1 Representations

1.1 Source Language

statement ::= compound-statement
| `if` `(` expression `)` compound-statement `else` compound-statement
| `while` `(` expression `)` compound-statement
| `for` `(` statement `;` statement `;` statement `)` compound-statement
| qualifier type name `;`
| qualifier type name `=` expression `;`
| `return` expression `;`
| expression `;`

compound-statement ::= `{` statement* `}`

primary-expression ::= identifier
| constant
| string-literal
| `(` expression `)`

postfix-expression ::= primary-expression
| postfix-expression `[` expression `]`
| name `(` argument-list `)`
| postfix-expression `++`
| postfix-expression `--`

unary-expression ::= postfix-expression
| `++` unary-expression
| `--` unary-expression
| unary-operator postfix-expression

multiplicative-expression ::= unary-expression
| multiplicative-expression `*` unary-expression
| multiplicative-expression `/` unary-expression
| multiplicative-expression `%` unary-expression

additive-expression ::= multiplicative-expression
| additive-expression `+` multiplicative-expression
| additive-expression `-` multiplicative-expression

shift-expression ::= additive-expression
| shift-expression `<<` additive-expression
| shift-expression `>>` additive-expression

relational-expression ::= shift-expression
| relational-expression `<` shift-expression
| relational-expression `>` shift-expression

| relational-expression `<=` shift-expression
| relational-expression `>=` shift-expression

equality-expression ::= relational-expression
| equality-expression `==` relational-expression
| equality-expression `!=` relational-expression

bitwise-and-expression ::= equality-expression
| and-expression `&` equality-expression

exclusive-or-expression ::= and-expression
| exclusive-or-expression `|` exclusive-or-expression

bitwise-or-expression ::= exclusive-or-expression
| inclusive-or-expression `|` exclusive-or-expression

logical-and-expression ::= bitwise-or-expression
| logical-and-expression `&&` bitwise-or-expression

logical-or-expression ::= logical-and-expression
| logical-or-expression `||` logical-and-expression

conditional-expression ::= logical-or-expression
| conditional-expression `?` expression `:` conditional-expression

assignment-expression ::= conditional-expression
| unary-expression assignment-operator assignment-expression

expression ::= assignment-expression

parameter-list ::= qualifier type name `,` argument-list
| type-qualifier type name

argument-list ::= expression `,` argument-list
| expression

assignment-operator ::= `=` | `*=` | `/=` | `%=` | `+=` | `-=`

unary-operator ::= `*` | `-` | `!` | `~`

type-qualifier ::= `const` | `volatile`

type-specifier ::= `int` | `unsigned` | `char`

type ::= type-qualifier type-specifier

function-definition ::= type name `(` parameter-list `)` compound-expression

2 Symbolic Execution

2.1 Metavariables

e - source language expressions
c - source language statements
s - symbolic expressions
x - names
v - symbolic values

2.2 Symbolic Expressions

τ - symbolic types
s - symbolic expressions
x, α - symbolic variables
v - symbolic values

$$
\begin{aligned}
s ::=\ & v \mid x \\
\mid\ & s \text{ binop } s \\
\mid\ & \text{unop } s \\
\mid\ & \text{sel}(s, s) \\
\mid\ & \text{upd}(s, s, s) \\
\mid\ & x(s, .., s) \\
\mid\ & s? : s : s \\
\mid\ & \emptyset
\end{aligned}
$$

2.3 Expressions

$$\langle S; v \rangle \Downarrow \langle S; v \rangle \ \textsc{Literal}$$

$$\frac{\langle S; e \rangle \Downarrow S'; s \rangle}{\langle S; \texttt{-}e \rangle \Downarrow \langle S'; -s : \tau(s) \rangle} \ \textsc{Negate}$$

$$\frac{\langle S; e_1 \rangle \Downarrow S_1; s_1 \rangle \qquad \langle S_1; e_2 \rangle \Downarrow S_2; s_2 \rangle \qquad \tau = \max(\tau(s_1), \tau(s_2))}{\langle S; e_1 \ \texttt{+} \ e_2 \rangle \Downarrow \langle S_2; s_1 + s_2 : \tau \rangle} \ \textsc{Add}$$

$$\frac{\langle S; e_1 \rangle \Downarrow \langle S_1; s_1) \rangle \qquad s_1 = \text{sel}(\mu(S_k, s_k)) : \tau \qquad \langle S_1; e_2 \rangle \Downarrow \langle S_2; s_2 \rangle}{\langle S; e_1 \ \texttt{+=} \ e_2 \rangle \Downarrow \langle S_2[\mu \mapsto \text{upd}(\mu(S_2), s_k, s_1 + s_2 : \tau)]; s_2 \rangle} \ \textsc{AssignAdd}$$

$$\frac{\rho(S_1)[x] = (\tau_1, ..\tau_n) \to \tau \qquad \forall i \in 1..n, \langle S_k; e_i \rangle \Downarrow \langle S_{i+1}; s_i \rangle}{\langle S_1; x\texttt{(}e_1\texttt{,}..\texttt{,}e_n\texttt{)} \rangle \Downarrow \langle S_{n+1}; x(s_1 : \tau_1, .., s_n : \tau_n) : \tau \rangle} \ \textsc{FunCall}$$

$$\frac{\langle S; e\rangle \Downarrow \langle S'; s\rangle}{\langle S; (\tau)\ e\rangle \Downarrow \langle S'; s : \tau\rangle}\ \textsc{Cast}$$

2.4 Statements

$$\frac{\langle S; e\rangle \Downarrow \langle S'; s\rangle}{\langle S; e\texttt{;}\rangle \Downarrow \langle S'; \emptyset : \mathrm{u8}\rangle}\ \textsc{Expression}$$

$$\frac{\forall i \in 1..n, \langle S_k; c_i\rangle \Downarrow \langle S_{i+1}; s_{i+1}\rangle}{\langle S_1; \{c_1..c_n\}\rangle \Downarrow \langle S_{n+1}[\rho \mapsto \rho(S_1)]; s_{n+1}\rangle}\ \textsc{CompoundStatement}$$

$$\frac{\begin{array}{c}\langle S; e\rangle \Downarrow \langle S_1; g_1\rangle \qquad g(S) \not\Longrightarrow g_1 \qquad g(S) \not\Longrightarrow \neg g_1 \\ \langle S_1[g \mapsto g(S_1) \wedge g_1]; c_1\rangle \Downarrow \langle S_2; s_2\rangle \\ \langle S_1[g \mapsto g(S_1) \wedge \neg g_1]; c_1\rangle \Downarrow \langle S_3; s_3\rangle \\ S' = \langle (g_1 ? g(S_2) : g(S_3)); (g_1 ? \rho(S_2) : \rho(S_3)); (g_1 ? \mu(S_2) : \mu(S_3))\rangle\end{array}}{\langle S; \texttt{if}\ e\ c_1\ \texttt{else}\ c_2\rangle \Downarrow \langle S'; \emptyset : \mathrm{u8}\rangle}\ \textsc{IfElse}$$

2.5 Memory

$$\frac{x \in \rho(S) \qquad \rho(S)[x] = \alpha : \mathrm{ptr}\ \tau}{\langle S; x\rangle \Downarrow \langle S; \mathrm{sel}(\mu(S), \alpha) : \tau\rangle}\ \textsc{Var}$$

$$\frac{x \in \rho(S) \qquad \rho(S)[x] = \alpha : \tau}{\langle S; \texttt{\&}x\rangle \Downarrow \langle S; \alpha : \tau\rangle}\ \textsc{Ref}$$

$$\frac{\langle S; e\rangle \Downarrow \langle S'; s\rangle \qquad \tau(s) = \mathrm{ptr}\ \tau}{\langle S; \texttt{*}e\rangle \Downarrow \langle S'; \mathrm{sel}(\mu(S'), s) : \tau\rangle}\ \textsc{Sel}$$

$$\frac{\langle S; e_1\rangle \Downarrow \langle S_1; s_1)\rangle \qquad s_1 = \mathrm{sel}(\mu(S_k), s_k) : \tau \qquad \langle S_1; e_2\rangle \Downarrow \langle S_2; s_2\rangle}{\langle S; e_1\ \texttt{=}\ e_2\rangle \Downarrow \langle S_2[\mu \mapsto \mathrm{upd}(\mu(S_2), s_k, s_2 : \tau)]; s_2\rangle}\ \textsc{Upd}$$

$$\frac{x \notin \mathrm{dom}\ (\rho(S)) \qquad \alpha \text{ is fresh}}{\langle S; \tau\ \ x\texttt{;}\rangle \Downarrow \langle S[\mu \mapsto \mathrm{upd}(\mu(S), \alpha, \emptyset : \tau); \rho \mapsto \rho(S)[x \mapsto \alpha : \mathrm{ptr}\ \tau]]; s\rangle}\ \textsc{Declare}$$

$$\frac{x \notin \mathrm{dom}\ (\rho(S)) \qquad \langle S; e\rangle \Downarrow \langle S'; s\rangle \qquad \alpha \text{ is fresh}}{\langle S; \tau\ \ x\ \texttt{=}\ e\texttt{;}\rangle \Downarrow \langle S'[\mu \mapsto \mathrm{upd}(\mu(S'), \alpha, s); \rho \mapsto \rho(S')[x \mapsto \alpha : \mathrm{ptr}\ \tau]]; s\rangle}\ \textsc{DeclareAssign}$$

$$\frac{\langle S; e\rangle \Downarrow \langle S'; s)\rangle \qquad s = \mathrm{sel}(\mu(S_k), s_k) : \tau}{\langle S; \texttt{++}e\rangle \Downarrow \langle S'[\mu \mapsto \mathrm{upd}(\mu(S'), s_k, s+1 : \tau); s+1 : \tau]\rangle} \ \textsc{IncPre}$$

$$\frac{\langle S; e\rangle \Downarrow \langle S'; s)\rangle \qquad s = \mathrm{sel}(\mu(S_k), s_k) : \tau}{\langle S; \texttt{++}e\rangle \Downarrow \langle S'[\mu \mapsto \mathrm{upd}(\mu(S'), s_k, s+1 : \tau); s]\rangle} \ \textsc{IncPost}$$