

Denotational semantics

Evaluating e with argument α and state σ results in the value $\llbracket e \rrbracket_v(\alpha, \sigma)$ and the updated state $\llbracket e \rrbracket_s(\alpha, \sigma)$.

$$\llbracket \cdot \rrbracket_v : \text{Value} \times \text{State} \rightarrow \text{Value} \quad \llbracket \cdot \rrbracket_s : \text{Value} \times \text{State} \rightarrow \text{State} \quad \llbracket e \rrbracket(\alpha, \sigma) \triangleq (\llbracket e \rrbracket_v(\alpha, \sigma), \llbracket e \rrbracket_s(\alpha, \sigma))$$

$$\text{State} \triangleq \text{Memory} \times \text{Log} \quad \llbracket \cdot \rrbracket_m \triangleq \text{fst} \circ \llbracket \cdot \rrbracket_s \quad \llbracket \cdot \rrbracket_l \triangleq \text{snd} \circ \llbracket \cdot \rrbracket_s$$

`malloc` takes a state and number of bytes n , finds a contiguous region of free memory starting at address p , and returns p and an updated state where addresses $[p, p + n)$ are marked as used.

$$\begin{aligned} \llbracket \text{Num } n \rrbracket(\alpha, \sigma) &\triangleq (n, \sigma) \\ \llbracket \text{Bool } b \rrbracket(\alpha, \sigma) &\triangleq (b, \sigma) \\ \llbracket \text{Arg} \rrbracket(\alpha, \sigma) &\triangleq (\alpha, \sigma) \\ \llbracket \text{Add } e_1 \ e_2 \rrbracket_v(\alpha, \sigma) &\triangleq \llbracket e_1 \rrbracket_v(\alpha, \sigma) + \llbracket e_2 \rrbracket_v(\alpha, \llbracket e_1 \rrbracket_s(\alpha, \sigma)) \\ \llbracket \text{Add } e_1 \ e_2 \rrbracket_s(\alpha, \sigma) &\triangleq \llbracket e_2 \rrbracket_s(\alpha, \llbracket e_1 \rrbracket_s(\alpha, \sigma)) \\ \llbracket \text{If } e_c \ e_t \ e_e \rrbracket(\alpha, \sigma) &\triangleq \begin{cases} \llbracket e_t \rrbracket(\alpha, \llbracket e_c \rrbracket_s(\alpha, \sigma)) & \llbracket e_c \rrbracket_v(\alpha, \sigma) = \top \\ \llbracket e_e \rrbracket(\alpha, \llbracket e_c \rrbracket_s(\alpha, \sigma)) & \llbracket e_c \rrbracket_v(\alpha, \sigma) = \perp \end{cases} \\ \llbracket \text{Switch } e_{pred} \ \vec{e} \rrbracket(\alpha, \sigma) &\triangleq \llbracket \vec{e} \llbracket e_{pred} \rrbracket_v(\alpha, \sigma) \rrbracket(\alpha, \llbracket e_{pred} \rrbracket_s(\alpha, \sigma)) \\ \llbracket \text{Let } e_{in} \ e_{out} \rrbracket &\triangleq \llbracket e_{out} \rrbracket \circ \llbracket e_{in} \rrbracket \\ \llbracket \text{Print } e \rrbracket_v(\alpha, \sigma) &\triangleq [] \\ \llbracket \text{Print } e \rrbracket_s(\alpha, \sigma) &\triangleq (\llbracket e \rrbracket_m(\alpha, \sigma), \llbracket e \rrbracket_l(\alpha, \sigma) ++ \llbracket e \rrbracket_v(\alpha, \sigma)) \\ \llbracket \text{Read } e \rrbracket_v(\alpha, \sigma) &\triangleq \llbracket e \rrbracket_m(\alpha, \sigma) [\llbracket e \rrbracket_v(\alpha, \sigma)] \\ \llbracket \text{Read } e \rrbracket_s &\triangleq \llbracket e \rrbracket_s \\ \llbracket \text{Alloc } e \ \tau \rrbracket(\alpha, \sigma) &\triangleq \text{malloc}(\llbracket e \rrbracket_s(\alpha, \sigma), \llbracket e \rrbracket_v(\alpha, \sigma) * \text{sizeof}(\tau)) \\ \llbracket \text{Write } e_p \ e_d \rrbracket_v(\alpha, \sigma) &\triangleq [] \\ \llbracket \text{Write } e_p \ e_d \rrbracket_m(\alpha, \sigma) &\triangleq \llbracket e_p \rrbracket_m(\alpha, \llbracket e_d \rrbracket_s(\alpha, \sigma)) [\llbracket e_p \rrbracket_v(\alpha, \sigma) \mapsto \llbracket e_d \rrbracket_v(\alpha, \sigma)] \\ \llbracket \text{Write } e_p \ e_d \rrbracket_l(\alpha, \sigma) &\triangleq \llbracket e_p \rrbracket_l(\alpha, \llbracket e_d \rrbracket_s(\alpha, \sigma)) \\ \llbracket \text{Single } e \rrbracket_v(\alpha, \sigma) &\triangleq \llbracket e \rrbracket_v(\alpha, \sigma) \\ \llbracket \text{Single } e \rrbracket_s &\triangleq \llbracket e \rrbracket_s \\ \llbracket \text{Concat Sequential } e_1 \ e_2 \rrbracket_v(\alpha, \sigma) &\triangleq \llbracket e_1 \rrbracket_v(\alpha, \sigma) ++ \llbracket e_2 \rrbracket_v(\alpha, \llbracket e_2 \rrbracket_s(\alpha, \sigma)) \\ \llbracket \text{Concat Sequential } e_1 \ e_2 \rrbracket_s(\alpha, \sigma) &\triangleq \llbracket e_2 \rrbracket_s(\alpha, \llbracket e_1 \rrbracket_s(\alpha, \sigma)) \\ \llbracket \text{DoWhile } e_{in} \ e_{po} \rrbracket(\alpha, \sigma) &\triangleq \text{let } (v_{in}, \sigma') = \llbracket e_{in} \rrbracket(\alpha, \sigma) \text{ in} \\ &\quad \text{let } ((v_{pred}, v_{out}), \sigma'') = \llbracket e_{po} \rrbracket(v_{in}, \sigma') \text{ in} \\ &\quad \begin{cases} (v_{out}, \sigma'') & v_{pred} = \perp \\ \llbracket \text{DoWhile Arg } e_{po} \rrbracket(v_{out}, \sigma'') & v_{pred} = \top \end{cases} \\ &\quad \text{(this should be written as a fixpoint instead)} \\ \llbracket \text{Assume (InLet } e_{in}) \ e \rrbracket(\alpha, \sigma) &\triangleq \begin{cases} \llbracket e \rrbracket(\alpha, \sigma) & \exists \alpha', \sigma'. \llbracket e_{in} \rrbracket_v(\alpha', \sigma') = \alpha \\ \text{undefined} & \text{otherwise} \end{cases} \\ \llbracket \text{Assume (InLoop } e_{in} \ e_{po}) \ e \rrbracket(\alpha, \sigma) &\triangleq \begin{cases} \llbracket e \rrbracket(\alpha, \sigma) & \exists \alpha', \sigma'. \llbracket e_{in} \rrbracket_v(\alpha', \sigma') = \alpha \\ & \vee \llbracket \text{Assume (InLoop } e_{in} \ e_{po}) \ e \rrbracket_v(\alpha', \sigma') = \alpha \\ \text{undefined} & \text{otherwise} \end{cases} \\ \llbracket \text{Assume (AfterWrite } e_p \ e_d) \ e \rrbracket(\alpha, \sigma) &\triangleq \begin{cases} \llbracket e \rrbracket(\alpha, \sigma) & \exists \alpha', \sigma'. \llbracket \text{Write } e_p \ e_d \rrbracket_s(\alpha', \sigma') = \sigma \\ \text{undefined} & \text{otherwise} \end{cases} \end{aligned}$$

Big-step operational semantics

$\langle e, \alpha, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$ means: with argument α and state σ , e evaluates to v and the resulting state is σ' .

A state is a pair (M, L) , containing memory and a print log.

$$\begin{array}{c}
\frac{\llbracket e_1 \rrbracket(\alpha, \sigma) = (v_1, \sigma') \quad \llbracket e_2 \rrbracket(\alpha, \sigma') = (v_2, \sigma'')}{\llbracket \text{Add } e_1 \ e_2 \rrbracket(\alpha, \sigma) \triangleq (v_1 + v_2, \sigma'')} \quad (\text{E-ADD}) \\
\\
\frac{\llbracket e_c \rrbracket(\alpha, \sigma) = (\top, \sigma') \quad \llbracket e_t \rrbracket(\alpha, \sigma') = (v_t, \sigma'')}{\llbracket \text{If } e_c \ e_t \ e_e \rrbracket(\alpha, \sigma) \triangleq (v_t, \sigma'')} \quad (\text{E-IFTRUE}) \\
\\
\frac{\llbracket e_c \rrbracket(\alpha, \sigma) = (\perp, \sigma') \quad \llbracket e_e \rrbracket(\alpha, \sigma') = (v_e, \sigma'')}{\llbracket \text{If } e_c \ e_t \ e_e \rrbracket(\alpha, \sigma) \triangleq (v_e, \sigma'')} \quad (\text{E-IFFALSE}) \\
\\
\frac{\llbracket e_{pred} \rrbracket(\alpha, \sigma) = (i, \sigma') \quad \llbracket e_i \rrbracket(\alpha, \sigma') = (v, \sigma'')}{\llbracket \text{Switch } e_{pred} \ (e_1, \dots, e_n) \rrbracket(\alpha, \sigma) \triangleq (v, \sigma'')} \quad (\text{E-SWITCH}) \\
\\
\frac{\llbracket e_{in} \rrbracket(\alpha, \sigma) = (v_{in}, \sigma') \quad \llbracket e_{out} \rrbracket(v_{in}, \sigma') = (v_{out}, \sigma'')}{\llbracket \text{Let } e_{in} \ e_{out} \rrbracket(\alpha, \sigma) \triangleq (v_{out}, \sigma'')} \quad (\text{E-LET}) \\
\\
\frac{\llbracket e \rrbracket(\alpha, \sigma) = (v, (M, L))}{\llbracket \text{Print } e \rrbracket(\alpha, \sigma) \triangleq ([], (M, L ++ v))} \quad (\text{E-PRINT}) \\
\\
\frac{\llbracket e \rrbracket(\alpha, \sigma) = (v, (M, L))}{\llbracket \text{Read } e \rrbracket(\alpha, \sigma) \triangleq (M[v], (M, L))} \quad (\text{E-READ}) \\
\\
\frac{\llbracket e \rrbracket(\alpha, \sigma) = (n, (M, L)) \quad (M', p) = \text{malloc}(M, n * \text{sizeof}(\tau))}{\llbracket \text{Alloc } e \ \tau \rrbracket(\alpha, \sigma) \triangleq (p, (M', L))} \quad (\text{E-ALLOC}) \\
\\
\frac{\llbracket e_p \rrbracket(\alpha, \sigma) = (v_p, \sigma') \quad \llbracket e_d \rrbracket(\alpha, \sigma') = (v_d, (M, L))}{\llbracket \text{Write } e_p \ e_d \rrbracket(\alpha, \sigma) \triangleq ([], (M[v_p \mapsto v_d], L))} \quad (\text{E-WRITE}) \\
\\
\frac{\llbracket e \rrbracket(\alpha, \sigma) = (v, \sigma')}{\llbracket \text{Single } e \rrbracket(\alpha, \sigma) \triangleq ([v], \sigma')} \quad (\text{E-SINGLE}) \\
\\
\frac{\llbracket e_1 \rrbracket(\alpha, \sigma) = (v_1, \sigma') \quad \llbracket e_2 \rrbracket(\alpha, \sigma') = (v_2, \sigma'')}{\llbracket \text{Concat Sequential } e_1 \ e_2 \rrbracket(\alpha, \sigma) \triangleq (v_1 ++ v_2, \sigma'')} \quad (\text{E-CONCATSEQ}) \\
\\
\frac{\llbracket e_{in} \rrbracket(\alpha, \sigma) = (\alpha', \sigma') \quad \llbracket e_{po} \rrbracket(\alpha', \sigma') = ([\perp, v], \sigma'')}{\llbracket \text{DoWhile } e_{in} \ e_{out} \rrbracket(\alpha, \sigma) \triangleq (v, \sigma'')} \quad (\text{E-DOWHILEFALSE}) \\
\\
\frac{\llbracket e_{in} \rrbracket(\alpha, \sigma) = (\alpha', \sigma') \quad \llbracket e_{po} \rrbracket(\alpha', \sigma') = ([\top, \alpha''], \sigma'') \quad \llbracket \text{DoWhile } e_{in} \ e_{po} \rrbracket(\alpha'', \sigma'') = (v, \sigma''')}{\llbracket \text{DoWhile } e_{in} \ e_{po} \rrbracket(\alpha, \sigma) \triangleq (v, \sigma''')} \quad (\text{E-DOWHILETRUE}) \\
\\
\frac{\llbracket e \rrbracket(\alpha, \sigma) = (v, \sigma') \quad \exists \alpha_2, \sigma_2, \sigma_3. \llbracket e_{in} \rrbracket(\alpha_2, \sigma_2) = (\alpha, \sigma_3)}{\llbracket \text{Assume (InLet } e_{in}) \ e \rrbracket(\alpha, \sigma) \triangleq (v, \sigma')} \quad (\text{E-ASSUMEINLET}) \\
\\
\frac{\llbracket e \rrbracket(\alpha, \sigma) = (v, \sigma') \quad \exists \alpha_2, v_2, \sigma_2. \llbracket \text{Write } e_p \ e_d \rrbracket(\alpha_2, \sigma_2) = (v_2, \sigma_3)}{\llbracket \text{Assume (AfterWrite } e_p \ e_d) \ e \rrbracket(\alpha, \sigma) \triangleq (v, \sigma')} \quad (\text{E-ASSUMEAFTERWRITE}) \\
\\
\frac{\llbracket e \rrbracket(\alpha, \sigma) = (v, \sigma') \quad \exists \alpha_2, \sigma_2, \sigma_3. \llbracket e_{in} \rrbracket(\alpha_2, \sigma_2) = (\alpha, \sigma_3)}{\llbracket \text{Assume (InLoop } e_{in} \ e_{po}) \ e \rrbracket(\alpha, \sigma) \triangleq (v, \sigma')} \quad (\text{E-ASSUMEINLOOP1}) \\
\\
\frac{\exists \alpha_2, \sigma_2, \sigma_3. \llbracket \text{Assume (InLoop } e_{in} \ e_{po}) \ e \rrbracket(\alpha_2, \sigma_2) = (\alpha, \sigma_3)}{\llbracket \text{Assume (InLoop } e_{in} \ e_{po}) \ e \rrbracket(\alpha, \sigma) \triangleq (v, \sigma')} \quad (\text{E-ASSUMEINLOOP2})
\end{array}$$

Abstract grammar

$\langle \text{basetype} \rangle ::= \text{IntT}$	$\langle \text{expr} \rangle ::= \text{Arg } \langle \text{type} \rangle$
BoolT	Int $\langle \mathbb{N} \rangle$
$\langle \text{type} \rangle ::= \langle \text{basetype} \rangle$	Bool $\langle \text{bool} \rangle$
PointerT $\langle \text{type} \rangle$	Empty
TupleT $\langle \text{basetype} \rangle^*$	Add $\langle \text{expr} \rangle \langle \text{expr} \rangle$
$\langle \text{bool} \rangle ::= \top$	Sub $\langle \text{expr} \rangle \langle \text{expr} \rangle$
\perp	Mul $\langle \text{expr} \rangle \langle \text{expr} \rangle$
$\langle \text{order} \rangle ::= \text{Parallel}$	LessThan $\langle \text{expr} \rangle \langle \text{expr} \rangle$
Sequential	And $\langle \text{expr} \rangle \langle \text{expr} \rangle$
$\langle \text{function} \rangle ::= \text{Function } \langle \text{type} \rangle \langle \text{expr} \rangle$	Or $\langle \text{expr} \rangle \langle \text{expr} \rangle$
$\langle \text{assumption} \rangle ::= \text{InLet } \langle \text{expr} \rangle$	Write $\langle \text{expr} \rangle \langle \text{expr} \rangle$
InLoop $\langle \text{expr} \rangle \langle \text{expr} \rangle$	Not $\langle \text{expr} \rangle$
AfterWrite $\langle \text{expr} \rangle \langle \text{expr} \rangle$	Print $\langle \text{expr} \rangle$
	Read $\langle \text{expr} \rangle$
	Get $\langle \text{expr} \rangle \langle \mathbb{N} \rangle$
	Alloc $\langle \text{expr} \rangle \langle \text{type} \rangle$
	Call $\langle \text{function} \rangle \langle \text{expr} \rangle$
	Single $\langle \text{expr} \rangle$
	Concat $\langle \text{order} \rangle \langle \text{expr} \rangle \langle \text{expr} \rangle$
	Switch $\langle \text{expr} \rangle \langle \text{expr} \rangle^*$
	If $\langle \text{expr} \rangle \langle \text{expr} \rangle \langle \text{expr} \rangle$
	Let $\langle \text{expr} \rangle \langle \text{expr} \rangle$
	DoWhile $\langle \text{expr} \rangle \langle \text{expr} \rangle$
	Assume $\langle \text{assumption} \rangle \langle \text{expr} \rangle$