Denotational semantics

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

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

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 [\![\vec{e}\,[[\![e_{pred}]\!]_v(\alpha,\sigma)]]\!]\,(\alpha, [\![e_{pred}]\!]_s(\alpha,\sigma))\\
[\![\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)) \quad (\eta\text{-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})\ 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)\ 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})\ e]\!](\alpha,\sigma) &\triangleq \begin{cases} [\![e]\!](\alpha,\sigma) & (\alpha,\sigma) \in \text{loop\_reachable}(e_{in},e_{po}) \end{cases}
\end{aligned}
$$

```
def loop_reachable(e_in, e_po):
  res = {[[e_in]](α,σ) | α ∈ Value, σ ∈ State}
  fixpoint:
    for (α,σ) ∈ res:
      (v_pred, v_out), σ' = [[e_po]](α,σ)
      if v_pred:
        res.insert((v_out, σ'))
  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.

$$\frac{\langle e_1, \alpha, \sigma\rangle \Downarrow \langle v_1, \sigma'\rangle \qquad \langle e_2, \alpha, \sigma'\rangle \Downarrow \langle v_2, \sigma''\rangle}{\langle \text{Add } e_1\ e_2, \alpha, \sigma\rangle \Downarrow \langle v_1 + v_2, \sigma''\rangle} \quad \text{(E-ADD)}$$

$$\frac{\langle e_c, \alpha, \sigma\rangle \Downarrow \langle \top, \sigma'\rangle \qquad \langle e_t, \alpha, \sigma'\rangle \Downarrow \langle v_t, \sigma''\rangle}{\langle \text{If } e_c\ e_t\ e_e, \alpha, \sigma\rangle \Downarrow \langle v_t, \sigma''\rangle} \quad \text{(E-IFTRUE)}$$

$$\frac{\langle e_c, \alpha, \sigma\rangle \Downarrow \langle \bot, \sigma'\rangle \qquad \langle e_e, \alpha, \sigma'\rangle \Downarrow \langle v_e, \sigma''\rangle}{\langle \text{If } e_c\ e_t\ e_e, \alpha, \sigma\rangle \Downarrow \langle v_e, \sigma''\rangle} \quad \text{(E-IFFALSE)}$$

$$\frac{\langle e_{pred}, \alpha, \sigma\rangle \Downarrow \langle i, \sigma'\rangle \qquad \langle e_i, \alpha, \sigma'\rangle \Downarrow \langle v, \sigma''\rangle}{\langle \text{Switch } e_{pred}\ (e_1, \ldots, e_n), \alpha, \sigma\rangle \Downarrow \langle v, \sigma''\rangle} \quad \text{(E-SWITCH)}$$

$$\frac{\langle e_{in}, \alpha, \sigma\rangle \Downarrow \langle v_{in}, \sigma'\rangle \qquad \langle e_{out}, v_{in}, \sigma'\rangle \Downarrow \langle v_{out}, \sigma''\rangle}{\langle \text{Let } e_{in}\ e_{out}, \alpha, \sigma\rangle \Downarrow \langle v_{out}, \sigma''\rangle} \quad \text{(E-LET)}$$

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

$$\frac{\langle e, \alpha, \sigma\rangle \Downarrow \langle v, (M, L)\rangle}{\langle \text{Read } e, \alpha, \sigma\rangle \Downarrow \langle M[v], (M, L)\rangle} \quad \text{(E-READ)}$$

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

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

$$\frac{\langle e, \alpha, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \text{Single } e, \alpha, \sigma\rangle \Downarrow \langle [v], \sigma'\rangle} \quad \text{(E-SINGLE)}$$

$$\frac{\langle e_1, \alpha, \sigma\rangle \Downarrow \langle v_1, \sigma'\rangle \qquad \langle e_2, \alpha, \sigma'\rangle \Downarrow \langle v_2, \sigma''\rangle}{\langle \text{Concat Sequential } e_1\ e_2, \alpha, \sigma\rangle \Downarrow \langle v_1 \mathbin{+\!\!+} v_2, \sigma''\rangle} \quad \text{(E-CONCATSEQ)}$$

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

$$\frac{\langle e_{in}, \alpha, \sigma\rangle \Downarrow \langle \alpha', \sigma'\rangle \qquad \langle e_{po}, \alpha', \sigma'\rangle \Downarrow \langle [\top, \alpha''], \sigma''\rangle \qquad \langle \text{DoWhile } e_{in}\ e_{po}, \alpha'', \sigma''\rangle \Downarrow \langle v, \sigma'''\rangle}{\langle \text{DoWhile } e_{in}\ e_{po}, \alpha, \sigma\rangle \Downarrow \langle v, \sigma'''\rangle} \quad \text{(E-DOWHILETRUE)}$$

$$\frac{\langle e, \alpha, \sigma\rangle \Downarrow \langle v, \sigma'\rangle \qquad \exists\, \alpha_2, \sigma_2, \sigma_3.\ \langle e_{in}, \alpha_2, \sigma_2\rangle \Downarrow \langle \alpha, \sigma_3\rangle}{\langle \text{Assume } (\text{InLet } e_{in})\ e, \alpha, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad \text{(E-ASSUMEINLET)}$$

$$\frac{\langle e, \alpha, \sigma\rangle \Downarrow \langle v, \sigma'\rangle \qquad \exists\, \alpha_2, v_2, \sigma_2.\ \langle \text{Write } e_p\ e_d, \alpha_2, \sigma_2\rangle \Downarrow \langle v_2, \sigma_3\rangle}{\langle \text{Assume } (\text{AfterWrite } e_p\ e_d)\ e, \alpha, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad \text{(E-ASSUMEAFTERWRITE)}$$

$$\frac{\langle e, \alpha, \sigma\rangle \Downarrow \langle v, \sigma'\rangle \qquad (\alpha, \sigma) \in \text{loop\_reachable}(e_{in}, e_{out})}{\langle \text{Assume } (\text{InLoop } e_{in}\ e_{po})\ e, \alpha, \sigma\rangle \Downarrow \langle v, \sigma'\rangle} \quad \text{(E-ASSUMEINLOOP)}$$

Abstract grammar

⟨*basetype*⟩ ::= IntT
| BoolT

⟨*type*⟩ ::= ⟨*basetype*⟩
| PointerT ⟨*type*⟩
| TupleT ⟨*basetype*⟩*

⟨*bool*⟩ ::= $\top$
| $\bot$

⟨*order*⟩ ::= Parallel
| Sequential

⟨*function*⟩ ::= Function ⟨*type*⟩ ⟨*expr*⟩

⟨*assumption*⟩ ::= InLet ⟨*expr*⟩
| InLoop ⟨*expr*⟩ ⟨*expr*⟩
| AfterWrite ⟨*expr*⟩ ⟨*expr*⟩

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