Denotational semantics

Evaluating e with argument α and state σ results in the value $[\![e]\!]_v(\alpha,\sigma)$ and the updated state $[\![e]\!]_s(\alpha,\sigma)$.
$[\![\cdot]\!]_v$: Value × State → Value $\quad$ $[\![\cdot]\!]_s$: Value × State → State $\quad$ $[\![e]\!](\alpha,\sigma) \triangleq ([\![e]\!]_v(\alpha,\sigma), [\![e]\!]_s(\alpha,\sigma))$

State $\triangleq$ Memory × Log $\quad$ $[\![\cdot]\!]_m \triangleq \text{fst} \circ [\![\cdot]\!]_s$ $\quad$ $[\![\cdot]\!]_l \triangleq \text{snd} \circ [\![\cdot]\!]_s$

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}
[\![\text{Num } n]\!](\alpha,\sigma) &\triangleq (n,\sigma) \\
[\![\text{Bool } b]\!](\alpha,\sigma) &\triangleq (b,\sigma) \\
[\![\text{Arg}]\!](\alpha,\sigma) &\triangleq (\alpha,\sigma) \\
[\![\text{Add } e_1\ e_2]\!]_v(\alpha,\sigma) &\triangleq [\![e_1]\!]_v(\alpha,\sigma) + [\![e_2]\!]_v(\alpha,[\![e_1]\!]_s(\alpha,\sigma)) \\
[\![\text{Add } e_1\ e_2]\!]_s(\alpha,\sigma) &\triangleq [\![e_2]\!]_s(\alpha,[\![e_1]\!]_s(\alpha,\sigma)) \\
[\![\text{If } e_c\ e_t\ e_e]\!](\alpha,\sigma) &\triangleq \begin{cases} [\![e_t]\!](\alpha,[\![e_c]\!]_s(\alpha,\sigma)) & [\![e_c]\!]_v(\alpha,\sigma) = \top \\ [\![e_e]\!](\alpha,[\![e_c]\!]_s(\alpha,\sigma)) & [\![e_c]\!]_v(\alpha,\sigma) = \bot \end{cases} \\
[\![\text{Switch } e_{pred}\ \vec{e}\,]\!](\alpha,\sigma) &\triangleq \left[\!\left[\vec{e}\,[[\![e_{pred}]\!]_v(\alpha,\sigma)]\right]\!\right]\left(\alpha,[\![e_{pred}]\!]_s(\alpha,\sigma)\right) \\
[\![\text{Let } e_{in}\ e_{out}]\!] &\triangleq [\![e_{out}]\!] \circ [\![e_{in}]\!] \qquad \text{(point-free version)} \\
[\![\text{Let } e_{in}\ e_{out}]\!](\alpha,\sigma) &\triangleq [\![e_{out}]\!]([\![e_{in}]\!](\alpha,\sigma)) \qquad \text{(η-expanded for clarity)} \\
[\![\text{Print } e]\!]_v(\alpha,\sigma) &\triangleq [] \\
[\![\text{Print } e]\!]_s(\alpha,\sigma) &\triangleq ([\![e]\!]_m(\alpha,\sigma),\ [\![e]\!]_l(\alpha,\sigma) \ +\!\!+\ [\![e]\!]_v(\alpha,\sigma)) \\
[\![\text{Read } e]\!]_v(\alpha,\sigma) &\triangleq [\![e]\!]_m(\alpha,\sigma)\,[[\![e]\!]_v(\alpha,\sigma)] \\
[\![\text{Read } e]\!]_s &\triangleq [\![e]\!]_s \\
[\![\text{Alloc } e\ \tau]\!](\alpha,\sigma) &\triangleq \text{malloc}([\![e]\!]_s(\alpha,\sigma),\ [\![e]\!]_v(\alpha,\sigma) * \text{sizeof}(\tau)) \\
[\![\text{Write } e_p\ e_d]\!]_v(\alpha,\sigma) &\triangleq [] \\
[\![\text{Write } e_p\ e_d]\!]_m(\alpha,\sigma) &\triangleq [\![e_p]\!]_m(\alpha,[\![e_d]\!]_s(\alpha,\sigma))\,[[\![e_p]\!]_v(\alpha,\sigma) \mapsto [\![e_d]\!]_v(\alpha,\sigma)] \\
[\![\text{Write } e_p\ e_d]\!]_l(\alpha,\sigma) &\triangleq [\![e_p]\!]_l(\alpha,[\![e_d]\!]_s(\alpha,\sigma)) \\
[\![\text{Single } e]\!]_v(\alpha,\sigma) &\triangleq [[\![e]\!]_v(\alpha,\sigma)] \\
[\![\text{Single } e]\!]_s &\triangleq [\![e]\!]_s \\
[\![\text{Concat Sequential } e_1\ e_2]\!]_v(\alpha,\sigma) &\triangleq [\![e_1]\!]_v(\alpha,\sigma) \ +\!\!+\ [\![e_2]\!]_v(\alpha,[\![e_1]\!]_s(\alpha,\sigma)) \\
[\![\text{Concat Sequential } e_1\ e_2]\!]_s(\alpha,\sigma) &\triangleq [\![e_2]\!]_s(\alpha,[\![e_1]\!]_s(\alpha,\sigma)) \\
[\![\text{DoWhile } e_{in}\ e_{po}]\!](\alpha,\sigma) &\triangleq \text{let } (v_{in},\sigma') = [\![e_{in}]\!](\alpha,\sigma) \text{ in} \\
&\quad\ \text{let } ((v_{pred},v_{out}),\sigma'') = [\![e_{po}]\!](v_{in},\sigma') \text{ in} \\
&\quad\ \begin{cases} (v_{out},\sigma'') & v_{pred} = \bot \\ [\![\text{DoWhile Arg } e_{po}]\!](v_{out},\sigma'') & v_{pred} = \top \end{cases} \\
&\quad\ \text{(this should be written as a fixpoint instead)} \\
[\![\text{Assume (InLet } e_{in}\text{) } e]\!](\alpha,\sigma) &\triangleq \begin{cases} [\![e]\!](\alpha,\sigma) & \exists\alpha',\sigma'.\ [\![e_{in}]\!]_v(\alpha',\sigma') = \alpha \end{cases} \\
[\![\text{Assume (AfterWrite } e_p\ e_d\text{) } e]\!](\alpha,\sigma) &\triangleq \begin{cases} [\![e]\!](\alpha,\sigma) & \exists\alpha',\sigma'.\ [\![\text{Write } e_p\ e_d]\!]_s(\alpha',\sigma') = \sigma \end{cases} \\
[\![\text{Assume (InLoop } e_{in}\ e_{po}\text{) } e]\!](\alpha,\sigma) &\triangleq \begin{cases} [\![e]\!](\alpha,\sigma) & \begin{array}{l} \exists\alpha',\sigma'. \\ [\![e_{in}]\!]_v(\alpha',\sigma') = \alpha\ \vee \\ (\text{let } (v_p,v_o) = [\![e_{po}]\!]_v(\alpha',\sigma') \text{ in} \\ v_p = \top \wedge \\ (v_o,\sigma') \in \text{domain}([\![\text{Assume (InLoop Arg } e_{po}\text{) } e]\!])) \end{array} \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.

$$\frac{[\![e_1]\!](\alpha,\sigma) = (v_1,\sigma') \qquad [\![e_2]\!](\alpha,\sigma') = (v_2,\sigma'')}{[\![\text{Add } e_1\ e_2]\!](\alpha,\sigma) \triangleq (v_1 + v_2, \sigma'')} \quad \text{(E-ADD)}$$

$$\frac{[\![e_c]\!](\alpha,\sigma) = (\top,\sigma') \qquad [\![e_t]\!](\alpha,\sigma') = (v_t,\sigma'')}{[\![\text{If } e_c\ e_t\ e_e]\!](\alpha,\sigma) \triangleq (v_t, \sigma'')} \quad \text{(E-IFTRUE)}$$

$$\frac{[\![e_c]\!](\alpha,\sigma) = (\bot,\sigma') \qquad [\![e_e]\!](\alpha,\sigma') = (v_e,\sigma'')}{[\![\text{If } e_c\ e_t\ e_e]\!](\alpha,\sigma) \triangleq (v_e, \sigma'')} \quad \text{(E-IFFALSE)}$$

$$\frac{[\![e_{pred}]\!](\alpha,\sigma) = (i,\sigma') \qquad [\![e_i]\!](\alpha,\sigma') = (v,\sigma'')}{[\![\text{Switch } e_{pred}\ (e_1,\ldots,e_n)]\!](\alpha,\sigma) \triangleq (v, \sigma'')} \quad \text{(E-SWITCH)}$$

$$\frac{[\![e_{in}]\!](\alpha,\sigma) = (v_{in},\sigma') \qquad [\![e_{out}]\!](v_{in},\sigma') = (v_{out},\sigma'')}{[\![\text{Let } e_{in}\ e_{out}]\!](\alpha,\sigma) \triangleq (v_{out}, \sigma'')} \quad \text{(E-LET)}$$

$$\frac{[\![e]\!](\alpha,\sigma) = (v,(M,L))}{[\![\text{Print } e]\!](\alpha,\sigma) \triangleq ([], (M, L \mathbin{+\!\!+} v))} \quad \text{(E-PRINT)}$$

$$\frac{[\![e]\!](\alpha,\sigma) = (v,(M,L))}{[\![\text{Read } e]\!](\alpha,\sigma) \triangleq (M[v], (M, L))} \quad \text{(E-READ)}$$

$$\frac{[\![e]\!](\alpha,\sigma) = (n,(M,L)) \qquad (M',p) = \text{malloc}(M, n * \text{sizeof}(\tau))}{[\![\text{Alloc } e\ \tau]\!](\alpha,\sigma) \triangleq (p, (M', L))} \quad \text{(E-ALLOC)}$$

$$\frac{[\![e_p]\!](\alpha,\sigma) = (v_p,\sigma') \qquad [\![e_d]\!](\alpha,\sigma') = (v_d,(M,L))}{[\![\text{Write } e_p\ e_d]\!](\alpha,\sigma) \triangleq ([], (M[v_p \mapsto v_d], L))} \quad \text{(E-WRITE)}$$

$$\frac{[\![e]\!](\alpha,\sigma) = (v,\sigma')}{[\![\text{Single } e]\!](\alpha,\sigma) \triangleq ([v], \sigma')} \quad \text{(E-SINGLE)}$$

$$\frac{[\![e_1]\!](\alpha,\sigma) = (v_1,\sigma') \qquad [\![e_2]\!](\alpha,\sigma') = (v_2,\sigma'')}{[\![\text{Concat Sequential } e_1\ e_2]\!](\alpha,\sigma) \triangleq (v_1 \mathbin{+\!\!+} v_2, \sigma'')} \quad \text{(E-CONCATSEQ)}$$

$$\frac{[\![e_{in}]\!](\alpha,\sigma) = (\alpha',\sigma') \qquad [\![e_{po}]\!](\alpha',\sigma') = ([\bot, v],\sigma'')}{[\![\text{DoWhile } e_{in}\ e_{out}]\!](\alpha,\sigma) \triangleq (v, \sigma'')} \quad \text{(E-DOWHILEFALSE)}$$

$$\frac{[\![e_{in}]\!](\alpha,\sigma) = (\alpha',\sigma') \qquad [\![e_{po}]\!](\alpha',\sigma') = ([\top, \alpha''],\sigma'') \qquad [\![\text{DoWhile } e_{in}\ e_{po}]\!](\alpha'',\sigma'') = (v,\sigma''')}{[\![\text{DoWhile } e_{in}\ e_{po}]\!](\alpha,\sigma) \triangleq (v, \sigma''')}$$

(E-DOWHILETRUE)

$$\frac{[\![e]\!](\alpha,\sigma) = (v,\sigma') \qquad \exists\, \alpha_2, \sigma_2, \sigma_3.\ [\![e_{in}]\!](\alpha_2,\sigma_2) = (\alpha,\sigma_3)}{[\![\text{Assume (InLet } e_{in}) \ e]\!](\alpha,\sigma) \triangleq (v, \sigma')} \quad \text{(E-ASSUMEINLET)}$$

$$\frac{[\![e]\!](\alpha,\sigma) = (v,\sigma') \qquad \exists\, \alpha_2, v_2, \sigma_2.\ [\![\text{Write } e_p\ e_d]\!](\alpha_2,\sigma_2) = (v_2,\sigma_3)}{[\![\text{Assume (AfterWrite } e_p\ e_d) \ e]\!](\alpha,\sigma) \triangleq (v, \sigma')} \quad \text{(E-ASSUMEAFTERWRITE)}$$

$$\frac{[\![e]\!](\alpha,\sigma) = (v,\sigma') \qquad \exists\, \alpha_2, \sigma_2, \sigma_3.\ [\![e_{in}]\!](\alpha_2,\sigma_2) = (\alpha,\sigma_3)}{[\![\text{Assume (InLoop } e_{in}\ e_{po}) \ e]\!](\alpha,\sigma) \triangleq (v, \sigma')} \quad \text{(E-ASSUMEINLOOP1)}$$

$$\frac{\exists\, \alpha_2, \sigma_2, \sigma_3.\ [\![\text{Assume (InLoop } e_{in}\ e_{po}) \ e]\!](\alpha_2,\sigma_2) = (\alpha,\sigma_3)}{[\![\text{Assume (InLoop } e_{in}\ e_{po}) \ e]\!](\alpha,\sigma) \triangleq (v, \sigma')} \quad \text{(E-ASSUMEINLOOP2)}$$

Abstract grammar

$\langle basetype \rangle$ ::= IntT
| BoolT

$\langle type \rangle$ ::= $\langle basetype \rangle$
| PointerT $\langle type \rangle$
| TupleT $\langle basetype \rangle$*

$\langle bool \rangle$ ::= $\top$
| $\bot$

$\langle order \rangle$ ::= Parallel
| Sequential

$\langle function \rangle$ ::= Function $\langle type \rangle$ $\langle expr \rangle$

$\langle assumption \rangle$ ::= InLet $\langle expr \rangle$
| InLoop $\langle expr \rangle$ $\langle expr \rangle$
| AfterWrite $\langle expr \rangle$ $\langle expr \rangle$

$\langle expr \rangle$ ::= Arg $\langle type \rangle$
| Int $\langle \mathbb{N} \rangle$
| Bool $\langle bool \rangle$
| Empty
| Add $\langle expr \rangle$ $\langle expr \rangle$
| Sub $\langle expr \rangle$ $\langle expr \rangle$
| Mul $\langle expr \rangle$ $\langle expr \rangle$
| LessThan $\langle expr \rangle$ $\langle expr \rangle$
| And $\langle expr \rangle$ $\langle expr \rangle$
| Or $\langle expr \rangle$ $\langle expr \rangle$
| Write $\langle expr \rangle$ $\langle expr \rangle$
| Not $\langle expr \rangle$
| Print $\langle expr \rangle$
| Read $\langle expr \rangle$
| Get $\langle expr \rangle$ $\langle \mathbb{N} \rangle$
| Alloc $\langle expr \rangle$ $\langle type \rangle$
| Call $\langle function \rangle$ $\langle expr \rangle$
| Single $\langle expr \rangle$
| Concat $\langle order \rangle$ $\langle expr \rangle$ $\langle expr \rangle$
| Switch $\langle expr \rangle$ $\langle expr \rangle$*
| If $\langle expr \rangle$ $\langle expr \rangle$ $\langle expr \rangle$
| Let $\langle expr \rangle$ $\langle expr \rangle$
| DoWhile $\langle expr \rangle$ $\langle expr \rangle$
| Assume $\langle assumption \rangle$ $\langle expr \rangle$