

<i>Z, len</i>	integer
<i>f</i>	float
<i>n</i>	int
<i>ident, id</i>	identifier
<i>label, l</i>	label
<i>dcls</i>	global variable declarations
<i>fndefns</i>	function declarations
<i>opt_tid</i>	optional thread id
<i>ef_sig</i>	external signature
<i>p</i>	pointer
<i>typ</i>	
<i>fundef, fd</i>	
<i>fn_body</i>	
<i>fn</i>	
<i>ge</i>	
δ	

<i>signedness</i>	<pre> ::= Signed Unsigned </pre>	Signedness
<i>intsize</i>	<pre> ::= I8 I16 I32 </pre>	Integer sizes
<i>floatsize</i>	<pre> ::= F32 F64 </pre>	Float sizes
<i>type, ty</i>	<pre> ::= void int(<i>intsize</i>, <i>signedness</i>) float(<i>floatsize</i>) pointer(<i>ty</i>) array(<i>ty</i>, <i>len</i>) function(<i>ty</i>*, <i>ty</i>) struct(<i>id</i>, <i>φ</i>) union(<i>id</i>, <i>φ</i>) comp_pointer(<i>id</i>) (<i>ty</i>) </pre>	Types the void type integer types floating-point types pointer types (*ty) array types (ty[<i>len</i>]) function types struct types union types pointer to named struct or union S
<i>typelist, ty</i> *	<pre> ::= nil <i>ty</i>::<i>ty</i>* </pre>	Type list
<i>fieldlist, φ</i>	<pre> ::= nil (<i>id</i>, <i>ty</i>)::<i>φ</i> </pre>	Field lists
<i>unary_operation, op</i> ₁	<pre> ::= ! ~ - </pre>	unary Boolean negation Integer complement opposite
<i>binary_operation, op</i> ₂	<pre> ::= + - * / % & ^ </pre>	binary addition subtraction multiplication division modulo bitwise and bitwise or bitwise xor

		<<	left shift
		>>	right shift
		==	equality
		!=	not equal
		<	less than
		>	greater than
		<=	less than equal
		>=	greater than equal
<i>expr, e</i>	::=		typed expression
		<i>a</i> ^{<i>ty</i>}	expression
<i>expr_descr, a</i>	::=		basic expressions
		<i>n</i>	integer literal
		<i>f</i>	float literal
		<i>id</i>	variable
		* <i>e</i>	unary pointer dereference
		& <i>e</i>	address-of
		<i>op</i> ₁ <i>e</i>	unary operation
		<i>e</i> ₁ <i>op</i> ₂ <i>e</i> ₂	binary operation
		(<i>ty</i>) <i>e</i>	type cast
		<i>e</i> ₁ ? <i>e</i> ₂ : <i>e</i> ₃	conditional
		<i>e</i> ₁ && <i>e</i> ₂	sequential and
		<i>e</i> ₁ <i>e</i> ₂	sequential or
		sizeof (<i>ty</i>)	size of a type
		<i>e</i> . <i>id</i>	access to a member of a struct or union
<i>opt_lhs</i>	::=		optional lhs expression
		(<i>id</i> : <i>ty</i>)=	
<i>opt_e</i>	::=		optional expression
		<i>e</i>	
<i>e</i> *	::=		expression list
<i>atomic_statement, astmt</i>	::=		atomic
		CAS	compare and swap
		ATOMIC_INC	locked inc
<i>statement, s</i>	::=		statements
		skip	do nothing
		<i>e</i> ₁ = <i>e</i> ₂	assignment [lvalue = rvalue]
		<i>opt_lhs</i> <i>e'</i> (<i>e</i> *	function or procedure call
		<i>s</i> ₁ ; <i>s</i> ₂	sequence
		if (<i>e</i> ₁) then <i>s</i> ₁ else <i>s</i> ₂	conditional

	while (<i>e</i>) do <i>s</i> do <i>s</i> while (<i>e</i>) for (<i>s</i> ₁ ; <i>e</i> ₂ ; <i>s</i> ₃) <i>s</i> break continue return <i>opt_e</i> switch (<i>e</i>) <i>ls</i> <i>l</i> : <i>s</i> goto <i>l</i> thread_create(<i>e</i> ₁ , <i>e</i> ₂) <i>opt_lhs astmt</i> (<i>e</i> [*]) mfence	while do while for loop break continue return switch labelled statement goto thread creation atomic operation mfence
<i>labeled_statements, ls</i>	::= default : <i>s</i> case <i>n</i> : <i>s</i> ; <i>ls</i>	labeled statements default labeled case
<i>arglist</i>	::= <i>ty id</i> <i>ty id, arglist</i>	Argument lists
<i>varlist</i>	::= <i>ty id; varlist</i>	Local variable lists
<i>fndefn_internal</i>	::= <i>ty id</i> (<i>arglist</i>){ <i>varlist s</i> }	function definition
<i>program</i>	::= <i>dcls fndefns</i> main= <i>id</i>	programs
<i>val, v</i>	::= <i>n</i> <i>f</i> <i>p</i> undef	untyped values integer value floating point value pointer undef
<i>extval, evl</i>	::= extint <i>n</i> extfloat <i>f</i>	external values external integer value external floating point value
<i>vs</i>	::= nil <i>v</i> :: <i>vs</i> <i>vs</i> @[<i>v</i>]	value list

<i>arg_list</i> , <i>args</i>	$::=$ <code>nil</code> <i>id^{ty}</i> :: <i>args</i>	argument lists
<i>evs</i>	$::=$ <code>nil</code> <i>evl</i> :: <i>evs</i>	eventval list
<i>memory_chunk</i> , <i>c</i>	$::=$ <code>Mint32</code>	
<i>mobject_kind</i>	$::=$ <code>MObjStack</code>	
<i>rmw_instr</i> , <i>rmwi</i>	$::=$ <code>ADD <i>v</i></code> <code>CAS <i>v v'</i></code> <code>SET <i>v</i></code>	
<i>mem_event</i> , <i>me</i>	$::=$ <code>write <i>p memory_chunk v</i></code> <code>read <i>p memory_chunk v</i></code> <code>alloc <i>p n mobject_kind</i></code> <code>free <i>p mobject_kind</i></code> <code>rmw <i>p memory_chunk v rmwi</i></code> <code>fence</code>	
<i>event</i> , <i>ev</i>	$::=$ <code>call <i>id evs</i></code> <code>return <i>typ evl</i></code> <code>exit <i>n</i></code> <code>fail</code>	
<i>thread_event</i> , <i>te</i>	$::=$ <code>ext <i>event</i></code> <code>mem <i>mem_event</i></code> <code>exit</code> <code>start <i>p vs</i></code>	thread events externally observable event memory event thread-local event normal exit thread start (bootstrap)
<i>opt_pty</i>	$::=$	optional pointer/type pair
<i>opt_v</i>	$::=$	optional value
<i>ps</i>	$::=$	pointer list
ρ , ρ' , ρ''	$::=$	environment

$cont, \kappa_S$	$::=$	statement continuation
	stop	
	$[-; s_2] \cdot \kappa_S$	sequence
	$[\text{while } (e) \text{ do } s] \cdot \kappa_S$	while
	$[\text{do } s \text{ while } (e)] \cdot \kappa_S$	do while
	$[\text{for}(; e_2; \diamond s_3) s] \cdot \kappa_S$	for loop, pending increment
	$[\text{for}(; \diamond e_2; s_3) s] \cdot \kappa_S$	for loop, pending condition evaluation
	$[\text{opt_lhs } fd(-) _{\rho}] \cdot \kappa_S$	call awaiting args
	$[\text{switch } \kappa_S]$	switch protecting break
	$[\text{free } ps; \text{return } opt_v] \cdot \kappa_S$	
$expr_cont, \kappa_e$	$::=$	expression continuations
	$[op_1^{ty} -] \cdot \kappa_e$	unary operation
	$[- op_2^{ty_1 * ty_2 \rightarrow ty} e_2] \cdot \kappa_e$	binary operation
	$[v op_2^{ty_1 * ty_2 \rightarrow ty} -] \cdot \kappa_e$	binary operation
	$[(ty)_{-}^{ty'}] \cdot \kappa_e$	
	$[-^{ty} ? e_2 : e_3] \cdot \kappa_e$	
	$[- \cdot \delta] \cdot \kappa_e$	access to member of struct
	$[*_{-}^{ty}] \cdot \kappa_e$	load
	$[-^{ty} = e_2] \cdot \kappa_S$	assignment
	$[v^{ty} = -] \cdot \kappa_S$	assignment
	$[\text{opt_lhs } -^{ty} (e^*)] \cdot \kappa_S$	call function
	$[\text{opt_lhs } v^{ty} (vs, e^*)] \cdot \kappa_S$	call args
	$[\text{opt_lhs } astmt(vs, e^*)] \cdot \kappa_S$	atomic args
	$[\text{if } (-^{ty}) \text{ then } s_1 \text{ else } s_2] \cdot \kappa_S$	if
	$[\text{while } (-e) \text{ do } s] \cdot \kappa_S$	while
	$[\text{do } s \text{ while } (-e)] \cdot \kappa_S$	dowhile
	$[\text{for } (; -e_2; s_3) s] \cdot \kappa_S$	for loop, pending test
	$[\text{return } -] \cdot \kappa_S$	function return
	$[\text{switch } (-) ls] \cdot \kappa_S$	switch
	$[\text{thread_create } (-, e_2)] \cdot \kappa_S$	thread creation
	$[\text{thread_create } (p, -)] \cdot \kappa_S$	thread creation
$state, \sigma$	$::=$	states
	$\text{lval } (e) \cdot \kappa_e _{\rho}$	
	$e \cdot \kappa_e _{\rho}$	
	$v \cdot \kappa_e _{\rho}$	
	$s \cdot \kappa_S _{\rho}$	
	$vs \cdot \kappa_S$	
	$\text{bind } (fn, vs, args) \cdot \kappa_S _{\rho}$	
	$\text{alloc } (vs, args) \cdot \kappa_S _{\rho}$	
	$opt_lhs \text{ ext } (-^{typ}) \cdot \kappa_S _{\rho}$	
	$opt_lhs v \cdot \kappa_S _{\rho}$	

$\boxed{\sigma \xrightarrow{te} \sigma'}$ Labelled Transitions (parameterised over ge)

Integer Constant

$$\frac{}{n^{ty} \cdot \kappa_e |_{\rho} \longrightarrow n \cdot \kappa_e |_{\rho}} \text{ STEPCONSTINT}$$

Float Constant

$$\frac{}{f^{ty} \cdot \kappa_e |_{\rho} \longrightarrow f \cdot \kappa_e |_{\rho}} \text{ STEPCONSTFLOAT}$$

Var

$$\begin{array}{c}
\frac{}{id^{ty} \cdot \kappa_e \mid_\rho \longrightarrow lval(id^{ty}) \cdot [*_{-ty}] \cdot \kappa_e \mid_\rho} \text{STEPVAREXPRBYVALUE} \\
\\
\frac{\rho!id = \text{Some } p}{lval(id^{ty}) \cdot \kappa_e \mid_\rho \longrightarrow p \cdot \kappa_e \mid_\rho} \text{STEPVARLOCAL} \\
\\
\frac{\rho!id = \text{None} \quad \text{Genv.find_symbol ge } id = \text{Some } p}{lval(id^{ty}) \cdot \kappa_e \mid_\rho \longrightarrow p \cdot \kappa_e \mid_\rho} \text{STEPVARGLOBAL}
\end{array}$$

Load

$$\begin{array}{c}
\frac{\text{access_mode } ty' = \text{By_value } c \quad typ = \text{type_of_chunk } c \quad \text{Val.has_type } v \text{ } typ}{p \cdot [*_{-ty'}] \cdot \kappa_e \mid_\rho \xrightarrow{\text{mem (read } p \text{ } c \text{ } v)} v \cdot \kappa_e \mid_\rho} \text{STEPLoadBYVALUE} \\
\\
\frac{\text{access_mode } ty' = \text{By_reference} \setminus / \text{access_mode } ty' = \text{By_nothing}}{p \cdot [*_{-ty'}] \cdot \kappa_e \mid_\rho \longrightarrow p \cdot \kappa_e \mid_\rho} \text{STEPLoadNOTBYVALUE}
\end{array}$$

AddrOf

$$\frac{}{\&e^{ty} \cdot \kappa_e \mid_\rho \longrightarrow lval(e) \cdot \kappa_e \mid_\rho} \text{STEPADDR}$$

Econdition

$$\begin{array}{c}
\frac{}{e_1?e_2:e_3^{ty} \cdot \kappa_e \mid_\rho \longrightarrow e_1 \cdot [_{\text{typeof } e_1?e_2:e_3}] \cdot \kappa_e \mid_\rho} \text{STEPCONDITION} \\
\\
\frac{\text{is_true } v \text{ } ty}{v \cdot [_{ty?e_2:e_3}] \cdot \kappa_e \mid_\rho \longrightarrow e_2 \cdot \kappa_e \mid_\rho} \text{STEPCONDITIONTRUE} \\
\\
\frac{\text{is_false } v \text{ } ty}{v \cdot [_{ty?e_2:e_3}] \cdot \kappa_e \mid_\rho \longrightarrow e_3 \cdot \kappa_e \mid_\rho} \text{STEPCONDITIONFALSE}
\end{array}$$

Dereference

$$\begin{array}{c}
\frac{}{*e^{ty} \cdot \kappa_e \mid_\rho \longrightarrow e \cdot [*_{-ty}] \cdot \kappa_e \mid_\rho} \text{STEPDEREF} \\
\\
\frac{}{lval(*e^{ty}) \cdot \kappa_e \mid_\rho \longrightarrow e \cdot \kappa_e \mid_\rho} \text{STEPDEREFLVAL}
\end{array}$$

Field

$$\begin{array}{c}
\frac{}{e.id^{ty} \cdot \kappa_e \mid_\rho \longrightarrow lval(e.id^{ty}) \cdot [*_{-ty}] \cdot \kappa_e \mid_\rho} \text{STEPFIELD} \\
\\
\frac{\text{typeof } e = \text{struct}(id', \phi) \quad \text{field_offset } id \text{ } \phi = \text{OK } \delta}{lval(e.id^{ty}) \cdot \kappa_e \mid_\rho \longrightarrow lval(e) \cdot [_{\cdot \delta}] \cdot \kappa_e \mid_\rho} \text{STEPFSTRUCT1} \\
\\
\frac{p' = \text{Ptr.add } p \text{ } (\text{Int.repr } \delta)}{p \cdot [_{\cdot \delta}] \cdot \kappa_e \mid_\rho \longrightarrow p' \cdot \kappa_e \mid_\rho} \text{STEPFSTRUCT2}
\end{array}$$

Field Union

$$\frac{\text{typeof } e = \text{union}(id', \phi)}{lval(e.id^{ty}) \cdot \kappa_e \mid_\rho \longrightarrow lval(e) \cdot \kappa_e \mid_\rho} \text{STEPFUNION}$$

SizeOf

$$\frac{v = \text{Vint } (\text{Int.repr } (\text{sizeof } ty'))}{\text{sizeof } (ty')^{ty} \cdot \kappa_e \mid_\rho \longrightarrow v \cdot \kappa_e \mid_\rho} \text{STEPsizeof}$$

Unop

$$\begin{array}{c}
\frac{}{op_1 \text{ } e^{ty} \cdot \kappa_e \mid_\rho \longrightarrow e \cdot [op_1^{\text{typeof } e} _] \cdot \kappa_e \mid_\rho} \text{STEPUNOP1} \\
\frac{\text{sem_unary_operation } op_1 \text{ } v \text{ } ty = \text{Some } v'}{v \cdot [op_1^{ty} _] \cdot \kappa_e \mid_\rho \longrightarrow v' \cdot \kappa_e \mid_\rho} \text{STEPUNOP}
\end{array}$$

Binop

$$\begin{array}{c}
\frac{}{(e_1 \text{ } op_2 \text{ } e_2)^{ty} \cdot \kappa_e \mid_\rho \longrightarrow e_1 \cdot [_{op_2^{\text{typeof } e_1 * \text{typeof } e_2 \rightarrow ty}} e_2] \cdot \kappa_e \mid_\rho} \text{STEPBINOP1} \\
\frac{\text{valid_arg } op_2 \text{ } ty_1 \text{ } ty_2 \text{ } v = \text{true}}{v \cdot [_{op_2^{ty_1 * ty_2 \rightarrow ty}} e_2] \cdot \kappa_e \mid_\rho \longrightarrow e_2 \cdot [v \text{ } op_2^{ty_1 * ty_2 \rightarrow ty} _] \cdot \kappa_e \mid_\rho} \text{STEPBINOP2} \\
\frac{\text{sem_binary_operation } op_2 \text{ } v_1 \text{ } ty_1 \text{ } v_2 \text{ } ty_2 = \text{Some } v}{v_2 \cdot [v_1 \text{ } op_2^{ty_1 * ty_2 \rightarrow ty} _] \cdot \kappa_e \mid_\rho \longrightarrow v \cdot \kappa_e \mid_\rho} \text{STEPBINOP}
\end{array}$$

Cast

$$\begin{array}{c}
\frac{}{(ty) \text{ } e^{ty'} \cdot \kappa_e \mid_\rho \longrightarrow e \cdot [(ty)_{\text{typeof } e}] \cdot \kappa_e \mid_\rho} \text{STEPCAST1} \\
\frac{\text{cast } v \text{ } ty' \text{ } ty \text{ } v'}{v \cdot [(ty)_{\text{typeof } e}] \cdot \kappa_e \mid_\rho \longrightarrow v' \cdot \kappa_e \mid_\rho} \text{STEPCAST2}
\end{array}$$

Short circuit and (&&)

$$\frac{\begin{array}{l} n_0 = \text{Int.repr } 0 \\ n_1 = \text{Int.repr } 1 \end{array}}{e_1 \&\& e_2^{ty} \cdot \kappa_e \mid_\rho \longrightarrow e_1 ? (e_2 ? (n_1^{ty}) : (n_0^{ty})^{ty}) : n_0^{ty ty} \cdot \kappa_e \mid_\rho} \text{STEPANDBOOL}$$

Short circuit or

$$\frac{\begin{array}{l} n_0 = \text{Int.repr } 0 \\ n_1 = \text{Int.repr } 1 \end{array}}{e_1 \mid \mid e_2^{ty} \cdot \kappa_e \mid_\rho \longrightarrow e_1 ? (n_1^{ty}) : (e_2 ? (n_1^{ty}) : (n_0^{ty})^{ty})^{ty} \cdot \kappa_e \mid_\rho} \text{STEPORBOOL}$$

Thread

$$\begin{array}{c}
\frac{}{\text{thread_create}(e_1, e_2) \cdot \kappa_s \mid_\rho \longrightarrow e_1 \cdot [\text{thread_create}(_, e_2)] \cdot \kappa_s \mid_\rho} \text{STEPTHREAD} \\
\frac{}{p \cdot [\text{thread_create}(_, e_2)] \cdot \kappa_s \mid_\rho \longrightarrow e_2 \cdot [\text{thread_create}(p, _)] \cdot \kappa_s \mid_\rho} \text{STEPTHREADFN} \\
\frac{}{v \cdot [\text{thread_create}(p, _)] \cdot \kappa_s \mid_\rho \xrightarrow{\text{start } p \text{ } v :: \text{nil}} \text{skip} \cdot \kappa_s \mid_\rho} \text{STEPTHREAD EVT}
\end{array}$$

Assignment

$$\begin{array}{c}
\frac{}{e_1 = e_2 \cdot \kappa_s \mid_\rho \longrightarrow \text{lval}(e_1) \cdot [_{\text{typeof } e_1 = e_2}] \cdot \kappa_s \mid_\rho} \text{STEPASSIGN1} \\
\frac{}{v_1 \cdot [_{ty = e_2}] \cdot \kappa_s \mid_\rho \longrightarrow e_2 \cdot [v_1^{ty} = _] \cdot \kappa_s \mid_\rho} \text{STEPASSIGN2} \\
\frac{\begin{array}{l} \text{type_to_chunk } ty_1 = \text{Some } c \\ \text{cast_value_to_chunk } c \text{ } v_1 = v_2 \end{array}}{v_1 \cdot [p_1^{ty_1} = _] \cdot \kappa_s \mid_\rho \xrightarrow{\text{mem}(\text{write } p_1 \text{ } c \text{ } v_2)} \text{skip} \cdot \kappa_s \mid_\rho} \text{STEPASSIGN}
\end{array}$$

Statement sequence

$$\frac{}{s_1 ; s_2 \cdot \kappa_s \mid_\rho \longrightarrow s_1 \cdot [_{; s_2}] \cdot \kappa_s \mid_\rho} \text{STEPSEQ}$$

Call

$$\begin{array}{c}
\frac{}{\text{opt_lhs } e'(e^*) \cdot \kappa_s \mid_\rho \longrightarrow e' \cdot [\text{opt_lhs } \text{typeof } e'(e^*)] \cdot \kappa_s \mid_\rho} \text{STEPCALL} \\
\frac{\text{Genv.find_funct ge } v = \text{Some } fd \quad \text{type_of_fundef } fd = ty}{v \cdot [\text{opt_lhs } \text{ty}(\text{nil})] \cdot \kappa_s \mid_\rho \longrightarrow \text{nil} \cdot [\text{opt_lhs } fd(_) \mid_\rho] \cdot \kappa_s} \text{STEPCALLARGSNONE} \\
\frac{}{v \cdot [\text{opt_lhs } \text{ty}(e :: e^*)] \cdot \kappa_s \mid_\rho \longrightarrow e \cdot [\text{opt_lhs } v^{ty}(\text{nil}, e^*)] \cdot \kappa_s \mid_\rho} \text{STEPCALLARGS1} \\
\frac{}{v_1 \cdot [\text{opt_lhs } v^{ty}(vs, e :: e^*)] \cdot \kappa_s \mid_\rho \longrightarrow e \cdot [\text{opt_lhs } v^{ty}(vs@[v_1], e^*)] \cdot \kappa_s \mid_\rho} \text{STEPCALLARGS2} \\
\frac{\text{Genv.find_funct ge } v = \text{Some } fd \quad \text{type_of_fundef } fd = ty}{v' \cdot [\text{opt_lhs } v^{ty}(vs, \text{nil})] \cdot \kappa_s \mid_\rho \longrightarrow vs@[v'] \cdot [\text{opt_lhs } fd(_) \mid_\rho] \cdot \kappa_s} \text{STEPCALLFINISH}
\end{array}$$

Atomic statement

$$\begin{array}{c}
\frac{}{\text{opt_lhs astmt}(e :: e^*) \cdot \kappa_s \mid_\rho \longrightarrow e \cdot [\text{opt_lhs astmt}(\text{nil}, e^*)] \cdot \kappa_s \mid_\rho} \text{STEPATOMIC} \\
\frac{}{v \cdot [\text{opt_lhs astmt}(vs, e :: e^*)] \cdot \kappa_s \mid_\rho \longrightarrow e \cdot [\text{opt_lhs astmt}(vs@[v], e^*)] \cdot \kappa_s \mid_\rho} \text{STEPATOMICARGS} \\
\frac{\text{sem_atomic_statement astmt } (vs \text{ ++ } v :: \text{nil}) = \text{Some } (p, rmwi) \quad \text{Val.has_type } v' \text{ (type_of_chunk Mint32)}}{v \cdot [\text{astmt}(vs, \text{nil})] \cdot \kappa_s \mid_\rho \xrightarrow{\text{mem (rmw } p \text{ Mint32 } v' \text{ rmwi)}} \text{skip} \cdot \kappa_s \mid_\rho} \text{STEPATOMICFINISHNONE} \\
\frac{\text{sem_atomic_statement astmt } (vs \text{ ++ } v :: \text{nil}) = \text{Some } (p, rmwi) \quad \text{Val.has_type } v' \text{ (type_of_chunk Mint32)}}{v \cdot [(id:ty) = \text{astmt}(vs, \text{nil})] \cdot \kappa_s \mid_\rho \xrightarrow{\text{mem (rmw } p \text{ Mint32 } v' \text{ rmwi)}} (id:ty) = v' \cdot \kappa_s \mid_\rho} \text{STEPATOMICFINISHSOME}
\end{array}$$

Fence

$$\frac{}{\text{mfence} \cdot \kappa_s \mid_\rho \xrightarrow{\text{mem fence}} \text{skip} \cdot \kappa_s \mid_\rho} \text{STEPFENCE}$$

Continue

$$\frac{}{\text{continue} \cdot [-; s] \cdot \kappa_s \mid_\rho \longrightarrow \text{continue} \cdot \kappa_s \mid_\rho} \text{STEPCONTINUE}$$

Break

$$\frac{}{\text{break} \cdot [-; s] \cdot \kappa_s \mid_\rho \longrightarrow \text{break} \cdot \kappa_s \mid_\rho} \text{STEPBREAK}$$

If-then-else

$$\begin{array}{c}
\frac{}{\text{if } (e) \text{ then } s_1 \text{ else } s_2 \cdot \kappa_s \mid_\rho \longrightarrow e \cdot [\text{if } (\text{typeof } e) \text{ then } s_1 \text{ else } s_2] \cdot \kappa_s \mid_\rho} \text{STEPIFTHENELSE} \\
\frac{\text{is_true } v \text{ ty}}{v \cdot [\text{if } (\text{ty}) \text{ then } s_1 \text{ else } s_2] \cdot \kappa_s \mid_\rho \longrightarrow s_1 \cdot \kappa_s \mid_\rho} \text{STEPIFTHENELSETRUE} \\
\frac{\text{is_false } v \text{ ty}}{v \cdot [\text{if } (\text{ty}) \text{ then } s_1 \text{ else } s_2] \cdot \kappa_s \mid_\rho \longrightarrow s_2 \cdot \kappa_s \mid_\rho} \text{STEPIFTHENELSEFALSE}
\end{array}$$

While

$$\begin{array}{c}
\frac{}{\text{while } (e) \text{ do } s \cdot \kappa_s \mid_\rho \longrightarrow e \cdot [\text{while } (\neg e) \text{ do } s] \cdot \kappa_s \mid_\rho} \text{STEPWHILE} \\
\frac{\text{is_true } v \text{ (typeof } e)}{v \cdot [\text{while } (\neg e) \text{ do } s] \cdot \kappa_s \mid_\rho \longrightarrow s \cdot [\text{while } (e) \text{ do } s] \cdot \kappa_s \mid_\rho} \text{STEPWHILETRUE}
\end{array}$$

$$\begin{array}{c}
\frac{\text{is_false } v \text{ (typeof } e)}{v \cdot [\text{while } (-e) \text{ do } s] \cdot \kappa_s |_\rho \longrightarrow \text{skip} \cdot \kappa_s |_\rho} \quad \text{STEPWHILEFALSE} \\
\\
\frac{}{\text{continue} \cdot [\text{while } (e) \text{ do } s] \cdot \kappa_s |_\rho \longrightarrow \text{while } (e) \text{ do } s \cdot \kappa_s |_\rho} \quad \text{STEPCONTINUEWHILE} \\
\\
\frac{}{\text{break} \cdot [\text{while } (e) \text{ do } s] \cdot \kappa_s |_\rho \longrightarrow \text{skip} \cdot \kappa_s |_\rho} \quad \text{STEPBREAKWHILE} \\
\\
\text{DoWhile} \\
\\
\frac{}{\text{do } s \text{ while } (e) \cdot \kappa_s |_\rho \longrightarrow s \cdot [\text{do } s \text{ while } (e)] \cdot \kappa_s |_\rho} \quad \text{STEPDOWHILE} \\
\\
\frac{\text{is_true } v \text{ (typeof } e)}{v \cdot [\text{do } s \text{ while } (-e)] \cdot \kappa_s |_\rho \longrightarrow \text{do } s \text{ while } (e) \cdot \kappa_s |_\rho} \quad \text{STEPDOWHILETRUE} \\
\\
\frac{\text{is_false } v \text{ (typeof } e)}{v \cdot [\text{do } s \text{ while } (-e)] \cdot \kappa_s |_\rho \longrightarrow \text{skip} \cdot \kappa_s |_\rho} \quad \text{STEPDOWHILEFALSE} \\
\\
\frac{}{\text{continue} \cdot [\text{do } s \text{ while } (e)] \cdot \kappa_s |_\rho \longrightarrow e \cdot [\text{do } s \text{ while } (-e)] \cdot \kappa_s |_\rho} \quad \text{STEPDOCONTINUEWHILE} \\
\\
\frac{}{\text{break} \cdot [\text{do } s \text{ while } (e)] \cdot \kappa_s |_\rho \longrightarrow \text{skip} \cdot \kappa_s |_\rho} \quad \text{STEPDOBREAKWHILE} \\
\\
\text{For} \\
\\
\frac{}{\text{for } (s_1; e_2; s_3) s \cdot \kappa_s |_\rho \longrightarrow s_1 \cdot [\text{for } (; \diamond e_2; s_3) s] \cdot \kappa_s |_\rho} \quad \text{STEPFORINIT} \\
\\
\frac{}{\text{skip} \cdot [\text{for } (; \diamond e_2; s_3) s] \cdot \kappa_s |_\rho \longrightarrow e_2 \cdot [\text{for } (; -e_2; s_3) s] \cdot \kappa_s |_\rho} \quad \text{STEPFORCOND} \\
\\
\frac{\text{is_true } v \text{ (typeof } e_2)}{v \cdot [\text{for } (; -e_2; s_3) s] \cdot \kappa_s |_\rho \longrightarrow s \cdot [\text{for } (; e_2; \diamond s_3) s] \cdot \kappa_s |_\rho} \quad \text{STEPFORTRUE} \\
\\
\frac{\text{is_false } v \text{ (typeof } e_2)}{v \cdot [\text{for } (; -e_2; s_3) s] \cdot \kappa_s |_\rho \longrightarrow \text{skip} \cdot \kappa_s |_\rho} \quad \text{STEPFORFALSE} \\
\\
\frac{}{\text{skip} \cdot [\text{for } (; e_2; \diamond s_3) s] \cdot \kappa_s |_\rho \longrightarrow s_3 \cdot [\text{for } (; \diamond e_2; s_3) s] \cdot \kappa_s |_\rho} \quad \text{STEPFORINCR} \\
\\
\frac{}{\text{break} \cdot [\text{for } (; e_2; \diamond s_3) s] \cdot \kappa_s |_\rho \longrightarrow \text{skip} \cdot \kappa_s |_\rho} \quad \text{STEPFORBREAK} \\
\\
\frac{}{\text{continue} \cdot [\text{for } (; e_2; \diamond s_3) s] \cdot \kappa_s |_\rho \longrightarrow s_3 \cdot [\text{for } (; \diamond e_2; s_3) s] \cdot \kappa_s |_\rho} \quad \text{STEPFORCONTINUE} \\
\\
\text{Return} \\
\\
\frac{\begin{array}{l} \text{call_cont } \kappa_s = (\text{Kcall None (Internal } fn) \rho'' \kappa'_s) \\ fn.(fn_return) = \text{Tvoid} \\ ps = \text{sorted_pointers_of_env } \rho' \end{array}}{\text{return} \cdot \kappa_s |_{\rho'} \longrightarrow \text{skip} \cdot [\text{free } ps; \text{return None}] \cdot \kappa_s |_{\rho'}} \quad \text{STEPRETURNNONE} \\
\\
\frac{}{\text{skip} \cdot [\text{free } p :: ps; \text{return } opt_v] \cdot \kappa_s |_\rho \xrightarrow{\text{mem (free } p \text{ MObjStack)}} \text{skip} \cdot [\text{free } ps; \text{return } opt_v] \cdot \kappa_s |_\rho} \quad \text{STEPRETURN} \\
\\
\frac{\begin{array}{l} \text{call_cont } \kappa_s = \kappa'_s \\ \text{get_fundef } \kappa'_s = \text{Some (Internal } fn) \\ fn.(fn_return) <> \text{Tvoid} \end{array}}{\text{return } e \cdot \kappa_s |_{\rho'} \longrightarrow e \cdot [\text{return } _] \cdot \kappa_s |_{\rho'}} \quad \text{STEPRETURNSOME} \\
\\
\frac{ps = \text{sorted_pointers_of_env } \rho}{v \cdot [\text{return } _] \cdot \kappa_s |_\rho \longrightarrow \text{skip} \cdot [\text{free } ps; \text{return (Some } v)] \cdot \kappa_s |_\rho} \quad \text{STEPRETURNSOME1}
\end{array}$$

Switch

$$\begin{array}{c}
\frac{}{\text{switch } (e) \text{ } ls \cdot \kappa_s \mid_\rho \longrightarrow e \cdot [\text{switch } (e) \text{ } ls] \cdot \kappa_s \mid_\rho} \text{STEP SWITCH} \\
\frac{s = \text{seq_of_labeled_statement } (\text{select_switch } n \text{ } ls)}{n \cdot [\text{switch } (e) \text{ } ls] \cdot \kappa_s \mid_\rho \longrightarrow s \cdot [\text{switch } \kappa_s] \mid_\rho} \text{STEP SELECT SWITCH} \\
\frac{}{\text{break} \cdot [\text{switch } \kappa_s] \mid_\rho \longrightarrow \text{skip} \cdot \kappa_s \mid_\rho} \text{STEP BREAK SWITCH} \\
\frac{}{\text{continue} \cdot [\text{switch } \kappa_s] \mid_\rho \longrightarrow \text{continue} \cdot \kappa_s \mid_\rho} \text{STEP CONTINUE SWITCH}
\end{array}$$

Label

$$\frac{}{l : s \cdot \kappa_s \mid_\rho \longrightarrow s \cdot \kappa_s \mid_\rho} \text{STEP LABEL}$$

Goto

$$\frac{\begin{array}{l} \text{call_cont } \kappa_s = \kappa'_s \\ \text{get_fundef } \kappa'_s = (\text{Some } (\text{Internal } fn)) \\ \text{find_label } l \text{ } fn.(fn_body) \text{ } \kappa'_s = \text{Some } (s', \kappa''_s) \end{array}}{\text{goto } l \cdot \kappa_s \mid_\rho \longrightarrow s' \cdot \kappa''_s \mid_\rho} \text{STEP GOTO}$$

Internal Call/Return

$$\frac{\begin{array}{l} args = fn.(fn_params) ++ fn.(fn_vars) \\ fd = \text{Internal } fn \end{array}}{vs \cdot [\text{opt_lhs } fd \text{ } (e) \mid_\rho] \cdot \kappa_s \longrightarrow \text{alloc } (vs, args) \cdot [\text{opt_lhs } fd \text{ } (e) \mid_\rho] \cdot \kappa_s \mid_{\rho_{\text{empty}}}} \text{STEP FUNCTION INTERNAL}$$

$$\frac{n = \text{Int.repr}(\text{sizeof } ty)}{\text{alloc } (vs, id^{ty} :: args) \cdot \kappa_s \mid_\rho \xrightarrow{\text{mem } (\text{alloc } p \text{ } n \text{ } \text{MObjStack})} \text{alloc } (vs, args) \cdot \kappa_s \mid_{\rho \oplus (id \mapsto p)}} \text{STEP ALLOC LOCAL}$$

$$\frac{\begin{array}{l} args = fn.(fn_params) \\ fd = (\text{Internal } fn) \end{array}}{\text{alloc } (vs, nil) \cdot [\text{opt_lhs } fd \text{ } (e) \mid_{\rho'}] \cdot \kappa_s \mid_{\rho''} \longrightarrow \text{bind } (fn, vs, args) \cdot [\text{opt_lhs } fd \text{ } (e) \mid_{\rho'}] \cdot \kappa_s \mid_{\rho''}} \text{STEP BIND ARGS}$$

$$\frac{\begin{array}{l} \rho ! id = \text{Some } p \\ \text{type_to_chunk } ty = (\text{Some } c) \\ \text{cast_value_to_chunk } c \text{ } v_1 = v_2 \end{array}}{\text{bind } (fn, v_1 :: vs, id^{ty} :: args) \cdot \kappa_s \mid_\rho \xrightarrow{\text{mem } (\text{write } p \text{ } c \text{ } v_2)} \text{bind } (fn, vs, args) \cdot \kappa_s \mid_\rho} \text{STEP BIND ARGS}$$

$$\frac{s = fn.(fn_body)}{\text{bind } (fn, nil, nil) \cdot \kappa_s \mid_\rho \longrightarrow s \cdot \kappa_s \mid_\rho} \text{STEP TRANSFER FUN}$$

External Call/Return

$$\frac{\begin{array}{l} \text{true } (* \text{ event_match } (\text{external_function } id \text{ } \text{targs } ty) \text{ } vs \text{ } t \text{ } vres \text{ } \rightarrow *) \\ fd = \text{External } id \text{ } ty^* \text{ } ty \\ vs = \text{map val_of_eval } evs \end{array}}{vs \cdot [\text{opt_lhs } fd \text{ } (e) \mid_\rho] \cdot \kappa_s \xrightarrow{\text{ext } (\text{call } id \text{ } evs)} \text{opt_lhs ext } (e) \cdot \kappa_s \mid_\rho} \text{STEP EXTERNAL CALL}$$

$$\frac{\begin{array}{l} \text{Val.has_type } v \text{ } typ \\ fd = \text{External } id \text{ } ty^* \text{ } ty \\ typ = \text{match } (\text{opttyp_of_type } ty) \text{ with } | \text{Some } x \Rightarrow x \mid \text{None} \Rightarrow \text{Ast.Tint end} \\ v = \text{val_of_eval } evl \end{array}}{\text{opt_lhs ext } (e) \cdot \kappa_s \mid_\rho \xrightarrow{\text{ext } (\text{return } typ \text{ } evl)} \text{opt_lhs } v \cdot \kappa_s \mid_\rho} \text{STEP EXTERNAL RET}$$

$$\begin{array}{c}
\rho!id = \text{Some } p \\
\text{type_to_chunk } ty = \text{Some } c \\
\text{cast_value_to_chunk } c \ v_1 = v_2 \\
\hline
(id:ty) = v_1 \cdot \kappa_s \mid_\rho \xrightarrow{\text{mem}(\text{write } p \ c \ v_2)} \text{skip} \cdot \kappa_s \mid_\rho \quad \text{STEPEXTERNALSTORESOMELOCAL}
\end{array}$$

$$\begin{array}{c}
\rho!id = \text{None} \\
\text{Genv.find_symbol } ge \ id = \text{Some } p \\
\text{type_to_chunk } ty = \text{Some } c \\
\text{cast_value_to_chunk } c \ v_1 = v_2 \\
\hline
(id:ty) = v_1 \cdot \kappa_s \mid_\rho \xrightarrow{\text{mem}(\text{write } p \ c \ v_2)} \text{skip} \cdot \kappa_s \mid_\rho \quad \text{STEPEXTERNALSTORESOMEGLOBAL}
\end{array}$$

$$\frac{}{v \cdot \kappa_s \mid_\rho \longrightarrow \text{skip} \cdot \kappa_s \mid_\rho} \quad \text{STEPEXTERNALSTORENONE}$$

Continuation Management

$$\frac{}{\text{skip} \cdot [-; s_2] \cdot \kappa_s \mid_\rho \longrightarrow s_2 \cdot \kappa_s \mid_\rho} \quad \text{STEPSKIP}$$

$$\frac{}{\text{skip} \cdot [\text{while } (e) \text{ do } s] \cdot \kappa_s \mid_\rho \longrightarrow \text{while } (e) \text{ do } s \cdot \kappa_s \mid_\rho} \quad \text{STEPWHILELOOP}$$

$$\frac{}{\text{skip} \cdot [\text{do } s \text{ while } (e)] \cdot \kappa_s \mid_\rho \longrightarrow e \cdot [\text{do } s \text{ while } (-e)] \cdot \kappa_s \mid_\rho} \quad \text{STEPDOWHILELOOP}$$

$$\frac{}{\text{skip} \cdot [\text{switch } \kappa_s] \mid_\rho \longrightarrow \text{skip} \cdot \kappa_s \mid_\rho} \quad \text{STEPSKIPSWITCH}$$

$$\frac{\text{call_cont } \kappa_s = [fd(_) \mid_{\rho'}] \cdot \kappa'_s}{\text{skip} \cdot [\text{free nil; return } opt_v] \cdot \kappa_s \mid_{\rho''} \longrightarrow \text{skip} \cdot \kappa'_s \mid_{\rho'}} \quad \text{STEPRETURNNONEFINISH}$$

$$\frac{\begin{array}{l} \text{type_to_chunk } ty = (\text{Some } c) \\ \rho'!id = \text{Some } p \\ \text{call_cont } \kappa_s = [(id:ty) = fd(_) \mid_{\rho'}] \cdot \kappa'_s \\ \text{cast_value_to_chunk } c \ v_1 = v_2 \end{array}}{\text{skip} \cdot [\text{free nil; return } (\text{Some } v_1)] \cdot \kappa_s \mid_{\rho''} \xrightarrow{\text{mem}(\text{write } p \ c \ v_2)} \text{skip} \cdot \kappa'_s \mid_{\rho'}} \quad \text{STEPRETURN SOME FINISH LOCAL}$$

$$\frac{\begin{array}{l} \text{type_to_chunk } ty = (\text{Some } c) \\ \rho'!id = \text{None} \\ \text{Genv.find_symbol } ge \ id = \text{Some } p \\ \text{call_cont } \kappa_s = [(id:ty) = fd(_) \mid_{\rho'}] \cdot \kappa'_s \\ \text{cast_value_to_chunk } c \ v_1 = v_2 \end{array}}{\text{skip} \cdot [\text{free nil; return } (\text{Some } v_1)] \cdot \kappa_s \mid_{\rho''} \xrightarrow{\text{mem}(\text{write } p \ c \ v_2)} \text{skip} \cdot \kappa'_s \mid_{\rho'}} \quad \text{STEPRETURN SOME FINISH GLOBAL}$$

$$\frac{}{\text{skip} \cdot \text{stop} \mid_\rho \xrightarrow{\text{exit}} \text{skip} \cdot \text{stop} \mid_\rho} \quad \text{STEPSTOP}$$

Definition rules: 94 good 0 bad
Definition rule clauses: 178 good 0 bad