The PREV programming language

(academic year 2016/17)

Boštjan Slivnik

1 Lexical structure

Programs in the PREV programming language are written in ASCII character set (no additional characters denoting post-alveolar consonants are allowed).

Programs in the PREV programming language consist of the following lexical elements:

- *Literals*:
  - literals of type void: `none`
  - literals of type bool: `true` `false`
  - literals of type char:
    An character with the character code in decimal range 32 . . . 126 (from space to `~`) enclosed in single quotes (`'`).
  - literals of type int:
    A nonempty finite string of digits (`0`. . . `9`) optionally preceded by a sign (`+` or `-`).
  - literals of pointer types: `null`
- *Symbols*:
  `!` `|` `^` `&` `==` `!=` `<=` `>=` `<` `>` `+` `-` `*` `/` `%` `$` `@` `=` `.` `,` `:` `;` `[` `]` `(` `)` `{` `}`
- *Keywords*:
  `arr` `bool` `char` `del` `do` `else` `end` `fun` `if` `int` `new` `ptr` `rec` `then` `typ` `var` `void` `where` `while`
- *Identifiers*:
  A nonempty finite string of letters (`A`. . . `Z` and `a`. . . `z`), digits (`0`. . . `9`), and underscores (`_`) that (a) starts with either a letter or an underscore and (b) is not a keyword.
- *Comments*:
  A string of characters starting with a hash (`#`) and extending to the end of line.
- *White space*:
  Space, horizontal tab (HT), line feed (LF) and carriage return (CR). Line feed alone denotes the end of line within a source file.

Lexical elements should be recognised from left to right using the longest match approach.

2 Syntax structure

The concrete syntax of the PREV programming language is defined by context free grammar with the start symbol *expr* and productions

(*literal*) $expr \longrightarrow literal$
(*unary expression*) $expr \longrightarrow unop\ expr$
(*binary expression*) $expr \longrightarrow expr\ binop\ expr$
(*variable access*) $expr \longrightarrow identifier$
(*function call*) $expr \longrightarrow identifier\texttt{(}[expr\,\{\texttt{,}\,expr\}]\texttt{)}$
(*element access*) $expr \longrightarrow expr\texttt{[}expr\texttt{]}$
(*component access*) $expr \longrightarrow expr\texttt{.}identifier$
(*type cast*) $expr \longrightarrow \texttt{[}type\texttt{]}\,expr$
(*memory allocation*) $expr \longrightarrow \texttt{new}\ type$
(*memory deallocation*) $expr \longrightarrow \texttt{del}\ expr$
(*compound expression*) $expr \longrightarrow \texttt{\{}\ stmt\,\{\texttt{;}\,stmt\}\ \texttt{:}\ expr\ [\texttt{where}\ decl\,\{\texttt{;}\,decl\}]\ \texttt{\}}$
(*enclosed expression*) $expr \longrightarrow \texttt{(}expr\texttt{)}$

(*atomic type*) $type \longrightarrow \texttt{void} \mid \texttt{bool} \mid \texttt{char} \mid \texttt{int}$
(*array type*) $type \longrightarrow \texttt{arr [}expr\texttt{]}\ type$
(*record type*) $type \longrightarrow \texttt{rec (}identifier\texttt{:}type\,\{\texttt{,}\,identifier\texttt{:}type\}\texttt{)}$
(*pointer type*) $type \longrightarrow \texttt{ptr}\ type$
(*named type*) $type \longrightarrow identifier$

(*expression*) $stmt \longrightarrow expr$
(*assignment*) $stmt \longrightarrow expr\texttt{=}expr$
(*conditional*) $stmt \longrightarrow \texttt{if}\ expr\ \texttt{then}\ stmt\,\{\texttt{;}\,stmt\}\ [\texttt{else}\ stmt\,\{\texttt{;}\,stmt\}]\ \texttt{end}$
(*loop*) $stmt \longrightarrow \texttt{while}\ expr\ \texttt{do}\ stmt\,\{\texttt{;}\,stmt\}\ \texttt{end}$

(*type declaration*) $decl \longrightarrow \texttt{typ}\ identifier\texttt{:}type$
(*variable declaration*) $decl \longrightarrow \texttt{var}\ identifier\texttt{:}type$
(*function declaration*) $decl \longrightarrow \texttt{fun}\ identifier\texttt{(}[identifier\texttt{:}type\,\{\texttt{,}\,identifier\texttt{:}type\}]\texttt{)}\texttt{:}type\ [\texttt{=}expr]$

where *literal* denotes any literal, *unop* denotes an unary operator (any of `!`, `+`, `-`, `$` and `@`) and *binop* denotes a binary operator (any of `|`, `^`, `&`, `==`, `!=`, `<=`, `>=`, `<`, `>`, `+`, `-`, `*`, `/` and `%`). In the grammar above, braces typeset as {} enclose sentential forms that can repeated zero or more times, brackets typeset as [] enclose sentential forms that can be present or not while braces and brackets typeset as `{}` and `[]` denote characters that are a part of the program text.

Relational operators are non-associative, all other binary operators are left associative.

The precedence of operators is as follows:

`| ^`	THE LOWEST PRECEDENCE
`&`	
`== != <= >= < >`	
`+ -`	(binary `+` and `-`)
`* / %`	
`! + - $ @ new del` *type-cast*	(unary `+` and `-`)
array-access component-access	THE HIGHEST PRECEDENCE

3 Semantic structure

3.1 Name binding

Let function $[\![\cdot]\!]_{\text{BIND}}$ binds a name to its declaration according to the rules of namespaces and scopes as described below. Hence, the value of function $[\![\cdot]\!]_{\text{BIND}}$ depends on the context of its argument.

Namespaces. There are two kinds of a namespaces:

1. Names of types, functions, variables and parameters belong to one single global namespace.
2. Names of record components belong to record-specific namespaces, i.e., each record defines its own namespace containing names of its components.

Scopes. A new scope is created in two ways:

1. Compound expression

   $$\texttt{\{}\ \textit{stmt}\ \{\texttt{;}\textit{stmt}\}\ \texttt{:}\ \textit{expr}\ [\texttt{where}\ \textit{decl}\ \{\texttt{;}\textit{decl}\}]\ \texttt{\}}$$

   creates a new scope. The scope starts right after `{` and ends just before `}`.

2. Function declaration

   $$\texttt{fun}\ \textit{identifier}\texttt{(}[\textit{identifier}\texttt{:}\textit{type}\ \{\texttt{,}\textit{identifier}\texttt{:}\textit{type}\}]\texttt{)}\texttt{:}\textit{type}\ [\texttt{=}\textit{expr}]$$

   creates a new scope. The name of a function, the types of parameters and the type of a result belong to the outer scope while the names of parameters and the expression denoting the function body belong to the scope created by the function declaration.

All names declared within a given scope are visible in the entire scope unless hidden by a declaration in the nested scope. A name can be declared within the same scope at most once.

3.2 Constant subexpressions

Let $\mathrm{I} = \{(-2^{63}) \ldots (2^{63}-1)\}$. Semantic function

$$[\![\cdot]\!]_{\mathrm{VAL}} : \mathcal{P} \to \mathrm{I}$$

maps phrases of PREV to the integer values they denote. It is defined by the following rules:

$$\frac{\textit{lexeme}(\mathrm{INTEGER}) \in \mathrm{I}}{[\![\mathrm{INTEGER}]\!]_{\mathrm{VAL}} = \textit{lexeme}(\mathrm{INTEGER})} \qquad \frac{[\![\textit{expr}]\!]_{\mathrm{VAL}} = n}{[\![\texttt{(}\textit{expr}\texttt{)}]\!]_{\mathrm{VAL}} = n}$$

$$\frac{[\![\textit{expr}]\!]_{\mathrm{VAL}} = n \quad \mathrm{op} \in \{\texttt{+}, \texttt{-}\}}{[\![\mathrm{op}\ \textit{expr}]\!]_{\mathrm{VAL}} = \mathrm{op}\ n} \qquad \frac{[\![\textit{expr}_1]\!]_{\mathrm{VAL}} = n_1 \quad [\![\textit{expr}_2]\!]_{\mathrm{VAL}} = n_2 \quad \mathrm{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{[\![\textit{expr}_1\ \mathrm{op}\ \textit{expr}_2]\!]_{\mathrm{VAL}} = n_1\ \mathrm{op}\ n_2}$$

In all other cases the value of $[\![\cdot]\!]_{\mathrm{VAL}}$ is undefined (denoted by $\perp$).

3.3 Lvalues

Semantic function

$$[\![\cdot]\!]_{\mathrm{LVAL}} : \mathcal{P} \to \{\mathbf{true}, \mathbf{false}\}$$

marks phrases representing lvalues. It is defined by the following rules:

$$\frac{[\![\mathrm{IDENTIFIER}]\!]_{\mathrm{BIND}} \in \{(\textit{variable declaration}), (\textit{parameter declaration})\ \}}{[\![\mathrm{IDENTIFIER}]\!]_{\mathrm{LVAL}} = \mathbf{true}}$$

$$\frac{[\![\textit{expr}]\!]_{\mathrm{LVAL}} = \mathbf{true}}{[\![\texttt{@}\ \textit{expr}]\!]_{\mathrm{LVAL}} = \mathbf{true}} \qquad \frac{[\![\textit{expr}]\!]_{\mathrm{LVAL}} = \mathbf{true}}{[\![\textit{expr}\,\texttt{[}\textit{expr}'\texttt{]}]\!]_{\mathrm{LVAL}} = \mathbf{true}} \qquad \frac{[\![\textit{expr}]\!]_{LVAL} = \mathbf{true}}{[\![\textit{expr}\,\texttt{.}\,\mathrm{IDENTIFIER}]\!]_{\mathrm{LVAL}} = \mathbf{true}}$$

In all other cases the value of $[\![\cdot]\!]_{\mathrm{LVAL}}$ equals **false**.

3.4 Type system

A set

$$\begin{aligned} T_d = {} & \{\mathbf{void}, \mathbf{bool}, \mathbf{char}, \mathbf{int}\} && \text{(atomic types)} \\ & \cup \{\mathbf{arr}(n \times \tau) \mid n \in I \wedge \tau \in T_d\} && \text{(arrays)} \\ & \cup \{\mathbf{rec}(\tau_1, \ldots, \tau_n) \mid n \in I \wedge \tau_1, \ldots, \tau_n \in T_d\} && \text{(records)} \\ & \cup \{\mathbf{ptr}(\tau) \mid \tau \in T_d\} && \text{(pointers)} \end{aligned}$$

denotes a set of all data types of PREV. A set

$$\begin{aligned} T = {} & T_d && \text{(data types)} \\ & \cup \{(\tau_1, \ldots, \tau_n) \to \tau \mid n \geq 0 \wedge \tau_1, \ldots, \tau_n, \tau \in T_d\} && \text{(functions)} \end{aligned}$$

denotes a set of all types of PREV.

Two types are equal if they share the same structure.

Type expressions. Semantic function

$$[\![\cdot]\!]_{\text{ISTYPE}} : \mathcal{P} \to T_d$$

maps phrases of PREV to types (as it is defined using $[\![\cdot]\!]_{\text{BIND}}$ it depends on the context of its argument). It is defined by the following rules:

$$\frac{}{[\![\texttt{void}]\!]_{\text{ISTYPE}} = \mathbf{void}} \quad \frac{}{[\![\texttt{bool}]\!]_{\text{ISTYPE}} = \mathbf{bool}} \quad \frac{}{[\![\texttt{char}]\!]_{\text{ISTYPE}} = \mathbf{char}} \quad \frac{}{[\![\texttt{int}]\!]_{\text{ISTYPE}} = \mathbf{int}}$$

$$\frac{[\![\mathit{expr}]\!]_{\text{VAL}} = n \quad [\![\mathit{type}]\!]_{\text{ISTYPE}} = \tau}{[\![\texttt{arr[}\mathit{expr}\texttt{]}\ \mathit{type}]\!]_{\text{ISTYPE}} = \mathbf{arr}(n \times \tau)} \quad \frac{[\![\mathit{type}]\!]_{\text{ISTYPE}} = \tau}{[\![\texttt{ptr}\ \mathit{type}]\!]_{\text{ISTYPE}} = \mathbf{ptr}(\tau)}$$

$$\frac{n > 0 \quad [\![\mathit{type}_1]\!]_{\text{ISTYPE}} = \tau_1, \ldots, [\![\mathit{type}_n]\!]_{\text{ISTYPE}} = \tau_n}{[\![\texttt{rec\{}\ \text{IDENTIFER}_1 \texttt{:}\ \mathit{type}_1, \ldots, \text{IDENTIFER}_n \texttt{:}\ \mathit{type}_n\ \texttt{\}}]\!]_{\text{ISTYPE}} = \mathbf{rec}(\tau_1, \ldots, \tau_n)}$$

$$\frac{[\![\text{IDENTIFIER}]\!]_{\text{BIND}} = \texttt{typ}\ \mathit{decl} \quad [\![\mathit{decl}]\!]_{\text{TYPED}} = \tau}{[\![\text{IDENTIFIER}]\!]_{\text{ISTYPE}} = \tau}$$

In all other cases the value of $[\![\cdot]\!]_{\text{ISTYPE}}$ is undefined.

Value expressions and statements. Semantic function

$$[\![\cdot]\!]_{\text{OFTYPE}} : \mathcal{P} \to T_d$$

maps phrases of PREV to types (as it is defined using $[\![\cdot]\!]_{\text{BIND}}$ it depends on the context of its argument). It is defined by the following rules:

$$\frac{}{[\![\texttt{none}]\!]_{\text{OFTYPE}} = \mathbf{void}} \quad \frac{}{[\![\texttt{null}]\!]_{\text{OFTYPE}} = \mathbf{ptr}(\mathbf{void})}$$

$$\frac{}{[\![\text{BOOLEAN}]\!]_{\text{OFTYPE}} = \mathbf{bool}} \quad \frac{}{[\![\text{CHAR}]\!]_{\text{OFTYPE}} = \mathbf{char}} \quad \frac{}{[\![\text{INTEGER}]\!]_{\text{OFTYPE}} = \mathbf{int}}$$

$$\frac{[\![\mathit{expr}]\!]_{\text{OFTYPE}} = \mathbf{bool}}{[\![\texttt{!}\ \mathit{expr}]\!]_{\text{OFTYPE}} = \mathbf{bool}} \quad \frac{[\![\mathit{expr}]\!]_{\text{OFTYPE}} = \mathbf{int} \quad \text{op} \in \{\texttt{+}, \texttt{-}\}}{[\![\text{op expr}]\!]_{\text{OFTYPE}} = \mathbf{int}}$$

$$\frac{[\![\mathit{expr}_1]\!]_{\text{OFTYPE}} = \mathbf{bool} \quad [\![\mathit{expr}_2]\!]_{\text{OFTYPE}} = \mathbf{bool} \quad \text{op} \in \{\texttt{|}, \texttt{\^{}}, \texttt{\&}\}}{[\![\mathit{expr}_1\ \text{op}\ \mathit{expr}_2]\!]_{\text{OFTYPE}} = \mathbf{bool}}$$

$$\frac{[\![expr_1]\!]_{\text{OFTYPE}} = \textbf{int} \quad [\![expr_2]\!]_{\text{OFTYPE}} = \textbf{int} \quad \text{op} \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\}}{[\![expr_1 \text{ op } expr_2]\!]_{\text{OFTYPE}} = \textbf{int}}$$

$$\frac{\begin{array}{c}[\![expr_1]\!]_{\text{OFTYPE}} = \tau \quad [\![expr_2]\!]_{\text{OFTYPE}} = \tau \quad \text{op} \in \{\texttt{==}, \texttt{!=}, \texttt{<}, \texttt{>}, \texttt{<=}, \texttt{>=}\} \\ \tau \in \{\textbf{bool}, \textbf{char}, \textbf{int}\} \cup \{\textbf{ptr}(\tau) \mid \tau \in T_d\}\end{array}}{[\![expr_1 \text{ op } expr_2]\!]_{\text{OFTYPE}} = \textbf{bool}}$$

$$\frac{[\![expr]\!]_{\text{OFTYPE}} = \tau \quad \tau \neq \textbf{void} \quad [\![expr]\!]_{\text{LVAL}} = \textbf{true}}{[\![\texttt{\$}\ expr]\!]_{\text{OFTYPE}} = \textbf{ptr}(\tau)} \qquad \frac{[\![expr]\!]_{\text{OFTYPE}} = \textbf{ptr}(\tau) \quad \tau \neq \textbf{void}}{[\![\texttt{@}\ expr]\!]_{\text{OFTYPE}} = \tau}$$

$$\frac{[\![type]\!]_{\text{ISTYPE}} = \tau \quad \tau \neq \textbf{void}}{[\![\texttt{new}\ type]\!]_{\text{OFTYPE}} = \textbf{ptr}(\tau)} \qquad \frac{[\![expr]\!]_{\text{OFTYPE}} = \textbf{ptr}(\tau) \quad \tau \neq \textbf{void}}{[\![\texttt{del}\ expr]\!]_{\text{OFTYPE}} = \textbf{void}}$$

$$\frac{\begin{array}{c}[\![type]\!]_{\text{ISTYPE}} = \tau_1 \quad [\![expr]\!]_{\text{OFTYPE}} = \tau_2 \\ \langle \tau_1, \tau_2 \rangle \in \{\langle \textbf{void}, \tau \rangle \mid \tau \in T_d\} \cup \\ \qquad \{\langle \textbf{int}, \textbf{int} \rangle, \langle \textbf{int}, \textbf{char} \rangle, \langle \textbf{int}, \textbf{bool} \rangle\} \cup \\ \qquad \{\langle \textbf{ptr}(\tau), \textbf{ptr}(\textbf{void}) \rangle \mid \tau \in T_d\}\end{array}}{[\![\texttt{[}type\texttt{]}\ expr]\!]_{\text{OFTYPE}} = \tau_1}$$

$$\frac{[\![\text{IDENTIFIER}]\!]_{\text{BIND}} = \texttt{var}\ decl \quad [\![decl]\!]_{\text{TYPED}} = \tau}{[\![\text{IDENTIFIER}]\!]_{\text{OFTYPE}} = \tau}$$

$$\frac{\begin{array}{c}n \geq 0 \quad [\![expr_1]\!]_{\text{OFTYPE}} = \tau_1, \ldots, [\![expr_n]\!]_{\text{OFTYPE}} = \tau_n \\ [\![\text{IDENTIFIER}]\!]_{\text{BIND}} = \texttt{fun}\ decl \quad [\![decl]\!]_{\text{TYPED}} = (\tau_1, \ldots, \tau_n) \to \tau\end{array}}{[\![\text{IDENTIFIER}(expr_1, \ldots, expr_n)]\!]_{\text{OFTYPE}} = \tau}$$

$$\frac{[\![expr_1]\!]_{\text{OFTYPE}} = \textbf{arr}(n \times \tau) \quad [\![expr_2]\!]_{\text{OFTYPE}} = \textbf{int}}{[\![expr_1\,\texttt{[}expr_2\texttt{]}]\!]_{\text{OFTYPE}} = \tau}$$

$$\frac{[\![expr]\!]_{\text{OFTYPE}} = \textbf{rec}(\tau_1, \ldots, \tau_n) \quad [\![\text{IDENTIFIER}]\!]_{\text{BIND}} = decl \quad [\![decl]\!]_{\text{TYPED}} = \tau}{[\![expr\,\texttt{.}\,\text{IDENTIFIER}]\!]_{\text{OFTYPE}} = \tau}$$

$$\frac{[\![stmts]\!]_{\text{OFTYPE}} = \textbf{void} \quad [\![expr]\!]_{\text{OFTYPE}} = \tau}{[\![\texttt{\{}stmts\texttt{:}\,expr\texttt{\}}]\!]_{\text{OFTYPE}} = \tau} \qquad \frac{[\![stmts]\!]_{\text{OFTYPE}} = \textbf{void} \quad [\![expr]\!]_{\text{OFTYPE}} = \tau}{[\![\texttt{\{}stmts\texttt{:}\,expr\ \texttt{where}\ decls\texttt{\}}]\!]_{\text{OFTYPE}} = \tau}$$

$$\frac{[\![expr]\!]_{\text{OFTYPE}} = \tau}{[\![\texttt{(}expr\texttt{)}]\!]_{\text{OFTYPE}} = \tau}$$

$$\frac{[\![expr]\!]_{\text{OFTYPE}} = \textbf{void} \quad expr = stmt}{[\![stmt]\!]_{\text{OFTYPE}} = \textbf{void}}$$

$$\frac{[\![expr_1]\!]_{\text{OFTYPE}} = \tau \quad [\![expr_2]\!]_{\text{OFTYPE}} = \tau \quad [\![expr_1]\!]_{\text{LVAL}} = \textbf{true}}{[\![expr_1\texttt{=}expr_2]\!]_{\text{OFTYPE}} = \textbf{void}}$$

$$\frac{[\![expr]\!]_{\text{OFTYPE}} = \textbf{bool} \quad [\![stmts]\!]_{\text{OFTYPE}} = \textbf{void}}{[\![\texttt{if}\ expr\ \texttt{then}\ stmts\ \texttt{end}]\!]_{\text{OFTYPE}} = \textbf{void}}$$

$$\frac{[\![expr]\!]_{\text{OFTYPE}} = \textbf{bool} \quad [\![stmts_1]\!]_{\text{OFTYPE}} = \textbf{void} \quad [\![stmts_2]\!]_{\text{OFTYPE}} = \textbf{void}}{[\![\texttt{if } expr \texttt{ then } stmts_1 \texttt{ else } stmts_2 \texttt{ end}]\!]_{\text{OFTYPE}} = \textbf{void}}$$

$$\frac{[\![expr]\!]_{\text{OFTYPE}} = \textbf{bool} \quad [\![stmts]\!]_{\text{OFTYPE}} = \textbf{void}}{[\![\texttt{while } expr \texttt{ do } stmts \texttt{ end}]\!]_{\text{OFTYPE}} = \textbf{void}}$$

$$\frac{[\![stmt_1]\!]_{\text{OFTYPE}} = \textbf{void}, \ldots, [\![stmt_n]\!]_{\text{OFTYPE}} = \textbf{void}}{[\![stmt_1 \texttt{;} \ldots \texttt{;} stmt_n]\!]_{\text{OFTYPE}} = \textbf{void}}$$

In all other cases the value of $[\![\cdot]\!]_{\text{OFTYPE}}$ is undefined.

Declarations. Semantic function

$$[\![\cdot]\!]_{\text{TYPED}} \colon \mathcal{P} \to T$$

maps phrases of PREV to types (as it is defined using $[\![\cdot]\!]_{\text{BIND}}$ it depends on the context of its argument). It is defined by the following rules:

$$\frac{[\![type]\!]_{\text{ISTYPE}} = \tau}{[\![\text{IDENTIFIER} \texttt{:} type]\!]_{\text{TYPED}} = \tau}$$

$$\frac{\begin{array}{c} n \geq 0 \quad [\![decl_1]\!]_{\text{TYPED}} = \tau_1, \ldots, [\![decl_n]\!]_{\text{TYPED}} = \tau_n \quad [\![type]\!]_{\text{ISTYPE}} = \tau \\ \tau_1, \ldots, \tau_n, \tau \in \{\textbf{void}, \textbf{bool}, \textbf{char}, \textbf{int}\} \cup \{\textbf{ptr}(\tau) \mid \tau \in T_d\} \end{array}}{[\![\text{IDENTIFIER}\texttt{(}decl_1, \ldots, decl_n\texttt{)} \texttt{:} type]\!]_{\text{TYPED}} = (\tau_1, \ldots, \tau_n) \to \tau}$$

In all other cases the value of $[\![\cdot]\!]_{\text{TYPED}}$ is undefined.

4 Operational semantics

Operational semantics is described by semantic functions

$$[\![\cdot]\!]_{\text{ADDR}} \colon \mathcal{P} \times M \to \mathcal{I} \times M, \quad [\![\cdot]\!]_{\text{EXPR}} \colon \mathcal{P} \times M \to \mathcal{I} \times M, \quad \text{and} \quad [\![\cdot]\!]_{\text{STMT}} \colon \mathcal{P} \times M \to M$$

where $\mathcal{P}$ denotes the set of phrases of PREV, $\mathcal{I}$ denotes the set of 64-bit integers, and M denotes possible states of the memory. Unary operators (except `$` and `@`) and binary operators perform 64-bit signed operations.

$$\frac{}{[\![\texttt{none}]\!]^M_{\text{EXPR}} = \langle 0, M\rangle} \quad \frac{}{[\![\texttt{null}]\!]^M_{\text{EXPR}} = \langle 0, M\rangle} \quad \frac{}{[\![\texttt{true}]\!]^M_{\text{EXPR}} = \langle 1, M\rangle} \quad \frac{}{[\![\texttt{false}]\!]^M_{\text{EXPR}} = \langle 0, M\rangle}$$

$$\frac{}{[\![int]\!]^M_{\text{EXPR}} = \langle int, M\rangle} \quad \frac{}{[\![\texttt{'}char\texttt{'}]\!]^M_{\text{EXPR}} = \langle \text{ASCII}(char), M\rangle}$$

$$\frac{[\![expr]\!]^M_{\text{EXPR}} = \langle val, M'\rangle \quad \text{op} \in \{\texttt{!}, \texttt{+}, \texttt{-}\}}{[\![\text{op } expr]\!]^M_{\text{EXPR}} = \langle \text{op } val, M'\rangle}$$

$$\frac{\begin{array}{c} [\![expr_1]\!]^M_{\text{EXPR}} = \langle val_1, M'\rangle \quad [\![expr_2]\!]^{M'}_{\text{EXPR}} = \langle val_2, M''\rangle \\ \text{op} \in \{\texttt{|}, \texttt{\^{}}, \texttt{\&}, \texttt{==}, \texttt{!=}, \texttt{<=}, \texttt{>=}, \texttt{<}, \texttt{>}, \texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}, \texttt{\%}\} \end{array}}{[\![expr_1 \text{ op } expr_2]\!]^M_{\text{EXPR}} = \langle val_1 \ \ op \ \ val_2, M''\rangle}$$

$$\frac{[\![expr]\!]^M_{\text{ADDR}} = \langle addr, M'\rangle}{[\![\texttt{\$}expr]\!]^M_{\text{EXPR}} = \langle addr, M'\rangle} \quad \frac{[\![expr]\!]^M_{\text{ADDR}} = \langle addr, M'\rangle}{[\![\texttt{@}expr]\!]^M_{\text{EXPR}} = \langle M'[addr], M'\rangle}$$

$$\frac{[\![expr]\!]^M_{\text{EXPR}} = \langle val, M' \rangle}{[\![\texttt{[}type\texttt{]}\ expr]\!]^M_{\text{EXPR}} = \langle val, M' \rangle} \qquad \frac{[\![expr]\!]^M_{\text{EXPR}} = \langle val, M' \rangle}{[\![\texttt{(}expr\texttt{)}]\!]^M_{\text{EXPR}} = \langle val, M' \rangle}$$

$$\frac{\text{addr}(varname) = addr}{[\![varname]\!]^M_{\text{ADDR}} = \langle addr, M \rangle} \qquad \frac{\text{addr}(varname) = addr}{[\![varname]\!]^M_{\text{EXPR}} = \langle M[addr], M \rangle}$$

$$\frac{[\![expr_1]\!]^{M_0}_{\text{EXPR}} = \langle val_0, M_1 \rangle, \ldots, [\![expr_n]\!]^{M_{n-1}}_{\text{EXPR}} = \langle val_n, M_n \rangle}{[\![funname\texttt{(}expr_1\texttt{,} \ldots \texttt{,}expr_n\texttt{)}]\!]^{M_0}_{\text{EXPR}} = \langle funname(val_1, \ldots, val_n), M_n \rangle}$$

$$\frac{[\![expr_1]\!]^M_{\text{ADDR}} = \langle addr, M' \rangle \quad [\![expr_2]\!]^{M'}_{\text{EXPR}} = \langle val, M'' \rangle}{[\![expr_1\texttt{[}expr_2\texttt{]}]\!]^M_{\text{ADDR}} = \langle addr + val \cdot element\text{-}size, M'' \rangle}$$

$$\frac{[\![expr]\!]^M_{\text{ADDR}} = \langle addr, M' \rangle \quad \text{addr}(compname) = offset}{[\![expr\,.\,compname]\!]^M_{\text{ADDR}} = \langle addr + offset, M' \rangle}$$

$$\frac{}{[\![\texttt{new}\ type]\!]^M_{\text{EXPR}} = \langle \text{malloc}(type\text{-}size), M \rangle} \qquad \frac{[\![expr]\!]^M_{\text{EXPR}} = \langle addr, M' \rangle}{[\![\texttt{del}\ expr]\!]^M_{\text{EXPR}} = \langle \text{free}(addr), M' \rangle}$$

$$\frac{[\![stmt_1]\!]^{M_0}_{\text{STMT}} = M_1, \ldots, [\![stmt_n]\!]^{M_{n-1}}_{\text{STMT}} = M_n \quad [expr]^{M_n}_{\text{EXPR}} = \langle val, M_{n+1} \rangle}{[\![\texttt{\{}stmt_1\texttt{;} \ldots \texttt{;}stmt_n\texttt{:}expr[\texttt{where}\ decls]\texttt{\}}]\!]^M_{\text{EXPR}} = \langle val, M_{n+1} \rangle}$$

$$\frac{[\![expr]\!]^M_{\text{EXPR}} = \langle val, M' \rangle}{[\![expr]\!]^M_{\text{STMT}} = M'}$$

$$\frac{\begin{array}{c}[\![expr_1]\!]^M_{\text{ADDR}} = \langle addr, M' \rangle \quad [\![expr_2]\!]^{M'}_{\text{EXPR}} = \langle val, M'' \rangle \\ M''' = M'' \text{ but } M'''[addr] = val)\end{array}}{[\![expr_1\texttt{=}expr_2]\!]^M_{\text{STMT}} = M'''}$$

$$\frac{[\![expr]\!]^M_{\text{EXPR}} = \langle \texttt{true}, M' \rangle \quad [stmt_1]^{M'}_{\text{STMT}} = M''}{[\![\texttt{if}\ expr\ \texttt{then}\ stmt_1\ [\texttt{else}\ stmt_2]\ \texttt{end}]\!]^M_{\text{STMT}} = M''}$$

$$\frac{[\![expr]\!]^M_{\text{EXPR}} = \langle \texttt{false}, M' \rangle \quad [stmt_2]^{M'}_{\text{STMT}} = M''}{[\![\texttt{if}\ expr\ \texttt{then}\ stmt_1\ \texttt{else}\ stmt_2\ \texttt{end}]\!]^M_{\text{STMT}} = M''}$$

$$\frac{[\![expr]\!]^M_{\text{EXPR}} = \langle \texttt{true}, M' \rangle \quad [stmt]^{M'}_{\text{STMT}} = M''}{[\![\texttt{while}\ expr\ \texttt{do}\ stmt\ \texttt{end}]\!]^M_{\text{STMT}} = [\![\texttt{while}\ expr\ \texttt{do}\ stmt\ \texttt{end}]\!]^{M''}_{\text{STMT}}}$$

$$\frac{[\![expr]\!]^M_{\text{EXPR}} = \langle \texttt{false}, M' \rangle}{[\![\texttt{while}\ expr\ \texttt{do}\ stmt\ \texttt{end}]\!]^M_{\text{STMT}} = M'}$$

Function addr returns the address or the offset of a variable, a parameter or a record component, depending on its context. Functions malloc and free are internal functions with labels `malloc` and `free`.