

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 \quad (\text{point-free version}) \\ \llbracket \text{Let } e_{in} e_{out} \rrbracket(\alpha, \sigma) &\triangleq \llbracket e_{out} \rrbracket(\llbracket e_{in} \rrbracket(\alpha, \sigma)) \quad (\eta\text{-expanded for clarity}) \\ \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_1 \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{Context (InLet } e_{in}) e \rrbracket(\alpha, \sigma) &\triangleq \left\{ \llbracket e \rrbracket(\alpha, \sigma) \mid \exists \alpha', \sigma'. \llbracket e_{in} \rrbracket_v(\alpha', \sigma') = \alpha \right. \\ \llbracket \text{Context (AfterWrite } e_p e_d) e \rrbracket(\alpha, \sigma) &\triangleq \left\{ \llbracket e \rrbracket(\alpha, \sigma) \mid \exists \alpha', \sigma'. \llbracket \text{Write } e_p e_d \rrbracket_s(\alpha', \sigma') = \sigma \right. \\ \llbracket \text{Context (InLoop } e_{in} e_{po}) e \rrbracket(\alpha, \sigma) &\triangleq \left\{ \llbracket e \rrbracket(\alpha, \sigma) \mid \exists \sigma'. (\alpha, \sigma') \in \text{loop_reachable}(e_{in}, e_{po}) \right\} \end{aligned}$$

```
def loop_reachable(e_in, e_po):
    res = {llbracket e_in \rrbracket(\alpha, \sigma) | \alpha \in \text{Value}, \sigma \in \text{State}}
    fixpoint:
        for (\alpha, \sigma) \in res:
            (v_pred, v_out), \sigma' = llbracket e_po \rrbracket(\alpha, \sigma)
            if v_pred:
                res.insert((v_out, \sigma'))
    return res
```

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

Abstract grammar

$\langle \text{basetype} \rangle ::= \text{IntT}$ BoolT	$\langle \text{expr} \rangle ::= \text{Arg } \langle \text{type} \rangle$ Int $\langle \mathbb{N} \rangle$
$\langle \text{type} \rangle ::= \langle \text{basetype} \rangle$ PointerT $\langle \text{type} \rangle$ TupleT $\langle \text{basetype} \rangle^*$	Bool $\langle \text{bool} \rangle$
$\langle \text{bool} \rangle ::= \top$ \perp	Empty
$\langle \text{order} \rangle ::= \text{Parallel}$ Sequential	Add $\langle \text{expr} \rangle \langle \text{expr} \rangle$
$\langle \text{function} \rangle ::= \text{Function } \langle \text{type} \rangle \langle \text{expr} \rangle$	Sub $\langle \text{expr} \rangle \langle \text{expr} \rangle$
$\langle \text{assumption} \rangle ::= \text{InLet } \langle \text{expr} \rangle$ InLoop $\langle \text{expr} \rangle \langle \text{expr} \rangle$ AfterWrite $\langle \text{expr} \rangle \langle \text{expr} \rangle$	Mul $\langle \text{expr} \rangle \langle \text{expr} \rangle$
	LessThan $\langle \text{expr} \rangle \langle \text{expr} \rangle$
	And $\langle \text{expr} \rangle \langle \text{expr} \rangle$
	Or $\langle \text{expr} \rangle \langle \text{expr} \rangle$
	Write $\langle \text{expr} \rangle \langle \text{expr} \rangle$
	Not $\langle \text{expr} \rangle$
	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 $\langle \text{expr} \rangle \langle \text{expr} \rangle$
	Context $\langle \text{assumption} \rangle \langle \text{expr} \rangle$