Specification on New Features of Decaf

朱俸民

September 28, 2018

1 Syntax

In the following EBNF, $sep(N, t)$ denotes a (possibly empty) list of non-terminal N separated by a token t. Optional parts are surrounded with $\langle\ \rangle$. As a convention, we use **bold** font for keywords and operators/delimiters, `monospace` font for (other) terminals, and sans serif font for non-terminals.

ClassDef ::= ⟨**sealed**⟩ **class** `identifier` ⟨**extends** `identifier`⟩ **{** Field* **}**
Stmt ::= ⋯ | OCStmt **;** | GuardedStmt | ForeachStmt
OCStmt ::= **scopy (** `identifier` **,** Expr **)**
GuardedStmt ::= **if {** *sep*(Guard, **|||**) **}**
Guard ::= Expr **:** Stmt
ForeachStmt ::= **foreach (** BoundVariable **in** Expr ⟨**while** BoolExpr⟩ **)** Stmt
BoundVariable ::= Type `identifier` | **var** `identifier`
LValue ::= ⋯ | **var** `identifier`
Expr ::= ⋯ | Expr **%%** Expr | Expr **++** Expr | Expr **[** Expr **:** Expr **]**
| Expr **[** Expr **]** **default** Expr
| **[** Expr **for** `identifier` **in** Expr ⟨**if** BoolExpr⟩ **]**
Constant ::= ⋯ | ArrayConstant
ArrayConstant ::= **[** *sep*(Constant, **,**) **]**

The operator precedence (the smaller number for the higher precedence) and associativity are shown in the following table:

1	**[default .**	
2	**! -**	
3	*** / %**	left associative
4	**+ -**	left associative
5	**< <= > >=**	not associative
6	**%%**	left associative
7	**++**	right associative
8	**== !=**	left associative
9	**&&**	left associative
10	**||**	left associative

2 Type Checking

Recall that in Decaf, types can be declared syntactically:

Type ::= **int** | **bool** | **string** | **void** | **class** `identifier` | Type **[ ]**

Semantically, *types* are:

$$\text{Type } T ::= \text{int} \mid \text{bool} \mid \text{string} \mid \text{void} \mid A \mid T[]$$

where A is the name for a class. Any object/instance of the class A has type A. Note that void is a special type that is only used for a statement or a method without return value. In this document, we distinguish types (semantically) and the types declared syntactically. We let $type(t)$ be the type (semantically) of the one t declared syntactically, i.e.

$$\begin{aligned} type(\texttt{int}) &= \text{int} \\ type(\texttt{bool}) &= \text{bool} \\ type(\texttt{string}) &= \text{string} \\ type(\texttt{void}) &= \text{void} \\ type(\texttt{class } A) &= A \\ type(t\ \texttt{[ ]}) &= type(t)[] \end{aligned}$$

2.1 Typing Rules for Expressions

Let $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ be a *type environment*, in which the variable (syntactically, an identifier) x_i has type T_i (for $1 \le i \le n$). Let $\Gamma, x : T$ denote the updated type environment by letting x have type T. In case x occurs in Γ, the original type is shadowed and x now has type T. Let $\Gamma \vdash E : T$ denote that in Γ, expression E has type T, and we say that E is *well-typed* under Γ. The typing rules are:

$$\text{T-Array-Repeat}\ \frac{\Gamma \vdash E : T \qquad \Gamma \vdash E_1 : \text{int}}{\Gamma \vdash E\ \texttt{\%\%}\ E_1 : T[]}$$

$$\text{T-Array-Concat}\ \frac{\Gamma \vdash E_1 : T[] \qquad \Gamma \vdash E_2 : T[]}{\Gamma \vdash E_1\ \texttt{++}\ E_2 : T[]}$$

$$\text{T-Array-Range}\ \frac{\Gamma \vdash E : T[] \qquad \Gamma \vdash E_1 : \text{int} \qquad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E\ \texttt{[}\ E_1\ \texttt{:}\ E_2\ \texttt{]} : T[]}$$

$$\text{T-Array-Access}\ \frac{\Gamma \vdash E : T[] \qquad \Gamma \vdash E_1 : \text{int} \qquad \Gamma \vdash E' : T}{\Gamma \vdash E\ \texttt{[}\ E_1\ \texttt{] default}\ E' : T}$$

$$\text{T-Array-Comp}\ \frac{\Gamma \vdash E : T[] \qquad \Gamma, x : T \vdash E' : T' \qquad \Gamma, x : T \vdash B : \text{bool}}{\Gamma \vdash \texttt{[}\ E'\ \texttt{for}\ x\ \texttt{in}\ E\ \texttt{if}\ B\ \texttt{]} : T'[]}$$

$$\text{T-Array-Const}\ \frac{\forall i : 0 \le i \le n, \Gamma \vdash C_i : T}{\Gamma \vdash \texttt{[}\ C_0\texttt{,}\ \ldots\texttt{,}C_n\ \texttt{]} : T[]}$$

2.2 Typing Rules for Statements

Notation $\Gamma \vdash S : \text{void}$ means that statement S is *well-typed* under Γ. The typing rules are:

$$\text{T-OCStmt}\ \frac{\Gamma \vdash x : A \qquad \Gamma \vdash E : A}{\Gamma \vdash \texttt{scopy (}\ x\ \texttt{,}\ E\ \texttt{) ;} : \text{void}}$$

$$\text{T-GuardedStmt}\ \frac{\forall i : 1 \le i \le n, \Gamma \vdash E_i : \text{bool} \wedge \Gamma \vdash S_i : \text{void}}{\Gamma \vdash \texttt{if \{}\ E_1\ \texttt{:}\ S_1\ \texttt{|||}\ \cdots\ \texttt{|||}\ E_n\ \texttt{:}\ S_n\ \texttt{\}} : \text{void}}$$

$$\text{T-ForeachStmt-V}\ \frac{\Gamma \vdash E : T[] \qquad \Gamma, x : T \vdash B : \text{bool} \qquad \Gamma, x : T \vdash S : \text{void}}{\Gamma \vdash \texttt{foreach (}\ \texttt{var}\ x\ \texttt{in}\ E\ \texttt{while}\ B\ \texttt{)}\ S : \text{void}}$$

$$\text{T-ForeachStmt-T}\ \frac{\begin{array}{cc} \Gamma \vdash E : T[] & T_1 = type(t) \\ \Gamma, x : T_1 \vdash B : \text{bool} \qquad \Gamma, x : T_1 \vdash S : \text{void} & T <: T_1 \end{array}}{\Gamma \vdash \texttt{foreach (}\ t\ x\ \texttt{in}\ E\ \texttt{while}\ B\ \texttt{)}\ S : \text{void}}$$

where $T_1 <: T_2$ denotes that type T_1 is a *subtype* of T_2. The subtype relation is formulated by the following four rules:

- (reflexivity) for any type T, $T <: T$;
- (transitivity) if $T_1 <: T_2$ and $T_2 <: T_3$, then $T_1 <: T_3$;
- (class inheritance) if class A extends class B, then $A <: B$;
- (array covariance) if $T_1 <: T_2$, then $T_1[] <: T_2[]$.

3 Semantics

Semantics are defined for well-typed expressions and statements (under their current typing environment). Let $e :: E$ denote a nonempty array $E' = [e_0, e_1, \ldots, e_n]$ where expression $e = e_0$ and array expression $E = [e_1, \ldots, e_n]$. In case E' has only one element, say $E' = [e_0]$, then E is empty, denoted by $E = []$.

3.1 Evaluating Expressions

Let $\sigma = \{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$ be a *value environment*, in which the variable x_i is bound to value v_i. Let $\sigma[x \mapsto v]$ denote a updated value environment in which the value of x has now been updated to v. Semantic function $[\![E]\!]\,\sigma$ returns the value of evaluating expression E in σ. The evaluation rules are:

$$\text{E-Array-Repeat}\ \frac{[\![E]\!]\,\sigma = v \qquad [\![E_1]\!]\,\sigma = n}{[\![E\ \texttt{\%\%}\ E_1]\!]\,\sigma = [\underbrace{v, v, \ldots, v}_{n}]}$$

$$\text{E-Array-Concat}\ \frac{[\![E_1]\!]\,\sigma = [v_0, \ldots, v_n] \qquad [\![E_2]\!]\,\sigma = [v'_0, \ldots, v'_m]}{[\![E_1\ \texttt{++}\ E_2]\!]\,\sigma = [v_0, \ldots, v_n, v'_0, \ldots, v'_m]}$$

$$\text{E-Array-Range}\ \frac{\begin{array}{c}[\![E]\!]\,\sigma = [v_0, \ldots, v_n] \qquad [\![E_1]\!]\,\sigma = n_1 \qquad [\![E_2]\!]\,\sigma = n_2 \\ n'_1 = \max\{0, n_1\} \qquad n'_2 = \min\{n, n_2\} \qquad n'_1 \le n'_2\end{array}}{[\![E\ \texttt{[}\ E_1\ \texttt{:}\ E_2\ \texttt{]}]\!]\,\sigma = [v_{n'_1}, \ldots, v_{n'_2}]}$$

$$\text{E-Array-Range-Empty}\ \frac{\begin{array}{c}[\![E]\!]\,\sigma = [v_0, \ldots, v_n] \qquad [\![E_1]\!]\,\sigma = n_1 \qquad [\![E_2]\!]\,\sigma = n_2 \\ n'_1 = \max\{0, n_1\} \qquad n'_2 = \min\{n, n_2\} \qquad n'_1 > n'_2\end{array}}{[\![E\ \texttt{[}\ E_1\ \texttt{:}\ E_2\ \texttt{]}]\!]\,\sigma = []}$$

$$\text{E-Array-Access}\ \frac{[\![E]\!]\,\sigma = [v_0, \ldots, v_n] \qquad [\![E_1]\!]\,\sigma = i \qquad 0 \le i \le n}{[\![E\ \texttt{[}\ E_1\ \texttt{]}\ \texttt{default}\ E']\!]\,\sigma = v_i}$$

$$\text{E-Array-Access-Default}\ \frac{[\![E]\!]\,\sigma = [v_0, \ldots, v_n] \qquad [\![E']\!]\,\sigma = v' \qquad [\![E_1]\!]\,\sigma = i \qquad i < 0 \vee i > n}{[\![E\ \texttt{[}\ E_1\ \texttt{]}\ \texttt{default}\ E']\!]\,\sigma = v'}$$

$$\text{E-Array-Comp-Empty}\ \frac{}{[\![\texttt{[}\ E'\ \texttt{for}\ x\ \texttt{in}\ []\ \texttt{if}\ B\ \texttt{]}]\!]\,\sigma = []}$$

$$\text{E-Array-Comp-True}\ \frac{[\![e]\!]\,\sigma = v \qquad [\![B]\!]\,\sigma[x \mapsto v] = \text{true} \qquad [\![E']\!]\,\sigma[x \mapsto v] = v'}{[\![\texttt{[}\ E'\ \texttt{for}\ x\ \texttt{in}\ e :: E\ \texttt{if}\ B\ \texttt{]}]\!]\,\sigma = v' :: [\![\texttt{[}\ E'\ \texttt{for}\ x\ \texttt{in}\ E\ \texttt{if}\ B\ \texttt{]}]\!]\,\sigma}$$

$$\text{E-Array-Comp-False}\ \frac{[\![e]\!]\,\sigma = v \qquad [\![B]\!]\,\sigma[x \mapsto v] = \text{false}}{[\![\texttt{[}\ E'\ \texttt{for}\ x\ \texttt{in}\ e :: E\ \texttt{if}\ B\ \texttt{]}]\!]\,\sigma = [\![\texttt{[}\ E'\ \texttt{for}\ x\ \texttt{in}\ E\ \texttt{if}\ B\ \texttt{]}]\!]\,\sigma}$$

$$\text{E-Array-Const}\ \frac{\forall i : 0 \le i \le n, [\![C_i]\!]\,\sigma = v_i}{[\![\texttt{[}\ C_0\texttt{,}\ \ldots\texttt{,}C_n\ \texttt{]}]\!]\,\sigma = [v_0, \ldots, v_n]}$$

3.2 Operational Semantics for Statements

Let $ref(o)$ be the reference of the object o. Let $(B, S) :: G$ denote a nonempty guarded block $G' = B\ \texttt{:}\ S\ \texttt{|||}\ B_1\ \texttt{:}\ S_1\ \texttt{|||}\ \cdots\ \texttt{|||}\ B_n\ \texttt{:}\ S_n$ where the guarded block $G = B_1\ \texttt{:}\ S_1\ \texttt{|||}\ \cdots\ \texttt{|||}\ B_n\ \texttt{:}\ S_n$. In case G' has only one guard, say $G' = B\ \texttt{:}\ S$, then G is empty, denoted by $G = \varepsilon$.

The operational semantics models the execution behaviors of the statement S. It has the form $\langle \sigma, S \rangle \rightarrow \sigma'$, meaning that in the value environment σ, after executing S, the value environment changes to σ'. Statements are executed sequentially, that is to say, after executing the first statement, the updated value environment is used as the input value environment for the rest statements, as formulated by the following rule:

$$\text{E-Seq}\ \frac{\langle \sigma, S_1 \rangle \rightarrow \sigma' \qquad \langle \sigma', S_2 \rangle \rightarrow \sigma''}{\langle \sigma, S_1; S_2 \rangle \rightarrow \sigma''}$$

The rules for the new featured statements are:

$$\text{E-OCStmt}\ \frac{[\![E]\!]\ \sigma = v \qquad \sigma' = \sigma[x \mapsto ref(v)]}{\langle \sigma, \texttt{scopy (} x \texttt{ , } E \texttt{) ;} \rangle \rightarrow \sigma'}$$

$$\text{E-GuardedStmt}\ \frac{\langle \sigma, G \rangle \rightarrow \sigma'}{\langle \sigma, \texttt{if \{ } G \texttt{ \}} \rangle \rightarrow \sigma'}$$

$$\text{E-GuardedBlock-Empty}\ \frac{}{\langle \sigma, \varepsilon \rangle \rightarrow \sigma}$$

$$\text{E-GuardedBlock-True}\ \frac{[\![B]\!]\ \sigma = \text{true} \qquad \langle \sigma, S \rangle \rightarrow \sigma'}{\langle \sigma, (B, S) :: G \rangle \rightarrow \sigma'}$$

$$\text{E-GuardedBlock-False}\ \frac{[\![B]\!]\ \sigma = \text{false} \qquad \langle \sigma, G \rangle \rightarrow \sigma'}{\langle \sigma, (B, S) :: G \rangle \rightarrow \sigma'}$$

$$\text{E-ForeachStmt-Empty}\ \frac{}{\langle \sigma, \texttt{foreach (var } x \texttt{ in [] while } B \texttt{) } S \rangle \rightarrow \sigma}$$

$$\text{E-ForeachStmt-False}\ \frac{[\![e]\!]\ \sigma = v \qquad [\![B]\!]\ \sigma[x \mapsto v] = \text{false}}{\langle \sigma, \texttt{foreach (var } x \texttt{ in } e :: E \texttt{ while } B \texttt{) } S \rangle \rightarrow \sigma}$$

$$\text{E-ForeachStmt-True}\ \frac{[\![e]\!]\ \sigma = v \qquad [\![B]\!]\ \sigma[x \mapsto v] = \text{true} \qquad \langle \sigma[x \mapsto v], S \rangle \rightarrow \sigma' \qquad \langle \sigma', \texttt{foreach (var } x \texttt{ in } E \texttt{ while } B \texttt{) } S \rangle \rightarrow \sigma''}{\langle \sigma, \texttt{foreach (var } x \texttt{ in } e :: E \texttt{ while } B \texttt{) } S \rangle \rightarrow \sigma''}$$

Particularly, we have to concern about the break-statement inside the loop body of foreach-statements. If the break-statement is reached in the last iteration, then the loop should be terminated immediately, no matter the condition holds or not. To model this, we introduce *signals*, with two values, $\not\triangleright$ for skip and $\triangleright$ for continue. The signal-aware operational semantics have the form $\langle (\sigma, s), S \rangle \rightarrow (\sigma', s')$, where s and s' are signals, meaning that in the value environment σ, given the history that the break-statement is reached (s is $\not\triangleright$) or not (s is $\triangleright$) in the last iteration, after executing S, then the value environment changes to σ', and s' denotes in this iteration the break-statement is reached or not.

Signal $\not\triangleright$ is generated only when the break-statement is reached:

$$\text{E-Break}\ \frac{}{\langle (\sigma, \triangleright), \texttt{break ;} \rangle \rightarrow (\sigma, \not\triangleright)}$$

The rules for the foreach-statement are:

$$\text{E-ForeachStmt-Skip}\ \frac{}{\langle (\sigma, \not\triangleright), \texttt{foreach (var } x \texttt{ in } e :: E \texttt{ while } B \texttt{) } S \rangle \rightarrow (\sigma, \triangleright)}$$

$$\text{E-ForeachStmt-Empty-SIG}\ \frac{}{\langle (\sigma, \triangleright), \texttt{foreach (var } x \texttt{ in [] while } B \texttt{) } S \rangle \rightarrow (\sigma, \triangleright)}$$

$$\text{E-ForeachStmt-False-SIG}\ \frac{[\![e]\!]\ \sigma = v \qquad [\![B]\!]\ \sigma[x \mapsto v] = \text{false}}{\langle (\sigma, \triangleright), \texttt{foreach (var } x \texttt{ in } e :: E \texttt{ while } B \texttt{) } S \rangle \rightarrow (\sigma, \triangleright)}$$

$$\text{E-ForeachStmt-True-SIG}\ \frac{[\![e]\!]\ \sigma = v \qquad [\![B]\!]\ \sigma[x \mapsto v] = \text{true} \qquad \langle (\sigma[x \mapsto v], \triangleright), S \rangle \rightarrow (\sigma', s') \qquad \langle (\sigma', s'), \texttt{foreach (var } x \texttt{ in } E \texttt{ while } B \texttt{) } S \rangle \rightarrow (\sigma'', s'')}{\langle (\sigma, \triangleright), \texttt{foreach (var } x \texttt{ in } e :: E \texttt{ while } B \texttt{) } S \rangle \rightarrow (\sigma'', s'')}$$

The situation is identical for **`foreach (`** T x **`in`** E **`while`** B **`)`** S except that variable x has the specified type T.

For a statement S in which the break-statement is not allowed,

$$\langle \sigma, S \rangle \rightarrow \sigma' \equiv \langle (\sigma, \triangleright), S \rangle \rightarrow (\sigma', \triangleright).$$