Progetto di Linguaggi e Compilatori

Edoardo Annunziata, Desire Akpo

26 luglio 2017

Per semplicità, nella presente relazione chiameremo il linguaggio implementato *Mini Eiffel*, abbreviato in *ME*.

1 Lessico

L'alfabeto di *ME* è costituito dai 95 caratteri stampabili. Un identificatore `<id>` è una stringa costituita da qualsiasi combinazione di lettere, cifre e istanze dei simboli `'` e `_` che inizi con una lettera, non sia virgolettata, e non sia una parola riservata.
Una stringa letterale `<string>` è una stringa della forma `"S"`, dove `S` è una qualsiasi sequenza di simboli tranne `"`, a meno che non sia preceduto da \
Un intero letterale `<int>` è una stringa non vuota di cifre.
Un carattere letterale `<char>` è una stringa della forma `'C'`, dove `C` è un qualsiasi simbolo.
Un reale letterale `<double>` è una stringa costituita da due stringhe non vuote di cifre separate dal simbolo `.` opzionalmente seguita dal simbolo `e` e da una stringa non vuota di cifre, opzionalmente preceduta dal simbolo `-`
Un booleano letterale `<bool>` è una stringa che è esattamente una delle seguenti due: `true`, `false`.
Le parole riservate di *ME* sono le seguenti:

and	array	bool	char
do	double	else	end
false	from	if	int
loop	name	or	ref
readDouble	readInt	readString	return
string	then	true	until
valres	value	writeDouble	writeInt
writeString			

2 Sintassi

Una espressione di *ME* è costituita da un qualsiasi numero di letterali, identificatori o chiamate a funzioni, combinate mediante vari operatori:

```
<exp>             →  <exp> or <exp>
                  |  <exp> and <exp>
                  |  <exp> == <exp>
                  |  <exp> > <exp>
                  |  <exp> < <exp>
                  |  <exp> + <exp>
                  |  <exp> - <exp>
                  |  <exp> * <exp>
                  |  <exp> / <exp>
                  |  !  <exp>
                  |  * <exp>
                  |  & <exp>
                  |  <assignable-exp>
<assignable-exp>  →  <assignable-exp>[<exp>]
                  |  <id>([<exp>])
                  |  <id> | <string> | <int>
                  |  <double> | <char> | <bool>
                  |  (<exp>)
```

Gli operatori sono riportati nella lista precedente in ordine crescente di precedenza. Osserviamo che le parentesi, producendo un'espressione di precedenza massima, modificano l'ordine di precedenza degli operatori.
ME prevede annotazioni di tipo esplicite. Obbediscono alle seguenti produzioni:

```
<type>  →  int | double | bool | string | char
        |  * <type>
        |  array [<type>]
```

ME prevede inoltre la possibilità di dichiarare la modalità di passaggio dei parametri alle funzioni.

```
<pdecl>  →  <pmet> :  <type>
<pmet>   →  value <ident> | valres <ident>
         |   name <ident> | ref <ident> | <ident>
```

Sono previsti i seguenti statement:

```
<stm>  →  <id> :  <type>
       |  <exp>
       |  <assignable-exp> := <exp>
       |  <f-decl
       |  if <exp> then [<stm>] else [<stm>]
       |  if <exp> then [<stm>]
       |  do [<stm>] end
       |  from <ass> until <exp> loop [<stm>] end
       |  until <exp> loop [<stm>] end
       |  writeInt(<expr>)
       |  writeDouble(<expr>)
       |  writeString(<expr>)
```

Lo statement di dichiarazione di una funzione ha la sintassi seguente:

`<f-decl>` → `$<ident> ( [<pdecl>] ) : <type> do [<stm>] end`

Infine, definiamo un programma *ME* come una successione finita di *statement*:

`<program>` → `[<statement>]`

L'unica decisione che devia dalla specifica è l'introduzione del glifo "$" a precedere la dichiarazione di funzioni. La decisione è motivata dal fatto che, a differenza dei linguaggi della famiglia del *C*, la sintassi di *Eiffel* postpone l'annotazione del tipo della funzione fino a dopo il termine della dichiarazione dei parametri formali. In un parser $LALR(1)$, questo causa un conflitto reduce/reduce con la produzione che identifica la chiamata di una funzione. È immediato verificare che l'accorgimento adottato elimina l'ambiguità.

3 Tipi

Per prima cosa definiamo l'insieme di tipi usato da *ME*. Sia:

$$T_0 = \{\texttt{int}, \texttt{double}, \texttt{string}, \texttt{char}, \texttt{bool}\}$$

Inoltre sia T_1 l'insieme minimale per inclusione tale che $T_0 \subseteq T_1$ e che $\forall \tau \in T_1.((*\tau \in T_1) \wedge (\texttt{array}[\tau] \in T_1))$. Definiamo quindi $T_2 = \{\&\tau \mid \tau \in T_1\}$. Sia infine $T = T_1 \cup T_2$. L'insieme dei tipi di *ME* è T.
Definiamo inoltre l'insieme delle segnature di funzione $F = \{(\tau, P) \mid t \in T, P \in T^*\}$.
Sia Γ_0 un contesto. Definiamo un *ambiente* Δ come una pila di contesti; formalmente,

- $\varnothing$ è un ambiente.
- Se Δ è un ambiente e Γ_0 è un contesto, allora $\Delta.\Gamma_0$ è un ambiente.

Sia $\Delta = \Gamma_n.\Gamma_{n-1} \ldots \Gamma_1.\Gamma_0$ un ambiente, x una variabile, k il minimo intero per cui x è in Γ_k, allora:

$$\frac{}{\Delta \vdash x : \tau} \text{ se } x : \tau \text{ in } \Gamma_k$$

Introduciamo ora le regole per i tipi delle espressioni. Nelle seguenti, Γ rappresenta un contesto.

Regola 1. Tipi Elementari

$$\frac{}{\Gamma \vdash x : \texttt{int}} \text{ se } x \text{ è un intero letterale}$$

Regole analoghe per gli altri tipi elementari.

Regola 2. Operatori booleani

$$\frac{\Gamma \vdash x : \texttt{bool} \qquad \Gamma \vdash y : \texttt{bool}}{\Gamma \vdash x \ \texttt{or} \ y : \texttt{bool}}$$

Regole analoghe per `or`.

Regola 3. Operatori di confronto

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash y : \tau}{\Gamma \vdash x \texttt{ == } y : \texttt{bool}}$$

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash y : \sigma}{\Gamma \vdash x \texttt{ == } y : \texttt{bool}} \text{ se } \exists \tau' \exists \sigma'.(\tau = *\tau' \vee \tau = \&\tau') \wedge (\sigma = *\sigma' \vee \sigma = \&\sigma')$$

Regole analoghe per `>` e `<`.

Regola 4. Somma

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash y : \sigma}{\Gamma \vdash x\texttt{+}y : \max(\tau, \sigma)} \text{ se } \tau, \sigma \in \{\texttt{int}, \texttt{double}, \texttt{string}\}$$

Dove $\texttt{int} \preceq \texttt{double} \preceq \texttt{string}$

Regola analoga per `-`.

Regola 5. Moltiplicazione

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash y : \sigma}{\Gamma \vdash x\texttt{*}y : \max(\tau, \sigma)} \text{ se } \tau, \sigma \in \{\texttt{int}, \texttt{double}\}$$

Dove $\texttt{int} \preceq \texttt{double}$

Regola analoga per `/`.

Regola 6. Negazione logica

$$\frac{\Gamma \vdash x : \texttt{bool}}{\Gamma \vdash \texttt{!}x : \texttt{bool}}$$

Regola 7. Dereferenziazione

$$\frac{\Gamma \vdash x : *\tau}{\Gamma \vdash *x : \tau}$$

Regola 8. Referenziazione

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash \&x : \&\tau} \text{ se } \tau \in T_1$$

Regola 9. Array

$$\frac{\Gamma \vdash x : \texttt{array}[\tau] \qquad \Gamma \vdash y : \texttt{int}}{\Gamma \vdash x[y] : \tau}$$

Introduciamo ora regole per dedurre la validità di uno statement rispetto ai tipi. Se s è uno statement, abbrevieremo tale nozione dicendo "s valido". Nelle seguenti, Δ indica un environment, Γ indica un contesto, S indica una lista di statement.

Regola 10. Espressioni

$$\frac{\exists \tau \in T(\Delta \vdash x : \tau)}{\Delta \vdash x \text{ valido}}$$

Regola 11. Assegnazioni

$$\frac{\Delta \vdash e : \tau \qquad \Delta \vdash S \text{ validi}}{\Delta \vdash (x\texttt{:=}e\texttt{;}S) \text{ valido}}\ x : \tau \text{ in } \Delta$$

$$\frac{\Delta \vdash e : *\tau \qquad \Delta \vdash S \text{ validi}}{\Delta \vdash (x\texttt{:=}e\texttt{;}S) \text{ valido}}\ x : \&\tau \text{ in } \Delta$$

Regola 12. Dichiarazione di variabili

$$\frac{\Delta.(\Gamma, x : \tau) \vdash S \text{ validi}}{\Delta.\Gamma \vdash (x\texttt{:}\tau\texttt{;}S) \text{ valido}}\ x \text{ non in } \Gamma$$

Regola 13. Blocchi

$$\frac{\Delta.\varnothing \vdash S_1 \text{ validi} \qquad \Delta \vdash S_2 \text{ validi}}{\Delta \vdash (\texttt{do}\ \ S_1\ \ \texttt{end}\ \ S_2) \text{ valido}}$$

Regola 14. Iterazione e condizionali

$$\frac{\Delta \vdash x : \texttt{bool} \qquad \Delta \vdash S \text{ validi}}{\Delta \vdash (\texttt{if}\ \ x\ \ \texttt{then}\ \ S\ \ \texttt{end}) \text{ valido}}$$

$$\frac{\Delta \vdash x : \texttt{bool} \qquad \Delta \vdash S_1 \text{ validi} \qquad \Delta \vdash S_2 \text{ validi}}{\Delta \vdash (\texttt{if}\ \ x\ \ \texttt{then}\ \ S_1\ \ \texttt{else}\ \ S_2\ \ \texttt{end}) \text{ valido}}$$

$$\frac{\Delta \vdash x : \texttt{bool} \qquad \Delta \vdash S \text{ validi}}{\Delta \vdash (\texttt{until}\ \ x\ \ \texttt{loop}\ \ S\ \ \texttt{end}) \text{ valido}}$$

$$\frac{\Delta \vdash x : \texttt{bool} \qquad \Delta \vdash s \text{ valido} \qquad \Delta \vdash S \text{ validi}}{\Delta \vdash (\texttt{from}\ \ s\ \ \texttt{until}\ \ x\ \ \texttt{loop}\ \ S\ \ \texttt{end}) \text{ valido}}$$

Regola 15. Dichiarazione di funzioni

$$\frac{\Delta.(\Gamma, f : (\tau_1 \dots \tau_n) \to \tau_r)).(\varnothing, x_1 : \tau_1 \dots x_n : \tau_n, f : (\tau_1 \dots \tau_n) \to \tau_r) \vdash S \text{ v.}}{\Delta.\Gamma \vdash (f(x_1 : \tau_1 \dots x_n : \tau_n) : \tau_r\ \ \texttt{do}\ \ S\ \ \texttt{end}) \text{ valido}}\ \phi$$

Con $\phi =$ "f non in Γ, $\{x_i\}$ a coppie distinti".

Regola 16. Applicazione di funzioni

$$\frac{\Gamma \vdash f : (\tau_1 \dots \tau_n) \to \tau_r \qquad \Gamma \vdash x_1 : \tau_1 \qquad \dots \qquad \Gamma \vdash x_n : \tau_n}{\Gamma \vdash f(x_1 \dots x_n) : \tau_r}$$

Regola 17. Funzioni builtin

$$\frac{\Delta \vdash x : \texttt{int} \qquad \Delta S \text{ validi}}{\Delta \vdash (\texttt{writeInt(}x\texttt{)}\texttt{;}S) \text{ valido}}$$

Regole analoghe per `writeDouble` e `writeString`.

4 Implementazione

Il lexer ed il parser sono stati generati automaticamente a partire dalla grammatica con BNFC.
Il tool di pretty printing compie una visita dell'albero della sintassi astratta, memorizzando un campo addizionale di chiave intera per rappresentare il numero di livelli di indentazione.
La realizzazione del type checker pone qualche questione di natura implementativa la cui soluzione non è banale. In primo luogo, dal momento che come da specifica il linguaggio supporta dichiarazioni di funzioni e variabili ad ogni livello di annidamento, è necessario affrontare il problema della visibilità delle dichiarazioni al variare del blocco attivo. Le regole di tipo che abbiamo definito suggeriscono di rappresentare i dati come una pila di contesti, devono quindi essere definite opportune funzioni per gestire i cambi di contesto. Le associazioni identificatore/tipo all'interno del singolo contesto sono gestite da una tabella di hash. In secondo luogo, non è immediato uniformare l'eterogeneità tra gli identificatori di variabili e di funzioni, se per i primi è sufficiente memorizzare un tipo, per i secondi è necessaria l'intera segnatura. La soluzione che abbiamo implementato consiste nella definizione di un tipo di dato ad-hoc, che possa rappresentare sia un singolo tipo, sia la segnatura di una funzione. I controlli di consistenza sono delegati a funzioni ausiliarie. Il cuore del type checker è costituito da due funzioni, che nel codice hanno nome `inferExp` e `checkStm`. Come i nomi stessi lasciano intuire, la prima determina il tipo di un'espressione, mentre la seconda determina se lo statement che riceve per argomento sia valido o meno. Il resto del codice è composto da funzioni che svolgono bookkeeping di varia natura, come tenere aggiornato lo stato o gestire gli errori.