A formal specification for OCaml: the Core Language

Scott Owens and Gilles Peskine and Peter Sewell

February 13, 2008

Contents

1 Introduction **4**

2 Syntax **6**

3 Type system **31**

3.1 $\boxed{\mathbf{dom}\,(\,EB\,)\ \triangleright\ name}$ Environment binding domain 31

3.2 $\boxed{\mathbf{dom}\,(\,E\,)\ \triangleright\ names}$ Environment domain 32

3.3 $\boxed{E\ \vdash\ name\ \ \triangleright\ \ EB}$ Environment lookup 32

3.4 $\boxed{E\ \vdash\ idx\ \mathbf{bound}}$ Well-formed index 33

3.5 $\boxed{\vdash\ type\_params\_opt\ :\ kind}$ Type parameter kinding 33

3.6 $\boxed{E\ \vdash\ \mathbf{ok}}$ Environment validity 33

3.7 $\boxed{E\ \vdash\ typeconstr\ :\ kind}$ Type constructor kinding 35

3.8 $\boxed{E\ \vdash\ typescheme\ :\ kind}$ de Bruijn type scheme well-formedness 36

3.9 $\boxed{E\ \vdash\ \forall\, type\_params\_opt\,,\ t\ :\ kind}$ Named type scheme well-formedness 37

3.10 $\boxed{E\ \vdash\ typexpr\ :\ kind}$ Type expression well-formedness 37

3.11 $\boxed{E\ \vdash\ typexpr\ \equiv\ typexpr'}$ Type equivalence 37

3.12 $\boxed{\{\!\{\, typexpr_1\,,\,..\,,\ typexpr_n\,\}\!\}\ typexpr'\ \ \triangleright\ \ typexpr''}$ de Bruin type substitution 38

3.13 $\boxed{E\ \vdash\ typexpr\ \leq\ typescheme}$ de Bruijn type scheme instantiation 39

3.14 $\boxed{E\ \vdash\ typexpr\ \leq\ \forall\, type\_params\_opt\,,\ typexpr'}$ Named type scheme instantiation 39

3.15 $\boxed{E\ \vdash\ typexpr\ \leq\ typexpr'}$ Wildcard type instantiation 40

3.16 $\boxed{E\ \vdash\ value\_name\ :\ typexpr}$ Variable typing 40

3.17 $\boxed{E\ \vdash\ field\_name\ :\ typexpr\ \rightarrow\ typexpr'}$ Field name typing 40

3.18 $\boxed{E\ \vdash\ constr\ :\ typexpr_1\ ...\ typexpr_n\ \rightarrow\ typexpr'}$ Non-constant constructor typing 41

3.19 $\boxed{E\ \vdash\ constr\ :\ typexpr}$ Constant constructor typing 41

3.20 $\boxed{E\ \vdash\ constant\ :\ typexpr}$ Constant typing 42

3.21 $\boxed{\sigma^T\ \&\ E\ \vdash\ pattern\ :\ typexpr\ \ \triangleright\ \ E'}$ Pattern typing and binding collection 43

3.22 $\boxed{E\ \vdash\ unary\_prim\ :\ typexpr}$ Unary primitive typing 44

3.23 $\boxed{E \vdash \mathit{binary\_prim} : \mathit{typexpr}}$ Binary primitive typing 45
3.24 $\boxed{\sigma^T \,\&\, E \vdash \mathit{expr} : \mathit{typexpr}}$ Expression typing 45
3.25 $\boxed{\sigma^T \,\&\, E \vdash \mathit{pattern\_matching} : \mathit{typexpr} \to \mathit{typexpr}'}$ Pattern matching/expression pair typing 49
3.26 $\boxed{\sigma^T \,\&\, E \vdash \mathit{let\_binding} \ \rhd\ E'}$ Let binding typing 49
3.27 $\boxed{\sigma^T \,\&\, E \vdash \mathit{letrec\_bindings} \ \rhd\ E'}$ Recursive let binding typing 49
3.28 $\boxed{\mathit{type\_params\_opt}\ \mathit{typeconstr} \vdash \mathit{constr\_decl} \ \rhd\ \mathit{EB}}$ Variant constructor declaration 49
3.29 $\boxed{\mathit{type\_params\_opt}\ \mathit{typeconstr\_name} \vdash \mathit{field\_decl} \ \rhd\ \mathit{EB}}$ Record field declaration 50
3.30 $\boxed{\vdash \mathit{typedef}_1 \ \mathbf{and} .. \mathbf{and}\ \mathit{typedef}_n \ \rhd\ E' \ \mathbf{and}\ E'' \ \mathbf{and}\ E'''}$ Type definitions collection 50
3.31 $\boxed{E \vdash \mathit{type\_definition} \ \rhd\ E'}$ Type definition well-formedness and binding collection 51
3.32 $\boxed{E \vdash \mathit{definition} : E'}$ Definition typing 52
3.33 $\boxed{E \vdash \mathit{definitions} : E'}$ Definition sequence typing 52
3.34 $\boxed{E \vdash \mathit{program} : E'}$ Program typing 53
3.35 $\boxed{E \vdash \mathit{store} : E'}$ Store typing 53
3.36 $\boxed{E \vdash \langle \mathit{program}, \mathit{store} \rangle}$ Top-level typing 53
3.37 $\boxed{\sigma^T \,\&\, E \vdash L}$ Label-to-environment extraction 53
3.38 $\boxed{\sigma^T \,\&\, E \vdash L \ \rhd\ E'}$ Label-to-environment extraction 54

4 Operational Semantics **54**
4.1 $\boxed{\vdash \mathit{expr}\ \mathbf{matches}\ \mathit{pattern}}$ Pattern matching 55
4.2 $\boxed{\vdash \mathit{expr}\ \mathbf{matches}\ \mathit{pattern} \ \rhd\ \{\!\{ \mathit{substs\_x} \}\!\}}$ Pattern matching with substitution creation 56
4.3 $\boxed{\mathbf{recfun}\,(\,\mathit{letrec\_bindings}\,,\,\mathit{pattern\_matching}\,) \ \rhd\ \mathit{expr}}$ Recursive function helper 57
4.4 $\boxed{\vdash \mathbf{funval}\,(\,e\,)}$ Function values 57
4.5 $\boxed{\vdash \mathit{unary\_prim}\ \mathit{expr} \xrightarrow{L} \mathit{expr}'}$ Unary primitive evaluation 57
4.6 $\boxed{\vdash \mathit{expr}_1\ \mathit{binary\_prim}\ \mathit{expr}_2 \xrightarrow{L} \mathit{expr}}$ Binary primitive evaluation 58
4.7 $\boxed{\vdash \mathit{expr}\ \mathbf{with}\ \mathit{pattern\_matching} \longrightarrow \mathit{pattern\_matching}'}$ Pattern matching step 59
4.8 $\boxed{\vdash \mathit{expr}\ \mathbf{with}\ \mathit{pattern\_matching} \longrightarrow \mathit{expr}'}$ Pattern matching finished 59

4.9 $\boxed{\vdash \mathit{expr} \xrightarrow{L} \mathit{expr}'}$ Expression evaluation . . . 60
4.10 $\boxed{\vdash \langle \mathit{definitions}, \mathit{program} \rangle \xrightarrow{L} \langle \mathit{definitions}', \mathit{program}' \rangle}$ Definition sequence evaluation . . . 64
4.11 $\boxed{\mathit{store}\,(\,\mathit{location}\,) \ \triangleright \ \mathit{expr}}$ Store lookup . . . 65
4.12 $\boxed{\vdash \mathit{store} \xrightarrow{L} \mathit{store}'}$ Store transition . . . 65
4.13 $\boxed{\vdash \langle \mathit{definitions}, \mathit{program}, \mathit{store} \rangle \longrightarrow \langle \mathit{definitions}', \mathit{program}', \mathit{store}' \rangle}$ Top-level reduction . . . 66
4.14 $\boxed{\vdash \mathit{expr}\, \textbf{behaves}}$ Expression behaviour . . . 66
4.15 $\boxed{\vdash \langle \mathit{definitions}, \mathit{program}, \mathit{store} \rangle \textbf{behaves}}$ structure body behaviour . . . 66

5 Statistics **67**

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`),
  - generative variant 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 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, but instead 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.
- "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).
- [S] indicates that the production (which must be a metaproduction) is implemented as syntactic sugar.
- d indicates a definition-level feature, if enabled.

$index$, 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	
	|	$(\ typexpr_1\ ,\ ...\ ,\ typexpr_n\ )\ typeconstr$		
	|	**shift** $num\ num'\ typexpr$	M	
		shifts as in **Tsigma** above		
	|	$t_1\ \rightarrow\ ...\ \rightarrow\ t_n\ \rightarrow\ t$	M	[I]
	|	$\sigma^T\ typexpr$	M	[I]
		apply the substitution		

src_typexpr, *src_t*	::=			types that can appear in source programs
	|	α		
	|	_		
	|	(*src_typexpr*)		
	|	$src\_typexpr_1 \ \rightarrow \ src\_typexpr_2$		
	|	$src\_typexpr_1 \ * \ .... \ * \ src\_typexpr_n$		
	|	*typeconstr*		
	|	*src_typexpr typeconstr*		
	|	$(\ src\_typexpr_1 \ , \ ... \ , \ src\_typexpr_n \ ) \ typeconstr$		
	|	**shift** *num num′ src_typexpr*		
$\alpha, \ \alpha$	::=			
	|	*′ident*		
typescheme, *ts*	::=			
	|	$\forall$ *typexpr*		[I]
	|	**shift** *num num′ typescheme*	M	[I]
		shifts as in **Tsigma** 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*) $\cup$ *value_name*	
	|	(*pattern*)	S	
	|	(*pattern* : *typexpr*)	xs = xs(*pattern*)	
	|	$pattern_1$ | $pattern_2$	xs = xs($pattern_1$)	
	|	*constr* ($pattern_1$, ... , $pattern_n$)	xs = xs($pattern_1$...$pattern_n$)	
	|	*constr* _	xs = {}	
	|	$pattern_1$, ... , $pattern_n$	xs = xs($pattern_1$....$pattern_n$)	
	|	{ $field_1$ = $pattern_1$; ... ; $field_n$ = $pattern_n$ }	xs = xs($pattern_1$...$pattern_n$)	[d]
	|	[$pattern_1$; ... ; $pattern_n$]	S	[L]
	|	$pattern_1$ :: $pattern_2$	xs = xs($pattern_1$) $\cup$ xs($pattern_2$)	[L]
unary_prim	::=			primitive functions with one argument
	|	**raise**		[L I]
	|	**not**		[L I]
	|	$\sim-$		[L I]
	|	**ref**		[L I]
	|	!		[L I]
binary_prim	::=			primitive functions with two arguments
	|	=		[L I]
	|	+		[L I]
	|	$-$		[L I]

	$\vert$	$*$		[L I]
	$\vert$	$/$		[L I]
	$\vert$	$:=$		[L I]
$expr,\ e$	$::=$			
	$\vert$	$(\ \%\mathbf{prim}\ unary\_prim\ )$		[L I]
		a unary primitive function value		
	$\vert$	$(\ \%\mathbf{prim}\ binary\_prim\ )$		[L I]
		a binary primitive function value		
	$\vert$	$value\_name$		
	$\vert$	$constant$		
	$\vert$	$(\ expr\ )$	S	
	$\vert$	$\mathbf{begin}\ expr\ \mathbf{end}$	S	
	$\vert$	$(\ expr\ :\ typexpr\ )$		
	$\vert$	$expr_1\ ,\ ....\ ,\ expr_n$		
	$\vert$	$constr\ (\ expr_1\ ,\ ..\ ,\ expr_n\ )$		
		potentially empty constructors to work around ott parser restriction		
	$\vert$	$expr_1 :: expr_2$		[L]
	$\vert$	$[\ expr_1\ ;\ ...\ ;\ expr_n\ ]$	S	[L]
	$\vert$	$\{\ field_1\ =\ expr_1\ ;\ ...\ ;\ field_n\ =\ expr_n\ \}$		[d]
	$\vert$	$\{\ expr\ \mathbf{with}\ field_1\ =\ expr_1\ ;\ ...\ ;\ field_n\ =\ expr_n\ \}$		[d]
	$\vert$	$expr_1\ expr_2$		
	$\vert$	$prefix\_symbol\ expr$	S	
	$\vert$	$expr_1\ infix\_op\ expr_2$	S	
	$\vert$	$expr_1\ \&\&\ expr_2$		[L]
	$\vert$	$\mathbf{AND}\ (\ expr_1\ \&\&\ ..\ \&\&\ expr_n\ )$	M	[L I]
		a delimited “and” operator with a list of arguments		
	$\vert$	$expr_1\ \vert\vert\ expr_2$		[L]
	$\vert$	$expr\ .\ field$		[d]
	$\vert$	$\mathbf{if}\ expr_0\ \mathbf{then}\ expr_1$	S	
	$\vert$	$\mathbf{if}\ expr_0\ \mathbf{then}\ expr_1\ \mathbf{else}\ expr_2$		
	$\vert$	$\mathbf{while}\ expr_1\ \mathbf{do}\ expr_2\ \mathbf{done}$		
	$\vert$	$\mathbf{for}\ x\ =\ expr_1\ [\mathbf{down}]\mathbf{to}\ expr_2\ \mathbf{do}\ expr_3\ \mathbf{done}$	bind x in $expr_3$	

	|	$expr_1$; $expr_2$		
	|	**match** *expr* **with** *pattern_matching*		
	|	**function** *pattern_matching*		
	|	**fun** $pattern_1 \ldots pattern_n \ \rightarrow\ expr$	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
	|	$value\_name_1 \leftarrow expr_1$, .. , $value\_name_n \leftarrow expr_n$		[I]
	|	$substs\_x_1$ @ .. @ $substs\_x_n$	M	[I]
pattern_matching, *pm*	::=			
	|	$pat\_exp_1 \mid \ldots \mid pat\_exp_n$		
	|	$\mid pat\_exp_1 \mid \ldots \mid pat\_exp_n$	S	
pat_exp	::=			
	|	$pattern\ \rightarrow\ expr$	bind xs(*pattern*) in *expr*	
let_binding	::=			
	|	$pattern\ =\ expr$	xs = xs(*pattern*)	

	$\mid$	$value\_name\ pattern_1 \ldots pattern_n\ =\ expr$	S	
	$\mid$	$value\_name\ pattern_1 \ldots pattern_n\ :\ typexpr\ =\ expr$	S	
	$\mid$	$\{\!\{\ \alpha_1 \leftarrow typexpr_1\ ,\ ..\ ,\ \alpha_n \leftarrow typexpr_n\ \}\!\}\ let\_binding$	M	
		substitution of types for type variables		
letrec_bindings	::=			
	$\mid$	$letrec\_binding_1$ **and** ... **and** $letrec\_binding_n$	$\mathrm{xs} = \mathrm{xs}(letrec\_binding_1 \ldots letrec\_binding_n)$	
	$\mid$	$\{\!\{\ \alpha_1 \leftarrow typexpr_1\ ,\ ..\ ,\ \alpha_n \leftarrow typexpr_n\ \}\!\}\ letrec\_bindings$	M	
		substitution of types for type variables		
letrec_binding	::=			
	$\mid$	$value\_name\ =$ **function** $pattern\_matching$	$\mathrm{xs} = value\_name$	
	$\mid$	$value\_name\ =$ **fun** $pattern\ pattern_1\ ..\ pattern_n\ \rightarrow\ expr$	S	
	$\mid$	$value\_name\ pattern\ pattern_1\ ..\ pattern_n\ =\ expr$	S	
	$\mid$	$value\_name\ pattern\ pattern_1\ ..\ pattern_n\ :\ typexpr\ =\ expr$	S	
type_definition	::=			
	$\mid$	**type** $typedef_1$ **and** .. **and** $typedef_n$	$\mathrm{type\_names} = \mathrm{type\_names}(typedef_1 .. typedef_n)$	[d]
		potentially empty definitions to work around Ott parser restrictions		
			$\mathrm{constr\_names} = \mathrm{constr\_names}(typedef_1 .. typedef_n)$	
typedef	::=			
	$\mid$	$type\_params\_opt\ typeconstr\_name\ type\_information$	bind $\mathrm{typevars}(type\_params\_opt)$ in $type\_information$	[d]
			$\mathrm{type\_names} = typeconstr\_name$	
			$\mathrm{constr\_names} = \mathrm{constr\_names}(type\_information)$	
type_information	::=			
	$\mid$	$type\_equation$	$\mathrm{constr\_names} = \{\}$	[d]
			$\mathrm{field\_names} = \{\}$	
	$\mid$	$type\_representation$	$\mathrm{constr\_names} = \mathrm{constr\_names}(type\_representation)$	[d]
			$\mathrm{field\_names} = \mathrm{field\_names}(type\_representation)$	
type_equation	::=			

	|	$=$ *typexpr*		[d]
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]
	|	(%**prim raise**) *expr*		[d]
value, *v*	::=			core value
	|	(%**prim** *unary_prim*)		[L I]
	|	(%**prim** *binary_prim*)		[L I]
	|	*binary_prim_app_value value*		[I]
		partially applied binary primitive		
	|	*constant*		[I]
	|	(*value*)		[I]
	|	$value_1$, ... , $value_n$		[I]
	|	*constr* ($value_1$, .. , $value_n$)		[I]
	|	$value_1$ :: $value_2$		[L I]
	|	[$value_1$; ... ; $value_n$]		[L I]
	|	{ $field_1$ = $value_1$; ... ; $field_n$ = $value_n$ }		[d I]
	|	**function** *pattern_matching*		[I]
	|	**fun** $pattern_1$ ... $pattern_n$ $\rightarrow$ *expr*		[I]
	|	*location*		[I]
binary_prim_app_value	::=			

	|	(%**prim** *binary_prim*)	[I]
definition_value, *d_value*	::=		
	|	*type_definition*	[d I]
	|	*exception_definition*	[d I]
definitions_value, *ds_value*	::=		
	|		[d I]
	|	*definition_value definitions_value*	[d I]
	|	*definition_value* ; ; *definitions_value*	[d I]
non_expansive, *nexp*	::=		nonexpansive expression (allowed in a polymorphic let)
	|	(%**prim** *unary_prim*)	[I]
	|	(%**prim** *binary_prim*)	[I]
	|	*binary_prim_app_value nexp* partially applied binary primitive	[I]
	|	*value_name*	[I]
	|	*constant*	[I]
	|	(*nexp*)	[I]
	|	(*nexp* : *typexpr*)	[I]
	|	$nexp_1$, ... , $nexp_n$	[I]
	|	*constr* ($nexp_1$, .. , $nexp_n$)	[I]
	|	$nexp_1$ :: $nexp_2$	[I]
	|	[$nexp_1$; ... ; $nexp_n$]	[L I]
	|	{ $field_1$ = $nexp_1$; ... ; $field_n$ = $nexp_n$ }	[d I]
	|	**let rec** *letrec_bindings* **in** *nexp*	[I]
	|	**function** *pattern_matching*	[I]
	|	**fun** $pattern_1$ ... $pattern_n$ → *expr*	[I]
	|	*location*	[I]
store, *st*	::=		
	|	**empty**	[I]
	|	*store* , *location* ↦ *expr*	[I]

	$\vert$	$store\ ,\ location\ \mapsto\ expr\ ,\ store'$	M	[I]
kind	::=			
	$\vert$	$\mathbf{Type}^{num} \rightarrow \mathbf{Type}$		[I]
	$\vert$	$\mathbf{Type}$	S	[I]
name	::=			environment lookup key
	$\vert$	$\mathbf{TV}$		[I]
	$\vert$	*value_name*		[I]
	$\vert$	*constr_name*		[d I]
	$\vert$	*typeconstr_name*		[d I]
	$\vert$	*field_name*		[d I]
	$\vert$	*location*		[I]
names	::=			
	$\vert$	$name_1\ ..\ name_n$		[I]
typexprs	::=			
	$\vert$	$typexpr_1\ ,\ \ldots\ ,\ typexpr_n$		[I]
	$\vert$	**shift** $num\ num'\ typexprs$	M	[I]
		shift the indices in the types in *typexprs* by *num*, ignoring indices lower than num'		
environment_binding, *EB*	::=			
	$\vert$	$\mathbf{TV}$		[I]
		type variable		
	$\vert$	$value\_name\ :\ typescheme$		[I]
		value binding		
	$\vert$	$value\_name\ :\ typexpr$	M	[I]
		value binding with no universal quantifier		
	$\vert$	*constr_name* **of** *typeconstr*		[d I]
		constant constructor		
	$\vert$	*constr_name* **of** $\forall$ *type_params_opt* , (*typexprs*) : *typeconstr*	bind typevars(*type_params_opt*) in *typexprs*	[d I]
		parameterised constructor		

	$\vert$	$field\_name : \forall\, type\_params\_opt\, ,\ typeconstr\_name \rightarrow typexpr$	bind typevars($type\_params\_opt$) in $typexpr$	[d I]
		field name a record destructor		
	$\vert$	$typeconstr\_name : kind$		[d I]
		type name, bound to a fresh type		
	$\vert$	$typeconstr\_name : kind\,\{\, field\_name_1 \,;\, \ldots \,;\, field\_name_n \,\}$		[d I]
		type name which is a record type definition		
	$\vert$	$type\_params\_opt\ typeconstr\_name = typexpr$	bind typevars($type\_params\_opt$) in $typexpr$	[d I]
		type name which is an abbreviation		
	$\vert$	$location : typexpr$		[I]
		location (memory cell)		
	$\vert$	$(\, EB\,)$	M	[I]
	$\vert$	**shift** $num\ num'\ EB$	M	[I]
		shift the indices in the types in EB by num, ignoring indices lower than num'		
$environment,\ E$	::=			
	$\vert$	**empty**		[I]
	$\vert$	E, EB		[I]
	$\vert$	$EB_1\, ,\, ..\, ,\, EB_n$	M	[I]
	$\vert$	$E_1\ @\ ..\ @\ E_n$	M	[I]
$trans\_label,\ L$	::=			reduction label (denoting a side effect)
	$\vert$			[I]
	$\vert$	**ref** $v = location$		[I]
	$\vert$	$!\, location = v$		[I]
	$\vert$	$location := v$		[I]
	$\vert$	L	M	[I]
$\xrightarrow{L}$	::=			
	$\vert$	$\xrightarrow{L}$		[I]
$formula$	::=			semantic judgements and their side condi
	$\vert$	$judgement$		
	$\vert$	$formula_1\quad ..\quad formula_n$		

	$\mid$	$\dot{n}_1 \dot{\leq} \dot{n}_2$	
	$\mid$	$\dot{n}_1 \dot{>} \dot{n}_2$	
	$\mid$	$num_1 < num_2$	
	$\mid$	$E = E'$	
	$\mid$	$expr = expr'$	
	$\mid$	$typexpr = typexpr'$	
	$\mid$	$typescheme = typescheme'$	
	$\mid$	$type\_params\_opt = type\_params\_opt'$	
	$\mid$	$letrec\_bindings = (letrec\_bindings')$	
	$\mid$	$\mathbf{length}\,(\,tp_1\,) \mathinner{..} (\,tp_n\,) = m$	
	$\mid$	$\mathbf{length}\,(\,t_1\,) \mathinner{..} (\,t_n\,) = num$	
	$\mid$	$\mathbf{length}\,(\,t_1\,) \mathinner{..} (\,t_n\,) \leq num$	
	$\mid$	$\mathbf{length}\,(\,t_1\,) \mathinner{..} (\,t_n\,) \geq num$	
	$\mid$	$\mathbf{length}\,(\,pat_1\,) \mathinner{..} (\,pat_n\,) \geq m$	
	$\mid$	$\mathbf{length}\,(\,e_1\,) \mathinner{..} (\,e_n\,) \geq m$	
	$\mid$	$name \notin names$	
	$\mid$	$field\_name\ \mathbf{in}\ field\_name_1 \mathinner{..} field\_name_n$	[d]
	$\mid$	$type\_param\ \mathbf{in}\ type\_params\_opt$	
	$\mid$	$name_1 \mathinner{..} name_n\ \mathbf{distinct}$	
	$\mid$	$tp_1 \mathinner{..} tp_n\ \mathbf{distinct}$	
	$\mid$	$E\ \mathbf{PERMUTES}\ E'$	
	$\mid$	$fn_1 \mathinner{..} fn_n\ \mathbf{PERMUTES}\ fn'_1 \mathinner{..} fn'_m$	[d]
	$\mid$	$fn_1 = e_1 \mathinner{..} fn_n = e_n\ \mathbf{PERMUTES}\ fn'_1 = e'_1 \mathinner{..} fn'_m = e'_m$	[d]
	$\mid$	$\neg(value\ \mathbf{matches}\ pattern)$	
	$\mid$	$constant \neq constant'$	
	$\mid$	$name \neq name'$	
	$\mid$	$store\,(\,location\,)\ \mathbf{unallocated}$	
	$\mid$	$\mathbf{type\_vars}\,(\,let\_binding\,) \rhd \alpha_1\,,\,..\,,\,\alpha_n$	
	$\mid$	$\mathbf{type\_vars}\,(\,letrec\_bindings\,) \rhd \alpha_1\,,\,..\,,\,\alpha_n$	
$terminals$	::=		prettyprinting specifications
	$\mid$	$\%\mathbf{prim}$	
	$\mid$	$*$	

| | $\to$
| | $\longrightarrow$
| | $\to$
| | $\leftarrow$
| | $\{\!\{$
| | $\leq$
| | $\equiv$
| | $\geq$
| | $\}\!\}$
| | $\forall$
| | $\triangleright$
| | $\notin$
| | $\neq$
| | $\vdash$
| | $\mapsto$
| | $\sim-$

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

$JTEok$ ::=
| $E \vdash$ **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

$JTidxsub$ ::=
| $\{\!\{ typexpr_1 , .. , typexpr_n \}\!\}\, typexpr' \ \triangleright\ typexpr''$
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 ::=
| | $E \vdash field\_name : typexpr \rightarrow typexpr'$
| | Field name typing

JTconstr_p ::=
| | $E \vdash constr : typexpr_1 \ldots typexpr_n \rightarrow typexpr'$
| | Non-constant constructor typing

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

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

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

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

JTbprim ::=
| | $E \vdash binary\_prim : typexpr$

Binary primitive typing

JTe	::=	
	|	$\sigma^T \mathbin{\&} E \vdash \mathit{expr} : \mathit{typexpr}$ Expression typing
	|	$\sigma^T \mathbin{\&} E \vdash \mathit{pattern\_matching} : \mathit{typexpr} \rightarrow \mathit{typexpr}'$ Pattern matching/expression pair typing
	|	$\sigma^T \mathbin{\&} E \vdash \mathit{let\_binding} \ \triangleright\ E'$ Let binding typing
	|	$\sigma^T \mathbin{\&} E \vdash \mathit{letrec\_bindings} \ \triangleright\ E'$ Recursive let binding typing
JTconstr_decl	::=	
	|	$\mathit{type\_params\_opt}\ \mathit{typeconstr} \vdash \mathit{constr\_decl} \ \triangleright\ \mathit{EB}$ Variant constructor declaration
JTfield_decl	::=	
	|	$\mathit{type\_params\_opt}\ \mathit{typeconstr\_name} \vdash \mathit{field\_decl} \ \triangleright\ \mathit{EB}$ Record field declaration
JTtypedef	::=	
	|	$\vdash \mathit{typedef}_1 \ \mathbf{and} \ldots \mathbf{and}\ \mathit{typedef}_n \ \triangleright\ E' \ \mathbf{and}\ E'' \ \mathbf{and}\ E'''$ 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	::=	

| | $E \vdash$ *definitions* $: E'$
Definition sequence typing

JTprog ::=
| $E \vdash$ *program* $: E'$
Program typing

JTstore ::=
| $E \vdash$ *store* $: E'$
Store typing

JTtop ::=
| $E \vdash \langle program, store \rangle$
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 ::=
| $\vdash$ *expr* **matches** *pattern*
Pattern matching

Jmatch ::=
| $\vdash$ *expr* **matches** *pattern* $\triangleright$ $\{\!\{$ *substs_x* $\}\!\}$
Pattern matching with substitution creation

Jrecfun ::=
| **recfun** (*letrec_bindings* , *pattern_matching*) $\triangleright$ *expr*

Recursive function helper

Jfunval	::=	
	|	$\vdash \textbf{funval}\,(\,e\,)$ Function values
JRuprim	::=	
	|	$\vdash \mathit{unary\_prim}\ \mathit{expr} \xrightarrow{L} \mathit{expr}'$ Unary primitive evaluation
JRbprim	::=	
	|	$\vdash \mathit{expr}_1\ \mathit{binary\_prim}\ \mathit{expr}_2 \xrightarrow{L} \mathit{expr}$ Binary primitive evaluation
JRmatching_step	::=	
	|	$\vdash \mathit{expr}\ \textbf{with}\ \mathit{pattern\_matching} \longrightarrow \mathit{pattern\_matching}'$ Pattern matching step
JRmatching_success	::=	
	|	$\vdash \mathit{expr}\ \textbf{with}\ \mathit{pattern\_matching} \longrightarrow \mathit{expr}'$ Pattern matching finished
Jred	::=	
	|	$\vdash \mathit{expr} \xrightarrow{L} \mathit{expr}'$ Expression evaluation
JRdefn	::=	
	|	$\vdash \langle \mathit{definitions}, \mathit{program}\rangle \xrightarrow{L} \langle \mathit{definitions}', \mathit{program}'\rangle$ Definition sequence evaluation
JSlookup	::=	
	|	$\mathit{store}\,(\,\mathit{location}\,) \ \triangleright\ \mathit{expr}$ Store lookup

JRstore	::=	
	|	$\vdash$ *store* $\xrightarrow{L}$ *store'* Store transition
JRtop	::=	
	|	$\vdash \langle \mathit{definitions}, \mathit{program}, \mathit{store} \rangle \longrightarrow \langle \mathit{definitions}', \mathit{program}', \mathit{store}' \rangle$ Top-level reduction
Jebehaviour	::=	
	|	$\vdash$ *expr* **behaves** Expression behaviour
Jdbehaviour	::=	
	|	$\vdash \langle \mathit{definitions}, \mathit{program}, \mathit{store} \rangle$**behaves** structure body behaviour
judgement	::=	
	|	*JdomEB*
	|	*JdomE*
	|	*Jlookup*
	|	*Jidx*
	|	*JTtps_kind*
	|	*JTEok*
	|	*JTeq*
	|	*JTidxsub*
	|	*JTinst*
	|	*JTinst_named*
	|	*JTinst_any*
	|	*JTval*
	|	*JTfield*
	|	*JTconstr_p*
	|	*JTconstr_c*
	|	*JTconst*

$$
\begin{array}{lcl}
 & | & \mathit{JTpat} \\
 & | & \mathit{JTuprim} \\
 & | & \mathit{JTbprim} \\
 & | & \mathit{JTe} \\
 & | & \mathit{JTconstr\_decl} \\
 & | & \mathit{JTfield\_decl} \\
 & | & \mathit{JTtypedef} \\
 & | & \mathit{JTtype\_definition} \\
 & | & \mathit{JTdefinition} \\
 & | & \mathit{JTdefinitions} \\
 & | & \mathit{JTprog} \\
 & | & \mathit{JTstore} \\
 & | & \mathit{JTtop} \\
 & | & \mathit{JTLin} \\
 & | & \mathit{JTLout} \\
 & | & \mathit{JmatchP} \\
 & | & \mathit{Jmatch} \\
 & | & \mathit{Jrecfun} \\
 & | & \mathit{Jfunval} \\
 & | & \mathit{JRuprim} \\
 & | & \mathit{JRbprim} \\
 & | & \mathit{JRmatching\_step} \\
 & | & \mathit{JRmatching\_success} \\
 & | & \mathit{Jred} \\
 & | & \mathit{JRdefn} \\
 & | & \mathit{JSlookup} \\
 & | & \mathit{JRstore} \\
 & | & \mathit{JRtop} \\
 & | & \mathit{Jebehaviour} \\
 & | & \mathit{Jdbehaviour} \\
 \\
\mathit{user\_syntax} & ::= & \\
 & | & \mathit{index}
\end{array}
$$

| *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**

$$
\begin{array}{ll}
| & \mathit{substs\_x} \\
| & \mathit{pattern\_matching} \\
| & \mathit{pat\_exp} \\
| & \mathit{let\_binding} \\
| & \mathit{letrec\_bindings} \\
| & \mathit{letrec\_binding} \\
| & \mathit{type\_definition} \\
| & \mathit{typedef} \\
| & \mathit{type\_information} \\
| & \mathit{type\_equation} \\
| & \mathit{type\_representation} \\
| & \mathit{type\_params\_opt} \\
| & \mathit{type\_param} \\
| & \mathit{constr\_decl} \\
| & \mathit{field\_decl} \\
| & \mathit{exception\_definition} \\
| & \mathit{definition} \\
| & \mathit{definitions} \\
| & \mathit{program} \\
| & \mathit{value} \\
| & \mathit{binary\_prim\_app\_value} \\
| & \mathit{definition\_value} \\
| & \mathit{definitions\_value} \\
| & \mathit{non\_expansive} \\
| & \mathit{store} \\
| & \mathit{kind} \\
| & \mathit{name} \\
| & \mathit{names} \\
| & \mathit{typexprs} \\
| & \mathit{environment\_binding} \\
| & \mathit{environment} \\
| & \mathit{trans\_label} \\
| & \xrightarrow{L}
\end{array}
$$

| *formula*
| *terminals*

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.
- We show how the system works with type abbreviations, but we do not use them in our theorem prover models because our soundness proof mechanization does not yet deal with them.

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

Gets the name of an environment entry.

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

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

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

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

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

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

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

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

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

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

Gets all of the names in an environment.

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

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

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

Returns the rightmost binding that matches the given name.

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

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

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

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.

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

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.

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{l}E \vdash \mathbf{ok} \\ \mathbf{dom}\,(\,E\,) \;\; \triangleright \;\; names \\ typeconstr\_name \notin names \\ field\_name_1\,...\,field\_name_n\;\mathbf{distinct}\end{array}}{E,(\,typeconstr\_name \;:\; kind\,\{\,field\_name_1\,;\,...\,;\,field\_name_n\,\}\,) \vdash \mathbf{ok}} \quad \textsf{JTEok\_typeconstr\_record}$$

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

3.7 $\boxed{E \vdash typeconstr \;:\; kind}$ 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.

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

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

$$\frac{\begin{array}{l}E \vdash \mathbf{ok} \\ E \vdash typeconstr\_name \;\; \triangleright \;\; typeconstr\_name \;:\; kind\,\{\,field\_name_1\,;\,...\,;\,field\_name_n\,\}\end{array}}{E \vdash typeconstr\_name \;:\; kind} \quad \textsf{JTtypeconstr\_record}$$

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

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

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

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

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

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

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

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

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

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

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.

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

3.9 $\boxed{E \vdash \forall\, type\_params\_opt\,,\ t\ :\ kind}$ 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.

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

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

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

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

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

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

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

3.11 $\boxed{E \vdash typexpr\ \equiv\ typexpr'}$ 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.

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

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

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

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

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

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

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

3.12 $\boxed{\{\!\{ \mathit{typexpr}_1, .., \mathit{typexpr}_n \}\!\}\, \mathit{typexpr}' \ \triangleright \ \mathit{typexpr}''}$ de Bruin type substitution

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

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

$$\frac{\mathbf{length}\,(t_1) .. (t_m) = \mathit{num}}{\{\!\{ t_1, .., t_m, t', t_1'', .., t_n'' \}\!\} < 0, \mathit{num} > \triangleright \ t'} \quad \textsf{JTinxsub\_idx0}$$

$$\frac{\mathbf{length}\,(t_1) .. (t_n) \leq \mathit{num}}{\{\!\{ t_1, .., t_n \}\!\} < 0, \mathit{num} > \triangleright \ \mathbf{unit}} \quad \textsf{JTinxsub\_idx1}$$

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

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

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

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

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

3.13 $\boxed{E \vdash \mathit{typexpr} \ \leq \ \mathit{typescheme}}$ 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 $\boxed{E \vdash \mathit{typexpr} \ \leq \ \forall \, \mathit{type\_params\_opt} \, , \, \mathit{typexpr}'}$ Named type scheme instantiation

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

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

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

Replaces _ type variables with arbitrary types.

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

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

$$\frac{\begin{array}{l} E \vdash t_1 \leq t'_1 \\ E \vdash t_2 \leq t'_2 \end{array}}{E \vdash t_1 \rightarrow t_2 \leq t'_1 \rightarrow t'_2} \quad \textsf{JTinst\_any\_arrow}$$

$$\frac{\begin{array}{l} E \vdash t_1 \leq t'_1 \quad .... \quad E \vdash t_n \leq t'_n \\ \mathbf{length}\,(t_1) .... (t_n) \geq 2 \end{array}}{E \vdash t_1 * .... * t_n \leq t'_1 * .... * t'_n} \quad \textsf{JTinst\_any\_tuple}$$

$$\frac{\begin{array}{l} E \vdash t_1 \leq t'_1 \quad ... \quad E \vdash t_n \leq t'_n \\ E \vdash typeconstr : \mathbf{Type}^n \rightarrow \mathbf{Type} \\ \mathbf{length}\,(t_1) ... (t_n) = n \end{array}}{E \vdash (t_1 , ... , t_n)\, typeconstr \leq (t'_1 , ... , t'_n)\, typeconstr} \quad \textsf{JTinst\_any\_ctor}$$

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 $\boxed{E \vdash field\_name : typexpr \rightarrow typexpr'}$ 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.

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

3.18 $\boxed{E \vdash constr : typexpr_1 \ldots typexpr_n \rightarrow typexpr'}$ 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Determines the type of a constant.

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

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

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

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

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

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

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

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

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

3.21 $\boxed{\sigma^T \,\&\, E \vdash \mathit{pattern} : \mathit{typexpr} \;\triangleright\; E'}$ 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.

$$\frac{E \vdash t : \textbf{Type}}{\sigma^T \,\&\, E \vdash x : t \;\triangleright\; x : t} \quad \text{JTpat\_var}$$

$$\frac{E \vdash t : \textbf{Type}}{\sigma^T \,\&\, E \vdash \_ : t \;\triangleright\; \textbf{empty}} \quad \text{JTpat\_any}$$

$$\frac{E \vdash \mathit{constant} : t}{\sigma^T \,\&\, E \vdash \mathit{constant} : t \;\triangleright\; \textbf{empty}} \quad \text{JTpat\_constant}$$

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

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

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

$$\frac{\begin{array}{l} E \vdash \mathit{constr} : t_1 \ldots t_n \to t \\ \sigma^T \,\&\, E \vdash \mathit{pattern}_1 : t_1 \;\triangleright\; E_1 \quad \ldots \quad \sigma^T \,\&\, E \vdash \mathit{pattern}_n : t_n \;\triangleright\; E_n \\ \textbf{dom}\,(E_1 \,@ \ldots @\, E_n) \;\triangleright\; \mathit{name}_1 \mathinner{..} \mathit{name}_m \\ \mathit{name}_1 \mathinner{..} \mathit{name}_m \;\textbf{distinct} \end{array}}{\sigma^T \,\&\, E \vdash \mathit{constr}\,(\mathit{pattern}_1 , \ldots , \mathit{pattern}_n) : t \;\triangleright\; E_1 \,@ \ldots @\, E_n} \quad \text{JTpat\_construct}$$

$$\frac{E \vdash \mathit{constr} : t_1 \ldots t_n \to t}{\sigma^T \,\&\, E \vdash \mathit{constr}\,\_ : t \;\triangleright\; \textbf{empty}} \quad \text{JTpat\_construct\_any}$$

$$\frac{\begin{array}{l} \sigma^T \,\&\, E \vdash \mathit{pattern}_1 : t_1 \;\triangleright\; E_1 \quad \ldots. \quad \sigma^T \,\&\, E \vdash \mathit{pattern}_n : t_n \;\triangleright\; E_n \\ \textbf{length}\,(\mathit{pattern}_1) \ldots. (\mathit{pattern}_n) \geq 2 \\ \textbf{dom}\,(E_1 \,@ \ldots. @\, E_n) \;\triangleright\; \mathit{name}_1 \mathinner{..} \mathit{name}_m \\ \mathit{name}_1 \mathinner{..} \mathit{name}_m \;\textbf{distinct} \end{array}}{\sigma^T \,\&\, E \vdash \mathit{pattern}_1 , \ldots. , \mathit{pattern}_n : t_1 * \ldots. * t_n \;\triangleright\; E_1 \,@ \ldots. @\, E_n} \quad \text{JTpat\_tuple}$$

$$\frac{\begin{array}{l}\sigma^T \,\&\, E \vdash \mathit{pattern}_1 : t_1 \;\triangleright\; E_1 \quad \ldots \quad \sigma^T \,\&\, E \vdash \mathit{pattern}_n : t_n \;\triangleright\; E_n \\ E \vdash \mathit{field\_name}_1 : t \to t_1 \quad \ldots \quad E \vdash \mathit{field\_name}_n : t \to t_n \\ \mathbf{length}\,(\,\mathit{pattern}_1\,) \ldots (\,\mathit{pattern}_n\,) \geq 1 \\ \mathbf{dom}\,(\,E_1 \,@ \ldots @\, E_n\,) \;\triangleright\; \mathit{name}_1 \,..\, \mathit{name}_m \\ \mathit{name}_1 \,..\, \mathit{name}_m \;\mathbf{distinct}\end{array}}{\sigma^T \,\&\, E \vdash \{\, \mathit{field\_name}_1 = \mathit{pattern}_1 \,;\, \ldots \,;\, \mathit{field\_name}_n = \mathit{pattern}_n \,\} : t \;\triangleright\; E_1 \,@ \ldots @\, E_n} \quad \mathsf{JTpat\_record}$$

$$\frac{\begin{array}{l}\sigma^T \,\&\, E \vdash \mathit{pattern} : t \;\triangleright\; E' \\ \sigma^T \,\&\, E \vdash \mathit{pattern}' : t\,\mathbf{list} \;\triangleright\; E'' \\ \mathbf{dom}\,(\,E'\,) \;\triangleright\; \mathit{name}_1 \,..\, \mathit{name}_m \\ \mathbf{dom}\,(\,E''\,) \;\triangleright\; \mathit{name}'_1 \,..\, \mathit{name}'_n \\ \mathit{name}_1 \,..\, \mathit{name}_m \; \mathit{name}'_1 \,..\, \mathit{name}'_n \;\mathbf{distinct}\end{array}}{\sigma^T \,\&\, E \vdash \mathit{pattern} :: \mathit{pattern}' : t\,\mathbf{list} \;\triangleright\; E' \,@\, E''} \quad \mathsf{JTpat\_cons}$$

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

Determines if a unary primitive has a given type.

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

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

$$\frac{E \vdash \mathbf{ok}}{E \vdash \sim- : \mathbf{int} \to \mathbf{int}} \quad \mathsf{JTuprim\_uminus}$$

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

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

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

Determines if a binary primitive has a given type.

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

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

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

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

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

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

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**.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{l}\sigma^T \,\&\, E \vdash e_1 : \mathbf{bool} \\ \sigma^T \,\&\, E \vdash e_2 : \mathbf{unit} \\ E \vdash \mathbf{unit} \equiv t\end{array}}{\sigma^T \,\&\, E \vdash \mathbf{while}\; e_1 \;\mathbf{do}\; e_2 \;\mathbf{done} : t} \quad \textsf{JTe\_while}$$

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

$$\frac{\begin{array}{l}\sigma^T \,\&\, E \vdash e_1 : \mathbf{unit} \\ \sigma^T \,\&\, E \vdash e_2 : t\end{array}}{\sigma^T \,\&\, E \vdash e_1 \,;\, e_2 : t} \quad \textsf{JTe\_sequence}$$

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.

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

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

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

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.

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

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

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

$$\frac{\sigma^T \,\&\, E \vdash e : \mathbf{bool} \qquad E \vdash \mathbf{unit} \equiv t}{\sigma^T \,\&\, E \vdash \mathbf{assert}\, e : t} \quad \textsf{JTe\_assert}$$

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

$$\frac{E \vdash \mathbf{ok} \qquad E \vdash \mathit{location} \ \triangleright\ \mathit{location} : t \qquad E \vdash t\, \mathbf{ref} \equiv t'}{\sigma^T \,\&\, E \vdash \mathit{location} : t'} \quad \textsf{JTe\_location}$$

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.

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

3.26 $\boxed{\sigma^T \,\&\, E \vdash \mathit{let\_binding} \;\triangleright\; E'}$ Let binding typing

Determines the types bound by a let bindings pattern.

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

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).

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

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.

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

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

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

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

$$\frac{}{(\,\alpha_1\,,\,...\,,\,\alpha_n\,)\,\mathit{typeconstr\_name} \vdash \mathit{fn}\,:\,t \;\;\triangleright\;\; \mathit{fn}\,:\,\forall\,(\,\alpha_1\,,\,...\,,\,\alpha_n\,)\,,\,\mathit{typeconstr\_name}\,\rightarrow\,t}\ \textsf{JTfield\_decl\_only}$$

3.30 $\boxed{\vdash\ \mathit{typedef}_1\ \mathbf{and}\,..\,\mathbf{and}\ \mathit{typedef}_n \;\;\triangleright\;\; E'\ \mathbf{and}\ E''\ \mathbf{and}\ E'''}$ 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.

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

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

$$\frac{\begin{array}{l}\vdash \overline{typedef_i}^{\,i} \;\triangleright\; E \;\mathbf{and}\; E' \;\mathbf{and}\; E''\\ \vdash type\_params\_opt \;:\; kind\\ type\_params\_opt\; typeconstr\_name \vdash constr\_decl_1 \;\triangleright\; EB_1 \quad ... \quad type\_params\_opt\; typeconstr\_name \vdash constr\_decl_n \;\triangleright\; EB_n\end{array}}{\begin{array}{c}\vdash type\_params\_opt\; typeconstr\_name \;=\; constr\_decl_1 \,|\, ... \,|\, constr\_decl_n \;\mathbf{and}\; \overline{typedef_i}^{\,i} \;\triangleright\\ E, (\, typeconstr\_name \;:\; kind \,) \;\mathbf{and}\; E' \;\mathbf{and}\; E'' \;@\; EB_1 \,, \,...\,, \, EB_n\end{array}} \quad \text{JTtypedef\_def\_sum}$$

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

$$\frac{\begin{array}{l}\vdash \overline{typedef_i}^{\,i} \;\triangleright\; E \;\mathbf{and}\; E' \;\mathbf{and}\; E''\\ \vdash type\_params\_opt \;:\; kind\\ type\_params\_opt\; typeconstr\_name \vdash field\_name_1 \;:\; t_1 \;\triangleright\; EB_1 \quad ... \quad type\_params\_opt\; typeconstr\_name \vdash field\_name_n \;:\; t_n \;\triangleright\; EB_n\end{array}}{\begin{array}{c}\vdash type\_params\_opt\; typeconstr\_name \;=\; \{\, field\_name_1 \;:\; t_1 \,;\, ... \,;\, field\_name_n \;:\; t_n \,\} \;\mathbf{and}\; \overline{typedef_i}^{\,i} \;\triangleright\\ E, (\, typeconstr\_name \;:\; kind \,\{\, field\_name_1 \,;\, ... \,;\, field\_name_n \,\}\,) \;\mathbf{and}\; E' \;\mathbf{and}\; E'' \;@\; EB_1 \,, \,...\,, \, EB_n\end{array}} \quad \text{JTtypedef\_def\_record}$$

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 $\boxed{E \vdash type\_definition \;\triangleright\; E'}$ 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.

$$\frac{\begin{array}{l}\vdash typedef_1 \;\mathbf{and} \ldots \mathbf{and}\; typedef_n \;\triangleright\; E' \;\mathbf{and}\; E'' \;\mathbf{and}\; E'''\\ E'''' \;=\; E' \,@\, E'' \,@\, E'''\\ E \,@\, E'''' \vdash \mathbf{ok}\end{array}}{E \vdash \mathbf{type}\; typedef_1 \;\mathbf{and} \ldots \mathbf{and}\; typedef_n \;\triangleright\; E''''} \quad \text{JTtype\_definition\_list}$$

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

3.32 $\boxed{E \vdash definition : E'}$ 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.

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

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

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

$$\frac{E \vdash \mathbf{type}\, typedef_1\, \mathbf{and} \ldots \mathbf{and}\, typedef_n \;\triangleright\; E'}{E \vdash \mathbf{type}\, typedef_1\, \mathbf{and} \ldots \mathbf{and}\, typedef_n : E'} \quad \text{JTdefinition\_typedef}$$

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

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

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

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

$$\frac{\begin{array}{l} E \vdash definition : E' \\ E \,@\, E' \vdash definitions' : E'' \end{array}}{E \vdash definition\; definitions' : E' \,@\, E''} \quad \text{JTdefinitions\_item}$$

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

Checks a progam.

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

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

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

Checks that the values in a store have types.

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

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

3.36 $\boxed{E \vdash \langle program, store \rangle}$ 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.

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

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

Used in the proof only

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

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

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

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

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

Used in the proof only

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

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

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

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

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: $\mathbf{let}\, f \;=\; \mathbf{function}\, 1 \;\to\; \mathbf{function}\, \_ \;\to\; 10\,;;\, \mathbf{let}\, \_ \;=\; f\, 2\,;;$ 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 $\boxed{\vdash \mathit{expr}\ \mathbf{matches}\ \mathit{pattern}}$ Pattern matching

Determines if a value matches a pattern.

$$\frac{}{\vdash v\ \mathbf{matches}\ x}\quad \textsf{JM\_matchP\_var}$$

$$\frac{}{\vdash v\ \mathbf{matches}\ \_}\quad \textsf{JM\_matchP\_any}$$

$$\frac{}{\vdash \mathit{constant}\ \mathbf{matches}\ \mathit{constant}}\quad \textsf{JM\_matchP\_constant}$$

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

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

$$\frac{\vdash v\ \mathbf{matches}\ \mathit{pat}_1}{\vdash v\ \mathbf{matches}\ \mathit{pat}_1 \mid \mathit{pat}_2}\quad \textsf{JM\_matchP\_or\_left}$$

$$\frac{\vdash v\ \mathbf{matches}\ \mathit{pat}_2}{\vdash v\ \mathbf{matches}\ \mathit{pat}_1 \mid \mathit{pat}_2}\quad \textsf{JM\_matchP\_or\_right}$$

$$\frac{\vdash v_1\ \mathbf{matches}\ \mathit{pat}_1 \quad \ldots \quad \vdash v_n\ \mathbf{matches}\ \mathit{pat}_n}{\vdash \mathit{constr}\ (\ v_1\ ,\ \ldots\ ,\ v_n\ )\ \mathbf{matches}\ \mathit{constr}\ (\ \mathit{pat}_1\ ,\ \ldots\ ,\ \mathit{pat}_n\ )}\quad \textsf{JM\_matchP\_construct}$$

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

$$\frac{\vdash v_1\ \mathbf{matches}\ \mathit{pat}_1 \quad \ldots. \quad \vdash v_n\ \mathbf{matches}\ \mathit{pat}_n}{\vdash (\ v_1\ ,\ \ldots.\ ,\ v_n\ )\ \mathbf{matches}\ (\ \mathit{pat}_1\ ,\ \ldots.\ ,\ \mathit{pat}_n\ )}\quad \textsf{JM\_matchP\_tuple}$$

$$\frac{\begin{array}{l}\mathit{field\_name}'_1\ =\ v'_1 \ldots \mathit{field\_name}'_n\ =\ v'_n\ \mathit{fn}_1\ =\ v''_1 \mathinner{..} \mathit{fn}_l\ =\ v''_l\ \mathbf{PERMUTES}\ \mathit{field\_name}_1\ =\ v_1 \ldots \mathit{field\_name}_m\ =\ v_m\\ \vdash v'_1\ \mathbf{matches}\ \mathit{pat}_1 \quad \ldots \quad \vdash v'_n\ \mathbf{matches}\ \mathit{pat}_n\\ \mathit{field\_name}_1 \ldots \mathit{field\_name}_m\ \mathbf{distinct}\end{array}}{\vdash \{\ \mathit{field\_name}_1\ =\ v_1\ ;\ \ldots ;\ \mathit{field\_name}_m\ =\ v_m\ \}\ \mathbf{matches}\ \{\ \mathit{field\_name}'_1\ =\ \mathit{pat}_1\ ;\ \ldots ;\ \mathit{field\_name}'_n\ =\ \mathit{pat}_n\ \}}\quad \textsf{JM\_matchP\_record}$$

$$\frac{\begin{array}{c}\vdash v_1\ \mathbf{matches}\ \mathit{pat}_1\\ \vdash v_2\ \mathbf{matches}\ \mathit{pat}_2\end{array}}{\vdash v_1 :: v_2\ \mathbf{matches}\ \mathit{pat}_1 :: \mathit{pat}_2}\quad \textsf{JM\_matchP\_cons}$$

4.2 $\boxed{\vdash \mathit{expr}\ \mathbf{matches}\ \mathit{pattern}\ \ \triangleright\ \ \{\!\!\{\ \mathit{substs\_x}\ \}\!\!\}}$ 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.

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

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

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

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

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

$$\frac{\vdash v\ \mathbf{matches}\ \mathit{pat}_1\ \ \triangleright\ \ \{\!\!\{\ x_1 \leftarrow v_1\ ,\ ..\ ,\ x_n \leftarrow v_n\ \}\!\!\}}{\vdash v\ \mathbf{matches}\ \mathit{pat}_1 \mid \mathit{pat}_2\ \ \triangleright\ \ \{\!\!\{\ x_1 \leftarrow v_1\ ,\ ..\ ,\ x_n \leftarrow v_n\ \}\!\!\}}\quad \text{JM\_match\_or\_left}$$

$$\frac{\begin{array}{l}\neg(v\ \mathbf{matches}\ \mathit{pat}_1)\\ \vdash v\ \mathbf{matches}\ \mathit{pat}_2\ \ \triangleright\ \ \{\!\!\{\ x_1 \leftarrow v_1\ ,\ ..\ ,\ x_n \leftarrow v_n\ \}\!\!\}\end{array}}{\vdash v\ \mathbf{matches}\ \mathit{pat}_1 \mid \mathit{pat}_2\ \ \triangleright\ \ \{\!\!\{\ x_1 \leftarrow v_1\ ,\ ..\ ,\ x_n \leftarrow v_n\ \}\!\!\}}\quad \text{JM\_match\_or\_right}$$

$$\frac{\vdash v_1\ \mathbf{matches}\ \mathit{pat}_1\ \ \triangleright\ \ \{\!\!\{\ \mathit{substs\_x}_1\ \}\!\!\}\quad ...\quad \vdash v_n\ \mathbf{matches}\ \mathit{pat}_n\ \ \triangleright\ \ \{\!\!\{\ \mathit{substs\_x}_n\ \}\!\!\}}{\vdash \mathit{constr}\ (\ v_1\ ,\ ...\ ,\ v_n\ )\ \mathbf{matches}\ \mathit{constr}\ (\ \mathit{pat}_1\ ,\ ...\ ,\ \mathit{pat}_n\ )\ \ \triangleright\ \ \{\!\!\{\ \mathit{substs\_x}_1\ @\ ...\ @\ \mathit{substs\_x}_n\ \}\!\!\}}\quad \text{JM\_match\_construct}$$

$$\frac{}{\vdash \mathit{constr}\ (\ v_1\ ,\ ...\ ,\ v_n\ )\ \mathbf{matches}\ \mathit{constr}\ \_\ \ \triangleright\ \ \{\!\!\{\ \}\!\!\}}\quad \text{JM\_match\_construct\_any}$$

$$\frac{\vdash v_1\ \mathbf{matches}\ \mathit{pat}_1\ \ \triangleright\ \ \{\!\!\{\ \mathit{substs\_x}_1\ \}\!\!\}\quad ....\quad \vdash v_n\ \mathbf{matches}\ \mathit{pat}_n\ \ \triangleright\ \ \{\!\!\{\ \mathit{substs\_x}_n\ \}\!\!\}}{\vdash (\ v_1\ ,\ ....\ ,\ v_n\ )\ \mathbf{matches}\ (\ \mathit{pat}_1\ ,\ ....\ ,\ \mathit{pat}_n\ )\ \ \triangleright\ \ \{\!\!\{\ \mathit{substs\_x}_1\ @\ ....\ @\ \mathit{substs\_x}_n\ \}\!\!\}}\quad \text{JM\_match\_tuple}$$

$$\frac{\begin{array}{l}\mathit{field\_name}'_1\ =\ v'_1\ ...\ \mathit{field\_name}'_n\ =\ v'_n\ \mathit{fn}_1\ =\ v''_1\ ..\ \mathit{fn}_l\ =\ v''_l\ \mathbf{PERMUTES}\ \mathit{field\_name}_1\ =\ v_1\ ...\ \mathit{field\_name}_m\ =\ v_m\\ \vdash v'_1\ \mathbf{matches}\ \mathit{pat}_1\ \ \triangleright\ \ \{\!\!\{\ \mathit{substs\_x}_1\ \}\!\!\}\quad ...\quad \vdash v'_n\ \mathbf{matches}\ \mathit{pat}_n\ \ \triangleright\ \ \{\!\!\{\ \mathit{substs\_x}_n\ \}\!\!\}\\ \mathit{field\_name}_1\ ...\ \mathit{field\_name}_m\ \mathbf{distinct}\end{array}}{\vdash \{\ \mathit{field\_name}_1\ =\ v_1\ ;\ ...\ ;\ \mathit{field\_name}_m\ =\ v_m\ \}\ \mathbf{matches}\ \{\ \mathit{field\_name}'_1\ =\ \mathit{pat}_1\ ;\ ...\ ;\ \mathit{field\_name}'_n\ =\ \mathit{pat}_n\ \}\ \ \triangleright\ \ \{\!\!\{\ \mathit{substs\_x}_1\ @\ ...\ @\ \mathit{substs\_x}_n\ \}\!\!\}}\quad \text{JM\_match\_record}$$

$$\frac{\begin{array}{l}\vdash v_1\ \mathbf{matches}\ \mathit{pat}_1\ \ \triangleright\ \ \{\!\!\{\ \mathit{substs\_x}_1\ \}\!\!\}\\ \vdash v_2\ \mathbf{matches}\ \mathit{pat}_2\ \ \triangleright\ \ \{\!\!\{\ \mathit{substs\_x}_2\ \}\!\!\}\end{array}}{\vdash v_1 :: v_2\ \mathbf{matches}\ \mathit{pat}_1 :: \mathit{pat}_2\ \ \triangleright\ \ \{\!\!\{\ \mathit{substs\_x}_1\ @\ \mathit{substs\_x}_2\ \}\!\!\}}\quad \text{JM\_match\_cons}$$

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

Expands a recursive definition.

$$\frac{\mathit{letrec\_bindings} = (x_1\ =\ \textbf{function}\,\mathit{pattern\_matching}_1\ \textbf{and}\ \ldots\ \textbf{and}\ x_n\ =\ \textbf{function}\,\mathit{pattern\_matching}_n)}{\textbf{recfun}\,(\,\mathit{letrec\_bindings}\,,\ \mathit{pattern\_matching}\,)\ \ \triangleright\ \ \{\!\{\,x_1 \leftarrow \textbf{let rec}\ \mathit{letrec\_bindings}\ \textbf{in}\ x_1\,,\ \ldots\,,\ x_n \leftarrow \textbf{let rec}\ \mathit{letrec\_bindings}\ \textbf{in}\ x_n\,\}\!\}\,(\,\textbf{function}\,\mathit{pattern\_matching}\,)}\ \textsf{Jrecfun\_letrec}$$

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

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

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

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

$$\frac{}{\vdash\ \textbf{funval}\,(\,(\,\%\textbf{prim}\,\mathit{binary\_prim}\,)\ v\,)}\ \textsf{Jfunval\_bp\_app}$$

$$\frac{}{\vdash\ \textbf{funval}\,(\,\textbf{function}\,\mathit{pattern\_matching}\,)}\ \textsf{Jfunval\_func}$$

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

Computes the result of a unary primitive application.

$$\frac{}{\vdash\ \textbf{not}\,\textbf{true}\ \longrightarrow\ \textbf{false}}\ \textsf{Juprim\_not\_true}$$

$$\frac{}{\vdash\ \textbf{not}\,\textbf{false}\ \longrightarrow\ \textbf{true}}\ \textsf{Juprim\_not\_false}$$

$$\frac{}{\vdash \sim\!-\ \dot{n}\ \longrightarrow\ 0\ \dot{-}\ \dot{n}}\ \textsf{Juprim\_uminus}$$

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.

$$\frac{}{\vdash\ \textbf{ref}\ v\ \xrightarrow{\textbf{ref}\ v = l}\ l}\ \textsf{Juprim\_ref\_alloc}$$

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

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

Computes the result of a binary primitive application.

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c} v'\ =\ \{\,fn_1''\ =\ v_1''\,;\ ...\,;\ fn_m''\ =\ v_m''\,\} \\ fn_1\ ...\ fn_n\ \mathbf{PERMUTES}\ fn_1''\ ...\ fn_m'' \end{array}}{\vdash\ \{\,fn_1\ =\ v_1\,;\ ...\,;\ fn_n\ =\ v_n\,\}\ =\ v' \longrightarrow \mathbf{AND}\,(\,(\,(\,\%\mathbf{prim}\ =\,)\ v_1\,)\,(\,v'\,.\,fn_1\,)\ \&\&\ ...\ \&\&\ (\,(\,\%\mathbf{prim}\ =\,)\ v_n\,)\,(\,v'\,.\,fn_n\,)\,)}\ \textsf{Jbprim\_equal\_rec}$$

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\neg(v\ \mathbf{matches}\ \mathit{pat})\\ \mathbf{length}\,(\,e_1\,) \ldots (\,e_n\,) \geq 1\end{array}}{\vdash v\,\mathbf{with}\,\mathit{pat} \rightarrow e \mid \mathit{pat}_1 \rightarrow e_1 \mid \ldots \mid \mathit{pat}_n \rightarrow e_n \longrightarrow \mathit{pat}_1 \rightarrow e_1 \mid \ldots \mid \mathit{pat}_n \rightarrow e_n} \quad \textsf{JRmatching\_next}$$

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

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

$$\frac{\vdash v\,\mathbf{matches}\,\mathit{pat} \ \triangleright\ \{\!\{\, x_1 \leftarrow v_1 \,,\, ..\,,\, x_m \leftarrow v_m \,\}\!\}}{\vdash v\,\mathbf{with}\,\mathit{pat} \rightarrow e \mid \mathit{pat}_1 \rightarrow e_1 \mid \ldots \mid \mathit{pat}_n \rightarrow e_n \longrightarrow \{\!\{\, x_1 \leftarrow v_1 \,,\, ..\,,\, x_m \leftarrow v_m \,\}\!\}\, e} \quad \textsf{JRmatching\_found}$$

$$\frac{\neg(v\ \mathbf{matches}\ \mathit{pat})}{\vdash v\,\mathbf{with}\,\mathit{pat} \rightarrow e \longrightarrow (\,\%\mathbf{prim\,raise}\,)\,\mathbf{Match\_failure}} \quad \textsf{JRmatching\_fail}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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).

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

$$\frac{}{\vdash\ \mathbf{match}\ (\,\%\mathbf{prim\,raise}\,)\ v\ \mathbf{with}\ \mathit{pattern\_matching} \ \longrightarrow\ (\,\%\mathbf{prim\,raise}\,)\ v} \quad \textsf{JR\_expr\_match\_raise}$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{}{\vdash\ (\ (\ \%\mathbf{prim}\,\mathbf{raise}\ )\ v\ ) :: v'\ \longrightarrow\ (\ \%\mathbf{prim}\,\mathbf{raise}\ )\ v}\quad \textsf{JR\_expr\_cons\_raise2}$$

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.

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

$$\frac{}{\vdash\ \{\ \mathit{fn}_1\ =\ e_1\ ;\ ...\ ;\ \mathit{fn}_m\ =\ e_m\ ;\ \mathit{fn}\ =\ (\ \%\mathbf{prim}\,\mathbf{raise}\ )\ v\ ;\ \mathit{fn}'_1\ =\ v_1\ ;\ ...\ ;\ \mathit{fn}'_n\ =\ v_n\ \}\ \longrightarrow\ (\ \%\mathbf{prim}\,\mathbf{raise}\ )\ v}\quad \textsf{JR\_expr\_record\_raise}$$

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

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

$$\frac{}{\vdash\ \{\ v'\ \mathbf{with}\ \mathit{fn}_1\ =\ e_1\ ;\ ...\ ;\ \mathit{fn}_m\ =\ e_m\ ;\ \mathit{fn}\ =\ (\ \%\mathbf{prim}\,\mathbf{raise}\ )\ v\ ;\ \mathit{fn}'_1\ =\ v_1\ ;\ ...\ ;\ \mathit{fn}'_n\ =\ v_n\ \}\ \longrightarrow\ (\ \%\mathbf{prim}\,\mathbf{raise}\ )\ v}\quad \textsf{JR\_expr\_record\_with\_raise1}$$

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

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

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

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

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

$$\frac{}{\vdash ((\%\mathbf{prim}\,\mathbf{raise})\, v)\,.\,\mathit{field\_name} \longrightarrow (\%\mathbf{prim}\,\mathbf{raise})\, v} \quad \mathsf{JR\_expr\_record\_access\_raise}$$

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

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

$$\frac{}{\vdash \mathbf{assert}\,((\%\mathbf{prim}\,\mathbf{raise})\, v) \longrightarrow (\%\mathbf{prim}\,\mathbf{raise})\, v} \quad \mathsf{JR\_expr\_assert\_raise}$$

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

$$\frac{}{\vdash \mathbf{assert}\ \mathbf{false} \longrightarrow (\%\mathbf{prim}\,\mathbf{raise})\ \mathbf{Assert\_failure}} \quad \mathsf{JR\_expr\_assert\_false}$$

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.

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

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

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

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

$$\frac{\begin{array}{l} letrec\_bindings = (x_1\ =\ \mathbf{function}\ pattern\_matching_1\ \mathbf{and} \ldots \mathbf{and}\ x_n\ =\ \mathbf{function}\ pattern\_matching_n) \\ \mathbf{recfun}\ (\ letrec\_bindings\ ,\ pattern\_matching_1\ )\ \ \triangleright\ \ e_1 \quad \ldots \quad \mathbf{recfun}\ (\ letrec\_bindings\ ,\ pattern\_matching_n\ )\ \ \triangleright\ \ e_n \end{array}}{\vdash \langle ds\_value, \mathbf{let}\ \mathbf{rec}\ letrec\_bindings\ ;;\ definitions\rangle \longrightarrow \langle ds\_value, \{\!\{\ x_1 \leftarrow \mathbf{remv\_tyvar}\ e_1\ ,\ \ldots\ ,\ x_n \leftarrow \mathbf{remv\_tyvar}\ e_n\ \}\!\}\ definitions\rangle}\quad \textsf{Jdefn\_letrec}$$

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

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

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

Gets the value stored at a given location.

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

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

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

Coordinates a store with a label.

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

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

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

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

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.

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

4.14 $\boxed{\vdash\ \mathit{expr}\ \mathbf{behaves}}$ 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.

$$\frac{}{\vdash\ v\ \mathbf{behaves}}\quad \mathsf{JRB\_ebehaviour\_value}$$

$$\frac{\vdash\ e\ \xrightarrow{L}\ e'}{\vdash\ e\ \mathbf{behaves}}\quad \mathsf{JRB\_ebehaviour\_reduces}$$

$$\frac{}{\vdash\ (\,\%\mathbf{prim}\,\mathbf{raise}\,)\ v\ \mathbf{behaves}}\quad \mathsf{JRB\_ebehaviour\_raises}$$

4.15 $\boxed{\vdash \langle \mathit{definitions}, \mathit{program}, \mathit{store}\rangle \mathbf{behaves}}$ 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.

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

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

$$\frac{}{\vdash \langle \mathit{definitions\_value}, (\%\mathbf{prim\,raise})\ v, \mathit{store}\rangle \mathbf{behaves}} \quad \mathsf{JRB\_behaviour\_raises}$$

5 Statistics

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