tag	OCaml type for struct/union tag
$impl\_const$	implementation-defined constant
x, y, $ident$	OCaml type variable for symbols
ty_mem_int	memory integer value
mem_val	memory value
$member$	C struct/union member name
τ	C type
$annots$	annotations
nat	OCaml arbitrary-width natural number
n, i	index variables
loc	OCaml type for C source
mem_iv_c	OCaml type for memory constraints on integer values
UB_name	undefined behaviour
$string$	OCaml string
$tyvar\_TY$	OCaml type variable for types
τ	OCaml type for an annotated C type
sym_prefix	OCaml type for symbol prefix
mem_order	OCaml type for memory order
$linux\_mem\_order$	OCaml type for Linux memory order
k	OCaml fixed-width integer

β	::=		Core base types
	|	`unit`	unit
	|	`bool`	boolean
	|	`integer`	integer
	|	`real`	rational numbers?
	|	`loc`	location
	|	`array` β	array
	|	$[\beta]$	list
	|	$(\beta_1, .., \beta_n)$	tuple
	|	`struct` *tag*	struct
	|	$\{\beta\}$	set
	|	`opt` (β)	option
	|	$\beta_1, .., \beta_n \rightarrow \beta$	parameter types

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
	|	`/\`	conjucntion
	|	`\/`	disjunction

polarity	::=		memory action polarities
	|	`Pos`	sequenced by `let weak` and `let strong`
	|	`Neg`	only sequenced by `let strong`
ident	::=		Ott-hack, ignore
name	::=		
	|	*ident*	Core identifier
	|	*impl_const*	implementation-defined constant
ptrval	::=		pointer values
	|	`nullptr`	null pointer
	|	`funcptr` *ident*	function pointer
	|	`concptr` *nat*	concrete pointer
object_value	::=		C object values
	|	*ty_mem_int*	integer value
	|	*ptrval*	pointer value
	|	`array` $(loaded\_value_1, .., loaded\_value_n)$	C array value
	|	$(\texttt{struct}\ ident)\{\ \overline{.member_i : \tau_i = mem\_val_i}^{\,i}\ \}$	C struct value
	|	$(\texttt{union}\ ident)\{.member = mem\_val\}$	C union value
loaded_value	::=		potentially unspecified C object values
	|	`specified` $(object\_value)$	specified loaded value
β	::=		Ott-hack, ignore
value	::=		Core values
	|	*object_value*	C object value
	|	*loaded_value*	loaded C object value

	`Unit`	unit
	`True`	boolean true
	`False`	boolean false
	$\beta[value_1, .., value_i]$	list
	$(value_1, .., value_i)$	tuple

ctor	::=	data constructors
	`Nil` β	empty list
	`Cons`	list cons
	`Tuple`	tuple
	`Array`	C array
	`Ivmax`	max integer value
	`Ivmin`	min integer value
	`Ivsizeof`	sizeof value
	`Ivalignof`	alignof value
	`IvCOMPL`	bitwise complement
	`IvAND`	bitwise AND
	`IvOR`	bitwise OR
	`IvXOR`	bitwise XOR
	`Specified`	non-unspecified loaded value
	`Unspecified`	unspecified loaded value
	`Fvfromint`	cast integer to floating value
	`Ivfromfloat`	cast floating to integer value

ident_opt_β	::=	type annotated optional identifier
	_ : β	
	ident : β	

pattern_aux	::=	
	ident_opt_β	

	| $ctor(\overline{pattern_i}^{\,i})$	
pattern	::=	
	| $loc\ annots\ pattern\_aux$	
ident_or_pattern	::=	
	| *ident*	
	| *pattern*	
ident	::=	Ott-hack, ignore
pexpr_aux	::=	pure expressions
	| *ident*	
	| *impl_const*	implementation-defined constant
	| *value*	
	| `constrained` $(\overline{mem\_iv\_c_i, ident_i}^{\,i})$	constrained value
	| `error` $(string, ident)$	impl-defined static error
	| $ctor(\overline{ident_i}^{\,i})$	data constructor application
	| `array_shift` $(ident_1, \tau, ident_2)$	pointer array shift
	| `member_shift` $(ident, ident, member)$	pointer struct/union member shift
	| `not` $(ident)$	boolean not
	| $ident_1\ binop\ ident_2$	binary operations
	| $(\texttt{struct}\ ident)\{\overline{.member_i = ident_i}^{\,i}\}$	C struct expression
	| $(\texttt{union}\ ident)\{.member = ident\}$	C union expression
	| `memberof` $(ident, member, ident)$	C struct/union member access
	| $name(ident_1, .., ident_n)$	pure function call
	| `assert_undef` $(ident, loc, UB\_name)$	
	| `bool_to_integer` $(ident)$	
	| `conv_int` $(\tau, ident)$	
	| `wrapI` $(\tau, ident)$	

pexpr	::=		pure expressions with location and annotations
	|	*loc annots tyvar_TY pexpr_aux*	
tpexpr_aux	::=		top-level pure expressions
	|	`undef` *loc UB_name*	undefined behaviour
	|	`case` *ident* `of` $\overline{\mid pattern_i \Rightarrow tpexpr_i}^{\,i}$ `end`	pattern matching
	|	`let` *ident_or_pattern* $=$ $tpexpr_1$ `in` $tpexpr_2$	pure let
	|	`if` *ident* `then` $tpexpr_1$ `else` $tpexpr_2$	pure if
	|	`done` *ident*	pure done
tpexpr	::=		pure top-level pure expressions with location and annotations
	|	*loc annots tyvar_TY tpexpr_aux*	
m_kill_kind	::=		
	|	`dynamic`	
	|	`static` τ	
bool	::=		OCaml booleans
	|	`true`	
	|	`false`	
action_aux	::=		memory actions
	|	`create` $(ident, \tau)sym\_prefix$	
	|	`create_readonly` $(ident_1, \tau, ident_2)sym\_prefix$	
	|	`alloc` $(ident_1, ident_2)sym\_prefix$	
	|	`kill` $(m\_kill\_kind, ident)$	the boolean indicates whether the action is dynamic (i.e. free())
	|	`store` $(bool, \tau, ident_1, ident_2, mem\_order)$	the boolean indicates whether the store is locking
	|	`load` $(\tau, ident, mem\_order)$	
	|	`rmw` $(\tau, ident_1, ident_2, ident_3, mem\_order_1, mem\_order_2)$	
	|	`fence` (mem_order)	

	$\texttt{cmp\_exch\_strong}\,(\tau, ident_1, ident_2, ident_3, mem\_order_1, mem\_order_2)$	
	$\texttt{cmp\_exch\_weak}\,(\tau, ident_1, ident_2, ident_3, mem\_order_1, mem\_order_2)$	
	$\texttt{linux\_fence}\,(linux\_mem\_order)$	
	$\texttt{linux\_load}\,(\tau, ident, linux\_mem\_order)$	
	$\texttt{linux\_store}\,(\tau, ident_1, ident_2, linux\_mem\_order)$	
	$\texttt{linux\_rmw}\,(\tau, ident_1, ident_2, linux\_mem\_order)$	

action ::=

	$loc\, action\_aux$	

memop ::=

		operations involving the memory state
	$ident_1 == ident_2$	pointer equality comparison
	$ident_1 \neq ident_2$	pointer inequality comparison
	$ident_1 < ident_2$	pointer less-than comparison
	$ident_1 > ident_2$	pointer greater-than comparison
	$ident_1 \leq ident_2$	pointer less-than comparison
	$ident_1 \geq ident_2$	pointer greater-than comparison
	$ident_1 -_\tau ident_2$	pointer subtraction
	$\texttt{intFromPtr}\,(\tau_1, \tau_2, ident)$	cast of pointer value to integer value
	$\texttt{ptrFromInt}\,(\tau_1, \tau_2, ident)$	cast of integer value to pointer value
	$\texttt{ptrValidForDeref}\,(\tau, ident)$	dereferencing validity predicate
	$\texttt{ptrWellAligned}\,(\tau, ident)$	
	$\texttt{ptrArrayShift}\,(ident_1, \tau, ident_2)$	
	$\texttt{memcpy}\,(ident_1, ident_2, ident_3)$	
	$\texttt{memcmp}\,(ident_1, ident_2, ident_3)$	
	$\texttt{realloc}\,(ident_1, ident_2, ident_3)$	TODO: not sure about this
	$\texttt{va\_start}\,(ident_1, ident_2)$	
	$\texttt{va\_copy}\,(ident)$	
	$\texttt{va\_arg}\,(ident, \tau)$	
	$\texttt{va\_end}\,(ident)$	

paction	::=		memory actions with polarity
	|	*action*	M positive, sequenced by both `let weak` and `let strong`
	|	$\neg$ (*action*)	M negative, only sequenced by `let strong`
expr_aux	::=		(effectful) expressions
	|	`pure` (*pexpr*)	
	|	`memop` (*memop*)	pointer op involving memory
	|	*paction*	memory action
	|	`skip`	
	|	`ccall` (τ, *ident*, $\overline{ident_i}^{\,i}$)	C function call
	|	`pcall` (*name*, $\overline{ident_i}^{\,i}$)	Core procedure call
expr	::=		(effectful) expressions with location and annotations
	|	*loc annots expr_aux*	
texpr_aux	::=		top-level expressions
	|	`let` *ident_or_pattern* = *pexpr* `in` *texpr*	
	|	`let weak` *pattern* = *expr* `in` *texpr*	weak sequencing
	|	`let strong` *ident_or_pattern* = *expr* `in` *texpr*	strong sequencing
	|	`case` *ident* `with` $\overline{\mid pattern_i \Rightarrow texpr_i}^{\,i}$ `end`	pattern matching
	|	`if` *ident* `then` $texpr_1$ `else` $texpr_2$	
	|	`bound` $[k]$(*texpr*)	...and boundary
	|	`unseq` ($expr_1$, .., $expr_n$)	unsequenced expressions
	|	`nd` ($texpr_1$, .., $texpr_n$)	nondeterministic sequencing
	|	`done` *ident*	
	|	`undef` *loc UB_name*	
	|	`run` *ident* $ident_1$, .., $ident_n$	run from label
texpr	::=		top-level expressions with location and annotations
	|	*loc annots texpr_aux*	

terminals	::=		
	|	λ	
	|	$\longrightarrow$	
	|	$\rightarrow$	
	|	$\Rightarrow$	
	|	$\Leftarrow$	
	|	$\vdash$	
	|	$\in$	
	|	Π	
	|	$\forall$	
	|	$\multimap$	
	|	$\supset$	
	|	Σ	
	|	$\exists$	
	|	$\star$	
	|	$\wedge$	
	|	$\bigwedge$	
	|	$\neg$	
	|	$=$	
	|	$\neq$	
	|	$\leq$	
	|	$\geq$	
	|	&	
$z,\ Z\_t$	::=		OCaml arbitrary-width integer
	|	of_mem_int(ty_mem_int)	M
	|	of_nat(nat)	M
lit	::=		
	|	*ident*	

$$
\begin{array}{lll}
 & | & () \\
 & | & \mathit{bool} \\
 & | & \texttt{int}\ Z\_t \\
 & | & \texttt{ptr}\ Z\_t \\
 & & \\
\mathit{bool\_op} & ::= & \\
 & | & \neg\ \mathit{index\_term} \\
 & | & \mathit{index\_term}_1 = \mathit{index\_term}_2 \\
 & | & \bigwedge(\mathit{index\_term}_1, .., \mathit{index\_term}_n) \\
 & & \\
\mathit{list\_op} & ::= & \\
 & | & [\mathit{index\_term}_1, .., \mathit{index\_term}_n] \\
 & | & \mathit{index\_term}^{(k)} \\
 & & \\
\mathit{tuple\_op} & ::= & \\
 & | & (\mathit{index\_term}_1, .., \mathit{index\_term}_n) \\
 & | & \mathit{index\_term}^{(k)} \\
 & & \\
\mathit{pointer\_op} & ::= & \\
 & | & \texttt{nullop} \\
 & & \\
\mathit{array\_op} & ::= & \\
 & | & \mathit{index\_term}_1[\,\texttt{int}\ Z\_t] \\
 & & \\
\mathit{param\_op} & ::= & \\
 & | & \mathit{index\_term}(\mathit{index\_term}_1, .., \mathit{index\_term}_n) \\
 & & \\
\mathit{struct\_op} & ::= & \\
 & | & \mathit{index\_term}.\mathit{member}
\end{array}
$$

index_term_aux	::=			
	|	*bool_op*		
	|	*list_op*		
	|	*pointer_op*		
	|	*array_op*		
	|	*param_op*		
bt	::=			OCaml type variable for base types
index_term	::=			
	|	*lit*		
	|	*index_term_aux bt*		
	|	(*index_term*)	S	parentheses
	|	$index\_term[index\_term_1/ident_1, .., index\_term_n/ident_n]$	M	
arg	::=			argument types
	|	$\Pi\, ident : \beta.arg$		
	|	$\forall\, ident : \texttt{logSort}.arg$		
	|	$\texttt{resource} \multimap arg$		
	|	$index\_term \supset arg$		
	|	`I`		
ret	::=			return types
	|	$\Sigma\, ident : \beta.ret$		
	|	$\exists\, ident : \texttt{logSort}.ret$		
	|	$\texttt{resource} \star ret$		
	|	$index\_term \wedge ret$		
	|	`I`		
Γ	::=			computational var env

		$\mid$ `empty`	
		$\mid$ $\Gamma, x : \beta$	
Λ	::=		logical var env
		$\mid$ `empty`	
		$\mid$ Λ, x	
Ξ	::=		constraints env
		$\mid$ `empty`	
		$\mid$ Ξ, `phi`	
formula	::=		
		$\mid$ *judgement*	
		$\mid$ `not` (*formula*)	
		$\mid$ $ident : \beta \in \Gamma$	
		$\mid$ $ident :$ `struct` $tag \& \overline{member_i : \tau_i}^{\,i} \in \Gamma$	
		$\mid$ $formula_1 \quad .. \quad formula_n$	
Jtype	::=		
		$\mid$ $\Gamma; \Lambda; \Xi \vdash value \Rightarrow ident, \beta, index\_term$	
		$\mid$ $\Gamma; \Lambda; \Xi \vdash pexpr\_aux \Rightarrow ret$	
judgement	::=		
		$\mid$ *Jtype*	
user_syntax	::=		
		$\mid$ *tag*	
		$\mid$ *impl_const*	
		$\mid$ x	
		$\mid$ *ty_mem_int*	

| mem_val
| $member$
| τ
| $annots$
| nat
| n
| loc
| mem_iv_c
| UB_name
| $string$
| $tyvar\_TY$
| τ
| sym_prefix
| mem_order
| $linux\_mem\_order$
| k
| β
| $binop$
| $polarity$
| $ident$
| $name$
| $ptrval$
| $object\_value$
| $loaded\_value$
| β
| $value$
| $ctor$
| $ident\_opt\_\beta$
| $pattern\_aux$

| *pattern*
| *ident_or_pattern*
| *ident*
| *pexpr_aux*
| *pexpr*
| *tpexpr_aux*
| *tpexpr*
| *m_kill_kind*
| *bool*
| *action_aux*
| *action*
| *memop*
| *paction*
| *expr_aux*
| *expr*
| *texpr_aux*
| *texpr*
| *terminals*
| *z*
| *lit*
| *bool_op*
| *list_op*
| *tuple_op*
| *pointer_op*
| *array_op*
| *param_op*
| *struct_op*
| *index_term_aux*
| *bt*

| *index_term*
| *arg*
| *ret*
| Γ
| Λ
| Ξ
| *formula*

$\boxed{\Gamma; \Lambda; \Xi \vdash value \Rightarrow ident, \beta, index\_term}$

$$\frac{}{\Gamma; \Lambda; \Xi \vdash ty\_mem\_int \Rightarrow y, \texttt{integer}, y = \texttt{int}\,\text{of\_mem\_int}(ty\_mem\_int)}\ \textsc{Val\_Obj\_Int}$$

$$\frac{}{\Gamma; \Lambda; \Xi \vdash \texttt{nullptr} \Rightarrow y, \texttt{loc}, y = \texttt{nullop}}\ \textsc{Val\_Obj\_Ptr\_Null}$$

$$\frac{}{\Gamma; \Lambda; \Xi \vdash \texttt{funcptr}\, ident \Rightarrow y, \texttt{loc}, y = ident}\ \textsc{Val\_Obj\_Ptr\_Func}$$

$$\frac{}{\Gamma; \Lambda; \Xi \vdash \texttt{concptr}\, nat \Rightarrow y, \texttt{loc}, y = \texttt{ptr}\,\text{of\_nat}(nat)}\ \textsc{Val\_Obj\_Ptr\_Conc}$$

$$\frac{\Gamma; \Lambda; \Xi \vdash loaded\_value_1 \Rightarrow y_1, \beta, index\_term_1 \quad .. \quad \Gamma; \Lambda; \Xi \vdash loaded\_value_n \Rightarrow y_n, \beta, index\_term_n}{\Gamma; \Lambda; \Xi \vdash \texttt{array}\,(loaded\_value_1, .., loaded\_value_n) \Rightarrow y, \texttt{array}\,\beta, \bigwedge(index\_term_1, .., index\_term_n)\,[y[\,\texttt{int}\,z_1]\,/y_1, .., y[\,\texttt{int}\,z_n]\,/y_n]}\ \textsc{Val\_Obj\_Arr}$$

$$\frac{}{\Gamma; \Lambda; \Xi \vdash \texttt{Unit} \Rightarrow y, \texttt{unit}, y = ()}\ \textsc{Val\_Unit}$$

$$\frac{}{\Gamma; \Lambda; \Xi \vdash \texttt{True} \Rightarrow y, \texttt{bool}, y = \texttt{true}}\ \textsc{Val\_True}$$

$$\frac{}{\Gamma; \Lambda; \Xi \vdash \texttt{False} \Rightarrow y, \texttt{bool}, y = \texttt{false}}\ \textsc{Val\_False}$$

$$\frac{\Gamma; \Lambda; \Xi \vdash value_1 \Rightarrow y_1, \beta, index\_term_1 \quad .. \quad \Gamma; \Lambda; \Xi \vdash value_n \Rightarrow y_n, \beta, index\_term_n}{\Gamma; \Lambda; \Xi \vdash \beta[value_1, .., value_n] \Rightarrow y, [\beta], (\bigwedge(index\_term_1, .., index\_term_n))[y^{(k_1)}\,/y_1, .., y^{(k_n)}\,/y_n]}\ \textsc{Val\_List}$$

$$\frac{\Gamma; \Lambda; \Xi \vdash value_1 \Rightarrow y_1, \beta_1, index\_term_1 \quad .. \quad \Gamma; \Lambda; \Xi \vdash value_n \Rightarrow y_n, \beta_n, index\_term_n}{\Gamma; \Lambda; \Xi \vdash (value_1, .., value_n) \Rightarrow y, (\beta_1, .., \beta_n), \bigwedge(index\_term_1, .., index\_term_n)\,[y^{(k_1)}\,/y_1, .., y^{(k_n)}\,/y_n]}\ \textsc{Val\_Tuple}$$

$\boxed{\Gamma; \Lambda; \Xi \vdash pexpr\_aux \Rightarrow ret}$

$$\frac{x : \beta \in \Gamma}{\Gamma; \Lambda; \Xi \vdash x \Rightarrow \Sigma\, y : \beta.\mathtt{I}} \quad \textsc{PExpr\_Var}$$

$$\frac{\Gamma; \Lambda; \Xi \vdash \mathit{value} \Rightarrow y, \beta, \mathit{index\_term}}{\Gamma; \Lambda; \Xi \vdash \mathit{value} \Rightarrow \Sigma\, y : \beta.\mathit{index\_term} \wedge \mathtt{I}} \quad \textsc{PExpr\_Val}$$

$$\frac{x : \mathtt{bool} \in \Gamma}{\Gamma; \Lambda; \Xi \vdash \mathtt{not}\,(x) \Rightarrow \Sigma\, y : \mathtt{bool}.y = (\neg\, x\,) \wedge \mathtt{I}} \quad \textsc{PExpr\_Not}$$

```
Definition rules:         13 good    0 bad
Definition rule clauses:  19 good    0 bad
```