

<i>ident, x, y, y_p, -</i>	subscript p is for pointers
<i>impl_const</i>	implementation-defined constant
<i>mem_int</i>	memory integer value
<i>member</i>	C struct/union member name
	Ott-hack, ignore (annotations)
<i>nat</i>	OCaml arbitrary-width natural number
<i>n, i</i>	index variables
<i>mem_ptr</i>	abstract pointer value
<i>mem_val</i>	abstract memory value
	Ott-hack, ignore (locations)
<i>mem_iv_c</i>	OCaml type for memory constraints on integer values
<i>UB_name</i>	undefined behaviour
<i>string</i>	OCaml string
	Ott-hack, ignore (OCaml type variable TY)
	Ott-hack, ignore (Symbol.prefix)
<i>mem_order, -</i>	OCaml type for memory order
<i>linux_mem_order</i>	OCaml type for Linux memory order
	Ott-hack, ignore (OCaml type variable bt)
<i>logical_val</i>	logical values (to be specified)

$Sctypes_t, \tau$	$::=$	C type
	τ^*	pointer to type τ
tag	$::=$	OCaml type for struct/union tag
	$ident$	
$\beta, -$	$::=$	base types
	unit	unit
	bool	boolean
	integer	integer
	real	rational numbers?
	loc	location
	array β	array
	$[\beta]$	list
	$\beta_1 \times \dots \times \beta_n$	tuple
	struct tag	struct
	$\{\beta\}$	set
	opt (β)	option
	$\beta_1, \dots, \beta_n \rightarrow \beta$	parameter types
	to_base (τ) M	of a C type
$binop$	$::=$	binary operators
	+	addition
	-	subtraction
	*	multiplication
	/	division
	rem_t	modulus
	rem_f	remainder
	^	exponentiation
	=	equality, defined both for integer and C types

		>	greater than
		<	less than
		>=	greater than or equal to
		<=	less than or equal to
		/\	conjunction
		\/	disjunction
<i>object_value</i>	::=		C object values (inhabitants of object types), which can be read/stored
		<i>mem_int</i>	integer value
		<i>mem_ptr</i>	pointer value
		array ($\overline{loaded_value_i}^i$)	C array value
		(struct <i>ident</i>) { $\overline{.member_i:\tau_i = mem_val_i}^i$ }	C struct value
		(union <i>ident</i>) { <i>.member</i> = <i>mem_val</i> }	C union value
<i>smt_object_value</i>	::=		like above, but can be embedded into the SMT value grammar
		<i>mem_int</i>	integer value
		<i>mem_ptr</i>	pointer value
		(struct <i>ident</i>) { $\overline{.member_i:\tau_i = mem_val_i}^i$ }	C struct value
		(union <i>ident</i>) { <i>.member</i> = <i>mem_val</i> }	C union value
<i>loaded_value</i>	::=		potentially unspecified C object values
		specified <i>object_value</i>	specified loaded value
<i>smt_loaded_value</i>	::=		like above, but can be embedded into SMT value grammar
		specified <i>smt_object_value</i>	specified loaded value
<i>value</i>	::=		Core values
		<i>object_value</i>	C object value
		<i>loaded_value</i>	loaded C object value
		Unit	unit

		True	boolean true
		False	boolean false
		$\beta[\overline{value_i}^i]$	list
		$(\overline{value_i}^i)$	tuple
<i>smt_value</i>	::=		like above, but can be embeded into SMT value grammar
		<i>smt_object_value</i>	C object value
		<i>smt_loaded_value</i>	loaded C object value
		Unit	unit
		True	boolean true
		False	boolean false
		$\beta[\overline{smt_value_i}^i]$	list
		$(\overline{smt_value_i}^i)$	tuple
<i>ctor_val</i>	::=		data constructors
		Nil β	empty list
		Cons	list cons
		Tuple	tuple
		Array	C array
		Specified	non-unspecified loaded value
<i>smt_ctor_val</i>	::=		like above, but can be embeded into SMT value grammar
		Nil β	empty list
		Cons	list cons
		Tuple	tuple
		Specified	non-unspecified loaded value
<i>ctor_expr</i>	::=		data constructors
		Ivmax	max integer value
		Ivmin	min integer value

		<code>Ivsizeof</code>	sizeof value
		<code>Ivalignof</code>	alignof value
		<code>IvCOMPL</code>	bitwise complement
		<code>IvAND</code>	bitwise AND
		<code>IvOR</code>	bitwise OR
		<code>IvXOR</code>	bitwise XOR
		<code>Fvfromint</code>	cast integer to floating value
		<code>Ivfromfloat</code>	cast floating to integer value
<i>name</i>	::=		
		<i>ident</i>	Core identifier
		<i>impl_const</i>	implementation-defined constant
<i>pval</i>	::=		pure values
		<i>ident</i>	Core identifier
		<i>impl_const</i>	implementation-defined constant
		<i>value</i>	Core values
		<code>constrained</code> ($\overline{mem_iv_c_i, pval_i}^i$)	constrained value
		<code>error</code> (<i>string</i> , <i>pval</i>)	impl-defined static error
		<code>ctor_val</code> ($\overline{pval_i}^i$)	data constructor application
		<code>(struct</code> <i>ident</i>) { $\overline{.member_i = pval_i}^i$ }	C struct expression
		<code>(union</code> <i>ident</i>) { $\overline{.member = pval}$ }	C union expression
<i>smt_pval</i>	::=		like above, but can be embeded into SMT value grammar
		<i>ident</i>	Core identifier
		<i>impl_const</i>	implementation-defined constant
		<i>smt_value</i>	Core values
		<code>constrained</code> ($\overline{mem_iv_c_i, smt_pval_i}^i$)	constrained value
		<code>error</code> (<i>string</i> , <i>smt_pval</i>)	impl-defined static error
		<code>smt_ctor_val</code> ($\overline{smt_pval_i}^i$)	data constructor application

	$\begin{array}{ l} (\text{struct } ident)\{\overline{.member_i = smt_pval_i}^i\} \\ (\text{union } ident)\{.member = smt_pval\} \end{array}$	<p>C struct expression</p> <p>C union expression</p>
<i>pexpr</i>	$\begin{array}{ l} ::= \\ \text{ } pval \\ \text{ } ctor_expr(\overline{pval_i}^i) \\ \text{ } array_shift(pval_1, \tau, pval_2) \\ \text{ } member_shift(pval, ident, member) \\ \text{ } not(pval) \\ \text{ } pval_1 \text{ } binop \text{ } pval_2 \\ \text{ } memberof(ident, member, pval) \\ \text{ } name(pval_1, \dots, pval_n) \\ \text{ } assert_undef(pval, UB_name) \\ \text{ } bool_to_integer(pval) \\ \text{ } conv_int(\tau, pval) \\ \text{ } wrapI(\tau, pval) \end{array}$	<p>pure expressions</p> <p>pure values</p> <p>data constructor application</p> <p>pointer array shift</p> <p>pointer struct/union member shift</p> <p>boolean not</p> <p>binary operations</p> <p>C struct/union member access</p> <p>pure function call</p>
<i>tpval</i>	$\begin{array}{ l} ::= \\ \text{ } undef \text{ } UB_name \\ \text{ } done \text{ } pval \end{array}$	<p>top-level pure values</p> <p>undefined behaviour</p> <p>pure done</p>
<i>ident_opt_β</i>	$\begin{array}{ l} ::= \\ \text{ } _:\beta \\ \text{ } ident:\beta \end{array}$	<p>type annotated optional identifier</p>
<i>pattern</i>	$\begin{array}{ l} ::= \\ \text{ } ident_opt_β \\ \text{ } ctor_val(\overline{pattern_i}^i) \end{array}$	
<i>ident_or_pattern</i>	$::=$	

	 	<i>ident</i> <i>pattern</i>	
<i>tpexpr</i>	::= 	<i>tpval</i> case <i>pval</i> of $\overline{\text{pattern}_i \Rightarrow \text{tpexpr}_i}^i$ end let <i>ident_or_pattern</i> = <i>pexpr</i> in <i>tpexpr</i> if <i>pval</i> then <i>tpexpr</i> ₁ else <i>tpexpr</i> ₂ [<i>C/C'</i>] <i>tpexpr</i>	top-level pure expressions top-level pure values pattern matching pure let pure if M simul-sub all vars in <i>C</i> for all vars in <i>C'</i> in <i>tpexpr</i>
<i>m_kill_kind</i>	::= 	dynamic static τ	
<i>bool</i> , $-$::= 	true false	OCaml booleans
<i>int</i> , $-$::= 	<i>i</i>	OCaml fixed-width integer literal integer
<i>mem_action</i>	::= 	create (<i>pval</i> , τ) create_readonly (<i>pval</i> ₁ , τ , <i>pval</i> ₂) alloc (<i>pval</i> ₁ , <i>pval</i> ₂) kill (<i>m_kill_kind</i> , <i>pval</i>) store (<i>bool</i> , τ , <i>pval</i> ₁ , <i>pval</i> ₂ , <i>mem_order</i>) load (τ , <i>pval</i> , <i>mem_order</i>) rmw (τ , <i>pval</i> ₁ , <i>pval</i> ₂ , <i>pval</i> ₃ , <i>mem_order</i> ₁ , <i>mem_order</i> ₂) fence (<i>mem_order</i>)	memory actions true means store is locking

	$\text{cmp_exch_strong}(\tau, pval_1, pval_2, pval_3, mem_order_1, mem_order_2)$ $\text{cmp_exch_weak}(\tau, pval_1, pval_2, pval_3, mem_order_1, mem_order_2)$ $\text{linux_fence}(linux_mem_order)$ $\text{linux_load}(\tau, pval, linux_mem_order)$ $\text{linux_store}(\tau, pval_1, pval_2, linux_mem_order)$ $\text{linux_rmw}(\tau, pval_1, pval_2, linux_mem_order)$	
<i>polarity</i>	$::=$ Pos Neg	polarities for memory actions sequenced by let weak and let strong only sequenced by let strong
<i>pol_mem_action</i>	$::=$ $\text{polarity } mem_action$	memory actions with polarity
<i>mem_op</i>	$::=$ $pval_1 == pval_2$ $pval_1 \neq pval_2$ $pval_1 < pval_2$ $pval_1 > pval_2$ $pval_1 \leq pval_2$ $pval_1 \geq pval_2$ $pval_1 -_{\tau} pval_2$ $\text{intFromPtr}(\tau_1, \tau_2, pval)$ $\text{ptrFromInt}(\tau_1, \tau_2, pval)$ $\text{ptrValidForDeref}(\tau, pval)$ $\text{ptrWellAligned}(\tau, pval)$ $\text{ptrArrayShift}(pval_1, \tau, pval_2)$ $\text{memcpy}(pval_1, pval_2, pval_3)$ $\text{memcmp}(pval_1, pval_2, pval_3)$ $\text{realloc}(pval_1, pval_2, pval_3)$	operations involving the memory state pointer equality comparison pointer inequality comparison pointer less-than comparison pointer greater-than comparison pointer less-than comparison pointer greater-than comparison pointer subtraction cast of pointer value to integer value cast of integer value to pointer value dereferencing validity predicate

		va_start ($pval_1, pval_2$)	
		va_copy ($pval$)	
		va_arg ($pval, \tau$)	
		va_end ($pval$)	
$tval$::=		(effectful) top-level values
		done $pval$	end of top-level expression
		undef UB_name	undefined behaviour
seq_expr	::=		sequential (effectful) expressions
		$pval$	pure values
		ccall ($\tau, pval, \overline{pval_i^i}$)	C function call
		pcall ($name, \overline{pval_i^i}$)	procedure call
seq_texpr	::=		sequential top-level (effectful) expressions
		$tval$	(effectful) top-level values
		run $ident\ pval_1, \dots, pval_n$	run from label
		nd ($pval_1, \dots, pval_n$)	nondeterministic choice
		let $ident_or_pattern = seq_expr$ in $texpr$	pure sequencing
		case $pval$ with $\overline{pattern_i \Rightarrow texpr_i^i}$ end	pattern matching
		if $pval$ then $texpr_1$ else $texpr_2$	conditional
		bound [int] (is_texpr)	limit scope of indet seq behaviour, absent at runtime
is_expr	::=		indet seq (effectful) expressions
		memop (mem_op)	pointer op involving memory
		pol_mem_action	memory action
		unseq ($texpr_1, \dots, texpr_n$)	unsequenced expressions
is_texpr	::=		indet seq top-level (effectful) expressions
		let weak $pattern = is_expr$ in μ_texpr_aux	weak sequencing

		<code>let strong <i>ident_or_pattern</i> = <i>is_expr</i> in mu_texpr_aux</code>	strong sequencing
<i>texpr</i>	::=		top-level (effectful) expressions
		<i>seq_texpr</i>	sequential (effectful) expressions
		<i>is_texpr</i>	indet seq (effectful) expressions
<i>terminals</i>	::=		
		λ	
		\longrightarrow	
		\rightarrow	
		\rightsquigarrow	
		\Rightarrow	
		\Leftarrow	
		\vdash	
		\in	
		Π	
		\forall	
		\dashv	
		\supset	
		Σ	
		\exists	
		\star	
		\times	
		\wedge	
		\bigwedge	
		\neg	
		$=$	
		\neq	
		\leq	
		\geq	

		&	
		.	
		+ _{ptr}	
		↦	
		*	
		::	
		✓	
		:	
		.	
		.	
		>>	
		::	
z	::=		OCaml arbitrary-width integer
		i	M literal integer
		<code>of_mem_int(<i>mem_int</i>)</code>	M
		<code>to_int(τ)</code>	M size of a C type
		<code>ptr_size</code>	M size of a pointer
\mathbb{Q}	::=		OCaml type for rational numbers
		$\frac{int_1}{int_2}$	
lit	::=		
		<i>ident</i>	
		<code>unit</code>	
		<i>bool</i>	
		z	
		\mathbb{Q}	

$bool_op$	$::=$	$\neg term$ $term_1 = term_2$ $\bigwedge (\overline{term_i})^i$
$arith_op$	$::=$	$term_1 \times term_2$
$list_op$	$::=$	nil $tl\ term$ $term^{(int)}$
$tuple_op$	$::=$	$(\overline{term_i})^i$ $term^{(int)}$
$pointer_op$	$::=$	$of_mem_ptr\ mem_ptr$ $term_1 +_{ptr} term_2$
$option_op$	$::=$	$none\ BT_t$ $some\ term$
$array_op$	$::=$	$term_1[term_2]$
$param_op$	$::=$	$term(term_1, .., term_n)$

<i>struct_op</i>	$::=$ <i>term.member</i>		
<i>ct_pred</i>	$::=$ representable ($\tau, term$) alignedI ($term_1, term_2$)		
<i>term, _</i>	$::=$ <i>lit</i> <i>arith_op</i> <i>bool_op</i> <i>tuple_op</i> <i>struct_op</i> <i>pointer_op</i> <i>list_op</i> <i>array_op</i> <i>ct_pred</i> <i>option_op</i> <i>param_op</i> (<i>term</i>) [<i>term</i> ₁ / <i>ident</i>] <i>term</i> ₂ <i>smt_pval</i> <i>resource</i>	S M M	parentheses substitute <i>term</i> ₁ for <i>ident</i> in <i>term</i> ₂ can be embeded into the SMT value grammar
<i>terms</i>	$::=$ [<i>term</i> ₁ , ..., <i>term</i> _{<i>n</i>}]		non-empty list of terms
<i>predicate_name</i>	$::=$ <i>Sctypes_t</i> <i>string</i>		names of predicates C type arbitrary

$init,$	$::=$ $\quad \quad \checkmark$ $\quad \quad \times$	initialisation status initialised uninitialised
$predicate$	$::=$ $\quad \quad terms_1 \mathbb{Q} \stackrel{init}{\mapsto} predicate_name terms_2$	arbitrary predicate
$resource$	$::=$ $\quad \quad predicate$	
$spine_elem$	$::=$ $\quad \quad pval$ $\quad \quad logical_val$ $\quad \quad resource$	spine element pure value logical variable resource
arg	$::=$ $\quad \quad \Pi ident:\beta. arg$ $\quad \quad \forall ident:\beta. arg$ $\quad \quad resource \multimap arg$ $\quad \quad term \supset arg$ $\quad \quad ret$ $\quad \quad [spine_elem/ident]arg$	argument types M
$ret, _$	$::=$ $\quad \quad \Sigma ident:\beta. ret$ $\quad \quad \exists ident:\beta. ret$ $\quad \quad resource \star ret$ $\quad \quad term \wedge ret$ $\quad \quad \mathbf{I}$	return types

\mathcal{C}	$::=$ $\mid \cdot$ $\mid \mathcal{C}, ident:BT_t$ $\mid \mathcal{C}, \mathcal{C}'$ $\mid \text{fresh}(\mathcal{C})$	computational var env M identical context except with fresh variable names
\mathcal{L}	$::=$ $\mid \cdot$ $\mid \mathcal{L}, ident$	logical var env
Φ	$::=$ $\mid \cdot$ $\mid \Phi, term$	constraints env
\mathcal{R}	$::=$ $\mid \cdot$ $\mid \mathcal{R}, resource$	resources env
<i>formula</i>	$::=$ $\mid judgement$ $\mid \text{smt}(\Phi \Rightarrow term)$ $\mid ident:\beta \in \mathcal{C}$ $\mid \overline{ident:\text{struct tag} \ \& \ member_i:\tau_i}^i \in \text{Globals}$ $\mid \overline{\mathcal{C}_i; \mathcal{L}_i; \Phi_i \vdash mem_val_i \Rightarrow \text{mem } y_i:\beta_i. term_i}^i$ $\mid \overline{\mathcal{C}_i; \mathcal{L}_i; \Phi_i \vdash pval_i \Rightarrow ident_i:\beta_i. term_i}^i$ $\mid \overline{pattern_i:\beta_i \rightsquigarrow \mathcal{C}_i}^i$ $\mid \overline{\mathcal{C}_i; \mathcal{L}_i; \Phi_i \vdash tpepr_i \Leftarrow y_i:\beta_i. term_i}^i$ $\mid \mathcal{L} \vdash logical_val:\beta$	dependent on memory object model
<i>object_value_jtype</i>	$::=$	

		$\mathcal{C}; \mathcal{L}; \Phi \vdash \text{object_value} \Rightarrow \text{obj ident}:\beta. \text{term}$
$pval_jtype$	$::=$	$\mathcal{C}; \mathcal{L}; \Phi \vdash pval \Rightarrow \text{ident}:\beta. \text{term}$
$spine_jtype$	$::=$	$\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash \overline{\text{spine_elem}_i}^i :: \text{arg} \gg \text{ret}$
$pexpr_jtype$	$::=$	$\mathcal{C}; \mathcal{L}; \Phi \vdash pexpr \Rightarrow \text{ident}:\beta. \text{term}$
$pattern_jtype$	$::=$	$pattern:\beta \rightsquigarrow \mathcal{C}$ $\text{ident_or_pattern}:\beta \rightsquigarrow \mathcal{C}$
$tpval_jtype$	$::=$	$\mathcal{C}; \mathcal{L}; \Phi \vdash tpval \Leftarrow \text{ident}:\beta. \text{term}$
$tpexpr_jtype$	$::=$	$\mathcal{C}; \mathcal{L}; \Phi \vdash tpexpr \Leftarrow \text{ident}:\beta. \text{term}$
$action_jtype$	$::=$	$\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash \text{mem_action} \Rightarrow \text{ret}$ $\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash \text{seq_expr} \Rightarrow \text{ret}$ $\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash \text{is_expr} \Rightarrow \text{ret}$
$judgement$	$::=$	$\text{object_value_jtype}$ $pval_jtype$ $spine_jtype$

		<i>pexpr_jtype</i>
		<i>pattern_jtype</i>
		<i>tpval_jtype</i>
		<i>tpexpr_jtype</i>
		<i>action_jtype</i>
<i>user_syntax</i>	::=	
		<i>ident</i>
		<i>impl_const</i>
		<i>mem_int</i>
		<i>member</i>
		<i>nat</i>
		<i>n</i>
		<i>mem_ptr</i>
		<i>mem_val</i>
		<i>mem_iv_c</i>
		<i>UB_name</i>
		<i>string</i>
		<i>mem_order</i>
		<i>linux_mem_order</i>
		<i>logical_val</i>
		<i>Sctypes_t</i>
		<i>tag</i>
		β
		<i>binop</i>

- | *ident*
- | τ
- | *ident*
- | *object_value*
- | *smt_object_value*
- | *loaded_value*
- | *smt_loaded_value*
- | β
- | *value*
- | *smt_value*
- | *ctor_val*
- | *smt_ctor_val*
- | *ctor_expr*
- | τ
- | *name*
- | *pval*
- | *smt_pval*
- | *pval*
- | *smt_pval*
- | *pexpr*
- | *pexpr*
- | *tpval*
- | *tpval*
- | *ident_opt_β*
- | *pattern*
- | *pattern*
- | *ident_or_pattern*
- | *tpexpr*
- | *tpexpr*

- | *m_kill_kind*
- | *bool*
- | *int*
- | *mem_action*
- | *mem_action*
- | *polarity*
- | *pol_mem_action*
- | *mem_op*
- | *tval*
- | *tval*
- | *seq_expr*
- | *seq_expr*
- | *seq_texpr*
- | *seq_texpr*
- | *is_expr*
- | *is_expr*
- | *is_texpr*
- | *is_texpr*
- | *texpr*
- | *terminals*
- | *z*
- | \mathbb{Q}
- | *lit*
- | *bool_op*
- | *arith_op*
- | *list_op*
- | *tuple_op*
- | *pointer_op*
- | *BT_t*

$option_op$
 $array_op$
 $param_op$
 $struct_op$
 ct_pred
 $term$
 $term$
 $term$
 $terms$
 $predicate_name$
 $init$
 $predicate$
 $resource$
 $spine_elem$
 arg
 ret
 \mathcal{C}
 \mathcal{L}
 Φ
 \mathcal{R}
 $formula$

$$\boxed{\mathcal{C}; \mathcal{L}; \Phi \vdash object_value \Rightarrow obj\ ident:\beta. term}$$

$$\frac{}{\mathcal{C}; \mathcal{L}; \Phi \vdash mem_int \Rightarrow obj\ y:integer. y = of_mem_int(mem_int)} \quad PVAL_OBJ_INT$$

$$\frac{}{\mathcal{C}; \mathcal{L}; \Phi \vdash mem_ptr \Rightarrow obj\ y:loc. y = of_mem_ptr\ mem_ptr} \quad PVAL_OBJ_PTR$$

$$\frac{\frac{}{\mathcal{C}; \mathcal{L}; \Phi \vdash loaded_value_i \Rightarrow y_i:\beta. term_i^i}}{\mathcal{C}; \mathcal{L}; \Phi \vdash array(loaded_value_i^i) \Rightarrow obj\ y:array\ \beta. \bigwedge ([y[i]/y_i] term_i^i)} \quad PVAL_OBJ_ARR$$

$$\frac{\frac{\text{ident}:\mathbf{struct\ tag} \ \& \ \overline{\text{member}_i:\tau_i}^i \in \mathbf{Globals}}{\mathcal{C};\mathcal{L};\Phi \vdash \text{mem_val}_i \Rightarrow \overline{\mathbf{mem} \ y_i:\beta_i. \text{term}_i}^i}}{\mathcal{C};\mathcal{L};\Phi \vdash (\mathbf{struct\ tag})\{.\overline{\text{member}_i:\tau_i = \text{mem_val}_i}^i\} \Rightarrow \mathbf{obj} \ y:\mathbf{struct\ tag}. \bigwedge ([y.\text{member}_i/y_i]\overline{\text{term}_i}^i)} \quad \text{PVAL_OBJ_STRUCT}$$

$$\boxed{\mathcal{C};\mathcal{L};\Phi \vdash \text{pval} \Rightarrow \text{ident}:\beta. \text{term}}$$

$$\frac{x:\beta \in \mathcal{C}}{\mathcal{C};\mathcal{L};\Phi \vdash x \Rightarrow y:\beta. y = x} \quad \text{PVAL_VAR}$$

$$\frac{\mathcal{C};\mathcal{L};\Phi \vdash \text{object_value} \Rightarrow \mathbf{obj} \ y:\beta. \text{term}}{\mathcal{C};\mathcal{L};\Phi \vdash \text{object_value} \Rightarrow y:\beta. \text{term}} \quad \text{PVAL_OBJ}$$

$$\frac{\mathcal{C};\mathcal{L};\Phi \vdash \text{object_value} \Rightarrow \mathbf{obj} \ y:\beta. \text{term}}{\mathcal{C};\mathcal{L};\Phi \vdash \mathbf{specified_object_value} \Rightarrow y:\beta. \text{term}} \quad \text{PVAL_LOADED}$$

$$\frac{}{\mathcal{C};\mathcal{L};\Phi \vdash \mathbf{Unit} \Rightarrow y:\mathbf{unit}. y = \mathbf{unit}} \quad \text{PVAL_UNIT}$$

$$\frac{}{\mathcal{C};\mathcal{L};\Phi \vdash \mathbf{True} \Rightarrow y:\mathbf{bool}. y = \mathbf{true}} \quad \text{PVAL_TRUE}$$

$$\frac{}{\mathcal{C};\mathcal{L};\Phi \vdash \mathbf{False} \Rightarrow y:\mathbf{bool}. y = \mathbf{false}} \quad \text{PVAL_FALSE}$$

$$\frac{\overline{\mathcal{C};\mathcal{L};\Phi \vdash \text{value}_i \Rightarrow y_i:\beta. \text{term}_i}^i}{\mathcal{C};\mathcal{L};\Phi \vdash \beta[\overline{\text{value}_i}^i] \Rightarrow y:[\beta]. \bigwedge ([y^{(i)}/y_i]\overline{\text{term}_i}^i)} \quad \text{PVAL_LIST}$$

$$\frac{\overline{\mathcal{C};\mathcal{L};\Phi \vdash \text{value}_i \Rightarrow y_i:\beta_i. \text{term}_i}^i}{\mathcal{C};\mathcal{L};\Phi \vdash (\overline{\text{value}_i}^i) \Rightarrow y:\overline{\beta_i}^i. \bigwedge ([y^{(i)}/y_i]\overline{\text{term}_i}^i)} \quad \text{PVAL_TUPLE}$$

$$\frac{\text{smt}(\Phi \Rightarrow \text{false})}{\mathcal{C}; \mathcal{L}; \Phi \vdash \text{error}(\text{string}, pval) \Rightarrow y:\beta. \text{term}} \quad \text{PVAL_ERROR}$$

$$\frac{}{\mathcal{C}; \mathcal{L}; \Phi \vdash \text{Nil } \beta() \Rightarrow y:[\beta]. y = \text{nil}} \quad \text{PVAL_CTOR_NIL}$$

$$\frac{\begin{array}{c} \mathcal{C}; \mathcal{L}; \Phi \vdash pval_1 \Rightarrow y_1:\beta. \text{term}_1 \\ \mathcal{C}; \mathcal{L}; \Phi \vdash pval_2 \Rightarrow y_2:[\beta]. \text{term}_2 \end{array}}{\mathcal{C}; \mathcal{L}; \Phi \vdash \text{Cons}(pval_1, pval_2) \Rightarrow y:[\beta]. [y^{(1)}/y_1] \text{term}_1 \wedge [\text{tl } y/y_2] \text{term}_2} \quad \text{PVAL_CTOR_CONS}$$

$$\frac{\overline{\mathcal{C}; \mathcal{L}; \Phi \vdash pval_i \Rightarrow y_i:\beta_i. \text{term}_i}^i}{\mathcal{C}; \mathcal{L}; \Phi \vdash \text{Tuple}(\overline{pval_i}^i) \Rightarrow y:\overline{\beta_i}^i. \bigwedge ([y^{(i)}/y_i] \text{term}_i^i)} \quad \text{PVAL_CTOR_TUPLE}$$

$$\frac{\overline{\mathcal{C}; \mathcal{L}; \Phi \vdash pval_i \Rightarrow y_i:\beta. \text{term}_i}^i}{\mathcal{C}; \mathcal{L}; \Phi \vdash \text{Array}(\overline{pval_i}^i) \Rightarrow y:\text{array } \beta. \bigwedge ([y[i]/y_i] \text{term}_i^i)} \quad \text{PVAL_CTOR_ARRAY}$$

$$\frac{\mathcal{C}; \mathcal{L}; \Phi \vdash pval \Rightarrow y:\beta. \text{term}}{\mathcal{C}; \mathcal{L}; \Phi \vdash \text{Specified}(pval) \Rightarrow y:\beta. \text{term}} \quad \text{PVAL_CTOR_SPECIFIED}$$

$$\frac{\overline{\mathcal{C}; \mathcal{L}; \Phi \vdash pval_i \Rightarrow y_i:\beta_i. \text{term}_i}^i}{\mathcal{C}; \mathcal{L}; \Phi \vdash (\text{struct } tag)\{.member_i = pval_i^i\} \Rightarrow y:\text{struct } tag. \bigwedge ([y.member_i/y_i] \text{term}_i^i)} \quad \text{PVAL_STRUCT}$$

$\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash \overline{spine_elem_i}^i :: \text{arg} \gg \text{ret}$

$$\frac{}{\mathcal{C}; \mathcal{L}; \Phi; \cdot \vdash :: \text{ret} \gg \text{ret}} \quad \text{SPINE_EMPTY}$$

$$\frac{\mathcal{C}; \mathcal{L}; \Phi \vdash pval \Rightarrow _:\beta. _}{\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash \overline{spine_elem_i}^i :: [pval/x]arg \gg ret} \text{ SPINE_COMPUTATIONAL}$$

$$\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash pval, \overline{spine_elem_i}^i :: \Pi x:\beta. arg \gg ret$$

$$\frac{\mathcal{L} \vdash logical_val:\beta}{\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash \overline{spine_elem_i}^i :: [logical_val/x]arg \gg ret} \text{ SPINE_LOGICAL}$$

$$\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash logical_val, \overline{spine_elem_i}^i :: \forall x:\beta. arg \gg ret$$

$$\frac{\text{smt}(\Phi \Rightarrow resource = resource')}{\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash \overline{spine_elem_i}^i :: arg \gg ret} \text{ SPINE_RESOURCE}$$

$$\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R}, resource \vdash resource', \overline{spine_elem_i}^i :: resource' \multimap arg \gg ret$$

$$\frac{\text{smt}(\Phi \Rightarrow term)}{\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash \overline{spine_elem_i}^i :: arg \gg ret} \text{ SPINE_CONSTRAINT}$$

$$\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash \overline{spine_elem_i}^i :: term \supset arg \gg ret$$

$$\boxed{\mathcal{C}; \mathcal{L}; \Phi \vdash pexpr \Rightarrow ident:\beta. term}$$

$$\frac{\mathcal{C}; \mathcal{L}; \Phi \vdash pval \Rightarrow y:\beta. term}{\mathcal{C}; \mathcal{L}; \Phi \vdash pval \Rightarrow y:\beta. term} \text{ PEXPR_VAL}$$

$$\frac{\mathcal{C}; \mathcal{L}; \Phi \vdash smt_pval_1 \Rightarrow _':loc. _'}{\mathcal{C}; \mathcal{L}; \Phi \vdash smt_pval_2 \Rightarrow _":integer. _"} \text{ PEXPR_ARRAY_SHIFT}$$

$$\mathcal{C}; \mathcal{L}; \Phi \vdash \text{array_shift}(smt_pval_1, \tau, smt_pval_2) \Rightarrow y:loc. y = smt_pval_1 +_{\text{ptr}} smt_pval_2 \times \text{to_int}(\tau)$$

$$\frac{\mathcal{C}; \mathcal{L}; \Phi \vdash smt_pval \Rightarrow _':bool. _'}{\mathcal{C}; \mathcal{L}; \Phi \vdash \text{not}(smt_pval) \Rightarrow y:bool. y = \neg smt_pval} \text{ PEXPR_NOT}$$

$$\begin{array}{c}
\mathcal{C}; \mathcal{L}; \Phi \vdash \text{name} \Rightarrow _:\beta. _ \\
\mathcal{C}; \mathcal{L}; \Phi; \cdot \vdash pval_1, \dots, pval_n :: \forall _:\beta. \mathbf{I} \gg \Sigma y:\beta'. \text{term}' \wedge \mathbf{I} \\
\hline
\mathcal{C}; \mathcal{L}; \Phi \vdash \text{name}(pval_1, \dots, pval_n) \Rightarrow y:\beta'. \text{term}'
\end{array}
\quad \text{PEXPR_CALL}$$

$$\boxed{\text{pattern}:\beta \rightsquigarrow \mathcal{C}}$$

$$\boxed{\text{ident_or_pattern}:\beta \rightsquigarrow \mathcal{C}}$$

$$\boxed{\mathcal{C}; \mathcal{L}; \Phi \vdash \text{tpval} \Leftarrow \text{ident}:\beta. \text{term}}$$

$$\frac{\text{smt}(\Phi \Rightarrow \text{false})}{\mathcal{C}; \mathcal{L}; \Phi \vdash \text{undef } UB_name \Leftarrow y:\beta. \text{term}}
\quad \text{TPVAL_UNDEF}$$

$$\frac{\begin{array}{c} \mathcal{C}; \mathcal{L}; \Phi \vdash pval \Rightarrow y:\beta. \text{term}' \\ \text{smt}(\Phi, \text{term}' \Rightarrow \text{term}) \end{array}}{\mathcal{C}; \mathcal{L}; \Phi \vdash \text{done } pval \Leftarrow y:\beta. \text{term}}
\quad \text{TPVAL_DONE}$$

$$\boxed{\mathcal{C}; \mathcal{L}; \Phi \vdash \text{tpepr} \Leftarrow \text{ident}:\beta. \text{term}}$$

$$\frac{\begin{array}{c} \mathcal{C}; \mathcal{L}; \Phi \vdash \text{smt_pval} \Rightarrow _:\text{bool}. _ \\ \mathcal{C}; \mathcal{L}, y'; \Phi, \text{smt_pval} = \text{true} \vdash \text{tpepr}_1 \Leftarrow y:\beta. \text{term} \\ \mathcal{C}; \mathcal{L}, y'; \Phi, \text{smt_pval} = \text{false} \vdash \text{tpepr}_2 \Leftarrow y:\beta. \text{term} \end{array}}{\mathcal{C}; \mathcal{L}; \Phi \vdash \text{if } \text{smt_pval} \text{ then } \text{tpepr}_1 \text{ else } \text{tpepr}_2 \Leftarrow y:\beta. \text{term}}
\quad \text{TPEXPR_IF}$$

$$\frac{\begin{array}{c} \mathcal{C}; \mathcal{L}; \Phi \vdash \text{pexpr} \Rightarrow _:\beta. _ \\ \text{ident_or_pattern}:\beta \rightsquigarrow \mathcal{C}' \\ \mathcal{C}, \text{fresh}(\mathcal{C}'); \mathcal{L}; \Phi \vdash [\text{fresh}(\mathcal{C}')/\mathcal{C}'] \text{tpepr} \Leftarrow y:\beta. \text{term} \end{array}}{\mathcal{C}; \mathcal{L}; \Phi \vdash \text{let } \text{ident_or_pattern} = \text{pexpr} \text{ in } \text{tpepr} \Leftarrow y:\beta. \text{term}}
\quad \text{TPEXPR_LET}$$

$$\begin{array}{c}
\mathcal{C}; \mathcal{L}; \Phi \vdash pval \Rightarrow \cdot : \beta. _ \\
\hline
pattern_i : \beta \rightsquigarrow \mathcal{C}_i^i \\
\hline
\mathcal{C}, \text{fresh}(\mathcal{C}_i); \mathcal{L}; \Phi \vdash [\text{fresh}(\mathcal{C}_i)/\mathcal{C}_i] tpepr_i \Leftarrow y : \beta. term^i \\
\hline
\mathcal{C}; \mathcal{L}; \Phi \vdash \text{case } pval \text{ of } \overline{pattern_i \Rightarrow tpepr_i^i} \text{ end} \Leftarrow y : \beta. term
\end{array}
\quad \text{TPEXPR_CASE}$$

$$\boxed{\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash mem_action \Rightarrow ret}$$

$$\begin{array}{c}
\mathcal{C}; \mathcal{L}; \Phi \vdash smt_pval \Rightarrow \cdot : \text{integer}. _ \\
\hline
\mathcal{C}; \mathcal{L}; \Phi; \cdot \vdash \text{create}(smt_pval, \tau) \Rightarrow \Sigma y_p : \text{loc}. \exists y : \text{to_base}(\tau). \text{representable}(\tau*, y_p) \wedge \text{alignedI}(smt_pval, y_p) \wedge [y_p] 1 \xrightarrow{\times}_{\tau} [y] \star \mathbf{I}
\end{array}
\quad \text{ACTION_CREATE}$$

$$\begin{array}{c}
\mathcal{C}; \mathcal{L}; \Phi \vdash smt_pval_1 \Rightarrow \cdot : \text{loc}. _ \\
\mathcal{C}; \mathcal{L}; \Phi \vdash smt_pval' \Rightarrow \cdot' : \text{to_base}(\tau). _ \\
\text{smt}(\Phi \Rightarrow \text{representable}(\tau, smt_pval')) \\
\text{smt}(\Phi \Rightarrow smt_pval_0 = smt_pval_1) \\
\hline
\mathcal{C}; \mathcal{L}; \Phi; \cdot, [smt_pval_0] 1 \mapsto_{\tau} [\cdot'] \vdash \text{store}(\cdot, \tau, smt_pval_1, smt_pval', \cdot) \Rightarrow \Sigma \cdot : \text{unit}. [smt_pval_0] 1 \xrightarrow{\check{\times}}_{\tau} [smt_pval'] \star \mathbf{I}
\end{array}
\quad \text{ACTION_STORE}$$

$$\begin{array}{c}
\mathcal{C}; \mathcal{L}; \Phi \vdash smt_pval_1 \Rightarrow \cdot : \text{loc}. _ \\
\text{smt}(\Phi \Rightarrow smt_pval_0 = smt_pval_1) \\
\hline
\mathcal{C}; \mathcal{L}; \Phi; \cdot, [smt_pval_0] 1 \mapsto_{\tau} [\cdot'] \vdash \text{kill}(\text{static } \tau, smt_pval_1) \Rightarrow \Sigma \cdot : \text{unit}. \mathbf{I}
\end{array}
\quad \text{ACTION_KILL_STATIC}$$

$$\boxed{\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash seq_expr \Rightarrow ret}$$

$$\begin{array}{c}
\mathcal{C}; \mathcal{L}; \Phi \vdash pval \Rightarrow y : \beta. term \\
\hline
\mathcal{C}; \mathcal{L}; \Phi; \cdot \vdash pval \Rightarrow \Sigma y : \beta. term \wedge \mathbf{I}
\end{array}
\quad \text{SEQ_EXPR_PURE}$$

$$\boxed{\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash is_expr \Rightarrow ret}$$

$$\begin{array}{c}
\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash mem_action \Rightarrow ret \\
\hline
\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash \text{Pos } mem_action \Rightarrow ret
\end{array}
\quad \text{IS_EXPR_ACTION}$$

$$\frac{\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash \text{mem_action} \Rightarrow \text{ret}}{\mathcal{C}; \mathcal{L}; \Phi; \mathcal{R} \vdash \text{Neg mem_action} \Rightarrow \text{ret}} \quad \text{IS_EXPR_NEG_ACTION}$$

Definition rules: 39 good 0 bad
Definition rule clauses: 90 good 0 bad