index, i, j, k, l, m, n	index variables (subscripts)
ident	
integer_literal	
float_literal	
char_literal	
string_literal	
infix_symbol	
prefix_symbol	
location, l	store locations (not in the source syntax)
lowercase_ident	
capitalized_ident	

value_name, *x*	::=		
	|	*lowercase_ident*	
	|	(*operator_name*)	
operator_name	::=		
	|	*prefix_symbol*	
	|	*infix_op*	
infix_op	::=		
	|	*infix_symbol*	
	|	*	[L]
	|	=	[L]
	|	:=	[L]
constr_name, *C*	::=		
	|	*capitalized_ident*	
typeconstr_name, *tcn*	::=		
	|	*lowercase_ident*	
field_name, *fn*	::=		
	|	*lowercase_ident*	[d]
value_path	::=		
	|	*value_name*	
constr	::=		constructors: named, and built-in (including exceptions)
	|	*constr_name*	
	|	**Invalid_argument**	[L]
	|	**Not_found**	[L]
	|	**Assert_failure**	[L]
	|	**Match_failure**	[L]
	|	**Division_by_zero**	[L]
	|	**None**	[L]
	|	**Some**	[L]

typeconstr	::=			type constructors: named, and built-in
	|	*typeconstr_name*		
	|	**int**		[L]
	|	**char**		[L]
	|	**string**		[L]
	|	**float**		[L]
	|	**bool**		[L]
	|	**unit**		[L]
	|	**exn**		[L]
	|	**list**		[L]
	|	**option**		[L]
	|	**ref**		[L]
field	::=			
	|	*field_name*		[d]
idx, *num*	::=			index arithmetic for the type system's deBruijn type variable representation
	|	m		[I]
	|	$idx_1 + idx_2$		[I]
	|	(num)	S	[I]
σ^T	::=			multiple substitutions of types for type variables
	|	$\{\!\{\alpha_1 \leftarrow typexpr_1, .., \alpha_n \leftarrow typexpr_n\}\!\}$		[I]
	|	**shift** $num\ num'\ \sigma^T$	M	[I]
		shift the indices in the types in σ^T by num, ignoring indices lower than num'		
typexpr, *t*	::=			
	|	α		
	|	$<idx, num>$		[I]
		de Bruijn represenataion of type variables. num allows each binder (i.e., a polymorphic **let**) to introduce an arbitrary number of binders		
	|	_		
	|	$(typexpr)$	S	
	|	$typexpr_1 \rightarrow typexpr_2$		
	|	$typexpr_1 * .... * typexpr_n$		
	|	*typeconstr*	S	

		in the theorem prover models we use a uniform representation for 0-, 1-, and n-ary type constructor applications		
	|	*typexpr typeconstr*	S	
	|	$(typexpr_1, \ldots, typexpr_n) typeconstr$		
	|	**shift** *num num′ typexpr*	M	
		shifts as in σ^T above		
	|	$t_1 \to \ldots \to t_n \to t$	M	[I]
	|	σ^T *typexpr*	M	[I]
		apply the substitution		
src_typexpr, *src_t*	::=			types that can appear in source programs
	|	α		
	|	_		
	|	(*src_typexpr*)		
	|	$src\_typexpr_1 \to src\_typexpr_2$		
	|	$src\_typexpr_1 * \ldots . * src\_typexpr_n$		
	|	*typeconstr*		
	|	*src_typexpr typeconstr*		
	|	$(src\_typexpr_1, \ldots, src\_typexpr_n) typeconstr$		
	|	**shift** *num num′ src_typexpr*		
α	::=			
	|	*′ident*		
typescheme, *ts*	::=			
	|	$\forall$ *typexpr*		[I]
	|	**shift** *num num′ typescheme*	M	[I]
		shifts as in σ^T above		
$\dot{n}$	::=			integer mathematical expressions, used to implement primitive operations and **for** loops
	|	*integer_literal*		
	|	$(\dot{n})$	M	[I]
	|	$\dot{n}_1 \dot{+} \dot{n}_2$	M	[I]
	|	$\dot{n}_1 \dot{-} \dot{n}_2$	M	[I]
	|	$\dot{n}_1 \dot{*} \dot{n}_2$	M	[I]
	|	$\dot{n}_1 \dot{/} \dot{n}_2$	M	[I]

constant	::=			
	|	$\dot{n}$		[L]
	|	*float_literal*		[L]
	|	*char_literal*		[L]
	|	*string_literal*		[L]
	|	**equal_error_string**	M	[L]
		The string constant "equal: functional value"		
	|	*constr*		[L]
	|	**false**		[L]
	|	**true**		[L]
	|	[]		[L]
	|	()		[L]

pattern, *pat*	::=			
	|	*value_name*	xs = *value_name*	
	|	_	xs = {}	
	|	*constant*	xs = {}	
	|	*pattern* **as** *value_name*	xs = xs(*pattern*) ∪ *value_name*	
	|	(*pattern*)	S	
	|	(*pattern* : *typexpr*)	xs = xs(*pattern*)	
	|	$pattern_1 \| pattern_2$	xs = xs($pattern_1$)	
	|	$constr(pattern_1, \ldots, pattern_n)$	xs = xs($pattern_1 \ldots pattern_n$)	
	|	*constr* _	xs = {}	
	|	$pattern_1, \ldots., pattern_n$	xs = xs($pattern_1 \ldots. pattern_n$)	
	|	$\{field_1 = pattern_1; \ldots; field_n = pattern_n\}$	xs = xs($pattern_1 \ldots pattern_n$)	[d]
	|	$[pattern_1; \ldots; pattern_n]$	S	[L]
	|	$pattern_1 :: pattern_2$	xs = xs($pattern_1$) ∪ xs($pattern_2$)	[L]

unary_prim	::=			primitive functions with one argument
	|	**raise**		[L I]
	|	**not**		[L I]
	|	~−		[L I]
	|	**ref**		[L I]
	|	!		[L I]

binary_prim	::=			primitive functions with two arguments
	|	$=$		[L I]
	|	$+$		[L I]
	|	$-$		[L I]
	|	$*$		[L I]
	|	$/$		[L I]
	|	$:=$		[L I]
expr, *e*	::=			
	|	(%**prim** *unary_prim*)		[L I]
		a unary primitive function value		
	|	(%**prim** *binary_prim*)		[L I]
		a binary primitive function value		
	|	*value_name*		
	|	*constant*		
	|	(*expr*)	S	
	|	**begin** *expr* **end**	S	
	|	(*expr* : *typexpr*)		
	|	$expr_1 , \ldots. , expr_n$		
	|	$constr(expr_1 , .. , expr_n)$		
		potentially empty constructors to work around ott parser restriction		
	|	$expr_1 :: expr_2$		[L]
	|	$[expr_1 ; \ldots ; expr_n]$	S	[L]
	|	$\{field_1 = expr_1 ; \ldots ; field_n = expr_n\}$		[d]
	|	$\{expr$ **with** $field_1 = expr_1 ; \ldots ; field_n = expr_n\}$		[d]
	|	$expr_1\ expr_2$		
	|	*prefix_symbol expr*	S	
	|	$expr_1$ *infix_op* $expr_2$	S	
	|	$expr_1 \&\& expr_2$		[L]
	|	**AND** $(expr_1 \&\& .. \&\& expr_n)$	M	[L I]
		a delimited "and" operator with a list of arguments		
	|	$expr_1 \|\| expr_2$		[L]
	|	*expr*.*field*		[d]
	|	**if** $expr_0$ **then** $expr_1$	S	
	|	**if** $expr_0$ **then** $expr_1$ **else** $expr_2$		
	|	**while** $expr_1$ **do** $expr_2$ **done**		

	|	**for** x = $expr_1$ [**down**]**to** $expr_2$ **do** $expr_3$ **done**	bind x in $expr_3$	
	|	$expr_1$; $expr_2$		
	|	**match** *expr* **with** *pattern_matching*		
	|	**function** *pattern_matching*		
	|	**fun** $pattern_1$ ... $pattern_n$ → *expr*	S	
	|	**try** *expr* **with** *pattern_matching*		
	|	**let** *let_binding* **in** *expr*	bind xs(*let_binding*) in *expr*	
		omitting multiple bindings, i.e. **and**		
	|	**let rec** *letrec_bindings* **in** *expr*	bind xs(*letrec_bindings*) in *letrec_bindings*	
			bind xs(*letrec_bindings*) in *expr*	
	|	**assert** *expr*		
	|	*location*		[I]
	|	{{*substs_x*}}*expr*	M	[I]
		substitution of expressions for variables		
	|	**remv_tyvar** *expr*	M	[I]
		replace the type variables in an expression's type annotations with _		
[**down**]**to**	::=			
	|	**to**		
	|	**downto**		
substs_x	::=			substitutions of expressions for variables
	|	$value\_name_1$←$expr_1$, .. , $value\_name_n$←$expr_n$		[I]
	|	$substs\_x_1$@ .. @$substs\_x_n$	M	[I]
pattern_matching, *pm*	::=			
	|	pat_exp_1 | ... | pat_exp_n		
	|	| pat_exp_1 | ... | pat_exp_n	S	
pat_exp	::=			
	|	*pattern* → *expr*	bind xs(*pattern*) in *expr*	
let_binding	::=			
	|	*pattern* = *expr*	xs = xs(*pattern*)	
	|	*value_name* $pattern_1$ ... $pattern_n$ = *expr*	S	

	|	$\mathit{value\_name}\ \mathit{pattern}_1\ ...\ \mathit{pattern}_n : \mathit{typexpr} = \mathit{expr}$	S	
	|	$\{\!\{\alpha_1 \leftarrow \mathit{typexpr}_1, \ ..\ , \alpha_n \leftarrow \mathit{typexpr}_n\}\!\}\mathit{let\_binding}$	M	
		substitution of types for type variables		
$\mathit{letrec\_bindings}$	::=			
	|	$\mathit{letrec\_binding}_1\ \textbf{and}\ ...\ \textbf{and}\ \mathit{letrec\_binding}_n$	$\mathrm{xs} = \mathrm{xs}(\mathit{letrec\_binding}_1 ... \mathit{letrec\_binding}_n)$	
	|	$\{\!\{\alpha_1 \leftarrow \mathit{typexpr}_1, \ ..\ , \alpha_n \leftarrow \mathit{typexpr}_n\}\!\}\mathit{letrec\_bindings}$	M	
		substitution of types for type variables		
$\mathit{letrec\_binding}$	::=			
	|	$\mathit{value\_name} = \ \textbf{function}\ \mathit{pattern\_matching}$	$\mathrm{xs} = \mathit{value\_name}$	
	|	$\mathit{value\_name} = \ \textbf{fun}\ \mathit{pattern}\ \mathit{pattern}_1\ ..\ \mathit{pattern}_n \to \mathit{expr}$	S	
	|	$\mathit{value\_name}\ \mathit{pattern}\ \mathit{pattern}_1\ ..\ \mathit{pattern}_n = \mathit{expr}$	S	
	|	$\mathit{value\_name}\ \mathit{pattern}\ \mathit{pattern}_1\ ..\ \mathit{pattern}_n : \mathit{typexpr} = \mathit{expr}$	S	
$\mathit{type\_definition}$	::=			
	|	$\textbf{type}\ \mathit{typedef}_1\ \textbf{and}\ ..\ \textbf{and}\ \mathit{typedef}_n$	$\mathrm{type\_names} = \mathrm{type\_names}(\mathit{typedef}_1 .. \mathit{typedef}_n)$	[d]
		potentially empty definitions to work around Ott parser restrictions		
			$\mathrm{constr\_names} = \mathrm{constr\_names}(\mathit{typedef}_1 .. \mathit{typedef}_n)$	
$\mathit{typedef}$	::=			
	|	$\mathit{type\_params\_opt}\ \mathit{typeconstr\_name}\ \mathit{type\_information}$	bind $\mathrm{typevars}(\mathit{type\_params\_opt})$ in $\mathit{type\_information}$	[d]
			$\mathrm{type\_names} = \mathit{typeconstr\_name}$	
			$\mathrm{constr\_names} = \mathrm{constr\_names}(\mathit{type\_information})$	
$\mathit{type\_information}$	::=			
	|	$\mathit{type\_equation}$	$\mathrm{constr\_names} = \{\}$	[d]
			$\mathrm{field\_names} = \{\}$	
	|	$\mathit{type\_representation}$	$\mathrm{constr\_names} = \mathrm{constr\_names}(\mathit{type\_representation})$	[d]
			$\mathrm{field\_names} = \mathrm{field\_names}(\mathit{type\_representation})$	
$\mathit{type\_equation}$	::=			
	|	$= \mathit{typexpr}$		[d]
$\mathit{type\_representation}$	::=			

	|	$=$ *constr_decl*$_1$ | ... | *constr_decl*$_n$	constr_names = constr_names(*constr_decl*$_1$...*constr_decl*$_n$) field_names = {}	[d]
	|	$=$ {*field_decl*$_1$; ... ; *field_decl*$_n$}	constr_names = {} field_names = field_names(*field_decl*$_1$...*field_decl*$_n$)	[d]
type_params_opt	::=			
	|		S	[d]
		in the theorem prover models we use a uniform representation for empty, singleton and multiple type paramaters		
	|	*type_param*	S	[d]
	|	(*type_param*$_1$, ... , *type_param*$_n$)	typevars = typevars(*type_param*$_1$...*type_param*$_n$)	[d]
type_param, *tp*	::=			
	|	α	typevars = α	[d]
constr_decl	::=			
	|	*constr_name*	constr_names = *constr_name*	[d]
	|	*constr_name* **of** *typexpr*$_1$ $*$ ... $*$ *typexpr*$_n$	constr_names = *constr_name*	[d]
field_decl	::=			
	|	*field_name* : *typexpr*	field_names = *field_name*	[d]
exception_definition	::=			
	|	**exception** *constr_decl*		[d]
definition, *d*	::=			
	|	**let** *let_binding*	xs = xs(*let_binding*)	[d]
		omitting multiple bindings, i.e. **and**		
	|	**let rec** *letrec_bindings*	xs = xs(*letrec_bindings*) bind xs(*letrec_bindings*) in *letrec_bindings*	[d]
	|	*type_definition*	xs = {}	[d]
	|	*exception_definition*	xs = {}	[d]
definitions, *ds*	::=			
	|			[d]
	|	*definition definitions*	bind xs(*definition*) in *definitions*	[d]

	|	*definition*; ; *definitions*	S	[d]
	|	$\{\!\{$ *substs_x* $\}\!\}$ *definitions*	M	[d]
		substitution of expressions for variables		
	|	*definitions definition*	M	[d]
		adding a definition to the end of a sequence		
	|	*definitions*; ; *definition*	M	[d]
program	::=			
	|	*definitions*		[d]
	|	(%*primraise*) *expr*		[d]
value, *v*	::=			core value
	|	(%**prim** *unary_prim*)		[L l]
	|	(%**prim** *binary_prim*)		[L l]
	|	*binary_prim_app_value value*		[l]
		partially applied binary primitive		
	|	*constant*		[l]
	|	(*value*)		[l]
	|	$value_1, \ldots., value_n$		[l]
	|	$constr(value_1, .., value_n)$		[l]
	|	$value_1 :: value_2$		[L l]
	|	$[value_1; \ldots; value_n]$		[L l]
	|	$\{field_1 = value_1; \ldots; field_n = value_n\}$		[d l]
	|	**function** *pattern_matching*		[l]
	|	**fun** $pattern_1 \ldots pattern_n \to expr$		[l]
	|	*location*		[l]
binary_prim_app_value	::=			
	|	(%**prim** *binary_prim*)		[l]
definition_value, *d_value*	::=			
	|	*type_definition*		[d l]
	|	*exception_definition*		[d l]
definitions_value, *ds_value*	::=			

	|			[d l]
	|	*definition_value definitions_value*		[d l]
	|	*definition_value*; ; *definitions_value*		[d l]
non_expansive, *nexp*	::=			nonexpansive expression (allowed in a polymorphic let)
	|	(%**prim** *unary_prim*)		[l]
	|	(%**prim** *binary_prim*)		[l]
	|	*binary_prim_app_value nexp*		[l]
		partially applied binary primitive		
	|	*value_name*		[l]
	|	*constant*		[l]
	|	(*nexp*)		[l]
	|	(*nexp* : *typexpr*)		[l]
	|	$nexp_1$, ... , $nexp_n$		[l]
	|	*constr*($nexp_1$, .. , $nexp_n$)		[l]
	|	$nexp_1$ :: $nexp_2$		[l]
	|	[$nexp_1$; ... ; $nexp_n$]		[L l]
	|	{$field_1$ = $nexp_1$; ... ; $field_n$ = $nexp_n$}		[d l]
	|	**let rec** *letrec_bindings* **in** *nexp*		[l]
	|	**function** *pattern_matching*		[l]
	|	**fun** $pattern_1$ ... $pattern_n$ → *expr*		[l]
	|	*location*		[l]
store, *st*	::=			
	|	**empty**		[l]
	|	*store*, *location* ↦ *expr*		[l]
	|	*store*, *location* ↦ *expr*, *store'*	M	[l]
kind	::=			
	|	$\mathbf{Type}^{num}$ → **Type**		[l]
	|	**Type**	S	[l]
name	::=			environment lookup key
	|	**TV**		[l]
	|	*value_name*		[l]

	|	*constr_name*		[d l]
	|	*typeconstr_name*		[d l]
	|	*field_name*		[d l]
	|	*location*		[l]
names	::=			
	|	$name_1$ .. $name_n$		[l]
typexprs	::=			
	|	$typexpr_1$, .. , $typexpr_n$		[l]
	|	**shift** *num* num' *typexprs*	M	[l]
		shift the indices in the types in *typexprs* by *num*, ignoring indices lower than num'		
environment_binding, *EB*	::=			
	|	**TV**		[l]
		type variable		
	|	*value_name* : *typescheme*		[l]
		value binding		
	|	*value_name* : *typexpr*	M	[l]
		value binding with no universal quantifier		
	|	*constr_name* **of** *typeconstr*		[d l]
		constant constructor		
	|	*constr_name* **of** $\forall$ *type_params_opt*, (*typexprs*) : *typeconstr*	bind typevars(*type_params_opt*) in *typexprs*	[d l]
		parameterised constructor		
	|	*field_name* : $\forall$ *type_params_opt*, *typeconstr_name* $\rightarrow$ *typexpr*	bind typevars(*type_params_opt*) in *typexpr*	[d l]
		field name a record destructor		
	|	*typeconstr_name* : *kind*		[d l]
		type name, bound to a fresh type		
	|	*typeconstr_name* : *kind*{$field\_name_1$; ... ; $field\_name_n$}		[d l]
		type name which is a record type definition		
	|	*type_params_opt* *typeconstr_name* = *typexpr*	bind typevars(*type_params_opt*) in *typexpr*	[d l]
		type name which is an abbreviation		
	|	*location* : *typexpr*		[l]
		location (memory cell)		
	|	(*EB*)	M	[l]

	|	$\mathbf{shift}\ num\ num'\ EB$	M	[I]
		shift the indices in the types in EB by num, ignoring indices lower than num'		
$environment,\ E$	::=			
	|	$\mathbf{empty}$		[I]
	|	E, EB		[I]
	|	$EB_1, .., EB_n$	M	[I]
	|	$E_1 @ .. @ E_n$	M	[I]
$trans\_label,\ L$	::=			reduction label (denoting a side effect)
	|			[I]
	|	$\mathbf{ref}\ v = location$		[I]
	|	$!location = v$		[I]
	|	$location := v$		[I]
	|	L	M	[I]
$\xrightarrow{L}$	::=			
	|	$\xrightarrow{L}$		[I]
$formula$	::=			
	|	$judgement$		
	|	$formula_1 \quad .. \quad formula_n$		
	|	$\dot{n}_1 \mathrel{\dot{\leq}} \dot{n}_2$		
	|	$\dot{n}_1 \mathrel{\dot{>}} \dot{n}_2$		
	|	$num_1 < num_2$		
	|	$E = E'$		
	|	$expr = expr'$		
	|	$typexpr = typexpr'$		
	|	$typescheme = typescheme'$		
	|	$type\_params\_opt = type\_params\_opt'$		
	|	$letrec\_bindings = (letrec\_bindings')$		
	|	$\mathbf{length}\,(tp_1)\,..\,(tp_n) = m$		
	|	$\mathbf{length}\,(t_1)\,..\,(t_n) = num$		
	|	$\mathbf{length}\,(t_1)\,..\,(t_n) \leq num$		
	|	$\mathbf{length}\,(t_1)\,..\,(t_n) \geq num$		

$$
\begin{array}{lcll}
 & | & \mathbf{length}\,(pat_1)\,..\,(pat_n) \geq m & \\
 & | & \mathbf{length}\,(e_1)\,..\,(e_n) \geq m & \\
 & | & name \notin names & \\
 & | & field\_name\ \mathbf{in}\ field\_name_1\,..\,field\_name_n & \textsf{[d]} \\
 & | & type\_param\ \mathbf{in}\ type\_params\_opt & \\
 & | & name_1\,..\,name_n\ \mathbf{distinct} & \\
 & | & tp_1\,..\,tp_n\ \mathbf{distinct} & \\
 & | & E\ \mathbf{PERMUTES}\ E' & \\
 & | & fn_1\,..\,fn_n\ \mathbf{PERMUTES}\ fn'_1\,..\,fn'_m & \textsf{[d]} \\
 & | & fn_1 = e_1\,..\,fn_n = e_n\ \mathbf{PERMUTES}\ fn'_1 = e'_1\,..\,fn'_m = e'_m & \textsf{[d]} \\
 & | & \neg(value\ \mathbf{matches}\ pattern) & \\
 & | & constant \neq constant' & \\
 & | & name \neq name' & \\
 & | & store(location)\ \mathbf{unallocated} & \\
 & | & \mathbf{type\_vars}\,(let\_binding) \ \triangleright\ \alpha_1,\,..\,,\alpha_n & \\
 & | & \mathbf{type\_vars}\,(letrec\_bindings) \ \triangleright\ \alpha_1,\,..\,,\alpha_n & \\
 & & & \\
terminals & ::= & & \text{prettyprinting specifications} \\
 & | & \%\mathbf{prim} & \\
 & | & * & \\
 & | & \rightarrow & \\
 & | & \longrightarrow & \\
 & | & \rightarrow & \\
 & | & \leftarrow & \\
 & | & \{\!\!\{ & \\
 & | & \leq & \\
 & | & \equiv & \\
 & | & \geq & \\
 & | & \}\!\!\} & \\
 & | & \forall & \\
 & | & \triangleright & \\
 & | & \notin & \\
 & | & \neq & \\
 & | & \vdash & \\
 & | & \mapsto & \\
 & | & \sim- & \\
\end{array}
$$

$JdomEB$ ::=
| $\mathbf{dom}\,(EB) \;\triangleright\; name$
  Environment binding domain

$JdomE$ ::=
| $\mathbf{dom}\,(E) \;\triangleright\; names$
  Environment domain

$Jlookup$ ::=
| $E \vdash name \;\triangleright\; EB$
  Environment lookup

$Jidx$ ::=
| $E \vdash idx\,\mathbf{bound}$
  Well-formed index

$JTtps\_kind$ ::=
| $\vdash type\_params\_opt : kind$
  Type parameter kinding

$JTEok$ ::=
| $E \vdash \mathbf{ok}$
  Environment validity
| $E \vdash typeconstr : kind$
  Type constructor kinding
| $E \vdash typescheme : kind$
  de Bruijn type scheme well-formedness
| $E \vdash \forall\, type\_params\_opt, t : kind$
  Named type scheme well-formedness
| $E \vdash typexpr : kind$
  Type expression well-formedness

$JTeq$ ::=
| $E \vdash typexpr \equiv typexpr'$
  Type equivalence

$JTidxsub$ ::=
| $\{\!\{ typexpr_1, \,..\, , typexpr_n \}\!\} typexpr' \quad \triangleright \quad typexpr''$
de Bruin type substitution

$JTinst$ ::=
| $E \vdash typexpr \leq typescheme$
de Bruijn type scheme instantiation

$JTinst\_named$ ::=
| $E \vdash typexpr \leq \forall\, type\_params\_opt, typexpr'$
Named type scheme instantiation

$JTinst\_any$ ::=
| $E \vdash typexpr \leq typexpr'$
Wildcard type instantiation

$JTval$ ::=
| $E \vdash value\_name : typexpr$
Variable typing

$JTfield$ ::=
| $E \vdash field\_name : typexpr \to typexpr'$
Field name typing

$JTconstr\_p$ ::=
| $E \vdash constr : typexpr_1 \ldots typexpr_n \to typexpr'$
Non-constant constructor typing

$JTconstr\_c$ ::=
| $E \vdash constr : typexpr$
Constant constructor typing

$JTconst$ ::=
| $E \vdash constant : typexpr$
Constant typing

$JTpat$ ::=
| $\sigma^T \& E \vdash pattern : typexpr \quad \triangleright \quad E'$
Pattern typing and binding collection

$JTuprim$ ::=
| $E \vdash unary\_prim : typexpr$
Unary primitive typing

$JTbprim$ ::=
| $E \vdash binary\_prim : typexpr$
Binary primitive typing

JTe ::=
| $\sigma^T \& E \vdash expr : typexpr$
Expression typing
| $\sigma^T \& E \vdash pattern\_matching : typexpr \rightarrow typexpr'$
Pattern matching/expression pair typing
| $\sigma^T \& E \vdash let\_binding \quad \triangleright \quad E'$
Let binding typing
| $\sigma^T \& E \vdash letrec\_bindings \quad \triangleright \quad E'$
Recursive let binding typing

$JTconstr\_decl$ ::=
| $type\_params\_opt\ typeconstr \vdash constr\_decl \quad \triangleright \quad EB$
Variant constructor declaration

$JTfield\_decl$ ::=
| $type\_params\_opt\ typeconstr\_name \vdash field\_decl \quad \triangleright \quad EB$
Record field declaration

$JTtypedef$ ::=
| $\vdash typedef_1 \ \mathbf{and} \ .. \ \mathbf{and} \ typedef_n \quad \triangleright \quad E' \ \mathbf{and} \ E'' \ \mathbf{and} \ E'''$
Type definitions collection

$JTtype\_definition$ ::=

| | $E \vdash \mathit{type\_definition} \;\; \triangleright \;\; E'$
Type definition well-formedness and binding collection

$JTdefinition$ ::=
| $E \vdash \mathit{definition} : E'$
Definition typing

$JTdefinitions$ ::=
| $E \vdash \mathit{definitions} : E'$
Definition sequence typing

$JTprog$ ::=
| $E \vdash \mathit{program} : E'$
Program typing

$JTstore$ ::=
| $E \vdash \mathit{store} : E'$
Store typing

$JTtop$ ::=
| $E \vdash \langle \mathit{program}, \mathit{store} \rangle$
Top-level typing

$JTLin$ ::=
| $\sigma^T \& E \vdash L$
Label-to-environment extraction

$JTLout$ ::=
| $\sigma^T \& E \vdash L \;\; \triangleright \;\; E'$
Label-to-environment extraction

$JmatchP$ ::=
| $\vdash \mathit{expr}\, \mathbf{matches}\, \mathit{pattern}$
Pattern matching

$Jmatch$	::=	
	|	$\vdash expr\ \mathbf{matches}\ pattern \ \ \triangleright\ \ \{\!\{ substs\_x \}\!\}$ Pattern matching with substitution creation
$Jrecfun$	::=	
	|	$\mathbf{recfun}\,(letrec\_bindings, pattern\_matching) \ \ \triangleright\ \ expr$ Recursive function helper
$Jfunval$	::=	
	|	$\vdash\ \mathbf{funval}\,(e)$ Function values
$JRuprim$	::=	
	|	$\vdash unary\_prim\ expr \xrightarrow{L} expr'$ Unary primitive evaluation
$JRbprim$	::=	
	|	$\vdash expr_1\ binary\_prim\ expr_2 \xrightarrow{L} expr$ Binary primitive evaluation
$JRmatching\_step$	::=	
	|	$\vdash expr\ \mathbf{with}\ pattern\_matching \longrightarrow pattern\_matching'$ Pattern matching step
$JRmatching\_success$	::=	
	|	$\vdash expr\ \mathbf{with}\ pattern\_matching \longrightarrow expr'$ Pattern matching finished
$Jred$	::=	
	|	$\vdash expr \xrightarrow{L} expr'$ Expression evaluation
$JRdefn$	::=	
	|	$\vdash \langle definitions, program\rangle \xrightarrow{L} \langle definitions', program'\rangle$ Definition sequence evaluation

$JSlookup$ ::=
| $store(location) \ \triangleright \ expr$
  Store lookup

$JRstore$ ::=
| $\vdash store \xrightarrow{L} store'$
  Store transition

$JRtop$ ::=
| $\vdash \langle definitions, program, store\rangle \longrightarrow \langle definitions', program', store'\rangle$
  Top-level reduction

$Jebehaviour$ ::=
| $\vdash expr$ **behaves**
  Expression behaviour

$Jdbehaviour$ ::=
| $\vdash \langle definitions, program, store\rangle$**behaves**
  structure body behaviour

$judgement$ ::=
| $JdomEB$
| $JdomE$
| $Jlookup$
| $Jidx$
| $JTtps\_kind$
| $JTEok$
| $JTeq$
| $JTidxsub$
| $JTinst$
| $JTinst\_named$
| $JTinst\_any$
| $JTval$
| $JTfield$
| $JTconstr\_p$

	$\vert$	$JTconstr\_c$
	$\vert$	$JTconst$
	$\vert$	$JTpat$
	$\vert$	$JTuprim$
	$\vert$	$JTbprim$
	$\vert$	JTe
	$\vert$	$JTconstr\_decl$
	$\vert$	$JTfield\_decl$
	$\vert$	$JTtypedef$
	$\vert$	$JTtype\_definition$
	$\vert$	$JTdefinition$
	$\vert$	$JTdefinitions$
	$\vert$	$JTprog$
	$\vert$	$JTstore$
	$\vert$	$JTtop$
	$\vert$	$JTLin$
	$\vert$	$JTLout$
	$\vert$	$JmatchP$
	$\vert$	$Jmatch$
	$\vert$	$Jrecfun$
	$\vert$	$Jfunval$
	$\vert$	$JRuprim$
	$\vert$	$JRbprim$
	$\vert$	$JRmatching\_step$
	$\vert$	$JRmatching\_success$
	$\vert$	$Jred$
	$\vert$	$JRdefn$
	$\vert$	$JSlookup$
	$\vert$	$JRstore$
	$\vert$	$JRtop$
	$\vert$	$Jebehaviour$
	$\vert$	$Jdbehaviour$
$user\_syntax$	$::=$	
	$\vert$	$index$
	$\vert$	$ident$

| *integer_literal*
| *float_literal*
| *char_literal*
| *string_literal*
| *infix_symbol*
| *prefix_symbol*
| *location*
| *lowercase_ident*
| *capitalized_ident*
| *value_name*
| *operator_name*
| *infix_op*
| *constr_name*
| *typeconstr_name*
| *field_name*
| *value_path*
| *constr*
| *typeconstr*
| *field*
| *idx*
| σ^T
| *typexpr*
| *src_typexpr*
| α
| *typescheme*
| $\dot{n}$
| *constant*
| *pattern*
| *unary_prim*
| *binary_prim*
| *expr*
| [**down**]**to**
| *substs_x*
| *pattern_matching*
| *pat_exp*
| *let_binding*

| *letrec_bindings*
| *letrec_binding*
| *type_definition*
| *typedef*
| *type_information*
| *type_equation*
| *type_representation*
| *type_params_opt*
| *type_param*
| *constr_decl*
| *field_decl*
| *exception_definition*
| *definition*
| *definitions*
| *program*
| *value*
| *binary_prim_app_value*
| *definition_value*
| *definitions_value*
| *non_expansive*
| *store*
| *kind*
| *name*
| *names*
| *typexprs*
| *environment_binding*
| *environment*
| *trans_label*
| $\xrightarrow{L}$
| *formula*
| *terminals*

A formal specification for OCaml: the Core Language

Scott Owens and Gilles Peskine and Peter Sewell

September 6, 2023

Contents

1 Introduction

This document describes the syntax and semantics of a substantial fragment of Objective Caml's core language. When writing this semantics, we have followed the structure of part 2 of the Objective Caml manual:

> The Objective Caml system
> release 3.09
> Documentation and user's manual
> Xavier Leroy (with Damien Doligez, Jacques Garrigue, Didier Rémy and Jérôme Vouillon)
> Copyright © 2005 Institut National de Recherche en Informatique et en Automatique

Our aim is to describe a real language, including theoretically redundant but practically useful features. We do not however cover the whole Objective Caml language: we have omitted some major semantic features, such as objects and modules. Our guideline is to retain the semantic features of core ML as implemented in Objective Caml. Our language corresponds roughly to the fragment presented in Chapter 1 of the Objective Caml manual.

Supported features include:

- the following primitive types and type constructors: `int`, `char`, `string`, `float`, `bool`, `unit`, `exn`, `list`, `option`, `ref`;
- tuple and function types
- type and type constructor definitions, including:
  - type abbreviations (e.g., `type t = int`),
  - variant data and record types (e.g., `type t = I of int || D of char` and `type t = {f:int}`),
  - parametric type constructors (e.g., `type 'a t = 'a -> 'a`),

- recursive and mutually recursive combinations of the above (although all recursion must go through a variant data or record type);

- let-based polymorphism (with the traditional ML-style value restriction);
- 31-bit word semantics for integers and IEEE-754 semantics for floating point numbers (in the version of the system generated for HOL);
- type annotations (e.g., `3:int`), list notation (e.g., [1; 2; 3]), record `with` expressions, `if` expressions, `while` expressions, `for` expressions, sequencing (`;`), `assert` expressions;
- (potentially) mutually-recursive function definitions;
- pattern matching with nested patterns and } patterns;
- mutable references through `ref`, `:=`, and `!`;
- exception definitions and handling (`try`, `raise`, `exception`);
- polymorphic equality (the `=` operator).

The following features are not supported:

- mutable records (e.g., {`mutable l1=e1;...;mutable ln=en`});
- arrays;
- modules;
- subtyping, labels, polymorphic variants, objects;
- pattern matching guards (`when`);
- features documented in the "language extensions" part of the manual;
- `-rectypes`, exhaustivity of pattern matching, and other compiler command-line options;
- support for type abbreviations in the HOL model (we explain in the commentary how they should be added);
- finiteness of memory.

This document contains a description of the language syntax (§2), a type system (§3) and an operational semantics (§4).

Metatheory This typeset definition is generated by `ott`. Well-formed definitions in HOL, Isabelle/HOL and Coq are also generated. We have mechanized the type soundness theorem for the system in HOL.

2 Syntax

We describe the syntax of the core OCaml language in BNF form, closely following the description in the Objective Caml manual, but omitting unsupported language features. The concrete syntax of Objective Caml includes lexical specifications as well as precedence rules to disambiguate the grammar; we do not reproduce these here.

Some productions mention annotations to the right of the right-hand side. The following annotations are understood by Ott.

- M indicates a metaproduction. These are not part of the free grammar for the relevant nonterminal. Instead they are given meaning (in the theorem prover models) by translation into non-metaproductions. These translations, specified in the Ott source, are specific to each theorem prover. We summarize their action in this document.
- S indicates a metaproduction that is implemented as syntactic sugar.
- “bind ...” and “*auxfun* = ...” are Ott binding specifications.

The following annotations are for informational purposes only.

- [I] indicates a production that is not intended to be available in user programs but is useful in the metatheory.
- [L] indicates a library facility (as opposed to a strictly language facility).
- d indicates a definition-level feature, if enabled.

3 Type system

The Objective Caml manual does not describe the type system. Therefore our semantics is driven by an attempt to mirror what the Objective Caml implementation does, drawing inspiration from various presentations of type systems for ML. Some notable aspects of the formalization follow:

- We give a declarative presentation of polymorphic typing, i.e., without unification.
- Polymorphic **let** introduces type variables which are encoded with de Bruijn indices.
- Several rules have premises that state there are at least 1 (or 2) elements of a list, despite there being 3 or 4 dots. This is because Ott does not use dot imposed length restrictions in the theorem prover models.
- Occasionally, we state that some list X1 .. Xm has length m. Ott does not impose this restriction in the theorem prover models either.

3.1 $\boxed{\mathbf{dom}\,(EB) \;\triangleright\; \mathit{name}}$ Environment binding domain

Gets the name of an environment entry.

$$\frac{}{\mathbf{dom}\,(\mathbf{TV}) \;\triangleright\; \mathbf{TV}} \quad \textsf{JdomEB\_type\_param}$$

$$\frac{}{\mathbf{dom}\,(\mathit{value\_name} : \mathit{typescheme}) \;\triangleright\; \mathit{value\_name}} \quad \textsf{JdomEB\_value\_name}$$

$$\frac{}{\mathbf{dom}\,(\mathit{constr\_name}\;\mathbf{of}\;\mathit{typeconstr}) \;\triangleright\; \mathit{constr\_name}} \quad \textsf{JdomEB\_const\_constr\_name}$$

$$\frac{}{\mathbf{dom}\,(\mathit{constr\_name}\;\mathbf{of}\;\forall\,\mathit{type\_params\_opt}, (t_1, \ldots, t_n) : \mathit{typeconstr}) \;\triangleright\; \mathit{constr\_name}} \quad \textsf{JdomEB\_constr\_name}$$

$$\frac{}{\mathbf{dom}\,(\mathit{typeconstr\_name} : \mathit{kind}) \;\triangleright\; \mathit{typeconstr\_name}} \quad \textsf{JdomEB\_opaque\_typeconstr\_name}$$

$$\frac{}{\mathbf{dom}\,(\mathit{type\_params\_opt}\;\mathit{typeconstr\_name} = t) \;\triangleright\; \mathit{typeconstr\_name}} \quad \textsf{JdomEB\_trans\_typeconstr\_name}$$

$$\frac{}{\mathbf{dom}\,(\mathit{typeconstr\_name} : \mathit{kind}\{\mathit{field\_name}_1; \ldots; \mathit{field\_name}_n\}) \;\triangleright\; \mathit{typeconstr\_name}} \quad \textsf{JdomEB\_record\_typeconstr\_name}$$

$$\frac{}{\mathbf{dom}\,(\mathit{field\_name} : \forall\,\mathit{type\_params\_opt}, \mathit{typeconstr\_name} \to \mathit{typexpr}) \;\triangleright\; \mathit{field\_name}} \quad \textsf{JdomEB\_record\_field\_name}$$

$$\frac{}{\mathbf{dom}\,(\mathit{location} : t) \;\triangleright\; \mathit{location}} \quad \textsf{JdomEB\_location}$$

3.2 $\boxed{\mathbf{dom}\,(E) \;\triangleright\; \mathit{names}}$ Environment domain

Gets all of the names in an environment.

$$\frac{}{\mathbf{dom}\,(\mathbf{empty}) \;\triangleright\;} \quad \textsf{JdomE\_empty}$$

$$\frac{\begin{array}{c}\mathbf{dom}\,(E) \;\triangleright\; \mathit{name}_1 \,..\, \mathit{name}_n \\ \mathbf{dom}\,(EB) \;\triangleright\; \mathit{name}\end{array}}{\mathbf{dom}\,(E, EB) \;\triangleright\; \mathit{name}\;\mathit{name}_1 \,..\, \mathit{name}_n} \quad \textsf{JdomE\_cons}$$

3.3 $\boxed{E \vdash name \;\; \triangleright \;\; EB}$ Environment lookup

Returns the rightmost binding that matches the given name.

$$\frac{\begin{array}{l}\mathbf{dom}\,(EB) \;\; \triangleright \;\; name' \\ name \neq name' \\ name' \neq \mathbf{TV} \\ E \vdash name \;\; \triangleright \;\; EB'\end{array}}{E, EB \vdash name \;\; \triangleright \;\; EB'} \quad \textsf{Jlookup\_EB\_rec1}$$

$$\frac{\begin{array}{l}name \neq \mathbf{TV} \\ E \vdash name \;\; \triangleright \;\; EB'\end{array}}{E, \mathbf{TV} \vdash name \;\; \triangleright \;\; \mathbf{shift}\, 0\, 1\, EB'} \quad \textsf{Jlookup\_EB\_rec2}$$

$$\frac{\mathbf{dom}\,(EB) \;\; \triangleright \;\; name}{E, EB \vdash name \;\; \triangleright \;\; EB} \quad \textsf{Jlookup\_EB\_head}$$

3.4 $\boxed{E \vdash idx\, \mathbf{bound}}$ Well-formed index

Determines whether an index is bound by an environment.

$$\frac{\begin{array}{l}E \vdash idx\, \mathbf{bound} \\ \mathbf{dom}\,(EB) \;\; \triangleright \;\; name \\ name \neq \mathbf{TV}\end{array}}{E, EB \vdash idx\, \mathbf{bound}} \quad \textsf{Jidx\_bound\_skip1}$$

$$\frac{E \vdash idx\, \mathbf{bound}}{E, \mathbf{TV} \vdash idx + 1\, \mathbf{bound}} \quad \textsf{Jidx\_bound\_skip2}$$

$$\frac{}{E, \mathbf{TV} \vdash 0\, \mathbf{bound}} \quad \textsf{Jidx\_bound\_found}$$

3.5 $\boxed{\vdash type\_params\_opt : kind}$ Type parameter kinding

Counts the number of parameters and ensures that none is repeated.

$$\frac{\begin{array}{l}tp_1 \ldots tp_n\, \mathbf{distinct} \\ \mathbf{length}\,(tp_1) \ldots (tp_n) = n\end{array}}{\vdash (tp_1, \ldots, tp_n) : \mathbf{Type}^n \to \mathbf{Type}} \quad \textsf{JTtps\_kind\_kind}$$

3.6 $\boxed{E \vdash \mathbf{ok}}$ Environment validity

Asserts that the various components of the environment are well-formed (including that there are no free type references), and regulates name shadowing. Environments contain identifiers related to type definitions and type variables as well as expression-level variables (i.e., value names), so they are dependent from left to right. Shadowing of type, constructor, field and label names is forbidden, but shadowing of value names is allowed.

$$\frac{}{\mathbf{empty} \vdash \mathbf{ok}} \quad \textsf{JTEok\_empty}$$

$$\frac{E \vdash \mathbf{ok}}{E, \mathbf{TV} \vdash \mathbf{ok}} \quad \textsf{JTEok\_typevar}$$

$$\frac{E \vdash \forall\, t : \mathbf{Type}}{E, (\mathit{value\_name} : \forall\, t) \vdash \mathbf{ok}} \quad \textsf{JTEok\_value\_name}$$

$$\frac{\begin{array}{l} E \vdash \mathbf{ok} \\ E \vdash \mathit{typeconstr\_name} \ \triangleright\ \mathit{typeconstr\_name} : \mathit{kind} \\ \mathbf{dom}\,(E) \ \triangleright\ \mathit{names} \\ \mathit{constr\_name} \notin \mathit{names} \end{array}}{E, (\mathit{constr\_name}\, \mathbf{of}\ \mathit{typeconstr\_name}) \vdash \mathbf{ok}} \quad \textsf{JTEok\_constr\_name\_c}$$

$$\frac{\begin{array}{l} E \vdash \mathbf{ok} \\ \mathbf{dom}\,(E) \ \triangleright\ \mathit{names} \\ \mathit{constr\_name} \notin \mathit{names} \end{array}}{E, (\mathit{constr\_name}\, \mathbf{of}\, \mathbf{exn}) \vdash \mathbf{ok}} \quad \textsf{JTEok\_exn\_constr\_name\_c}$$

$$\frac{\begin{array}{l} \mathit{type\_params\_opt} = (\alpha_1, \ldots, \alpha_m) \\ E \vdash \forall\, \mathit{type\_params\_opt}, t_1 : \mathbf{Type} \quad \ldots \quad E \vdash \forall\, \mathit{type\_params\_opt}, t_n : \mathbf{Type} \\ E \vdash \mathit{typeconstr\_name} \ \triangleright\ \mathit{typeconstr\_name} : \mathbf{Type}^m \to \mathbf{Type} \\ \mathbf{dom}\,(E) \ \triangleright\ \mathit{names} \\ \mathit{constr\_name} \notin \mathit{names} \\ \mathbf{length}\,(t_1) \ldots (t_n) \geq 1 \\ \mathbf{length}\,(\alpha_1) \ldots (\alpha_m) = m \end{array}}{E, (\mathit{constr\_name}\, \mathbf{of}\, \forall\, (\alpha_1, \ldots, \alpha_m), (t_1, \ldots, t_n) : \mathit{typeconstr\_name}) \vdash \mathbf{ok}} \quad \textsf{JTEok\_constr\_name\_p}$$

$$\frac{\begin{array}{l} E \vdash t_1 : \mathbf{Type} \quad \ldots \quad E \vdash t_n : \mathbf{Type} \\ \mathbf{dom}\,(E) \;\; \rhd \;\; \mathit{names} \\ \mathit{constr\_name} \;\notin\; \mathit{names} \\ \mathbf{length}\,(t_1) \ldots (t_n) \geq 1 \end{array}}{E, (\mathit{constr\_name}\, \mathbf{of}\, \forall\,, (t_1,\, \ldots,\, t_n) : \mathbf{exn}) \vdash \mathbf{ok}} \quad \text{JTEok\_exn\_constr\_name\_p}$$

$$\frac{\begin{array}{l} E \vdash \forall\,(\alpha_1,\, \ldots,\, \alpha_m), t : \mathbf{Type} \\ \mathbf{dom}\,(E) \;\; \rhd \;\; \mathit{names} \\ \mathit{field\_name} \;\notin\; \mathit{names} \\ E \vdash \mathit{typeconstr\_name} \;\; \rhd \;\; \mathit{typeconstr\_name} : \mathbf{Type}^m \to \mathbf{Type}\{\mathit{field\_name}_1;\, \ldots;\, \mathit{field\_name}_n\} \\ \mathbf{length}\,(\alpha_1) \ldots (\alpha_m) = m \\ \mathit{field\_name}\, \mathbf{in}\, \mathit{field\_name}_1 \ldots \mathit{field\_name}_n \end{array}}{E, (\mathit{field\_name} :\, \forall\,(\alpha_1,\, \ldots,\, \alpha_m), \mathit{typeconstr\_name} \to t) \vdash \mathbf{ok}} \quad \text{JTEok\_record\_destr}$$

$$\frac{\begin{array}{l} E \vdash \mathbf{ok} \\ \mathbf{dom}\,(E) \;\; \rhd \;\; \mathit{names} \\ \mathit{typeconstr\_name} \;\notin\; \mathit{names} \end{array}}{E, (\mathit{typeconstr\_name} : \mathit{kind}) \vdash \mathbf{ok}} \quad \text{JTEok\_typeconstr\_name}$$

$$\frac{\begin{array}{l} \mathbf{dom}\,(E) \;\; \rhd \;\; \mathit{names} \\ \mathit{typeconstr\_name} \;\notin\; \mathit{names} \\ E \vdash \forall\,(\alpha_1,\, \ldots,\, \alpha_m), t : \mathbf{Type} \end{array}}{E, ((\alpha_1,\, \ldots,\, \alpha_m)\; \mathit{typeconstr\_name} = t) \vdash \mathbf{ok}} \quad \text{JTEok\_typeconstr\_eqn}$$

$$\frac{\begin{array}{l} E \vdash \mathbf{ok} \\ \mathbf{dom}\,(E) \;\; \rhd \;\; \mathit{names} \\ \mathit{typeconstr\_name} \;\notin\; \mathit{names} \\ \mathit{field\_name}_1 \ldots \mathit{field\_name}_n\; \mathbf{distinct} \end{array}}{E, (\mathit{typeconstr\_name} : \mathit{kind}\{\mathit{field\_name}_1;\, \ldots;\, \mathit{field\_name}_n\}) \vdash \mathbf{ok}} \quad \text{JTEok\_typeconstr\_record}$$

$$\frac{\begin{array}{l} E \vdash t : \mathbf{Type} \\ \mathbf{dom}\,(E) \;\; \rhd \;\; \mathit{names} \\ \mathit{location} \;\notin\; \mathit{names} \end{array}}{E, (\mathit{location} :\, t) \vdash \mathbf{ok}} \quad \text{JTEok\_location}$$

3.7 $\boxed{E \vdash \mathit{typeconstr} : \mathit{kind}}$ Type constructor kinding

Ensures that the type constructor is either defined in the environment or built-in. The result kind indicates how many parameters the type constructor expects.

$$\frac{\begin{array}{l} E \vdash \mathbf{ok} \\ E \vdash \mathit{typeconstr\_name} \ \ \triangleright \ \ \mathit{typeconstr\_name} : \mathit{kind} \end{array}}{E \vdash \mathit{typeconstr\_name} : \mathit{kind}} \quad \textsf{JTtypeconstr\_abstract}$$

$$\frac{\begin{array}{l} E \vdash \mathbf{ok} \\ E \vdash \mathit{typeconstr\_name} \ \ \triangleright \ \ \mathit{type\_params\_opt}\ \mathit{typeconstr\_name} = t \\ \vdash \mathit{type\_params\_opt} : \mathit{kind} \end{array}}{E \vdash \mathit{typeconstr\_name} : \mathit{kind}} \quad \textsf{JTtypeconstr\_concrete}$$

$$\frac{\begin{array}{l} E \vdash \mathbf{ok} \\ E \vdash \mathit{typeconstr\_name} \ \ \triangleright \ \ \mathit{typeconstr\_name} : \mathit{kind}\{\mathit{field\_name}_1; \ \ldots; \mathit{field\_name}_n\} \end{array}}{E \vdash \mathit{typeconstr\_name} : \mathit{kind}} \quad \textsf{JTtypeconstr\_record}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{int} : \mathbf{Type}} \quad \textsf{JTtypeconstr\_int}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{char} : \mathbf{Type}} \quad \textsf{JTtypeconstr\_char}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{string} : \mathbf{Type}} \quad \textsf{JTtypeconstr\_string}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{float} : \mathbf{Type}} \quad \textsf{JTtypeconstr\_float}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{bool} : \mathbf{Type}} \quad \textsf{JTtypeconstr\_bool}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{unit} : \mathbf{Type}} \quad \textsf{JTtypeconstr\_unit}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{exn} : \mathbf{Type}} \quad \textsf{JTtypeconstr\_exn}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{list} : \mathbf{Type}^1 \to \mathbf{Type}} \quad \textsf{JTtypeconstr\_list}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{option} : \mathbf{Type}^1 \to \mathbf{Type}} \quad \textsf{JTtypeconstr\_option}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{ref} : \mathbf{Type}^1 \to \mathbf{Type}} \quad \textsf{JTtypeconstr\_ref}$$

3.8 $\boxed{E \vdash \mathit{typescheme} : \mathit{kind}}$ de Bruijn type scheme well-formedness

Ensures that the type is well-formed in an extended environment.

$$\frac{E, \mathbf{TV} \vdash t : \mathbf{Type}}{E \vdash \forall\, t : \mathbf{Type}} \quad \textsf{JTts\_forall}$$

3.9 $\boxed{E \vdash \forall\, \mathit{type\_params\_opt}, t : \mathit{kind}}$ Named type scheme well-formedness

Ensures that the named type paramaters are distinct, and that the type is well-formed. Instead of extending the environment, this simply substitutes a collection of well-formed types, here **unit**. This works because the type well-formedness judgment below only depends on well-formedness of sub-expressions, and not on the exact form of sub-expressions.

$$\frac{\begin{array}{l} E \vdash \{\!\{\alpha_1 \leftarrow \mathbf{unit}, \ldots, \alpha_n \leftarrow \mathbf{unit}\}\!\}\, t : \mathbf{Type} \\ \alpha_1 \ldots \alpha_n \ \mathbf{distinct} \end{array}}{E \vdash \forall\, (\alpha_1, \ldots, \alpha_n), t : \mathbf{Type}} \quad \textsf{JTtsnamed\_forall}$$

3.10 $\boxed{E \vdash \mathit{typexpr} : \mathit{kind}}$ Type expression well-formedness

Ensures that all of the indices and constructors that appear in a type are bound in the environment.

$$\frac{\begin{array}{c} E \vdash \mathbf{ok} \\ E \vdash \mathit{idx}\ \mathbf{bound} \end{array}}{E \vdash < \mathit{idx}, \mathit{num} >: \mathbf{Type}} \quad \textsf{JTt\_var}$$

$$\frac{\begin{array}{c} E \vdash t : \mathbf{Type} \\ E \vdash t' : \mathbf{Type} \end{array}}{E \vdash t \to t' : \mathbf{Type}} \quad \textsf{JTt\_arrow}$$

$$\frac{\begin{array}{l} E \vdash t_1 : \mathbf{Type} \quad \ldots. \quad E \vdash t_n : \mathbf{Type} \\ \mathbf{length}\,(t_1) \ldots. (t_n) \geq 2 \end{array}}{E \vdash t_1 * \ldots. * t_n : \mathbf{Type}} \quad \textsf{JTt\_tuple}$$

$$\frac{\begin{array}{l}E \vdash \mathit{typeconstr} : \mathbf{Type}^n \to \mathbf{Type}\\ E \vdash t_1 : \mathbf{Type} \quad \ldots \quad E \vdash t_n : \mathbf{Type}\\ \mathbf{length}\,(t_1) \ldots (t_n) = n\end{array}}{E \vdash (t_1, \ldots, t_n)\mathit{typeconstr} : \mathbf{Type}} \quad \textsf{JTt\_constr}$$

3.11 $\boxed{E \vdash \mathit{typexpr} \equiv \mathit{typexpr}'}$ Type equivalence

Checks whether two types are related (potentially indirectly) by the type abbreviations in the environment. The system does not allow recursive types that do not pass through an opaque (generative) type constructor, i.e., a variant or record. Therefore all type expressions have a canonical form obtained by expanding all type abbreviations.

$$\frac{E \vdash t : \mathbf{Type}}{E \vdash t \equiv t} \quad \textsf{JTeq\_refl}$$

$$\frac{E \vdash t' \equiv t}{E \vdash t \equiv t'} \quad \textsf{JTeq\_sym}$$

$$\frac{\begin{array}{l}E \vdash t \equiv t'\\ E \vdash t' \equiv t''\end{array}}{E \vdash t \equiv t''} \quad \textsf{JTeq\_trans}$$

$$\frac{\begin{array}{l}E \vdash \mathbf{ok}\\ E \vdash \mathit{typeconstr\_name} \ \triangleright \ (\alpha_1, \ldots, \alpha_n)\ \mathit{typeconstr\_name} = t\\ E \vdash t_1 : \mathbf{Type} \quad \ldots \quad E \vdash t_n : \mathbf{Type}\end{array}}{E \vdash (t_1, \ldots, t_n)\mathit{typeconstr\_name} \equiv \{\!\{\alpha_1 \leftarrow t_1, \ldots, \alpha_n \leftarrow t_n\}\!\}\ t} \quad \textsf{JTeq\_expand}$$

$$\frac{\begin{array}{l}E \vdash t_1 \equiv t_1'\\ E \vdash t_2 \equiv t_2'\end{array}}{E \vdash t_1 \to t_2 \equiv t_1' \to t_2'} \quad \textsf{JTeq\_arrow}$$

$$\frac{\begin{array}{l}E \vdash t_1 \equiv t_1' \quad \ldots. \quad E \vdash t_n \equiv t_n'\\ \mathbf{length}\,(t_1) \ldots. (t_n) \geq 2\end{array}}{E \vdash t_1 * \ldots. * t_n \equiv t_1' * \ldots. * t_n'} \quad \textsf{JTeq\_tuple}$$

$$\frac{\begin{array}{l}E \vdash \mathit{typeconstr} : \mathbf{Type}^n \to \mathbf{Type}\\ E \vdash t_1 \equiv t_1' \quad \ldots \quad E \vdash t_n \equiv t_n'\\ \mathbf{length}\,(t_1) \ldots (t_n) = n\end{array}}{E \vdash (t_1, \ldots, t_n)\mathit{typeconstr} \equiv (t_1', \ldots, t_n')\mathit{typeconstr}} \quad \textsf{JTeq\_constr}$$

3.12 $\boxed{\{\!\{ typexpr_1, \,..\,, typexpr_n \}\!\} typexpr' \;\triangleright\; typexpr''}$ de Bruin type substitution

Replaces index 0 position n with the nth type in the list, and reduces all other indices by 1.

$$\frac{}{\{\!\{ t_1, \,..\,, t_n \}\!\} \alpha \;\triangleright\; \alpha} \quad \text{JTinxsub\_alpha}$$

$$\frac{\textbf{length}\,(t_1)\,..\,(t_m) = num}{\{\!\{ t_1, \,..\,, t_m, t', t''_1, \,..\,, t''_n \}\!\} < 0, num > \;\triangleright\; t'} \quad \text{JTinxsub\_idx0}$$

$$\frac{\textbf{length}\,(t_1)\,..\,(t_n) \leq num}{\{\!\{ t_1, \,..\,, t_n \}\!\} < 0, num > \;\triangleright\; \textbf{unit}} \quad \text{JTinxsub\_idx1}$$

$$\frac{}{\{\!\{ t_1, \,..\,, t_n \}\!\} < idx + 1, num > \;\triangleright\; < idx, num >} \quad \text{JTinxsub\_idx2}$$

$$\frac{}{\{\!\{ t_1, \,..\,, t_n \}\!\} \_ \;\triangleright\; \_} \quad \text{JTinxsub\_any}$$

$$\frac{\begin{array}{l}\{\!\{ t_1, \,..\,, t_n \}\!\} t'_1 \;\triangleright\; t''_1 \\ \{\!\{ t_1, \,..\,, t_n \}\!\} t'_2 \;\triangleright\; t''_2\end{array}}{\{\!\{ t_1, \,..\,, t_n \}\!\} (t'_1 \to t'_2) \;\triangleright\; t''_1 \to t''_2} \quad \text{JTinxsub\_arrow}$$

$$\frac{\{\!\{ t_1, \,..\,, t_n \}\!\} t'_1 \;\triangleright\; t''_1 \quad .... \quad \{\!\{ t_1, \,..\,, t_n \}\!\} t'_m \;\triangleright\; t''_m}{\{\!\{ t_1, \,..\,, t_n \}\!\} (t'_1 * \,....\, * t'_m) \;\triangleright\; (t''_1 * \,....\, * t''_m)} \quad \text{JTinxsub\_tuple}$$

$$\frac{\{\!\{ t_1, \,..\,, t_n \}\!\} t'_1 \;\triangleright\; t''_1 \quad ... \quad \{\!\{ t_1, \,..\,, t_n \}\!\} t'_m \;\triangleright\; t''_m}{\{\!\{ t_1, \,..\,, t_n \}\!\} (t'_1, \,...\,, t'_m) typeconstr \;\triangleright\; (t''_1, \,...\,, t''_m) typeconstr} \quad \text{JTinxsub\_tc}$$

3.13 $\boxed{E \vdash typexpr \leq typescheme}$ de Bruijn type scheme instantiation

Replaces all of the bound variables of a type scheme.

$$\frac{\begin{array}{l}E \vdash \forall\, t' : \textbf{Type} \\ E \vdash t_1 : \textbf{Type} \quad .. \quad E \vdash t_n : \textbf{Type} \\ \{\!\{ t_1, \,..\,, t_n \}\!\} t' \;\triangleright\; t''\end{array}}{E \vdash t'' \leq \forall\, t'} \quad \text{JTinst\_idx}$$

3.14 $\boxed{E \vdash typexpr \leq \forall\, type\_params\_opt, typexpr'}$ Named type scheme instantiation

Replaces all of the bound variables of a named type scheme.

$$\frac{\begin{array}{l} E \vdash \forall\,(\alpha_1, \ldots, \alpha_n), t : \mathbf{Type} \\ E \vdash t_1 : \mathbf{Type} \quad \ldots \quad E \vdash t_n : \mathbf{Type} \end{array}}{E \vdash \{\!\{\alpha_1 \leftarrow t_1, \ldots, \alpha_n \leftarrow t_n\}\!\}\, t \leq \forall\,(\alpha_1, \ldots, \alpha_n), t} \quad \textsf{JTinst\_named\_named}$$

3.15 $\boxed{E \vdash typexpr \leq typexpr'}$ Wildcard type instantiation

Replaces _ type variables with arbitrary types.

$$\frac{E \vdash < idx, num >: \mathbf{Type}}{E \vdash < idx, num > \leq < idx, num >} \quad \textsf{JTinst\_any\_tyvar}$$

$$\frac{E \vdash t : \mathbf{Type}}{E \vdash t \leq \_} \quad \textsf{JTinst\_any\_any}$$

$$\frac{\begin{array}{l} E \vdash t_1 \leq t_1' \\ E \vdash t_2 \leq t_2' \end{array}}{E \vdash t_1 \to t_2 \leq t_1' \to t_2'} \quad \textsf{JTinst\_any\_arrow}$$

$$\frac{\begin{array}{l} E \vdash t_1 \leq t_1' \quad \ldots. \quad E \vdash t_n \leq t_n' \\ \mathbf{length}\,(t_1) \ldots. (t_n) \geq 2 \end{array}}{E \vdash t_1 * \ldots. * t_n \leq t_1' * \ldots. * t_n'} \quad \textsf{JTinst\_any\_tuple}$$

$$\frac{\begin{array}{l} E \vdash t_1 \leq t_1' \quad \ldots \quad E \vdash t_n \leq t_n' \\ E \vdash typeconstr : \mathbf{Type}^n \to \mathbf{Type} \\ \mathbf{length}\,(t_1) \ldots (t_n) = n \end{array}}{E \vdash (t_1, \ldots, t_n) typeconstr \leq (t_1', \ldots, t_n') typeconstr} \quad \textsf{JTinst\_any\_ctor}$$

3.16 $\boxed{E \vdash value\_name : typexpr}$ Variable typing

Determines if a variable can have a specified type.

$$\frac{\begin{array}{l} E \vdash value\_name \ \ \triangleright\ \ value\_name : ts \\ E \vdash t \leq ts \end{array}}{E \vdash value\_name : t} \quad \textsf{JTvalue\_name\_value\_name}$$

3.17 $\boxed{E \vdash \mathit{field\_name} : \mathit{typexpr} \to \mathit{typexpr}'}$ Field name typing

Determines the type constructor associated with a given field name. Since field names are used to destructure record data, the type is a function type from a record to the type of the corresponding position.

$$\frac{\begin{array}{l} E \vdash \mathit{field\_name} \;\; \triangleright \;\; \mathit{field\_name} : \forall\,(\alpha_1, \ldots, \alpha_m), \mathit{typeconstr\_name} \to t \\ E \vdash (t'_1, \ldots, t'_m)\mathit{typeconstr\_name} \to t'' \leq \forall\,(\alpha_1, \ldots, \alpha_m), (\alpha_1, \ldots, \alpha_m)\mathit{typeconstr\_name} \to t \end{array}}{E \vdash \mathit{field\_name} : (t'_1, \ldots, t'_m)\mathit{typeconstr\_name} \to t''} \quad \textsf{JTfield\_name}$$

3.18 $\boxed{E \vdash \mathit{constr} : \mathit{typexpr}_1 \ldots \mathit{typexpr}_n \to \mathit{typexpr}'}$ Non-constant constructor typing

Determines the type constructor associated with a given value constructor. Non-constant constructors are attributed types for each argument as well as a return type.

$$\frac{\begin{array}{l} E \vdash \mathit{constr\_name} \;\; \triangleright \;\; \mathit{constr\_name}\,\mathbf{of}\,\forall\,(\alpha_1, \ldots, \alpha_m), (t_1, \ldots., t_n) : \mathit{typeconstr} \\ E \vdash (t'_1 * \ldots. * t'_n) \to (t''_1, \ldots, t''_m)\mathit{typeconstr} \leq \forall\,(\alpha_1, \ldots, \alpha_m), (t_1 * \ldots. * t_n) \to (\alpha_1, \ldots, \alpha_m)\mathit{typeconstr} \end{array}}{E \vdash \mathit{constr\_name} : t'_1 \ldots. t'_n \to (t''_1, \ldots, t''_m)\mathit{typeconstr}} \quad \textsf{JTconstr\_p\_name}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{Invalid\_argument} : \mathbf{string} \to \mathbf{exn}} \quad \textsf{JTconstr\_p\_invarg}$$

$$\frac{E \vdash t : \mathbf{Type}}{E \vdash \mathbf{Some} : t \to (t\,\mathbf{option})} \quad \textsf{JTconstr\_p\_some}$$

3.19 $\boxed{E \vdash \mathit{constr} : \mathit{typexpr}}$ Constant constructor typing

Constant constructors are typed like non-constant constructors without arguments.

$$\frac{\begin{array}{l} E \vdash \mathbf{ok} \\ E \vdash \mathit{constr\_name} \;\; \triangleright \;\; \mathit{constr\_name}\,\mathbf{of}\,\mathit{typeconstr\_name} \\ E \vdash \mathit{typeconstr\_name} \;\; \triangleright \;\; \mathit{typeconstr\_name} : \mathbf{Type}^n \to \mathbf{Type} \\ E \vdash t_1 : \mathbf{Type} \quad \ldots \quad E \vdash t_n : \mathbf{Type} \\ \mathbf{length}\,(t_1) \ldots (t_n) = n \end{array}}{E \vdash \mathit{constr\_name} : (t_1, \ldots, t_n)\mathit{typeconstr\_name}} \quad \textsf{JTconstr\_c\_constr}$$

$$\frac{\begin{array}{l} E \vdash \mathbf{ok} \\ E \vdash \mathit{constr\_name} \quad \triangleright \quad \mathit{constr\_name}\,\mathbf{of\ exn} \end{array}}{E \vdash \mathit{constr\_name} : \mathbf{exn}} \quad \mathsf{JTconstr\_c\_exn\_constr}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{Not\_found} : \mathbf{exn}} \quad \mathsf{JTconstr\_c\_notfound}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{Assert\_failure} : \mathbf{exn}} \quad \mathsf{JTconstr\_c\_assert\_fail}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{Match\_failure} : \mathbf{exn}} \quad \mathsf{JTconstr\_c\_match\_fail}$$

Dropping the source location arguments for **Assert_failure** and **Match_failure**.

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{Division\_by\_zero} : \mathbf{exn}} \quad \mathsf{JTconstr\_c\_div\_by\_0}$$

$$\frac{E \vdash t : \mathbf{Type}}{E \vdash \mathbf{None} : t\,\mathbf{option}} \quad \mathsf{JTconstr\_c\_none}$$

3.20 $\boxed{E \vdash \mathit{constant} : \mathit{typexpr}}$ Constant typing

Determines the type of a constant.

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathit{integer\_literal} : \mathbf{int}} \quad \mathsf{JTconst\_int}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathit{float\_literal} : \mathbf{float}} \quad \mathsf{JTconst\_float}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathit{char\_literal} : \mathbf{char}} \quad \mathsf{JTconst\_char}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathit{string\_literal} : \mathbf{string}} \quad \mathsf{JTconst\_string}$$

$$\frac{E \vdash \mathit{constr} : t}{E \vdash \mathit{constr} : t} \quad \mathsf{JTconst\_constr}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{false} : \mathbf{bool}} \quad \mathsf{JTconst\_false}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{true} : \mathbf{bool}} \quad \textsf{JTconst\_true}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash () : \mathbf{unit}} \quad \textsf{JTconst\_unit}$$

$$\frac{E \vdash t : \mathbf{Type}}{E \vdash [\,] : t\,\mathbf{list}} \quad \textsf{JTconst\_nil}$$

3.21 $\boxed{\sigma^T \& E \vdash \mathit{pattern} : \mathit{typexpr} \quad \triangleright \quad E'}$ Pattern typing and binding collection

Determines if a pattern matches a value of a certain type, and calculates the types of the variables it binds. A pattern must bind any given variable at most once, except that the two alternatives of an or-pattern must bind the same set of variables. σ^T gives the types that should replace type variables in explicitly type-annotated patterns.

$$\frac{E \vdash t : \mathbf{Type}}{\sigma^T \& E \vdash x : t \quad \triangleright \quad x : t} \quad \textsf{JTpat\_var}$$

$$\frac{E \vdash t : \mathbf{Type}}{\sigma^T \& E \vdash \_ : t \quad \triangleright \quad \mathbf{empty}} \quad \textsf{JTpat\_any}$$

$$\frac{E \vdash \mathit{constant} : t}{\sigma^T \& E \vdash \mathit{constant} : t \quad \triangleright \quad \mathbf{empty}} \quad \textsf{JTpat\_constant}$$

$$\frac{\begin{array}{l} \sigma^T \& E \vdash \mathit{pattern} : t \quad \triangleright \quad E' \\ \mathbf{dom}\,(E', x : t) \quad \triangleright \quad \mathit{name}_1 \mathinner{..} \mathit{name}_n \\ \mathit{name}_1 \mathinner{..} \mathit{name}_n\,\mathbf{distinct} \end{array}}{\sigma^T \& E \vdash \mathit{pattern}\,\mathbf{as}\,x : t \quad \triangleright \quad E', x : t} \quad \textsf{JTpat\_alias}$$

$$\frac{\begin{array}{l} \sigma^T \& E \vdash \mathit{pattern} : t \quad \triangleright \quad E' \\ E \vdash t' \leq \sigma^T\,\mathit{src\_t} \\ E \vdash t \equiv t' \end{array}}{\sigma^T \& E \vdash (\mathit{pattern} : \mathit{src\_t}) : t \quad \triangleright \quad E'} \quad \textsf{JTpat\_typed}$$

$$\frac{\begin{array}{l} \sigma^T \& E \vdash \mathit{pattern} : t \quad \triangleright \quad E' \\ \sigma^T \& E \vdash \mathit{pattern'} : t \quad \triangleright \quad E'' \\ E'\,\mathbf{PERMUTES}\,E'' \end{array}}{\sigma^T \& E \vdash \mathit{pattern} | \mathit{pattern'} : t \quad \triangleright \quad E'} \quad \textsf{JTpat\_or}$$

$$\frac{\begin{array}{l} E \vdash constr : t_1 \ldots t_n \to t \\ \sigma^T \& E \vdash pattern_1 : t_1 \quad \rhd \quad E_1 \quad \ldots \quad \sigma^T \& E \vdash pattern_n : t_n \quad \rhd \quad E_n \\ \mathbf{dom}\,(E_1 @ \ldots @ E_n) \quad \rhd \quad name_1 \mathinner{..} name_m \\ name_1 \mathinner{..} name_m\ \mathbf{distinct} \end{array}}{\sigma^T \& E \vdash constr(pattern_1,\ \ldots, pattern_n) : t \quad \rhd \quad E_1 @ \ldots @ E_n} \quad \text{JTpat\_construct}$$

$$\frac{E \vdash constr : t_1 \ldots t_n \to t}{\sigma^T \& E \vdash constr\ \_ : t \quad \rhd \quad \mathbf{empty}} \quad \text{JTpat\_construct\_any}$$

$$\frac{\begin{array}{l} \sigma^T \& E \vdash pattern_1 : t_1 \quad \rhd \quad E_1 \quad \ldots. \quad \sigma^T \& E \vdash pattern_n : t_n \quad \rhd \quad E_n \\ \mathbf{length}\,(pattern_1) \ldots. (pattern_n) \geq 2 \\ \mathbf{dom}\,(E_1 @ \ldots. @ E_n) \quad \rhd \quad name_1 \mathinner{..} name_m \\ name_1 \mathinner{..} name_m\ \mathbf{distinct} \end{array}}{\sigma^T \& E \vdash pattern_1,\ \ldots., pattern_n : t_1 * \ \ldots. * t_n \quad \rhd \quad E_1 @ \ldots. @ E_n} \quad \text{JTpat\_tuple}$$

$$\frac{\begin{array}{l} \sigma^T \& E \vdash pattern_1 : t_1 \quad \rhd \quad E_1 \quad \ldots \quad \sigma^T \& E \vdash pattern_n : t_n \quad \rhd \quad E_n \\ E \vdash field\_name_1 : t \to t_1 \quad \ldots \quad E \vdash field\_name_n : t \to t_n \\ \mathbf{length}\,(pattern_1) \ldots (pattern_n) \geq 1 \\ \mathbf{dom}\,(E_1 @ \ldots @ E_n) \quad \rhd \quad name_1 \mathinner{..} name_m \\ name_1 \mathinner{..} name_m\ \mathbf{distinct} \end{array}}{\sigma^T \& E \vdash \{field\_name_1 = pattern_1; \ldots ; field\_name_n = pattern_n\} : t \quad \rhd \quad E_1 @ \ldots @ E_n} \quad \text{JTpat\_record}$$

$$\frac{\begin{array}{l} \sigma^T \& E \vdash pattern : t \quad \rhd \quad E' \\ \sigma^T \& E \vdash pattern' : t\,\mathbf{list} \quad \rhd \quad E'' \\ \mathbf{dom}\,(E') \quad \rhd \quad name_1 \mathinner{..} name_m \\ \mathbf{dom}\,(E'') \quad \rhd \quad name'_1 \mathinner{..} name'_n \\ name_1 \mathinner{..} name_m\ name'_1 \mathinner{..} name'_n\ \mathbf{distinct} \end{array}}{\sigma^T \& E \vdash pattern :: pattern' : t\,\mathbf{list} \quad \rhd \quad E' @ E''} \quad \text{JTpat\_cons}$$

3.22 $\boxed{E \vdash unary\_prim : typexpr}$ Unary primitive typing

Determines if a unary primitive has a given type.

$$\frac{E \vdash t : \mathbf{Type}}{E \vdash \mathbf{raise} : \mathbf{exn} \to t} \quad \text{JTuprim\_raise}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \mathbf{not} : \mathbf{bool} \to \mathbf{bool}} \quad \text{JTuprim\_not}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash \sim\!-: \mathbf{int} \rightarrow \mathbf{int}} \quad \textsf{JTuprim\_uminus}$$

$$\frac{E \vdash t : \mathbf{Type}}{E \vdash \mathbf{ref} : t \rightarrow (t\,\mathbf{ref})} \quad \textsf{JTuprim\_ref}$$

$$\frac{E \vdash t : \mathbf{Type}}{E \vdash ! : (t\,\mathbf{ref}) \rightarrow t} \quad \textsf{JTuprim\_deref}$$

3.23 $\boxed{E \vdash \mathit{binary\_prim} : \mathit{typexpr}}$ Binary primitive typing

Determines if a binary primitive has a given type.

$$\frac{E \vdash t : \mathbf{Type}}{E \vdash =: t \rightarrow (t \rightarrow \mathbf{bool})} \quad \textsf{JTbprim\_equal}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash + : \mathbf{int} \rightarrow (\mathbf{int} \rightarrow \mathbf{int})} \quad \textsf{JTbprim\_plus}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash - : \mathbf{int} \rightarrow (\mathbf{int} \rightarrow \mathbf{int})} \quad \textsf{JTbprim\_minus}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash * : \mathbf{int} \rightarrow (\mathbf{int} \rightarrow \mathbf{int})} \quad \textsf{JTbprim\_times}$$

$$\frac{E \vdash \mathbf{ok}}{E \vdash / : \mathbf{int} \rightarrow (\mathbf{int} \rightarrow \mathbf{int})} \quad \textsf{JTbprim\_div}$$

$$\frac{E \vdash t : \mathbf{Type}}{E \vdash :=: t\,\mathbf{ref} \rightarrow (t \rightarrow \mathbf{unit})} \quad \textsf{JTbprim\_assign}$$

3.24 $\boxed{\sigma^T \& E \vdash \mathit{expr} : \mathit{typexpr}}$ Expression typing

Detremines if an expression has a given type. Note that t is a type, not a type scheme, but it may contain type variables (which are recorded in E). σ^T gives the types that should replace type variables in explicitly type-annotated patterns.

While the choice of a rule is mostly syntax-directed (for any given constructor, a single rule applies, except for **let** and **assert**), polymorphism is handled in a purely declarative manner. The choice of instantiation for a polymorphic bound variable or primitive is free, as is the number of variables introduced by a polymorphic **let**.

$$\frac{\begin{array}{l} E \vdash \mathit{unary\_prim} : t \\ E \vdash t \equiv t' \end{array}}{\sigma^T \& E \vdash (\%\mathbf{prim}\,\mathit{unary\_prim}) : t'} \quad \textsf{JTe\_uprim}$$

$$\frac{\begin{array}{l} E \vdash \mathit{binary\_prim} : t \\ E \vdash t \equiv t' \end{array}}{\sigma^T \& E \vdash (\%\mathbf{prim}\,\mathit{binary\_prim}) : t'} \quad \textsf{JTe\_bprim}$$

$$\frac{\begin{array}{l} E \vdash \mathit{value\_name} : t \\ E \vdash t \equiv t' \end{array}}{\sigma^T \& E \vdash \mathit{value\_name} : t'} \quad \textsf{JTe\_ident}$$

$$\frac{\begin{array}{l} E \vdash \mathit{constant} : t \\ E \vdash t \equiv t' \end{array}}{\sigma^T \& E \vdash \mathit{constant} : t'} \quad \textsf{JTe\_constant}$$

$$\frac{\begin{array}{l} \sigma^T \& E \vdash e : t \\ E \vdash t' \leq \sigma^T\ \mathit{src\_t} \\ E \vdash t \equiv t' \end{array}}{\sigma^T \& E \vdash (e : \mathit{src\_t}) : t} \quad \textsf{JTe\_typed}$$

$$\frac{\begin{array}{l} \sigma^T \& E \vdash e_1 : t_1 \quad .... \quad \sigma^T \& E \vdash e_n : t_n \\ \mathbf{length}\,(e_1) \ldots. (e_n) \geq 2 \\ E \vdash t_1 * \ldots. * t_n \equiv t' \end{array}}{\sigma^T \& E \vdash e_1 ,\ \ldots. ,\ e_n : t'} \quad \textsf{JTe\_tuple}$$

$$\frac{\begin{array}{l} E \vdash \mathit{constr} : t_1 \ldots t_n \to t \\ \sigma^T \& E \vdash e_1 : t_1 \quad \ldots \quad \sigma^T \& E \vdash e_n : t_n \\ E \vdash t \equiv t' \end{array}}{\sigma^T \& E \vdash \mathit{constr}(e_1 ,\ \ldots ,\ e_n) : t'} \quad \textsf{JTe\_construct}$$

$$\frac{\begin{array}{l} \sigma^T \& E \vdash e_1 : t \\ \sigma^T \& E \vdash e_2 : t\,\mathbf{list} \\ E \vdash t\,\mathbf{list} \equiv t' \end{array}}{\sigma^T \& E \vdash e_1 :: e_2 : t'} \quad \textsf{JTe\_cons}$$

$$\frac{\begin{array}{l}\sigma^T \& E \vdash e_1 : t_1 \quad \ldots \quad \sigma^T \& E \vdash e_n : t_n \\ E \vdash \mathit{field\_name}_1 : t \rightarrow t_1 \quad \ldots \quad E \vdash \mathit{field\_name}_n : t \rightarrow t_n \\ t = (t'_1, \ldots, t'_l)\mathit{typeconstr\_name} \\ E \vdash \mathit{typeconstr\_name} \;\; \triangleright \;\; \mathit{typeconstr\_name} : \mathit{kind}\{\mathit{field\_name}'_1; \ldots; \mathit{field\_name}'_m\} \\ \mathit{field\_name}_1 \ldots \mathit{field\_name}_n \,\mathbf{PERMUTES}\, \mathit{field\_name}'_1 \ldots \mathit{field\_name}'_m \\ \mathbf{length}\,(e_1) \ldots (e_n) \geq 1 \\ E \vdash t \equiv t' \end{array}}{\sigma^T \& E \vdash \{\mathit{field\_name}_1 = e_1; \ldots; \mathit{field\_name}_n = e_n\} : t'} \quad \textsf{JTe\_record\_constr}$$

$$\frac{\begin{array}{l}\sigma^T \& E \vdash \mathit{expr} : t \\ E \vdash \mathit{field\_name}_1 : t \rightarrow t_1 \quad \ldots \quad E \vdash \mathit{field\_name}_n : t \rightarrow t_n \\ \sigma^T \& E \vdash e_1 : t_1 \quad \ldots \quad \sigma^T \& E \vdash e_n : t_n \\ \mathit{field\_name}_1 \ldots \mathit{field\_name}_n \,\mathbf{distinct} \\ \mathbf{length}\,(e_1) \ldots (e_n) \geq 1 \\ E \vdash t \equiv t' \end{array}}{\sigma^T \& E \vdash \{\mathit{expr}\,\mathbf{with}\,\mathit{field\_name}_1 = e_1; \ldots; \mathit{field\_name}_n = e_n\} : t'} \quad \textsf{JTe\_record\_with}$$

$$\frac{\begin{array}{l}\sigma^T \& E \vdash e : t_1 \rightarrow t \\ \sigma^T \& E \vdash e_1 : t_1 \end{array}}{\sigma^T \& E \vdash e\ e_1 : t} \quad \textsf{JTe\_apply}$$

$$\frac{\begin{array}{l}\sigma^T \& E \vdash e : t \\ E \vdash \mathit{field\_name} : t \rightarrow t' \\ E \vdash t' \equiv t'' \end{array}}{\sigma^T \& E \vdash e.\mathit{field\_name} : t''} \quad \textsf{JTe\_record\_proj}$$

$$\frac{\begin{array}{l}\sigma^T \& E \vdash e_1 : \mathbf{bool} \\ \sigma^T \& E \vdash e_2 : \mathbf{bool} \\ E \vdash \mathbf{bool} \equiv t \end{array}}{\sigma^T \& E \vdash e_1 \&\& e_2 : t} \quad \textsf{JTe\_and}$$

$$\frac{\begin{array}{l}\sigma^T \& E \vdash e_1 : \mathbf{bool} \\ \sigma^T \& E \vdash e_2 : \mathbf{bool} \\ E \vdash \mathbf{bool} \equiv t \end{array}}{\sigma^T \& E \vdash e_1 || e_2 : t} \quad \textsf{JTe\_or}$$

$$\frac{\begin{array}{l}\sigma^T \& E \vdash e_1 : \mathbf{bool} \\ \sigma^T \& E \vdash e_2 : t \\ \sigma^T \& E \vdash e_3 : t \end{array}}{\sigma^T \& E \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : t} \quad \textsf{JTe\_ifthenelse}$$

$$\frac{\sigma^T \& E \vdash e_1 : \mathbf{bool} \quad \sigma^T \& E \vdash e_2 : \mathbf{unit} \quad E \vdash \mathbf{unit} \equiv t}{\sigma^T \& E \vdash \mathbf{while}\, e_1\, \mathbf{do}\, e_2\, \mathbf{done} : t} \quad \mathsf{JTe\_while}$$

$$\frac{\sigma^T \& E \vdash e_1 : \mathbf{int} \quad \sigma^T \& E \vdash e_2 : \mathbf{int} \quad \sigma^T \& E, \mathit{lowercase\_ident} : \mathbf{int} \vdash e_3 : \mathbf{unit} \quad E \vdash \mathbf{unit} \equiv t}{\sigma^T \& E \vdash \mathbf{for}\, \mathit{lowercase\_ident} = e_1\, [\mathbf{down}]\mathbf{to}\, e_2\, \mathbf{do}\, e_3\, \mathbf{done} : t} \quad \mathsf{JTe\_for}$$

$$\frac{\sigma^T \& E \vdash e_1 : \mathbf{unit} \quad \sigma^T \& E \vdash e_2 : t}{\sigma^T \& E \vdash e_1 ; e_2 : t} \quad \mathsf{JTe\_sequence}$$

In the above rule, e_1 must have type **unit**. Ocaml lets the programmer off with a warning, unless `-warn-error S` is passed on the compiler command line.

$$\frac{\sigma^T \& E \vdash e : t \quad \sigma^T \& E \vdash \mathit{pattern\_matching} : t \to t'}{\sigma^T \& E \vdash \mathbf{match}\, e\, \mathbf{with}\, \mathit{pattern\_matching} : t'} \quad \mathsf{JTe\_match}$$

$$\frac{\sigma^T \& E \vdash \mathit{pattern\_matching} : t \to t' \quad E \vdash t \to t' \equiv t''}{\sigma^T \& E \vdash \mathbf{function}\, \mathit{pattern\_matching} : t''} \quad \mathsf{JTe\_function}$$

$$\frac{\sigma^T \& E \vdash e : t \quad \sigma^T \& E \vdash \mathit{pattern\_matching} : \mathbf{exn} \to t}{\sigma^T \& E \vdash \mathbf{try}\, e\, \mathbf{with}\, \mathit{pattern\_matching} : t} \quad \mathsf{JTe\_try}$$

We give three rules for **let** expressions. The rule JTe˙let˙mono describes "monomorphic let": it does not allow the type of *expr* to be generalised. The rule JTe˙let˙poly describes "polymorphic let": it allows any number of type variables in the type of *nexp* to be generalised (more precisely, this generalisation applies simultaneously to the types of all the variables bound by *pat*), at the cost of requiring *nexp* to be non-expansive (which is described syntactically through the grammar for *nexp*). The rule JTe˙letrec allows mutually recursive functions to be defined; since immediate functions are values, thus nonexpansive, there is no need for a monomorphic **let rec** rule.

$$\frac{\sigma^T \& E \vdash \mathit{pat} = \mathit{expr} \;\; \triangleright \;\; x_1 : t_1, .., x_n : t_n \quad \sigma^T \& E @ x_1 : t_1, .., x_n : t_n \vdash e : t}{\sigma^T \& E \vdash \mathbf{let}\, \mathit{pat} = \mathit{expr}\, \mathbf{in}\, e : t} \quad \mathsf{JTe\_let\_mono}$$

$$\frac{\mathbf{shift}\, 0\, 1\, \sigma^T \& E, \mathbf{TV} \vdash \mathit{pat} = \mathit{nexp} \;\; \triangleright \;\; x_1 : t_1, .., x_n : t_n \quad \sigma^T \& E @ x_1 : \forall\, t_1, .., x_n : \forall\, t_n \vdash e : t}{\sigma^T \& E \vdash \mathbf{let}\, \mathit{pat} = \mathit{nexp}\, \mathbf{in}\, e : t} \quad \mathsf{JTe\_let\_poly}$$

$$\frac{\begin{array}{l}\textbf{shift}\,0\,1\,\sigma^T\&E,\mathbf{TV}\vdash \mathit{letrec\_bindings}\quad\rhd\quad x_1:t_1,\,\ldots,x_n:t_n\\ \sigma^T\&E@(x_1:\forall\,t_1),\,\ldots,(x_n:\forall\,t_n)\vdash e:t\end{array}}{\sigma^T\&E\vdash \textbf{let rec}\,\mathit{letrec\_bindings}\,\textbf{in}\,e:t}\quad \textsf{JTe\_letrec}$$

$$\frac{\begin{array}{l}\sigma^T\&E\vdash e:\textbf{bool}\\ E\vdash \textbf{unit}\equiv t\end{array}}{\sigma^T\&E\vdash \textbf{assert}\,e:t}\quad \textsf{JTe\_assert}$$

$$\frac{E\vdash t:\textbf{Type}}{\sigma^T\&E\vdash \textbf{assert false}:t}\quad \textsf{JTe\_assertfalse}$$

$$\frac{\begin{array}{l}E\vdash\ \textbf{ok}\\ E\vdash \mathit{location}\quad\rhd\quad \mathit{location}:t\\ E\vdash t\,\textbf{ref}\equiv t'\end{array}}{\sigma^T\&E\vdash \mathit{location}:t'}\quad \textsf{JTe\_location}$$

3.25 $\boxed{\sigma^T\&E\vdash \mathit{pattern\_matching}:\mathit{typexpr}\rightarrow \mathit{typexpr}'}$ Pattern matching/expression pair typing

Determines the function type of a sequence of pattern/expression pairs. The function type desribes the type of the value matched by all of the patterns and the type of the value returned by all of the expressions. σ^T gives the types that should replace type variables in explicitly type-annotated patterns.

$$\frac{\begin{array}{l}\sigma^T\&E\vdash \mathit{pattern}_1:t\quad\rhd\quad E_1\quad\ldots\quad \sigma^T\&E\vdash \mathit{pattern}_n:t\quad\rhd\quad E_n\\ \sigma^T\&E@E_1\vdash e_1:t'\quad\ldots\quad \sigma^T\&E@E_n\vdash e_n:t'\\ \textbf{length}\,(\mathit{pattern}_1)\ldots(\mathit{pattern}_n)\geq 1\end{array}}{\sigma^T\&E\vdash \mathit{pattern}_1\rightarrow e_1|\ldots|\mathit{pattern}_n\rightarrow e_n:t\rightarrow t'}\quad \textsf{JTpat\_matching\_pm}$$

3.26 $\boxed{\sigma^T\&E\vdash \mathit{let\_binding}\quad\rhd\quad E'}$ Let binding typing

Determines the types bound by a let bindings pattern.

$$\frac{\begin{array}{l}\sigma^T\&E\vdash \mathit{pattern}:t\quad\rhd\quad x_1:t_1,\,..,x_n:t_n\\ \sigma^T\&E\vdash \mathit{expr}:t\end{array}}{\sigma^T\&E\vdash \mathit{pattern}=\mathit{expr}\quad\rhd\quad (x_1:t_1),\,..,(x_n:t_n)}\quad \textsf{JTlet\_binding\_poly}$$

3.27 $\boxed{\sigma^T \& E \vdash \mathit{letrec\_bindings} \ \triangleright \ E'}$ Recursive let binding typing

Determines the types bound by a recursive let's patterns (which are always just variables).

$$\frac{\begin{array}{l} E' = E@\mathit{value\_name}_1 : t_1 \to t'_1, \ldots, \mathit{value\_name}_n : t_n \to t'_n \\ \sigma^T \& E' \vdash \mathit{pattern\_matching}_1 : t_1 \to t'_1 \quad \ldots \quad \sigma^T \& E' \vdash \mathit{pattern\_matching}_n : t_n \to t'_n \\ \mathit{value\_name}_1 \ldots \mathit{value\_name}_n \ \mathbf{distinct} \end{array}}{\begin{array}{r} \sigma^T \& E \vdash \mathit{value\_name}_1 = \mathbf{function}\ \mathit{pattern\_matching}_1 \ \mathbf{and} \ldots \mathbf{and}\ \mathit{value\_name}_n = \mathbf{function}\ \mathit{pattern\_matching}_n \ \triangleright \\ \mathit{value\_name}_1 : t_1 \to t'_1, \ldots, \mathit{value\_name}_n : t_n \to t'_n \end{array}} \ \text{JTletrec\_binding\_equal\_function}$$

3.28 $\boxed{\mathit{type\_params\_opt}\ \mathit{typeconstr} \vdash \mathit{constr\_decl} \ \triangleright \ EB}$ Variant constructor declaration

Collects the constructors of a variant type declaration using named type schemes for the type parameters.

$$\frac{}{(\alpha_1, \ldots, \alpha_n)\ \mathit{typeconstr} \vdash \mathit{constr\_name} \ \triangleright \ \mathit{constr\_name}\ \mathbf{of}\ \mathit{typeconstr}} \ \text{JTconstr\_decl\_nullary}$$

$$\frac{}{(\alpha_1, \ldots, \alpha_n)\ \mathit{typeconstr} \vdash \mathit{constr\_name}\ \mathbf{of}\ t_1 * \ldots * t_n \ \triangleright \ \mathit{constr\_name}\ \mathbf{of}\ \forall\, (\alpha_1, \ldots, \alpha_n), (t_1, \ldots, t_n) : \mathit{typeconstr}} \ \text{JTconstr\_decl\_nary}$$

3.29 $\boxed{\mathit{type\_params\_opt}\ \mathit{typeconstr\_name} \vdash \mathit{field\_decl} \ \triangleright \ EB}$ Record field declaration

Collects the fields of a record type using named type schemes for the type parameters.

$$\frac{}{(\alpha_1, \ldots, \alpha_n)\ \mathit{typeconstr\_name} \vdash \mathit{fn} : t \ \triangleright \ \mathit{fn} : \forall\, (\alpha_1, \ldots, \alpha_n), \mathit{typeconstr\_name} \to t} \ \text{JTfield\_decl\_only}$$

3.30 $\boxed{\vdash \mathit{typedef}_1 \ \mathbf{and} \ldots \mathbf{and}\ \mathit{typedef}_n \ \triangleright \ E' \ \mathbf{and}\ E'' \ \mathbf{and}\ E'''}$ Type definitions collection

A type definition declares several sorts of names: type constructors (some of them corresponding to freshly generated types, others to type abbreviations), and data constructors and destructors. These names are collected into three environments:

- E' contains generative type definitions (variant and record types);

- E'' contains type abbreviations;
- E''' contains constructors and destructors for generative datatypes.

The order E', E'', E''' is chosen so that their concatenation is well-formed, because no component may refer to a subsequent one. The first component E', only contains declarations of names which do not depend on anything. The second component E'' contains type abbreviations topologically sorted according to their dependency order, which is possible since we do not allow recursive type abbreviations (in Objective Caml, without the `-rectypes` compiler option, recursive type abbreviations are only allowed when guarded polymorphic variants and object types) — recursive types must be guarded by a generative datatype. Finally E''' declares constructors and destructors for the types declared in E'; E''' may refer to types declared in E' or E'' in the types of the arguments to these constructors and destructors.

This judgement form does not directly assert the correctness of the definitions: this is performed by the rule JTtype˙definition˙list below, which states that the environment assembled here must be well-formed.

$$\frac{}{\vdash\ \triangleright\ \ \mathbf{empty\ and\ empty\ and\ empty}}\ \ \textsf{JTtypedef\_empty}$$

$$\frac{\vdash\ \overline{\mathit{typedef}_i}^{\,i}\ \ \triangleright\ \ E\ \mathbf{and}\ E'\ \mathbf{and}\ E''}{\vdash\ \mathit{type\_params\_opt}\ \mathit{typeconstr\_name}\ = t\ \mathbf{and}\ \overline{\mathit{typedef}_i}^{\,i}\ \ \triangleright\ \ E\ \mathbf{and}\ E', (\mathit{type\_params\_opt}\ \mathit{typeconstr\_name} = t)\ \mathbf{and}\ E''}\ \ \textsf{JTtypedef\_eq}$$

$$\frac{\begin{array}{l}\vdash\ \overline{\mathit{typedef}_i}^{\,i}\ \ \triangleright\ \ E\ \mathbf{and}\ E'\ \mathbf{and}\ E''\\ \vdash\ \mathit{type\_params\_opt} : \mathit{kind}\\ \mathit{type\_params\_opt}\ \mathit{typeconstr\_name} \vdash \mathit{constr\_decl}_1\ \ \triangleright\ \ EB_1\ \ \ldots\ \ \mathit{type\_params\_opt}\ \mathit{typeconstr\_name} \vdash \mathit{constr\_decl}_n\ \ \triangleright\ \ EB_n\end{array}}{\begin{array}{c}\vdash\ \mathit{type\_params\_opt}\ \mathit{typeconstr\_name}\ = \mathit{constr\_decl}_1 \,|\, \ldots \,|\, \mathit{constr\_decl}_n\ \mathbf{and}\ \overline{\mathit{typedef}_i}^{\,i}\ \ \triangleright\\ E, (\mathit{typeconstr\_name} : \mathit{kind})\ \mathbf{and}\ E'\ \mathbf{and}\ E''@EB_1, \ldots, EB_n\end{array}}\ \ \textsf{JTtypedef\_def\_sum}$$

A variant type definition yields two sorts of bindings: one for the type constructor name and one for each constructor.

$$\frac{\begin{array}{l}\vdash\ \overline{\mathit{typedef}_i}^{\,i}\ \ \triangleright\ \ E\ \mathbf{and}\ E'\ \mathbf{and}\ E''\\ \vdash\ \mathit{type\_params\_opt} : \mathit{kind}\\ \mathit{type\_params\_opt}\ \mathit{typeconstr\_name} \vdash \mathit{field\_name}_1 : t_1\ \ \triangleright\ \ EB_1\ \ \ldots\ \ \mathit{type\_params\_opt}\ \mathit{typeconstr\_name} \vdash \mathit{field\_name}_n : t_n\ \ \triangleright\ \ EB_n\end{array}}{\begin{array}{c}\vdash\ \mathit{type\_params\_opt}\ \mathit{typeconstr\_name}\ = \{\mathit{field\_name}_1 : t_1; \ldots ; \mathit{field\_name}_n : t_n\}\ \mathbf{and}\ \overline{\mathit{typedef}_i}^{\,i}\ \ \triangleright\\ E, (\mathit{typeconstr\_name} : \mathit{kind}\{\mathit{field\_name}_1; \ldots ; \mathit{field\_name}_n\})\ \mathbf{and}\ E'\ \mathbf{and}\ E''@EB_1, \ldots, EB_n\end{array}}\ \ \textsf{JTtypedef\_def\_record}$$

A record type definition yields two sorts of bindings: one for the type constructor name and one for each field. The field names are also recorded with the type constructor binding; this information is used in the rule JTe˙record˙constr to make sure that record expressions specify all fields. (We would similarly tag type constructor bindings for variant types with their constructor names if we wanted to check the exhaustivity of pattern matching.)

3.31 $\boxed{E \vdash \mathit{type\_definition} \ \ \triangleright \ \ E'}$ Type definition well-formedness and binding collection

Collects the bindings of a type definition and ensures that they are well-formed. Any given name may be defined at most once, and all names used must have been bound previously or earlier in the same type definition phrase. The conditions are checked by the premise $E@E'''' \vdash \mathbf{ok}$ in the rule JTtype˙definition˙list and the assembly is performed by the type definitions collection rules above. This implies that the type abbreviations must be topologically sorted in their dependency order. (Generative type definitions are exempt from such constraints.) Programmers do not have to abide by this constraint: they may order type abbreviations in any way. Therefore the rule JTtype˙definition˙swap allows an arbitrary reordering of type definitions — it suffices for a type definition to be correct that there exist a reordering that makes the type abbreviations properly ordered.

$$\frac{\begin{array}{l}\vdash \mathit{typedef}_1 \ \mathbf{and} \ldots \mathbf{and} \ \mathit{typedef}_n \ \ \triangleright \ \ E' \ \mathbf{and} \ E'' \ \mathbf{and} \ E''' \\ E'''' = E'@E''@E''' \\ E@E'''' \vdash \mathbf{ok}\end{array}}{E \vdash \mathbf{type} \ \mathit{typedef}_1 \ \mathbf{and} \ldots \mathbf{and} \ \mathit{typedef}_n \ \ \triangleright \ \ E''''} \quad \text{JTtype\_definition\_list}$$

$$\frac{E \vdash \mathbf{type} \ \overline{\mathit{typedef}_i}^{\,i} \ \mathbf{and} \ \mathit{typedef}' \ \mathbf{and} \ \mathit{typedef} \ \mathbf{and} \ \overline{\mathit{typedef}''_j}^{\,j} \ \ \triangleright \ \ E'}{E \vdash \mathbf{type} \ \overline{\mathit{typedef}_i}^{\,i} \ \mathbf{and} \ \mathit{typedef} \ \mathbf{and} \ \mathit{typedef}' \ \mathbf{and} \ \overline{\mathit{typedef}''_j}^{\,j} \ \ \triangleright \ \ E'} \quad \text{JTtype\_definition\_swap}$$

3.32 $\boxed{E \vdash \mathit{definition} : E'}$ Definition typing

Collects the bindings of a definition and ensures that they are well-formed. Each definition can bind zero, one or more names. Type variables that are mentionned by the programmer in type annotations are scoped at this level. Thus, the σ^T substitution is arbitrarily created for each definition to ensure that each type variable is used consistently in the definition.

$$\frac{\sigma^T \& E, \mathbf{TV} \vdash \mathit{pat} = \mathit{nexp} \ \ \triangleright \ \ (x_1 : t'_1), \ldots, (x_k : t'_k)}{E \vdash \mathbf{let} \ \mathit{pat} = \mathit{nexp} : (x_1 : \forall\, t'_1), \ldots, (x_k : \forall\, t'_k)} \quad \text{JTdefinition\_let\_poly}$$

$$\frac{\sigma^T \& E \vdash \mathit{pat} = \mathit{expr} \ \ \triangleright \ \ (x_1 : t'_1), \ldots, (x_k : t'_k)}{E \vdash \mathbf{let} \ \mathit{pat} = \mathit{expr} : (x_1 : t'_1), \ldots, (x_k : t'_k)} \quad \text{JTdefinition\_let\_mono}$$

$$\frac{\sigma^T \& E, \mathbf{TV} \vdash \mathit{letrec\_bindings} \ \ \triangleright \ \ (x_1 : t'_1), \ldots, (x_k : t'_k)}{E \vdash \mathbf{let\,rec} \ \mathit{letrec\_bindings} : (x_1 : \forall\, t'_1), \ldots, (x_k : \forall\, t'_k)} \quad \text{JTdefinition\_letrec}$$

$$\frac{E \vdash \mathbf{type} \ \mathit{typedef}_1 \ \mathbf{and} \ldots \mathbf{and} \ \mathit{typedef}_n \ \ \triangleright \ \ E'}{E \vdash \mathbf{type} \ \mathit{typedef}_1 \ \mathbf{and} \ldots \mathbf{and} \ \mathit{typedef}_n : E'} \quad \text{JTdefinition\_typedef}$$

$$\frac{\begin{array}{l}E \vdash \mathbf{ok} \\ \mathbf{exn} \vdash \mathit{constr\_decl} \ \ \triangleright \ \ EB\end{array}}{E \vdash \mathbf{exception} \ \mathit{constr\_decl} : EB} \quad \text{JTdefinition\_exndef}$$

3.33 $\boxed{E \vdash \mathit{definitions} : E'}$ Definition sequence typing

Collects the bindings of a definition and ensures that they are well-typed.

$$\frac{E \vdash \mathbf{ok}}{E \vdash :} \quad \textsf{JTdefinitions\_empty}$$

$$\frac{\begin{array}{c} E \vdash \mathit{definition} : E' \\ E@E' \vdash \mathit{definitions}' : E'' \end{array}}{E \vdash \mathit{definition}\ \mathit{definitions}' : E'@E''} \quad \textsf{JTdefinitions\_item}$$

3.34 $\boxed{E \vdash \mathit{program} : E'}$ Program typing

Checks a progam.

$$\frac{E \vdash \mathit{definitions} : E'}{E \vdash \mathit{definitions} : E'} \quad \textsf{JTprog\_defs}$$

$$\frac{\sigma^T \& E \vdash v : t}{E \vdash (\%primraise) v :} \quad \textsf{JTprog\_raise}$$

3.35 $\boxed{E \vdash \mathit{store} : E'}$ Store typing

Checks that the values in a store have types.

$$\frac{}{E \vdash \mathbf{empty} :} \quad \textsf{JTstore\_empty}$$

$$\frac{\begin{array}{c} E \vdash \mathit{store} : E' \\ \{\!\{\ \}\!\} \& E \vdash v : t \end{array}}{E \vdash \mathit{store}, l \mapsto v : E', (l : t)} \quad \textsf{JTstore\_map}$$

3.36 $\boxed{E \vdash \langle \mathit{program}, \mathit{store} \rangle}$ Top-level typing

Checks the combination of a program with a store. The store is typed in an environment that includes its bindings, so that it can contain cyclic structures.

$$\frac{\begin{array}{l} E @ E' \vdash \mathit{store} : E' \\ E @ E' \vdash \mathit{program} : E'' \end{array}}{E \vdash \langle \mathit{program}, \mathit{store} \rangle} \quad \mathsf{JTtop\_defs}$$

3.37 $\boxed{\sigma^T \& E \vdash L}$ Label-to-environment extraction

Used in the proof only

$$\frac{}{\sigma^T \& E \vdash} \quad \mathsf{JTLin\_nil}$$

$$\frac{\begin{array}{l} \mathbf{dom}\,(E) \;\; \triangleright \;\; \mathit{names} \\ \mathit{location} \notin \mathit{names} \end{array}}{\sigma^T \& E \vdash \mathbf{ref}\; v = \mathit{location}} \quad \mathsf{JTLin\_alloc}$$

$$\frac{\begin{array}{l} \sigma^T \& E \vdash v : t \\ E \vdash \mathit{location} \;\; \triangleright \;\; (\mathit{location} : t) \end{array}}{\sigma^T \& E \vdash !\mathit{location} = v} \quad \mathsf{JTLin\_deref}$$

$$\frac{}{\sigma^T \& E \vdash \mathit{location} := v} \quad \mathsf{JTLin\_assign}$$

3.38 $\boxed{\sigma^T \& E \vdash L \;\; \triangleright \;\; E'}$ Label-to-environment extraction

Used in the proof only

$$\frac{}{\sigma^T \& E \vdash \;\; \triangleright} \quad \mathsf{JTLout\_nil}$$

$$\frac{\sigma^T \& E \vdash v : t}{\sigma^T \& E \vdash \mathbf{ref}\; v = \mathit{location} \;\; \triangleright \;\; (\mathit{location} : t)} \quad \mathsf{JTLout\_alloc}$$

$$\frac{}{\sigma^T \& E \vdash !\mathit{location} = v \;\; \triangleright} \quad \mathsf{JTLout\_deref}$$

$$\frac{\begin{array}{l} \sigma^T \& E \vdash v : t \\ E \vdash \mathit{location} \;\; \triangleright \;\; (\mathit{location} : t) \end{array}}{\sigma^T \& E \vdash \mathit{location} := v \;\; \triangleright} \quad \mathsf{JTLout\_assign}$$

4 Operational Semantics

The operational semantics is a labelled transition system that lifts imperative and non-deterministic behavior our of the core evaluation rules. Notable aspects of the formalization include:

- explicit rules for evaluation in context (instead of a grammar of evaluation contexts),
- small-step propogation of exceptions,
- substitution-based function application,
- right-to-left evaluation ordering, which is overspecified compared to the OCaml manual; furthermore, this choice of evaluation ordering for record expressions differs from the implementation's choice, which is based on the type declaration,
- unlike the implementation, we do not treat curried functions specially, the difference can be seen in this program: $\textbf{let}\, f \;=\; \textbf{function}\, 1 \;\rightarrow\; \textbf{function}\, \_ \;\rightarrow\; 10;;\textbf{let}\, \_ = f\, 2;;$ which does not raise an exception in the implementation.
- As in the type system, several rules have premises that state there are at least 1 (or 2) elements of a list, despite there being 3 or 4 dots. This is because Ott does not use dot imposed length restrictions in the theorem prover models.

4.1 $\boxed{\vdash \mathit{expr}\, \textbf{matches}\, \mathit{pattern}}$ Pattern matching

Determines if a value matches a pattern.

$$\frac{}{\vdash v\, \textbf{matches}\, x}\ \textsf{JM\_matchP\_var}$$

$$\frac{}{\vdash v\, \textbf{matches}\, \_}\ \textsf{JM\_matchP\_any}$$

$$\frac{}{\vdash \mathit{constant}\, \textbf{matches}\, \mathit{constant}}\ \textsf{JM\_matchP\_constant}$$

$$\frac{\vdash v\, \textbf{matches}\, \mathit{pat}}{\vdash v\, \textbf{matches}\, \mathit{pat}\, \textbf{as}\, x}\ \textsf{JM\_matchP\_alias}$$

$$\frac{\vdash v\, \textbf{matches}\, \mathit{pat}}{\vdash v\, \textbf{matches}\, (\mathit{pat} : t)}\ \textsf{JM\_matchP\_typed}$$

$$\frac{\vdash v\, \textbf{matches}\, \mathit{pat}_1}{\vdash v\, \textbf{matches}\, \mathit{pat}_1 | \mathit{pat}_2}\ \textsf{JM\_matchP\_or\_left}$$

$$\frac{\vdash v\, \textbf{matches}\, \mathit{pat}_2}{\vdash v\, \textbf{matches}\, \mathit{pat}_1 | \mathit{pat}_2}\ \textsf{JM\_matchP\_or\_right}$$

$$\frac{\vdash v_1 \ \textbf{matches}\ pat_1 \quad \ldots \quad \vdash v_n \ \textbf{matches}\ pat_n}{\vdash constr(v_1, \ \ldots, v_n) \ \textbf{matches}\ constr(pat_1, \ \ldots, pat_n)} \quad \text{JM\_matchP\_construct}$$

$$\frac{}{\vdash constr(v_1, \ \ldots, v_n) \ \textbf{matches}\ constr \ \_} \quad \text{JM\_matchP\_construct\_any}$$

$$\frac{\vdash v_1 \ \textbf{matches}\ pat_1 \quad \ldots. \quad \vdash v_n \ \textbf{matches}\ pat_n}{\vdash (v_1, \ \ldots., v_n) \ \textbf{matches}\ (pat_1, \ \ldots., pat_n)} \quad \text{JM\_matchP\_tuple}$$

$$\frac{\begin{array}{l} field\_name'_1 = v'_1 \ldots field\_name'_n = v'_n \ fn_1 = v''_1 \ldots fn_l = v''_l \ \textbf{PERMUTES}\ field\_name_1 = v_1 \ldots field\_name_m = v_m \\ \vdash v'_1 \ \textbf{matches}\ pat_1 \quad \ldots \quad \vdash v'_n \ \textbf{matches}\ pat_n \\ field\_name_1 \ldots field\_name_m \ \textbf{distinct} \end{array}}{\vdash \{field\_name_1 = v_1; \ \ldots; field\_name_m = v_m\} \ \textbf{matches}\ \{field\_name'_1 = pat_1; \ \ldots; field\_name'_n = pat_n\}} \quad \text{JM\_matchP\_record}$$

$$\frac{\begin{array}{c} \vdash v_1 \ \textbf{matches}\ pat_1 \\ \vdash v_2 \ \textbf{matches}\ pat_2 \end{array}}{\vdash v_1 :: v_2 \ \textbf{matches}\ pat_1 :: pat_2} \quad \text{JM\_matchP\_cons}$$

4.2 $\boxed{\vdash expr \ \textbf{matches}\ pattern \ \ \triangleright \ \ \{\!\!\{ substs\_x \}\!\!\}}$ Pattern matching with substitution creation

Determines if a value matches a pattern and destructures the value into a substitution according to the pattern's variables. The previous pattern matching relation is used to get deterministic behavior for | patterns.

$$\frac{}{\vdash v \ \textbf{matches}\ x \ \ \triangleright \ \ \{\!\!\{ x \leftarrow v \}\!\!\}} \quad \text{JM\_match\_var}$$

$$\frac{}{\vdash v \ \textbf{matches}\ \_ \ \ \triangleright \ \ \{\!\!\{ \ \}\!\!\}} \quad \text{JM\_match\_any}$$

$$\frac{}{\vdash constant \ \textbf{matches}\ constant \ \ \triangleright \ \ \{\!\!\{ \ \}\!\!\}} \quad \text{JM\_match\_constant}$$

$$\frac{\vdash v \ \textbf{matches}\ pat \ \ \triangleright \ \ \{\!\!\{ x_1 \leftarrow v_1, \ \ldots, x_n \leftarrow v_n \}\!\!\}}{\vdash v \ \textbf{matches}\ pat \ \textbf{as}\ x \ \ \triangleright \ \ \{\!\!\{ x_1 \leftarrow v_1, \ \ldots, x_n \leftarrow v_n, x \leftarrow v \}\!\!\}} \quad \text{JM\_match\_alias}$$

$$\frac{\vdash v \ \textbf{matches}\ pat \ \ \triangleright \ \ \{\!\!\{ x_1 \leftarrow v_1, \ \ldots, x_n \leftarrow v_n \}\!\!\}}{\vdash v \ \textbf{matches}\ (pat : t) \ \ \triangleright \ \ \{\!\!\{ x_1 \leftarrow v_1, \ \ldots, x_n \leftarrow v_n \}\!\!\}} \quad \text{JM\_match\_typed}$$

$$\frac{\vdash v \ \textbf{matches}\ pat_1 \ \ \triangleright \ \ \{\!\!\{ x_1 \leftarrow v_1, \ \ldots, x_n \leftarrow v_n \}\!\!\}}{\vdash v \ \textbf{matches}\ pat_1 | pat_2 \ \ \triangleright \ \ \{\!\!\{ x_1 \leftarrow v_1, \ \ldots, x_n \leftarrow v_n \}\!\!\}} \quad \text{JM\_match\_or\_left}$$

$$\frac{\neg(v\ \mathbf{matches}\ pat_1) \quad \vdash v\ \mathbf{matches}\ pat_2\ \triangleright\ \{\!\{x_1 \leftarrow v_1, .., x_n \leftarrow v_n\}\!\}}{\vdash v\ \mathbf{matches}\ pat_1 | pat_2\ \triangleright\ \{\!\{x_1 \leftarrow v_1, .., x_n \leftarrow v_n\}\!\}}\ \textsf{JM\_match\_or\_right}$$

$$\frac{\vdash v_1\ \mathbf{matches}\ pat_1\ \triangleright\ \{\!\{substs\_x_1\}\!\} \quad \ldots \quad \vdash v_n\ \mathbf{matches}\ pat_n\ \triangleright\ \{\!\{substs\_x_n\}\!\}}{\vdash constr(v_1, \ldots, v_n)\ \mathbf{matches}\ constr(pat_1, \ldots, pat_n)\ \triangleright\ \{\!\{substs\_x_1 @ \ldots @ substs\_x_n\}\!\}}\ \textsf{JM\_match\_construct}$$

$$\frac{}{\vdash constr(v_1, \ldots, v_n)\ \mathbf{matches}\ constr\ \_\ \triangleright\ \{\!\{\}\!\}}\ \textsf{JM\_match\_construct\_any}$$

$$\frac{\vdash v_1\ \mathbf{matches}\ pat_1\ \triangleright\ \{\!\{substs\_x_1\}\!\} \quad \ldots. \quad \vdash v_n\ \mathbf{matches}\ pat_n\ \triangleright\ \{\!\{substs\_x_n\}\!\}}{\vdash (v_1, \ldots., v_n)\ \mathbf{matches}\ (pat_1, \ldots., pat_n)\ \triangleright\ \{\!\{substs\_x_1 @ \ldots. @ substs\_x_n\}\!\}}\ \textsf{JM\_match\_tuple}$$

$$\frac{\begin{array}{l} field\_name'_1 = v'_1 \ldots field\_name'_n = v'_n\ fn_1 = v''_1 .. fn_l = v''_l\ \mathbf{PERMUTES}\ field\_name_1 = v_1 \ldots field\_name_m = v_m \\ \vdash v'_1\ \mathbf{matches}\ pat_1\ \triangleright\ \{\!\{substs\_x_1\}\!\} \quad \ldots \quad \vdash v'_n\ \mathbf{matches}\ pat_n\ \triangleright\ \{\!\{substs\_x_n\}\!\} \\ field\_name_1 \ldots field\_name_m\ \mathbf{distinct} \end{array}}{\vdash \{field\_name_1 = v_1; \ldots; field\_name_m = v_m\}\ \mathbf{matches}\ \{field\_name'_1 = pat_1; \ldots; field\_name'_n = pat_n\}\ \triangleright\ \{\!\{substs\_x_1 @ \ldots @ substs\_x_n\}\!\}}\ \textsf{JM\_match\_record}$$

$$\frac{\begin{array}{c} \vdash v_1\ \mathbf{matches}\ pat_1\ \triangleright\ \{\!\{substs\_x_1\}\!\} \\ \vdash v_2\ \mathbf{matches}\ pat_2\ \triangleright\ \{\!\{substs\_x_2\}\!\} \end{array}}{\vdash v_1 :: v_2\ \mathbf{matches}\ pat_1 :: pat_2\ \triangleright\ \{\!\{substs\_x_1 @ substs\_x_2\}\!\}}\ \textsf{JM\_match\_cons}$$

4.3 $\boxed{\mathbf{recfun}\,(letrec\_bindings, pattern\_matching)\ \triangleright\ expr}$ Recursive function helper

Expands a recursive definition.

$$\frac{letrec\_bindings = (x_1 = \mathbf{function}\ pattern\_matching_1\ \mathbf{and} \ldots \mathbf{and}\ x_n = \mathbf{function}\ pattern\_matching_n)}{\mathbf{recfun}\,(letrec\_bindings, pattern\_matching)\ \triangleright\ \{\!\{x_1 \leftarrow \mathbf{let\ rec}\ letrec\_bindings\ \mathbf{in}\ x_1, \ldots, x_n \leftarrow \mathbf{let\ rec}\ letrec\_bindings\ \mathbf{in}\ x_n\}\!\}(\mathbf{function}\ pattern\_matching)}\ \textsf{Jrecfun\_letrec}$$

4.4 $\boxed{\vdash \mathbf{funval}\,(e)}$ Function values

Determines if an expression is a function value, for use in (Jbprim_equal_fun).

$$\frac{}{\vdash \mathbf{funval}\,((\%\mathbf{prim}\,unary\_prim))}\ \textsf{Jfunval\_up}$$

$$\frac{}{\vdash \mathbf{funval}\,((\%\mathbf{prim}\,binary\_prim))}\ \textsf{Jfunval\_bp}$$

$$\frac{}{\vdash \mathbf{funval}\,((\%\mathbf{prim}\, binary\_prim)\ v)} \quad \text{Jfunval\_bp\_app}$$

$$\frac{}{\vdash \mathbf{funval}\,(\mathbf{function}\ pattern\_matching)} \quad \text{Jfunval\_func}$$

4.5 $\boxed{\vdash unary\_prim\ expr \xrightarrow{L} expr'}$ Unary primitive evaluation

Computes the result of a unary primitive application.

$$\frac{}{\vdash \mathbf{not}\,\mathbf{true} \longrightarrow \mathbf{false}} \quad \text{Juprim\_not\_true}$$

$$\frac{}{\vdash \mathbf{not}\,\mathbf{false} \longrightarrow \mathbf{true}} \quad \text{Juprim\_not\_false}$$

$$\frac{}{\vdash \sim\!-\ \dot{n} \longrightarrow 0 \mathbin{\dot{-}} \dot{n}} \quad \text{Juprim\_uminus}$$

The effect of creating a reference is communicated to the store via the label on the reduction arrow. Similarly the reduction arrow carries the value read from the store when accessing a location.

$$\frac{}{\vdash \mathbf{ref}\ v \xrightarrow{\mathbf{ref}\ v=l} l} \quad \text{Juprim\_ref\_alloc}$$

$$\frac{}{\vdash !\, l \xrightarrow{!l=v} v} \quad \text{Juprim\_deref}$$

4.6 $\boxed{\vdash expr_1\ binary\_prim\ expr_2 \xrightarrow{L} expr}$ Binary primitive evaluation

Computes the result of a binary primitive application.

$$\frac{\vdash \mathbf{funval}\,(v)}{\vdash v\ =\ v' \longrightarrow (\%\mathbf{primraise})\,(\mathbf{Invalid\_argument}(\mathbf{equal\_error\_string}))} \quad \text{Jbprim\_equal\_fun}$$

$$\frac{}{\vdash constant\ =\ constant \longrightarrow \mathbf{true}} \quad \text{Jbprim\_equal\_const\_true}$$

$$\frac{constant \neq constant'}{\vdash constant\ =\ constant' \longrightarrow \mathbf{false}} \quad \text{Jbprim\_equal\_const\_false}$$

$$\frac{}{\vdash l\ =\ l' \longrightarrow ((\%\mathbf{prim}\ =)\,((\%\mathbf{prim}!)\ l))\,((\%\mathbf{prim}!)\ l')} \quad \text{Jbprim\_equal\_loc}$$

$$\frac{}{\vdash (v_1 :: v_2) \;=\; (v_1' :: v_2') \;\longrightarrow\; (((\%\mathbf{prim}\, =)\; v_1)\; v_1')\&\&(((\%\mathbf{prim}\, =)\; v_2)\; v_2')} \quad \textsf{Jbprim\_equal\_cons}$$

$$\frac{}{\vdash (v_1 :: v_2) \;=\; [] \;\longrightarrow\; \mathbf{false}} \quad \textsf{Jbprim\_equal\_cons\_nil}$$

$$\frac{}{\vdash [] \;=\; (v_1 :: v_2) \;\longrightarrow\; \mathbf{false}} \quad \textsf{Jbprim\_equal\_nil\_cons}$$

$$\frac{\mathbf{length}\,(v_1)\,....\,(v_n) \geq 2}{\vdash (v_1,\, ...\, ,v_n) \;=\; (v_1',\, ...\, ,v_n') \;\longrightarrow\; \mathbf{AND}\,(((\%\mathbf{prim}\, =)\; v_1)\; v_1'\&\&\, ...\, \&\&((\%\mathbf{prim}\, =)\; v_n)\; v_n')} \quad \textsf{Jbprim\_equal\_tuple}$$

$$\frac{}{\vdash (\mathit{constr}(v_1,\, ...\, ,v_n)) \;=\; (\mathit{constr}(v_1',\, ...\, ,v_n')) \;\longrightarrow\; \mathbf{AND}\,(((\%\mathbf{prim}\, =)\; v_1)\; v_1'\&\&\, ...\, \&\&((\%\mathbf{prim}\, =)\; v_n)\; v_n')} \quad \textsf{Jbprim\_equal\_constr}$$

$$\frac{\mathit{constr} \;\neq\; \mathit{constr}'}{\vdash \mathit{constr}(v_1,\, ...\, ,v_m) \;=\; \mathit{constr}'(v_1',\, ...\, ,v_n') \;\longrightarrow\; \mathbf{false}} \quad \textsf{Jbprim\_equal\_constr\_false}$$

$$\frac{}{\vdash \mathit{constr}' \;=\; \mathit{constr}(v_1,\, ...\, ,v_n) \;\longrightarrow\; \mathbf{false}} \quad \textsf{Jbprim\_equal\_const\_constr\_false}$$

$$\frac{}{\vdash \mathit{constr}(v_1,\, ...\, ,v_n) \;=\; \mathit{constr}' \;\longrightarrow\; \mathbf{false}} \quad \textsf{Jbprim\_equal\_constr\_const\_false}$$

$$\frac{\begin{array}{c} v' = \{\mathit{fn}_1'' = v_1'';\, ...\,; \mathit{fn}_m'' = v_m''\} \\ \mathit{fn}_1 \ldots \mathit{fn}_n\ \mathbf{PERMUTES}\ \mathit{fn}_1'' \ldots \mathit{fn}_m'' \end{array}}{\vdash \{\mathit{fn}_1 = v_1;\, ...\,; \mathit{fn}_n = v_n\} \;=\; v' \;\longrightarrow\; \mathbf{AND}\,(((\%\mathbf{prim}\, =)\; v_1)\; (v'.\mathit{fn}_1)\&\&\, ...\, \&\&((\%\mathbf{prim}\, =)\; v_n)\; (v'.\mathit{fn}_n))} \quad \textsf{Jbprim\_equal\_rec}$$

$$\frac{}{\vdash \dot{n}_1 \;+\; \dot{n}_2 \;\longrightarrow\; \dot{n}_1 \;\dot{+}\; \dot{n}_2} \quad \textsf{Jbprim\_plus}$$

$$\frac{}{\vdash \dot{n}_1 \;-\; \dot{n}_2 \;\longrightarrow\; \dot{n}_1 \;\dot{-}\; \dot{n}_2} \quad \textsf{Jbprim\_minus}$$

$$\frac{}{\vdash \dot{n}_1 \;*\; \dot{n}_2 \;\longrightarrow\; \dot{n}_1 \;\dot{*}\; \dot{n}_2} \quad \textsf{Jbprim\_times}$$

$$\frac{}{\vdash \dot{n} \,/\, 0 \;\longrightarrow\; (\%\mathbf{primraise})\;\mathbf{Division\_by\_zero}} \quad \textsf{Jbprim\_div0}$$

$$\frac{\dot{n}_2 \;\neq\; 0}{\vdash \dot{n}_1 \,/\, \dot{n}_2 \;\longrightarrow\; \dot{n}_1 \;\dot{/}\; \dot{n}_2} \quad \textsf{Jbprim\_div}$$

The side effect of an assignment is communicated to the store via the label on the reduction arrow.

$$\frac{}{\vdash l \;:=\; v \;\xrightarrow{l:=v}\; ()} \quad \textsf{Jbprim\_assign}$$

4.7 $\boxed{\vdash expr\ \mathbf{with}\ pattern\_matching \longrightarrow pattern\_matching'}$ Pattern matching step

Proceeding to the next case because the first, but not only, case has failed to match.

$$\frac{\neg(v\ \mathbf{matches}\ pat) \qquad \mathbf{length}\,(e_1)\ldots(e_n) \geq 1}{\vdash v\ \mathbf{with}\ pat \to e | pat_1 \to e_1 | \ldots | pat_n \to e_n \longrightarrow pat_1 \to e_1 | \ldots | pat_n \to e_n} \quad \text{JRmatching\_next}$$

4.8 $\boxed{\vdash expr\ \mathbf{with}\ pattern\_matching \longrightarrow expr'}$ Pattern matching finished

Proceeding to an expression because the first case matches, or the only case does not match.

$$\frac{\vdash v\ \mathbf{matches}\ pat \ \rhd\ \{\!\{x_1 \leftarrow v_1, \ldots, x_m \leftarrow v_m\}\!\}}{\vdash v\ \mathbf{with}\ pat \to e | pat_1 \to e_1 | \ldots | pat_n \to e_n \longrightarrow \{\!\{x_1 \leftarrow v_1, \ldots, x_m \leftarrow v_m\}\!\} e} \quad \text{JRmatching\_found}$$

$$\frac{\neg(v\ \mathbf{matches}\ pat)}{\vdash v\ \mathbf{with}\ pat \to e \longrightarrow (\%\mathbf{primraise})\ \mathbf{Match\_failure}} \quad \text{JRmatching\_fail}$$

4.9 $\boxed{\vdash expr \xrightarrow{L} expr'}$ Expression evaluation

Reduces an expression one-step. Most evaluation contexts require two rules, one for normal evaluation and one for exception propagation.

$$\frac{\vdash unary\_prim\ v \xrightarrow{L} e}{\vdash (\%\mathbf{prim}\, unary\_prim)\ v \xrightarrow{L} e} \quad \text{JR\_expr\_uprim}$$

$$\frac{\vdash v_1\ binary\_prim\ v_2 \xrightarrow{L} e}{\vdash ((\%\mathbf{prim}\, binary\_prim)\ v_1)\ v_2 \xrightarrow{L} e} \quad \text{JR\_expr\_bprim}$$

$$\frac{}{\vdash (e : t) \longrightarrow e} \quad \text{JR\_expr\_typed\_ctx}$$

Right-to-left evaluation order for application (i.e., argument before function).

$$\frac{\vdash e_0 \xrightarrow{L} e_0'}{\vdash e_1\ e_0 \xrightarrow{L} e_1\ e_0'} \quad \text{JR\_expr\_apply\_ctx\_arg}$$

$$\frac{}{\vdash e\,((\%\mathbf{primraise})\ v) \ \longrightarrow\ (\%\mathbf{primraise})\ v} \quad \textsf{JR\_expr\_apply\_raise1}$$

$$\frac{\vdash e_1 \xrightarrow{L} e_1'}{\vdash e_1\ v_0 \xrightarrow{L} e_1'\ v_0} \quad \textsf{JR\_expr\_apply\_ctx\_fun}$$

$$\frac{}{\vdash ((\%\mathbf{primraise})\ v)\ v' \ \longrightarrow\ (\%\mathbf{primraise})\ v} \quad \textsf{JR\_expr\_apply\_raise2}$$

$$\frac{}{\vdash (\mathbf{function}\ \mathit{pattern\_matching}\ v_0) \ \longrightarrow\ \mathbf{match}\ v_0\ \mathbf{with}\ \mathit{pattern\_matching}} \quad \textsf{JR\_expr\_apply}$$

$$\frac{\vdash e_0 \xrightarrow{L} e_0'}{\vdash \mathbf{let}\ \mathit{pat} = e_0\ \mathbf{in}\ e \xrightarrow{L} \mathbf{let}\ \mathit{pat} = e_0'\ \mathbf{in}\ e} \quad \textsf{JR\_expr\_let\_ctx}$$

$$\frac{}{\vdash \mathbf{let}\ \mathit{pat} = (\%\mathbf{primraise})\ v\ \mathbf{in}\ e \ \longrightarrow\ (\%\mathbf{primraise})\ v} \quad \textsf{JR\_expr\_let\_raise}$$

$$\frac{\vdash v\ \mathbf{matches}\ \mathit{pat} \ \rhd\ \{\!\{x_1 \leftarrow v_1, \ldots, x_m \leftarrow v_m\}\!\}}{\vdash \mathbf{let}\ \mathit{pat} = v\ \mathbf{in}\ e \ \longrightarrow\ \{\!\{x_1 \leftarrow v_1, \ldots, x_m \leftarrow v_m\}\!\}\, e} \quad \textsf{JR\_expr\_let\_subst}$$

$$\frac{\neg(v\ \mathbf{matches}\ \mathit{pat})}{\vdash \mathbf{let}\ \mathit{pat} = v\ \mathbf{in}\ e \ \longrightarrow\ (\%\mathbf{primraise})\ \mathbf{Match\_failure}} \quad \textsf{JR\_expr\_let\_fail}$$

$$\frac{\begin{array}{l}\mathit{letrec\_bindings} = (x_1 = \mathbf{function}\ \mathit{pattern\_matching}_1\ \mathbf{and} \ldots \mathbf{and}\ x_n = \mathbf{function}\ \mathit{pattern\_matching}_n) \\ \mathbf{recfun}\,(\mathit{letrec\_bindings}, \mathit{pattern\_matching}_1) \ \rhd\ e_1 \quad \ldots \quad \mathbf{recfun}\,(\mathit{letrec\_bindings}, \mathit{pattern\_matching}_n) \ \rhd\ e_n\end{array}}{\vdash \mathbf{let\ rec}\ \mathit{letrec\_bindings}\ \mathbf{in}\ e \ \longrightarrow\ \{\!\{x_1 \leftarrow e_1, \ldots, x_n \leftarrow e_n\}\!\}\, e} \quad \textsf{JR\_expr\_letrec}$$

$$\frac{\vdash e_1 \xrightarrow{L} e_1'}{\vdash e_1; e_2 \xrightarrow{L} e_1'; e_2} \quad \textsf{JR\_expr\_sequence\_ctx\_left}$$

$$\frac{}{\vdash ((\%\mathbf{primraise})\ v); e \ \longrightarrow\ (\%\mathbf{primraise})\ v} \quad \textsf{JR\_expr\_sequence\_raise}$$

$$\frac{}{\vdash v; e_2 \ \longrightarrow\ e_2} \quad \textsf{JR\_expr\_sequence}$$

$$\frac{\vdash e_1 \xrightarrow{L} e_1'}{\vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \xrightarrow{L} \mathbf{if}\ e_1'\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3} \quad \textsf{JR\_expr\_ifthenelse\_ctx}$$

$$\frac{}{\vdash \mathbf{if}\ (\%\mathbf{primraise})\ v\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 \ \longrightarrow\ (\%\mathbf{primraise})\ v} \quad \textsf{JR\_expr\_if\_raise}$$

$$\frac{}{\vdash \mathbf{if}\ \mathbf{true}\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \ \longrightarrow\ e_2} \quad \textsf{JR\_expr\_ifthenelse\_true}$$

$$\frac{}{\vdash \textbf{if false then } e_2 \textbf{ else } e_3 \ \longrightarrow\ e_3} \quad \textsf{JR\_expr\_ifthenelse\_false}$$

We treat matching one pattern against one value as atomic (this would be relevant when matching the contents of a reference after introducing concurrent evaluation).

$$\frac{\vdash e \xrightarrow{L} e'}{\vdash \textbf{match } e \textbf{ with } \mathit{pattern\_matching} \xrightarrow{L} \textbf{match } e' \textbf{ with } \mathit{pattern\_matching}} \quad \textsf{JR\_expr\_match\_ctx}$$

$$\frac{}{\vdash \textbf{match } (\%\textbf{primraise})\ v \textbf{ with } \mathit{pattern\_matching} \ \longrightarrow\ (\%\textbf{primraise})\ v} \quad \textsf{JR\_expr\_match\_raise}$$

$$\frac{\vdash v \textbf{ with } \mathit{pattern\_matching} \longrightarrow \mathit{pattern\_matching}'}{\vdash \textbf{match } v \textbf{ with } \mathit{pattern\_matching} \ \longrightarrow\ \textbf{match } v \textbf{ with } \mathit{pattern\_matching}'} \quad \textsf{JR\_expr\_match\_step}$$

$$\frac{\vdash v \textbf{ with } \mathit{pattern\_matching} \longrightarrow e'}{\vdash \textbf{match } v \textbf{ with } \mathit{pattern\_matching} \ \longrightarrow\ e'} \quad \textsf{JR\_expr\_match\_success}$$

$$\frac{}{\vdash e_1 \&\& e_2 \ \longrightarrow\ \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else false}} \quad \textsf{JR\_expr\_and}$$

$$\frac{}{\vdash e_1 || e_2 \ \longrightarrow\ \textbf{if } e_1 \textbf{ then true else } e_2} \quad \textsf{JR\_expr\_or}$$

$$\frac{}{\vdash \textbf{while } e_1 \textbf{ do } e_2 \textbf{ done} \ \longrightarrow\ \textbf{if } e_1 \textbf{ then } (e_2; \textbf{while } e_1 \textbf{ do } e_2 \textbf{ done})} \quad \textsf{JR\_expr\_while}$$

We specify the evaluation of e_1 before e_2 in **for** loops, which appears to follow the implementation.

$$\frac{\vdash e_1 \xrightarrow{L} e_1'}{\vdash \textbf{for } x = e_1 \ [\textbf{down}]\textbf{to } e_2 \textbf{ do } e_3 \textbf{ done} \xrightarrow{L} \textbf{for } x = e_1' \ [\textbf{down}]\textbf{to } e_2 \textbf{ do } e_3 \textbf{ done}} \quad \textsf{JR\_expr\_for\_ctx1}$$

$$\frac{}{\vdash \textbf{for } x = (\%\textbf{primraise})\ v \ [\textbf{down}]\textbf{to } e_2 \textbf{ do } e_3 \textbf{ done} \ \longrightarrow\ (\%\textbf{primraise})\ v} \quad \textsf{JR\_expr\_for\_raise1}$$

$$\frac{\vdash e_2 \xrightarrow{L} e_2'}{\vdash \textbf{for } x = v_1 \ [\textbf{down}]\textbf{to } e_2 \textbf{ do } e_3 \textbf{ done} \xrightarrow{L} \textbf{for } x = v_1 \ [\textbf{down}]\textbf{to } e_2' \textbf{ do } e_3 \textbf{ done}} \quad \textsf{JR\_expr\_for\_ctx2}$$

$$\frac{}{\vdash \textbf{for } x = v \ [\textbf{down}]\textbf{to } (\%\textbf{primraise})\ v' \textbf{ do } e_3 \textbf{ done} \ \longrightarrow\ (\%\textbf{primraise})\ v'} \quad \textsf{JR\_expr\_for\_raise2}$$

$$\frac{\dot{n}_1 \mathrel{\dot{\le}} \dot{n}_2}{\vdash \textbf{for } x = \dot{n}_1 \textbf{ to } \dot{n}_2 \textbf{ do } e \textbf{ done} \ \longrightarrow\ (\textbf{let } x = \dot{n}_1 \textbf{ in } e); \textbf{for } x = \dot{n}_1 \mathbin{\dot{+}} 1 \textbf{ to } \dot{n}_2 \textbf{ do } e \textbf{ done}} \quad \textsf{JR\_expr\_for\_to\_do}$$

$$\frac{\dot{n}_1 \mathrel{\dot{>}} \dot{n}_2}{\vdash \textbf{for}\, x = \dot{n}_1 \,\textbf{to}\, \dot{n}_2 \,\textbf{do}\, e \,\textbf{done} \longrightarrow ()} \quad \textsf{JR\_expr\_for\_to\_done}$$

$$\frac{\dot{n}_2 \mathrel{\dot{\leq}} \dot{n}_1}{\vdash \textbf{for}\, x = \dot{n}_1 \,\textbf{downto}\, \dot{n}_2 \,\textbf{do}\, e \,\textbf{done} \longrightarrow (\textbf{let}\, x = \dot{n}_1 \,\textbf{in}\, e); \textbf{for}\, x = \dot{n}_1 \mathbin{\dot{-}} 1 \,\textbf{downto}\, \dot{n}_2 \,\textbf{do}\, e \,\textbf{done}} \quad \textsf{JR\_expr\_for\_downto\_do}$$

$$\frac{\dot{n}_2 \mathrel{\dot{>}} \dot{n}_1}{\vdash \textbf{for}\, x = \dot{n}_1 \,\textbf{downto}\, \dot{n}_2 \,\textbf{do}\, e \,\textbf{done} \longrightarrow ()} \quad \textsf{JR\_expr\_for\_downto\_done}$$

$$\frac{\vdash e \xrightarrow{L} e'}{\vdash \textbf{try}\, e \,\textbf{with}\, \mathit{pattern\_matching} \xrightarrow{L} \textbf{try}\, e' \,\textbf{with}\, \mathit{pattern\_matching}} \quad \textsf{JR\_expr\_try\_ctx}$$

$$\frac{}{\vdash \textbf{try}\, v \,\textbf{with}\, \mathit{pattern\_matching} \longrightarrow v} \quad \textsf{JR\_expr\_try\_return}$$

$$\frac{}{\vdash \textbf{try}\, (\%\textbf{primraise})\, v \,\textbf{with}\, \mathit{pat\_exp}_1 | \ldots | \mathit{pat\_exp}_n \longrightarrow \textbf{match}\, v \,\textbf{with}\, \mathit{pat\_exp}_1 | \ldots | \mathit{pat\_exp}_n | \_ \rightarrow ((\%\textbf{primraise})\, v)} \quad \textsf{JR\_expr\_try\_catch}$$

We specify right-to-left evaluation order for tuples, applied variant constructors, and ::.

$$\frac{\vdash e \xrightarrow{L} e'}{\vdash e_1, .., e_m, e, v_1, .., v_n \xrightarrow{L} e_1, .., e_m, e', v_1, .., v_n} \quad \textsf{JR\_expr\_tuple\_ctx}$$

$$\frac{}{\vdash e_1, .., e_m, ((\%\textbf{primraise})\, v), v_1, .., v_n \longrightarrow (\%\textbf{primraise})\, v} \quad \textsf{JR\_expr\_tuple\_raise}$$

$$\frac{\vdash e \xrightarrow{L} e'}{\vdash \mathit{constr}(e_1, .., e_m, e, v_1, .., v_n) \xrightarrow{L} \mathit{constr}(e_1, .., e_m, e', v_1, .., v_n)} \quad \textsf{JR\_expr\_constr\_ctx}$$

$$\frac{}{\vdash \mathit{constr}(e_1, .., e_m, ((\%\textbf{primraise})\, v), v_1, .., v_n) \longrightarrow (\%\textbf{primraise})\, v} \quad \textsf{JR\_expr\_constr\_raise}$$

$$\frac{\vdash e \xrightarrow{L} e'}{\vdash e_0 :: e \xrightarrow{L} e_0 :: e'} \quad \textsf{JR\_expr\_cons\_ctx1}$$

$$\frac{}{\vdash e :: ((\%\textbf{primraise})\, v) \longrightarrow (\%\textbf{primraise})\, v} \quad \textsf{JR\_expr\_cons\_raise1}$$

$$\frac{\vdash e \xrightarrow{L} e'}{\vdash e :: v \xrightarrow{L} e' :: v} \quad \textsf{JR\_expr\_cons\_ctx2}$$

$$\frac{}{\vdash ((\%\mathbf{primraise})\ v) :: v' \longrightarrow (\%\mathbf{primraise})\ v} \quad \text{JR\_expr\_cons\_raise2}$$

We specify right-to-left evaluation for records. The bytecode implementation appears to go right to left after first reordering the record to correspond to the field ordering in the record type definition.

$$\frac{\vdash \mathit{expr} \xrightarrow{L} \mathit{expr}'}{\begin{array}{l}\vdash \{\mathit{fn}_1 = e_1;\ \ldots;\mathit{fn}_m = e_m;\mathit{field\_name} = \mathit{expr};\mathit{fn}'_1 = v_1;\ \ldots;\mathit{fn}'_n = v_n\} \xrightarrow{L} \\ \qquad \{\mathit{fn}_1 = e_1;\ \ldots;\mathit{fn}_m = e_m;\mathit{field\_name} = \mathit{expr}';\mathit{fn}'_1 = v_1;\ \ldots;\mathit{fn}'_n = v_n\}\end{array}} \quad \text{JR\_expr\_record\_ctx}$$

$$\frac{}{\vdash \{\mathit{fn}_1 = e_1;\ \ldots;\mathit{fn}_m = e_m;\mathit{fn} = (\%\mathbf{primraise})\ v;\mathit{fn}'_1 = v_1;\ \ldots;\mathit{fn}'_n = v_n\} \longrightarrow (\%\mathbf{primraise})\ v} \quad \text{JR\_expr\_record\_raise}$$

The bytecode implementation appears to evaluate the leftmost position first in **with** expressions, so we follow that here.

$$\frac{\vdash e \xrightarrow{L} e'}{\begin{array}{l}\vdash \{v\ \mathbf{with}\ \mathit{fn}_1 = e_1;\ \ldots;\mathit{fn}_m = e_m;\mathit{field\_name} = e;\mathit{fn}'_1 = v_1;\ \ldots;\mathit{fn}'_n = v_n\} \xrightarrow{L} \\ \qquad \{v\ \mathbf{with}\ \mathit{fn}_1 = e_1;\ \ldots;\mathit{fn}_m = e_m;\mathit{field\_name} = e';\mathit{fn}'_1 = v_1;\ \ldots;\mathit{fn}'_n = v_n\}\end{array}} \quad \text{JR\_expr\_record\_with\_ctx1}$$

$$\frac{}{\vdash \{v'\ \mathbf{with}\ \mathit{fn}_1 = e_1;\ \ldots;\mathit{fn}_m = e_m;\mathit{fn} = (\%\mathbf{primraise})\ v;\mathit{fn}'_1 = v_1;\ \ldots;\mathit{fn}'_n = v_n\} \longrightarrow (\%\mathbf{primraise})\ v} \quad \text{JR\_expr\_record\_with\_raise1}$$

$$\frac{\vdash e \xrightarrow{L} e'}{\vdash \{e\ \mathbf{with}\ \mathit{field\_name}_1 = e_1;\ \ldots;\mathit{field\_name}_n = e_n\} \xrightarrow{L} \{e'\ \mathbf{with}\ \mathit{field\_name}_1 = e_1;\ \ldots;\mathit{field\_name}_n = e_n\}} \quad \text{JR\_expr\_record\_with\_ctx2}$$

$$\frac{}{\vdash \{(\%\mathbf{primraise})\ v\ \mathbf{with}\ \mathit{field\_name}_1 = e_1;\ \ldots;\mathit{field\_name}_n = e_n\} \longrightarrow (\%\mathbf{primraise})\ v} \quad \text{JR\_expr\_record\_raise\_ctx2}$$

$$\frac{\begin{array}{c}\mathbf{length}\,(v''_1)\ldots(v''_l) \geq 1 \\ \mathit{field\_name} \notin \mathit{fn}_1 \ldots \mathit{fn}_m\end{array}}{\begin{array}{l}\vdash \{\{\mathit{fn}_1 = v_1;\ \ldots;\mathit{fn}_m = v_m;\mathit{field\_name} = v;\mathit{fn}'_1 = v'_1;\ \ldots;\mathit{fn}'_n = v'_n\}\ \mathbf{with}\ \mathit{field\_name} = v';\mathit{fn}''_1 = v''_1;\ \ldots;\mathit{fn}''_l = v''_l\} \longrightarrow \\ \qquad \{\{\mathit{fn}_1 = v_1;\ \ldots;\mathit{fn}_m = v_m;\mathit{field\_name} = v';\mathit{fn}'_1 = v'_1;\ \ldots;\mathit{fn}'_n = v'_n\}\ \mathbf{with}\ \mathit{fn}''_1 = v''_1;\ \ldots;\mathit{fn}''_l = v''_l\}\end{array}} \quad \text{JR\_expr\_record\_with\_many}$$

$$\frac{\mathit{field\_name} \notin \mathit{fn}_1 \ldots \mathit{fn}_m}{\begin{array}{l}\vdash \{\{\mathit{fn}_1 = v_1;\ \ldots;\mathit{fn}_m = v_m;\mathit{field\_name} = v;\mathit{fn}'_1 = v'_1;\ \ldots;\mathit{fn}'_n = v'_n\}\ \mathbf{with}\ \mathit{field\_name} = v'\} \longrightarrow \\ \qquad \{\mathit{fn}_1 = v_1;\ \ldots;\mathit{fn}_m = v_m;\mathit{field\_name} = v';\mathit{fn}'_1 = v'_1;\ \ldots;\mathit{fn}'_n = v'_n\}\end{array}} \quad \text{JR\_expr\_record\_with\_1}$$

$$\frac{\vdash e \xrightarrow{L} e'}{\vdash e.\mathit{field\_name} \xrightarrow{L} e'.\mathit{field\_name}} \quad \text{JR\_expr\_record\_access\_ctx}$$

$$\frac{}{\vdash ((\%\mathbf{primraise})\ v).\mathit{field\_name} \longrightarrow (\%\mathbf{primraise})\ v} \quad \text{JR\_expr\_record\_access\_raise}$$

$$\frac{\mathit{field\_name} \notin \mathit{fn}_1 \ldots \mathit{fn}_n}{\vdash \{\mathit{fn}_1 = v_1; \ldots; \mathit{fn}_n = v_n; \mathit{field\_name} = v; \mathit{fn}'_1 = v'_1; \ldots; \mathit{fn}'_m = v'_m\}.\mathit{field\_name} \longrightarrow v} \quad \text{JR\_expr\_record\_access}$$

$$\frac{\vdash e \xrightarrow{L} e'}{\vdash \mathbf{assert}\ e \xrightarrow{L} \mathbf{assert}\ e'} \quad \text{JR\_expr\_assert\_ctx}$$

$$\frac{}{\vdash \mathbf{assert}\ ((\%\mathbf{primraise})\ v) \longrightarrow (\%\mathbf{primraise})\ v} \quad \text{JR\_expr\_assert\_raise}$$

$$\frac{}{\vdash \mathbf{assert}\ \mathbf{true} \longrightarrow ()} \quad \text{JR\_expr\_assert\_true}$$

$$\frac{}{\vdash \mathbf{assert}\ \mathbf{false} \longrightarrow (\%\mathbf{primraise})\ \mathbf{Assert\_failure}} \quad \text{JR\_expr\_assert\_false}$$

4.10 $\boxed{\vdash \langle \mathit{definitions}, \mathit{program} \rangle \xrightarrow{L} \langle \mathit{definitions}', \mathit{program}' \rangle}$ Definition sequence evaluation

Reduces a definition one-step. Type and exception definitions are moved into the tuple left sequence as encountered to support typing of intermediate states.

$$\frac{\vdash e \xrightarrow{L} e'}{\vdash \langle \mathit{ds\_value}, \mathbf{let}\ \mathit{pat} = e;;\ \mathit{definitions} \rangle \xrightarrow{L} \langle \mathit{ds\_value}, \mathbf{let}\ \mathit{pat} = e';;\ \mathit{definitions} \rangle} \quad \text{Jdefn\_let\_ctx}$$

$$\frac{}{\vdash \langle \mathit{ds\_value}, \mathbf{let}\ \mathit{pat} = (\%\mathbf{primraise})\ v;;\ \mathit{definitions} \rangle \longrightarrow \langle \mathit{ds\_value}, (\%\mathit{primraise}) v \rangle} \quad \text{Jdefn\_let\_raise}$$

$$\frac{\vdash v\ \mathbf{matches}\ \mathit{pat}\ \triangleright\ \{\!\{x_1 \leftarrow v_1, \ldots, x_m \leftarrow v_m\}\!\}}{\vdash \langle \mathit{ds\_value}, \mathbf{let}\ \mathit{pat} = v;;\ \mathit{definitions} \rangle \longrightarrow \langle \mathit{ds\_value}, \{\!\{x_1 \leftarrow \mathbf{remv\_tyvar}\ v_1, \ldots, x_m \leftarrow \mathbf{remv\_tyvar}\ v_m\}\!\}\ \mathit{definitions} \rangle} \quad \text{Jdefn\_let\_match}$$

$$\frac{\neg(v\ \mathbf{matches}\ \mathit{pat})}{\vdash \langle \mathit{ds\_value}, \mathbf{let}\ \mathit{pat} = v;;\ \mathit{definitions} \rangle \longrightarrow \langle \mathit{ds\_value}, (\%\mathit{primraise}) \mathbf{Match\_failure} \rangle} \quad \text{Jdefn\_let\_not\_match}$$

$$\frac{\begin{array}{l} \mathit{letrec\_bindings} = (x_1 = \mathbf{function}\ \mathit{pattern\_matching}_1\ \mathbf{and} \ldots \mathbf{and}\ x_n = \mathbf{function}\ \mathit{pattern\_matching}_n) \\ \mathbf{recfun}\,(\mathit{letrec\_bindings}, \mathit{pattern\_matching}_1)\ \triangleright\ e_1 \quad \ldots \quad \mathbf{recfun}\,(\mathit{letrec\_bindings}, \mathit{pattern\_matching}_n)\ \triangleright\ e_n \end{array}}{\vdash \langle \mathit{ds\_value}, \mathbf{let\ rec}\ \mathit{letrec\_bindings};;\ \mathit{definitions} \rangle \longrightarrow \langle \mathit{ds\_value}, \{\!\{x_1 \leftarrow \mathbf{remv\_tyvar}\ e_1, \ldots, x_n \leftarrow \mathbf{remv\_tyvar}\ e_n\}\!\}\ \mathit{definitions} \rangle} \quad \text{Jdefn\_letrec}$$

$$\frac{}{\vdash \langle \mathit{ds\_value}, \mathit{type\_definition};;\ \mathit{definitions} \rangle \longrightarrow \langle \mathit{ds\_value};;\ \mathit{type\_definition}, \mathit{definitions} \rangle} \quad \text{Jdefn\_type}$$

$$\frac{}{\vdash \langle \mathit{ds\_value}, \mathit{exception\_definition};;\ \mathit{definitions} \rangle \longrightarrow \langle \mathit{ds\_value};;\ \mathit{exception\_definition}, \mathit{definitions} \rangle} \quad \text{Jdefn\_exn}$$

4.11 $\boxed{\mathit{store}(\mathit{location}) \quad \triangleright \quad \mathit{expr}}$ Store lookup

Gets the value stored at a given location.

$$\frac{\begin{array}{c} st(l) \quad \triangleright \quad e' \\ l \neq l' \end{array}}{st, l' \mapsto e(l) \quad \triangleright \quad e'} \quad \textsf{JSstlookup\_rec}$$

$$\frac{}{st, l \mapsto e(l) \quad \triangleright \quad e} \quad \textsf{JSstlookup\_found}$$

4.12 $\boxed{\vdash \mathit{store} \xrightarrow{L} \mathit{store}'}$ Store transition

Coordinates a store with a label.

$$\frac{}{\vdash st \longrightarrow st} \quad \textsf{JRstore\_empty}$$

$$\frac{st(l) \quad \triangleright \quad v}{\vdash st \xrightarrow{!l=v} st} \quad \textsf{JRstore\_lookup}$$

$$\frac{st'(l)\ \textbf{unallocated}}{\vdash st, l \mapsto \mathit{expr}, st' \xrightarrow{l:=v} st, l \mapsto \textbf{remv\_tyvar}\ v, st'} \quad \textsf{JRstore\_assign}$$

$$\frac{st(l)\ \textbf{unallocated}}{\vdash st \xrightarrow{\textbf{ref}\ v=l} st, l \mapsto \textbf{remv\_tyvar}\ v} \quad \textsf{JRstore\_alloc}$$

4.13 $\boxed{\vdash \langle \mathit{definitions}, \mathit{program}, \mathit{store} \rangle \longrightarrow \langle \mathit{definitions}', \mathit{program}', \mathit{store}' \rangle}$ Top-level reduction

The semantics of a machine is described as the parallel evolution of a structure body (the program) and a store. Each program evaluation step labelled L must be matched by a store evaluation step with the same label.

$$\frac{\begin{array}{l} \vdash \mathit{store} \xrightarrow{L} \mathit{store}' \\ \vdash \langle \mathit{definitions\_value}, \mathit{program} \rangle \xrightarrow{L} \langle \mathit{definitions}, \mathit{program}' \rangle \end{array}}{\vdash \langle \mathit{definitions\_value}, \mathit{program}, \mathit{store} \rangle \longrightarrow \langle \mathit{definitions}, \mathit{program}', \mathit{store}' \rangle} \quad \textsf{JRtop\_defs}$$

4.14 $\boxed{\vdash \mathit{expr}\ \textbf{behaves}}$ Expression behaviour

This relation describes expressions whose behaviour is defined. This includes values, expressions that reduce, and raised exceptions. An expression with no behaviour is said to be stuck. In this definition of expression behaviour, we treat any reducing expression as behaving, no matter what (satisfiable) constraint is imposed on the label.

$$\frac{}{\vdash v\ \textbf{behaves}} \quad \text{JRB\_ebehaviour\_value}$$

$$\frac{\vdash e \xrightarrow{L} e'}{\vdash e\ \textbf{behaves}} \quad \text{JRB\_ebehaviour\_reduces}$$

$$\frac{}{\vdash (\%\textbf{primraise})\ v\ \textbf{behaves}} \quad \text{JRB\_ebehaviour\_raises}$$

4.15 $\boxed{\vdash \langle \mathit{definitions}, \mathit{program}, \mathit{store} \rangle \textbf{behaves}}$ structure body behaviour

As for expressions, a definition sequence behaves if it is a value, if it reduces (under any label), or if it raises an exception.

$$\frac{}{\vdash \langle \mathit{definitions\_value}, , \mathit{store} \rangle \textbf{behaves}} \quad \text{JRB\_behaviour\_value}$$

$$\frac{\vdash \langle \mathit{definitions\_value}, \mathit{program}, \mathit{store} \rangle \longrightarrow \langle \mathit{definitions}', \mathit{program}', \mathit{store}' \rangle}{\vdash \langle \mathit{definitions\_value}, \mathit{program}, \mathit{store} \rangle \textbf{behaves}} \quad \text{JRB\_behaviour\_reduces}$$

$$\frac{}{\vdash \langle \mathit{definitions\_value}, (\%\mathit{primraise}) v, \mathit{store} \rangle \textbf{behaves}} \quad \text{JRB\_behaviour\_raises}$$

5 Statistics

```
Definition rules:        310 good    0 bad
Definition rule clauses: 696 good    0 bad
```