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
$member$	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
ty_mem_ptr	::=		pointer values
	|	`nullptr`	null pointer
	|	`funcptr` *ident*	function pointer
	|	`concptr` *nat*	concrete pointer
mem_val	::=		memory value
	|	`int` *ty_mem_int*	
	|	`ptr` *ty_mem_ptr*	
	|	`array` $mem\_val \,..\, mem\_val$	
	|	`struct` $ident\; member_1\; mem\_val \,..\, member_n\; mem\_val$	
	|	`union` $ident\; member$	
object_value	::=		C object values (inhabitants of object types), which can be read/stored
	|	*ty_mem_int*	integer value
	|	*ty_mem_ptr*	pointer value
	|	`array` $(loaded\_value_1, \,..\, , loaded\_value_n)$	C array value
	|	$(\texttt{struct}\; ident)\{\, \overline{.member_i : \tau_i = mem\_val}^{\,i} \,\}$	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
	|	$\_ : \beta$	
	|	$ident : \beta$	
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*	
	|	$\texttt{constrained}\,(\overline{mem\_iv\_c_i, ident_i}^{\,i})$	constrained value
	|	$\texttt{error}\,(string, ident)$	impl-defined static error
	|	$ctor(\overline{ident_i}^{\,i})$	data constructor application
	|	$\texttt{array\_shift}\,(ident_1, \tau, ident_2)$	pointer array shift
	|	$\texttt{member\_shift}\,(ident, ident, member)$	pointer struct/union member shift
	|	$\texttt{not}\,(ident)$	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

	$\mid$ `memberof` $(ident, member, ident)$	C struct/union member access
	$\mid$ $name(ident_1, .., ident_n)$	pure function call
	$\mid$ `assert_undef` $(ident, loc, UB\_name)$	
	$\mid$ `bool_to_integer` $(ident)$	
	$\mid$ `conv_int` $(\tau, ident)$	
	$\mid$ `wrapI` $(\tau, ident)$	
$pexpr$	::=	pure expressions with location and annotations
	$\mid$ $loc\ annots\ tyvar\_TY\ pexpr\_aux$	
$tpexpr\_aux$	::=	top-level pure expressions
	$\mid$ `undef` $loc\ UB\_name$	undefined behaviour
	$\mid$ `case` $ident$ `of` $\overline{\mid pattern_i \Rightarrow tpexpr_i}^{\,i}$ `end`	pattern matching
	$\mid$ `let` $ident\_or\_pattern = tpexpr_1$ `in` $tpexpr_2$	pure let
	$\mid$ `if` $ident$ `then` $tpexpr_1$ `else` $tpexpr_2$	pure if
	$\mid$ `done` $ident$	pure done
$tpexpr$	::=	pure top-level pure expressions with location and annotations
	$\mid$ $loc\ annots\ tyvar\_TY\ tpexpr\_aux$	
m_kill_kind	::=	
	$\mid$ `dynamic`	
	$\mid$ `static` τ	
$bool$	::=	OCaml booleans
	$\mid$ `true`	
	$\mid$ `false`	
$action\_aux$	::=	memory actions
	$\mid$ `create` $(ident, \tau)sym\_prefix$	

	$\texttt{create\_readonly}\,(ident_1, \tau, ident_2)sym\_prefix$	
	$\texttt{alloc}\,(ident_1, ident_2)sym\_prefix$	
	$\texttt{kill}\,(m\_kill\_kind, ident)$	the boolean indicates whether the action is dynamic (i.e. free())
	$\texttt{store}\,(bool, \tau, ident_1, ident_2, mem\_order)$	the boolean indicates whether the store is locking
	$\texttt{load}\,(\tau, ident, mem\_order)$	
	$\texttt{rmw}\,(\tau, ident_1, ident_2, ident_3, mem\_order_1, mem\_order_2)$	
	$\texttt{fence}\,(mem\_order)$	
	$\texttt{cmp\_exch\_strong}\,(\tau, ident_1, ident_2, ident_3, mem\_order_1, mem\_order_2)$	
	$\texttt{cmp\_exch\_weak}\,(\tau, ident_1, ident_2, ident_3, mem\_order_1, mem\_order_2)$	
	$\texttt{linux\_fence}\,(linux\_mem\_order)$	
	$\texttt{linux\_load}\,(\tau, ident, linux\_mem\_order)$	
	$\texttt{linux\_store}\,(\tau, ident_1, ident_2, linux\_mem\_order)$	
	$\texttt{linux\_rmw}\,(\tau, ident_1, ident_2, linux\_mem\_order)$	

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
	$\texttt{ptrWellAligned}\,(\tau, ident)$	
	$\texttt{ptrArrayShift}\,(ident_1, \tau, ident_2)$	

	$\mid$	$\texttt{memcpy}\,(ident_1, ident_2, ident_3)$	
	$\mid$	$\texttt{memcmp}\,(ident_1, ident_2, ident_3)$	
	$\mid$	$\texttt{realloc}\,(ident_1, ident_2, ident_3)$	
	$\mid$	$\texttt{va\_start}\,(ident_1, ident_2)$	
	$\mid$	$\texttt{va\_copy}\,(ident)$	
	$\mid$	$\texttt{va\_arg}\,(ident, \tau)$	
	$\mid$	$\texttt{va\_end}\,(ident)$	
paction	::=		memory actions with polarity
	$\mid$	*action*	M positive, sequenced by both `let weak` and `let strong`
	$\mid$	$\neg\,(action)$	M negative, only sequenced by `let strong`
expr_aux	::=		(effectful) expressions
	$\mid$	$\texttt{pure}\,(pexpr)$	
	$\mid$	$\texttt{memop}\,(memop)$	pointer op involving memory
	$\mid$	*paction*	memory action
	$\mid$	`skip`	
	$\mid$	$\texttt{ccall}\,(\tau, ident, \overline{ident_i}^{\,i})$	C function call
	$\mid$	$\texttt{pcall}\,(name, \overline{ident_i}^{\,i})$	Core procedure call
expr	::=		(effectful) expressions with location and annotations
	$\mid$	*loc annots expr_aux*	
texpr_aux	::=		top-level expressions
	$\mid$	$\texttt{let}\,ident\_or\_pattern = pexpr\,\texttt{in}\,texpr$	
	$\mid$	$\texttt{let weak}\,pattern = expr\,\texttt{in}\,texpr$	weak sequencing
	$\mid$	$\texttt{let strong}\,ident\_or\_pattern = expr\,\texttt{in}\,texpr$	strong sequencing
	$\mid$	$\texttt{case}\,ident\,\texttt{with}\,\overline{\mid pattern_i \Rightarrow texpr_i}^{\,i}\,\texttt{end}$	pattern matching
	$\mid$	$\texttt{if}\,ident\,\texttt{then}\,texpr_1\,\texttt{else}\,texpr_2$	
	$\mid$	$\texttt{bound}\,[k](texpr)$	...and boundary

	$\texttt{unseq}\,(expr_1, .., expr_n)$	unsequenced expressions
	$\texttt{nd}\,(texpr_1, .., texpr_n)$	nondeterministic sequencing
	$\texttt{done}\,ident$	
	$\texttt{undef}\,loc\,UB\_name$	
	$\texttt{run}\,ident\,ident_1, .., ident_n$	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*	
	|	()	
	|	*bool*	
	|	`int` Z_t	
	|	`ptr` Z_t	
bool_op	::=		
	|	$\neg\, index\_term$	
	|	$index\_term_1 = index\_term_2$	
	|	$\bigwedge(index\_term_1, .., index\_term_n)$	
list_op	::=		
	|	$[index\_term_1, .., index\_term_n]$	
	|	$index\_term^{(k)}$	
tuple_op	::=		
	|	$(index\_term_1, .., index\_term_n)$	
	|	$index\_term^{(k)}$	
pointer_op	::=		
	|	`nullop`	

array_op	::=			
	|	$index\_term_1[\,\texttt{int}\ Z\_t]$		
param_op	::=			
	|	$index\_term(index\_term_1, .., index\_term_n)$		
struct_op	::=			
	|	$index\_term.member$		
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
	|	$\Pi\, ident : \beta.arg$		
	|	$\forall\, ident : \texttt{logSort}\,.arg$		
	|	$\texttt{resource} \multimap arg$		
	|	$index\_term \supset arg$		
	|	$\texttt{I}$		

ret	::=		return types
	|	$\Sigma\, ident : \beta . ret$	
	|	$\exists\, ident :$ `logSort` $. ret$	
	|	`resource` $\star\, ret$	
	|	$index\_term \wedge ret$	
	|	`I`	
Γ	::=		computational var env
	|	`empty`	
	|	$\Gamma, x : \beta$	
Λ	::=		logical var env
	|	`empty`	
	|	Λ, x	
Ξ	::=		constraints env
	|	`empty`	
	|	Ξ, `phi`	
formula	::=		
	|	*judgement*	
	|	`inconsistent` $(\Gamma; \Lambda; \Xi)$	
	|	$ident : \beta \in \Gamma$	
	|	$ident :$ `struct` $tag \& \overline{member_i : \tau_i}^{\,i} \in \Gamma$	
	|	$formula_1 \quad .. \quad formula_n$	
mem_value_jtypes	::=		
	|	$\Gamma; \Lambda; \Xi \vdash mem\_val \Rightarrow$ `mem` $y, \beta, index\_term$	
value_jtypes	::=		

$$
\begin{array}{lll}
 & | & \Gamma; \Lambda; \Xi \vdash \mathit{object\_value} \Rightarrow \texttt{obj}\, \mathit{ident}, \beta, \mathit{index\_term} \\
 & | & \Gamma; \Lambda; \Xi \vdash \mathit{value} \Rightarrow \mathit{ident}, \beta, \mathit{index\_term}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{pexpr\_jtypes} & ::= & \\
 & | & \Gamma; \Lambda; \Xi \vdash \mathit{pexpr\_aux} \Rightarrow \mathit{ret}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{judgement} & ::= & \\
 & | & \mathit{mem\_value\_jtypes} \\
 & | & \mathit{value\_jtypes} \\
 & | & \mathit{pexpr\_jtypes}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{user\_syntax} & ::= & \\
 & | & \mathit{tag} \\
 & | & \mathit{impl\_const} \\
 & | & x \\
 & | & \mathit{ty\_mem\_int} \\
 & | & \mathit{member} \\
 & | & \tau \\
 & | & \mathit{annots} \\
 & | & \mathit{nat} \\
 & | & n \\
 & | & \mathit{loc} \\
 & | & \mathit{mem\_iv\_c} \\
 & | & \mathit{UB\_name} \\
 & | & \mathit{string} \\
 & | & \mathit{tyvar\_TY} \\
 & | & \tau \\
 & | & \mathit{sym\_prefix} \\
 & | & \mathit{mem\_order} \\
 & | & \mathit{linux\_mem\_order}
\end{array}
$$

| k
| β
| $binop$
| $polarity$
| $ident$
| $name$
| ty_mem_ptr
| mem_val
| $object\_value$
| $loaded\_value$
| 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*

$\boxed{\Gamma; \Lambda; \Xi \vdash \mathit{mem\_val} \Rightarrow \texttt{mem}\, y, \beta, \mathit{index\_term}}$

$\boxed{\Gamma; \Lambda; \Xi \vdash \mathit{object\_value} \Rightarrow \texttt{obj}\, \mathit{ident}, \beta, \mathit{index\_term}}$

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

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

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

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

$$\frac{\Gamma;\Lambda;\Xi \vdash \mathit{object\_value}_1 \Rightarrow y_1, \beta, \mathit{index\_term}_1 \quad .. \quad \Gamma;\Lambda;\Xi \vdash \mathit{object\_value}_n \Rightarrow y_n, \beta, \mathit{index\_term}_n}{\Gamma;\Lambda;\Xi \vdash \texttt{array}\,(\mathit{loaded\_value}_1, .., \mathit{loaded\_value}_n) \Rightarrow \texttt{obj}\,y, \texttt{array}\,\beta, \bigwedge(\mathit{index\_term}_1, .., \mathit{index\_term}_n)\,[y[\,\texttt{int}\,z_1]\,/y_1, .., y[\,\texttt{int}\,z_n]\,/y_n]}\ \textsc{Val\_Obj\_Arr}$$

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

$$\frac{\Gamma;\Lambda;\Xi \vdash \mathit{object\_value} \Rightarrow \texttt{obj}\,y, \beta, \mathit{index\_term}}{\Gamma;\Lambda;\Xi \vdash \mathit{object\_value} \Rightarrow y, \beta, \mathit{index\_term}}\ \textsc{Val\_Obj}$$

$$\frac{\Gamma;\Lambda;\Xi \vdash \mathit{object\_value} \Rightarrow \texttt{obj}\,y, \beta, \mathit{index\_term}}{\Gamma;\Lambda;\Xi \vdash \texttt{specified}\,\mathit{object\_value} \Rightarrow y, \beta, \mathit{index\_term}}\ \textsc{Val\_Loaded}$$

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

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

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

$$\frac{\Gamma;\Lambda;\Xi \vdash \mathit{value}_1 \Rightarrow y_1, \beta, \mathit{index\_term}_1 \quad .. \quad \Gamma;\Lambda;\Xi \vdash \mathit{value}_n \Rightarrow y_n, \beta, \mathit{index\_term}_n}{\Gamma;\Lambda;\Xi \vdash \beta[\mathit{value}_1, .., \mathit{value}_n] \Rightarrow y, [\beta], (\bigwedge(\mathit{index\_term}_1, .., \mathit{index\_term}_n))[y^{(k_1)}\,/y_1, .., y^{(k_n)}\,/y_n]}\ \textsc{Val\_List}$$

$$\frac{\Gamma;\Lambda;\Xi \vdash \mathit{value}_1 \Rightarrow y_1, \beta_1, \mathit{index\_term}_1 \quad .. \quad \Gamma;\Lambda;\Xi \vdash \mathit{value}_n \Rightarrow y_n, \beta_n, \mathit{index\_term}_n}{\Gamma;\Lambda;\Xi \vdash (\mathit{value}_1, .., \mathit{value}_n) \Rightarrow y, (\beta_1, .., \beta_n), \bigwedge(\mathit{index\_term}_1, .., \mathit{index\_term}_n)\,[y^{(k_1)}\,/y_1, .., y^{(k_n)}\,/y_n]}\ \textsc{Val\_Tuple}$$

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

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

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

$$\frac{\texttt{inconsistent}\,(\Gamma;\Lambda;\Xi)}{\Gamma;\Lambda;\Xi \vdash \texttt{error}\,(\mathit{string}, \mathit{ident}) \Rightarrow \Sigma\, y : \beta.\mathit{index\_term} \wedge \texttt{I}}\ \textsc{PExpr\_Error}$$

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

```
Definition rules:         16 good    0 bad
Definition rule clauses:  25 good    0 bad
```