FUNJAVA

Montana Nicholas - Davide Belluomo

Corso Linguaggi di Programazione - A.A. 21/22

Contents

1 Architettura

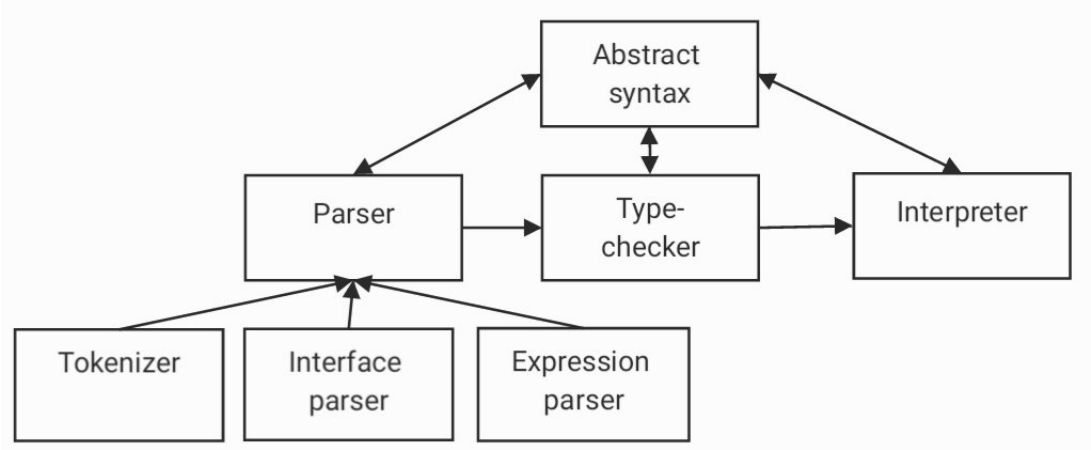

Figure 1: Architettura Software

Il nostro programma e' composto da diverse componenti:

- La Sintassi Astratta del linguaggio FUNJAVA;
- Un parser che, dato in input un programma FUNJAVA, lo traduce in sintassi astratta processabile da SML;
- Il Type-Checker, che esegue il type-checking su un programma FUNJAVA;
- L'interprete, che interpreta i progammi FUNJAVA.

1.1 Abstract Syntax

Il primo passo del nostro progetto e' stato quello di approcciarci ed utilizzare la sintassi astratta di FUNJAVA nel linguaggio SML. Cio' e' stato possibile grazie all'utilizzo del costrutto *datatype* che permette la definizione di un nuovo tipo nel contesto SML. La seguente lista di definizioni costruisce la sintassi astratta del linguaggio.

datatype **Variable** = **Var** of *char*;

datatype **Name** = **N** of *char*;

datatype **Type** = **ClassInterfaceType** of **Name** | **Int** | **Boolean**;

datatype **Expression** = **Cons** of *int* | **BoolCons** of *int* | **VarExp** of **Variable** | **Plus** of **Expression** * **Expression** | **Lambda** of **Variable** *list* * **Expression**

| **Apply** of **Variable** * **Expression** *list*;

datatype **Declaration** = **Interface** of **Name** * **Type** * **Type** *list* * **Variable** *list*;

datatype **Program** = **Prog** of **Declaration** *list* * **Type** * **Variable** * **Expression** * **Expression**;

1.2 Parser

La scelta di implementare un parser deriva dalla necessita' di avere un modo veloce e affidabile di trasformare programmi scritti nella sintassi astratta di FUNJAVA in una sintassi processabile da SML.

Esempio 1.1

```
1  interface I {
2      int m (int x);
3  }
4
5  interface J {
6      int m (I x, I y, int z);
7  }
8
9  class P {
10     public static void main(String[] args) {
11     J w = (x, y, z) -> x.m(y.m(z));
12     System.out.println(w.m((u)-> u+1, (v) -> 41, 8));
13     }
14 }
```

Figure 2: Esempio di programma FUNJAVA

```
Prog
    ([Interface (N #"I",Int,[Int],[Var #"x"]),
      Interface
        (N #"J",Int,
         [ClassInterfaceType (N #"I"),ClassInterfaceType (N #"I"),Int],
         [Var #"x",Var #"y",Var #"z"])],ClassInterfaceType (N #"J"),Var #"w",
     Lambda
       ([Var #"x",Var #"y",Var #"z"],
        Apply (Var #"x",[Apply (Var #"y",[VarExp (Var #"z")])])),
     Apply
       (Var #"w",
        [Lambda ([Var #"u"],Plus (VarExp (Var #"u"),Cons 1)),
         Lambda ([Var #"v"],Cons 41),Cons 8]))
```

Figure 3: Corrispondente programma dopo il parsing del programma in figura 1.1

E' facile pensare come la sintassi in figura 1.2 sia difficile da derivare a mano.

1.2.1 Tokenizzazione

Il primo passo per il parsing e' la tokenizzazzione. Un processo che suddivide il programma dato in input nelle sue singole stringhe e simboli attraverso l'uso di delimitatori, quali lo spazio, per poi generare una sequenza di token. Questi ultimi rappresentano gli elementi chiave del programma necessari al parser per la traduzione.

Esempio 1.2

Il seguente programma
2 + 2
Viene tokenizzato nel seguente modo
TokenCons 2 TokenPlus TokenCons 2

Come si puo' notare, ad ogni simbolo del programma viene associato un token distinto con un determinato significato. In questo caso il parser riconoscera' la presenza del token *TokenPlus* per dare in output l'espressione *Plus(Cons 2, Cons 2)* seguendo le regole della sintassi da noi definita.

La funzione principale *parse* riceve in input il risultato della tokenizzazione e attraverso la chiamata sequenziale a varie funzioni genera in output il risultato finale del processo di parsing. Ognuna di queste funzioni ha il compito di lavorare su una determinata parte del programma.

1.2.2 Interface parser

Sicuramente il parsing della definizione delle interfacce funzionali del programma e' fondamentale. Lo scopo di tale sezione e' proprio quella di riconoscere correttamente le varie interfacce definite.
In particolare per ogni interfaccia la funzione ricorsiva *parse_method* analizza determinati token per riconoscere il nome dell'interfaccia e il suo metodo. Di quest'ultimo e' infatti di importante identificare il tipo di ritorno e i vari parametri con i rispettivi tipi.

1.2.3 Expression parser

Il corretto riconoscimento delle due espressioni che compongono il programma e' un'ulteriore passo del processo di parsing.
La funzione ricorsiva *parse_expression* viene chiamata infatti due volte e sfrutta altre funzioni quali *parse_lambda*, *parse_apply* e *parse_atomic* per identificare senza errori le espressioni presenti.
Da notare come sia nel caso di una lambda espressione, sia di una applicazione di metodo di un' interfaccia che di una somma non si e' gia' a conoscenza del numero di parametri; cio' comporta che ogni funzione sia ricorsiva e che sfrutti la presenza di token come *RPAREN*(che identifica la presenza di una chiusura di parentesi ")") per riconoscere l'inizio e la fine delle espressioni.

1.3 Interpreter

L'interprete e' quella parte del progetto che si occupa di interpretare i programmi, ovvero di determinare quale e' il loro "valore". L'approccio adottato si basa sulla definizione di semantica operazionale **eager statica** vista a lezione. Vengono infatti sfruttate sia la nozione di **ambiente** che la **definizione induttiva** della sintassi astratta per la valutazione.
Esso consiste della funzione induttiva *eval_ exp* che essenzialmente implementa le regole di valutazione della semantica operazionale stessa, ovvero:

- $E \vdash k \rightsquigarrow k$
- $E \vdash x \rightsquigarrow v$ (se $E(x) = v$)
- $\dfrac{E \vdash e1 \rightsquigarrow v1 \quad E \vdash e2 \rightsquigarrow v2}{E \vdash e1 + e2 \rightsquigarrow v}$ $(v1 + v2 = v)$
- $E \vdash (x_1, ..., x_n) \rightarrow M \rightsquigarrow (< x_1, ..., x_n >, M, E)$
- $\dfrac{E \vdash x \rightsquigarrow (< x_1, .., x_n >, M, E') \quad E \vdash M_1 \rightsquigarrow v_1 .. E \vdash M_n \rightsquigarrow M_n \quad E'(x_1, v_1)..(x_n, v_n) \vdash M \rightsquigarrow v}{E \vdash x.m(M_1, ..., M_n) \rightsquigarrow v}$

L'ambiente nel programma e' implementato tramite liste di coppie della forma: $[(x_1, v_1), ..., (x_n, v_n)]$.
Nella progettazione dell'interprete ci siamo scontrati con due problemi aventi la stessa origine. Le funzioni in SML non possono tornare valori di tipo diverso e le liste fornite da SML non sono eterogenee (non possono contenere elementi di tipo diverso).
Dato che la valutazione di una lambda espressione tornava una chiusura (una coppia di tipo (char list) * Expression * (char * value)) e tutte le altre valutazioni tornavano un termine di tipo Expression questo creava un conflitto di tipi.
Abbiamo dovuto quindi astrarre il concetto di **valore** creando un datatype che rappresenta tutti i possibili valori che una espressione puo restituire di modo che questo problema fosse risolto.

Infine la valutazione dell'intero programma consiste nel valutare sequenzialmente le due espressioni che lo compongono, ovvero l'espressione nell'eventuale dichiarazione del main, e l'espressione presente nella funzione *System.out.println()*.
Durante la valutazione delle espressioni se non si riesce a trovare una variabile nell'ambiente, l'interprete alza l'eccezione *UnboundVariable* che sta ad indicare che la variabile acceduta in una parte di codice non e' stata mai dichiarata prima dell'accesso.

1.4 Type-checker

Il type-checker e' quella componente del progetto che si occupa di "tipare" il programma FUNJAVA dato in input. Nel fare questo il type-checker controlla che tutti i tipi dichiarati nel programma siano concordi e che non ci siano *TypeMismatch*. I passi presi dal type-checker sono i seguenti:

1. Si estrae il tipo della prima espressione nella eventuale dichiarazione del main.
2. Si aumenta il contesto della nuova informazione ottenuta nel passo precedente.
3. Si estrae il tipo della seconda espressione (quella nella chiamata *System.out.println()*) nel contesto ottenuto al passo precedente
4. Si controlla se il tipo estratto dalla prima espressione era concorde con il tipo dichiarato nel programma FUNJAVA.
5. Se si, allora si ritorna il tipo della seconda espressione, altrimenti il typechecker alza l'eccezione *TypeMismatch*.

Il type-checking avviene tutto nella funzione ricorsiva *typecheck_ exp*, che ancora una volta sfrutta la struttura induttiva della sintassi astratta. Essa si basa sulle seguenti regole di inferenza:

- $\Gamma \vdash k : K$ (dove K e' il tipo corrispondente alla costante k.)
- $\Gamma(x, T) \vdash x : T$
- $\Gamma \vdash M + N : Int$
- $\dfrac{\Gamma(x_1 : I_{tp1}) \ldots (x_n : I_{tpn}) \vdash M : I_{ret}}{\Gamma \vdash (x_1, ..., x_n) \to M : I}$ ($se\ I\ tipo\ dichiarato\ per\ la\ lambda\ e\ n = I_n$)
- $\dfrac{\Gamma \vdash x : I \quad \Gamma \vdash M_1 : I_{tp1} \ldots \Gamma \vdash M_n : I_{tpn}}{\Gamma \vdash x.m(M_1, ...M_n) : T}$ ($se\ T = I_{ret}$)

Dove, per una generica interfaccia I abbiamo:

- I_n =numero di parametri dichiarati;
- I_{tpi} = tipo del parametro i esimo;
- I_{ret} = tipo di ritorno.