
- In **Java**, il valore restituito è **2**, in quanto il metodo **m** della classe **C** non effettua un override.
- Nella versione descritta nelle dispense il valore restituito sarebbe **3**, in quanto il primo metodo compatibile che si incontra è quello della classe **C**. Ciò è un errore in quanto tale metodo non effettua alcun override, e perciò non dovrebbe essere chiamato.
- In **MiniJava**: il valore restituito è **2**. Il problema è stato risolto cercando il metodo che verrebbe chiamato se il tipo dell'oggetto fosse lo stesso del tipo della variabile che lo contiene. In altre parole, nella prima fase di selezione del metodo si considera il tipo statico, cioè il tipo della variabile, e si cerca nella classe corrispondente un metodo che sia compatibile al richiesto. Una volta trovato questo, si considera invece il tipo dinamico dell'oggetto, e si cerca, nella classe ora corrispondente, un metodo con lo stesso nome e gli stessi identici tipi dei parametri, risalendo la gerarchia. Quindi, se si trova un override, si seleziona quest'ultimo, altrimenti la scelta ricade sul metodo precedentemente selezionato.
- Quando viene chiamato un metodo, l'ambiente viene modificato aggiungendo l'associazione fra *this* e l'oggetto su cui viene chiamato il metodo. In questo modo all'interno del metodo è possibile usare la keyword *this* per riferirsi all'oggetto corretto.
- Sono state aggiunte le regole per la valutazione dell'espressione *this*.
- Nelle dispense viene trattato il caso in cui un blocco di comandi non produca un valore a causa dell'assenza di un comando di return. Noi non consideriamo questo caso, in quanto il controllo dei tipi ci assicura la presenza di almeno un comando di return in ogni blocco di comandi.

Controllo dei tipi

In questa sezione riportiamo i principali controlli sui tipi che sono stati aggiunti o modificati rispetto ai controlli presentati nelle dispense, inglobati nelle regole operazionali.

- Viene controllato che i metodi abbiano almeno un “return”, non necessariamente come ultima istruzione. I comandi che seguono il primo return semplicemente non verranno eseguiti.
- Viene controllato che non siano presenti definizioni multiple di classi, campi, metodi, parametri o variabili.
- Viene controllato che un metodo che effettua un Override, in caso cambi il tipo di ritorno definito, lo cambi in un tipo compatibile. Cioè, non deve essere possibile effettuare un operazione di questo tipo:

```
class A extends Object
{
    public int m(Object o)
    {
        return 1;
    }
}
class B extends A
{
    @Override
    public Object m(Object o)
    {
        return 2;
    }
}
```



- Viene controllato che una classe estenda una classe valida, cioè una classe esistente diversa da sé stessa.
- Viene controllato che le variabili utilizzate in un'espressione siano state prima inizializzate.
- Viene controllato che i tipi, in particolare i tipi di classe, siano tipi validi; ovunque essi siano usati.
- Viene controllato che durante l'inizializzazione di un campo non venga utilizzato un altro campo della classe non ancora definito.
- Viene controllato che gli assegnamenti a variabili e campi, i tipi di ritorno dei metodi e le inizializzazioni dei campi abbiano tipo coerenti.
- Viene controllato che le chiamate a metodo e gli accessi a campo avvengano su oggetti, e non su altre espressioni.
- Altri controlli per verificare che le classi, i metodi, i campi e le variabili usate siano definite.

Principio di funzionamento

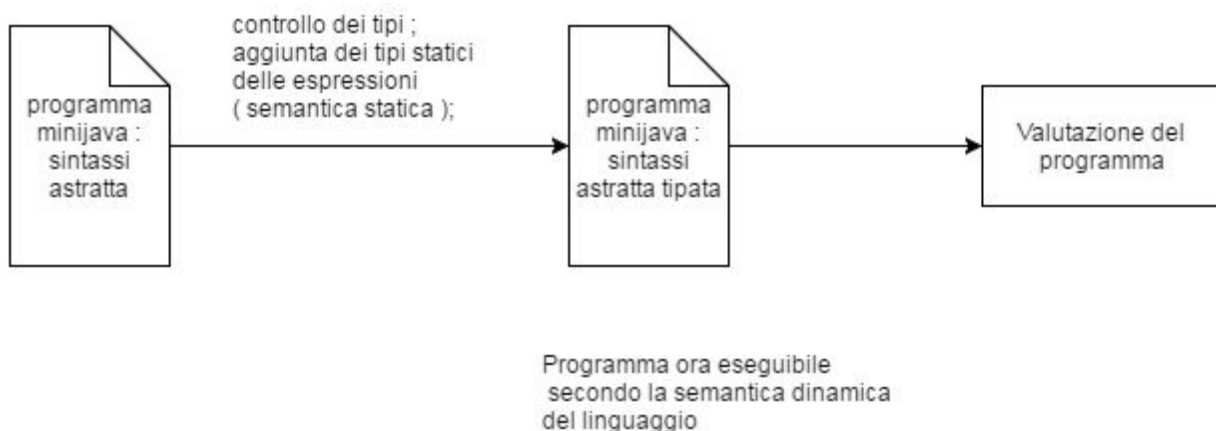
Il sistema miniJava è composto da tre *blocchi* principali: definizione della **sintassi astratta**, traduzione in sintassi astratta tipata (**semantica statica**) e esecuzione della sintassi astratta tipata (**semantica dinamica**).

Oltre queste tre parti ci sono altre parti di rilevanza minore, come quella che definisce le funzioni per tradurre da sintassi astratta (tipata e non) in linguaggio Java, e quella per la gestione delle eccezioni (ovviamente integrata nelle altre parti).

L'esecuzione di un programma MiniJava avviene passando per tre fasi principali, che riprendono i blocchi sopra descritti.

I programmi MiniJava sono scritti rispettando la sintassi astratta definita, ne viene quindi effettuato il controllo dei tipi, durante il quale il codice viene convertito in una nuova sintassi astratta che aggiunge informazioni sui tipi delle espressioni.

Il nuovo codice, scritto ora secondo una sintassi "tipata", viene eseguito ed il risultato dell'esecuzione è il risultato della valutazione dell'intero programma.



Per la gestione dell'ambiente, dell'heap e del contesto, ma anche di altri tipi di dati generici, abbiamo introdotto un datatype *dataList* che rappresenta una lista di coppie generiche, con diversi costruttori a seconda del tipo di oggetto che vogliamo costruire (abbiamo preferito introdurre questo datatype e mantenere separati i vari costruttori per maggiore chiarezza).

Su tale datatype sono state definite diverse funzioni polimorfe, in questo modo abbiamo evitato di riscrivere più volte la stessa funzione, una volta per ogni tipologia di oggetto presente (es. "cercaInEnv", "cercaInHeap", etc).

Sono state definite funzioni polimorfe di utilità anche su liste semplici, utili principalmente per convertire liste in diversi modi.

Materiale

Il file "Main.sml" contiene la definizione della funzione "eval" che deve essere chiamata per valutare un programma. Nei due file "ProgrammiEsempioCorretti" e "ProgrammiEsempioSbagliati" sono disponibili alcuni programmi scritti nella sintassi astratta pronti per essere eseguiti, i nomi sono riportati nel file "Main.sml". Per eseguire un programma è sufficiente dare il comando:

eval(<nome del programma d'esempio>);