Report del Progetto di Compilatori e Interpreti

Giuseppe De Palma 0000854846, Andrew Memma 1900068459

Corso di Laurea Magistrale in Informatica 2017/2018

Indice

1 Introduzione

Il lavoro presentato riguarda un compilatore per un semplice linguaggio di programmazione ad oggetti: FOOL. Il progetto è parte dell'esame del corso "Compilatori e Intepreti" della Magistrale in Informatica, Università di Bologna. Il lavoro è stato svolto da 2 studenti del suddetto corso.

L'obiettivo di tale progetto è volto allo studio dello sviluppo e funzionamento dei compilatori. Partendo dal testo di un programma scritto in FOOL, il compilatore lo analizza sia per ricavarne le informazioni necessarie per generare codice eseguibile, sia per controllarne la correttezza. Si fa uso di diverse tecniche per cercare errori nei programmi (a tempo di compilazione). In particolare, il linguaggio si avvale di un semplice sistema di tipi e il compilatore esegue un controllo di tipi statico che permette di segnalare all'utente errori di tipo. Con questo controllo si ha il vantaggio di evitare i cosiddetti errori *untrapped* e una parte degli errori *trapped*, cioè l'insieme degli errori "proibiti".[1]

FOOL non è l'unico linguaggio utilizzato, infatti il compilatore deve generare del codice eseguibile (da un elaboratore) partendo da un programma scritto in FOOL. Questo nuovo codice è scritto in un linguaggio *assembly* per una *stack virtual machine* con registri (SVM), il quale offre istruzioni per alcuni calcoli aritmentici, per gestione di *stack* e *heap* e salti condizionati. In accordo con le estensioni fatte a FOOL, anche il linguaggio SVM è stato arricchito.

1.1 Outline

Nei capitoli successivi è presentato in modo approfondito il progetto e le sue caratteristiche principali. Nel capitolo successivo viene descritta la struttura del compilatore e come utilizzarlo e una breve analisi sulle diverse parti che lo costituiscono. Il capitolo 3 discute la grammatica del linguaggio e come da questa si ricavino le informazioni per proseguire con la compilazione. Nel capitolo 4 è discussa l'analisi semantica, di particolare importanza è la sezione del *type checking*. Prima di concludere è descritta, nel capitolo 5, la generazione del codice *assembly*.

[1] Distinguiamo due generi di errore in un programma: errori *trapped*, ovvero quelli che provocano il fallimento della computazione, e gli errori *untrapped*, che non presentano sintomi immediatamente visibili, e sono perciò più insidiosi. Al primo tipo appartengono, ad esempio, la divisione per zero o un accesso proibito alla memoria; al secondo la lettura erronea di una porzione non significativa della memoria.

2 Descrizione del Progetto

Il compilatore è stato sviluppato nel linguaggio di programmazione Java, utilizzando il *tool* Antlr nella sua versione 4.6 per la generazione automatica di *lexer* e *parser*. Avendo, quindi, definito la grammatica di FOOL (nel file *FOOL.g4* per essere compatibile con Antlr), il *tool* può così generare le varie classi che permettono di effettuare l'analisi lessicale e sintattica sui programmi scritti in FOOL. In questo modo la creazione di *tokens* e poi creare l'albero astratto di sintassi (*Abstract Syntax Tree*) sono già gestite da Antlr, il quale fornisce due metodi per visitare ed utilizzare l'albero: i *visitor* e *listener design patterns*. Per questo progetto il *visitor pattern* è stato scelto per percorrere l'albero ed ottenere il necessario per effettuare l'analisi semantica e la generazione di codice.

Parlare della struttura del progetto, dei nodi etc.

2.1 Istruzioni Per l'Utilizzo

Si spiega con dettaglio come lanciare e utilizzare il progetto.

3 Sintassi

Punto di inizio dello sviluppo. Si discute FOOL. Si discute l'analisi sintattica, l'AST etc. Anche qui sarà breve.
Si discute anche l'implementazione dell'AST e le altre parti inerenti alla sintassi.

3.1 FOOL

4 Semantica

Si discute brevemente lo scopo dell'analisi semantica (attraversare l'albero di sintassi astratta costruito prima, gestione degli scope e controllo tipi).
Si spiega brevemente le decisioni di design per le symbol tables ed il sistema dei tipi (statico).

4.1 Scope

4.2 Type Checking

È stato sviluppato un sistema di tipi statico per il linguaggio FOOL. Qui di seguito vengono riportate le varie regole di inferenza.

Partendo dalle regole più semplici, FOOL può essere sintetizzato con il linguaggio di espressioni seguente:

$$\begin{array}{lcl} N & ::= & 0 \mid 1 \mid \dots \\ B & ::= & true \mid false \\ T & ::= & int \mid bool \\ E & ::= & N \mid B \mid E + E' \mid E - E' \mid E \times E' \mid E \div E' \mid \\ & & E == E' \mid E < E' \mid E > E' \mid E \leq E' \mid E \geq E' \mid \\ & & E \vee E' \mid E \wedge E' \mid \neg E \end{array}$$

Dove N sono i numeri, B i valori booleani, T il tipo che un'espressione può avere ed E espressioni semplici, aritmetiche e booleane.

4.2.1 Regole di Inferenza

I controlli che il compilatore esegue sulla correttezza delle espressioni usate, sono dati dalle seguenti regole.

1. Num Rule

$$\Gamma \vdash Num : int$$

2. False Rule

$$\Gamma \vdash false : bool$$

3. True Rule

$$\Gamma \vdash true : bool$$

4. Plus Rule

$$\frac{E : int \qquad E' : int}{\Gamma \vdash E + E' : int}$$

5. Sub Rule

$$\frac{E : int \quad E' : int}{\Gamma \vdash E - E' : int}$$

6. Mult Rule

$$\frac{E : int \quad E' : int}{\Gamma \vdash E \times E' : int}[Mul]$$

7. Div Rule

$$\frac{E : int \quad E' : int}{\Gamma \vdash E \div E' : int}$$

8. Equal Rule

$$\frac{E : T \quad E' : T}{\Gamma \vdash E == E' : bool}$$

9. Less Than Rule

$$\frac{E : int \quad E' : int}{\Gamma \vdash E < E' : bool}$$

10. Less Equal Rule

$$\frac{E : int \quad E' : int}{\Gamma \vdash E \leq E' : bool}$$

11. Greater Than Rule

$$\frac{E : int \quad E' : int}{\Gamma \vdash E > E' : bool}$$

12. Greater Equal Rule

$$\frac{E : int \quad E' : int}{\Gamma \vdash E \geq E' : bool}$$

13. Or Rule

$$\frac{E : bool \quad E' : bool}{\Gamma \vdash E \vee E' : bool}$$

14. And Rule

$$\frac{E : bool \quad E' : bool}{\Gamma \vdash E \wedge E' : bool}$$

15. Not Rule

$$\frac{E : bool}{\Gamma \vdash \neg E : bool}$$

4.2.2 Altri comandi

FOOL offre anche comandi di tipo imperativo e orientato agli oggetti, quindi dichiarazioni di variabili, di classi, il costrutto $if\ then\ else$. Il linguaggio mostrato precedentemente viene esteso con questi nuovi comandi.

$$\begin{array}{lll} D & ::= & T\ id = E \mid T\ id \\ F & ::= & \text{let } D \text{ in } P \mid T\ f(D)\ P \\ P & ::= & E \mid id \mid T\ id = E \mid id = E \mid \text{if } E \text{ then } P \text{ else } P' \mid \\ & & \text{class } C \text{ extends } C'(D)\ F \mid \text{new } C() \end{array}$$

Dove "*id*" rappresenta l'utilizzo di una generica variabile che può assumere un valore numerico, booleano o classe. Essa viene "dichiarata" prefissando il tipo alla variabile quando c'è un assegnamento con un'espressione, usando il simbolo =. Non dichiarando il tipo, invece, il comando diventa uno "statement", cioè una assegnazione di una variabile dichiarata in precedenza. Infine, usiamo f, e C per indicare nomi generici che una funzione o una classe possono avere.

1. Var Rule

$$\frac{\Gamma(id) = T}{\Gamma \vdash id : T}$$

5 Generazione del Codice

Capitolo sulla code generation. Si discute come si trasforma il codice scritto nella grammatica ad alto livello di FOOL a codice macchina. Come si gestisce lo stack, i pointers, i frames etc etc.

Magari si può dividere questo capitolo con le due sezioni sotto, dipende da come implementiamo la code generation. Se la facciamo in due fasi passando prima per una generazione ad un codice intermedio e poi al bytecode, allora questo capitolo si scrive con le due sezioni sotto. La generazione intermedia si fa quando si usa anche un interprete, dipende da come vogliamo fare.

5.1 Heap

5.2 Dispatch Table

5.3 SVM

5.4 Virtual Machine

6 Conclusioni

Brevissimo capitolo conclusivo, si fa un piccolo riassunto di tutto il report e si tirano le somme.