A formal specification for BIL: BIL Instruction Language

October 5, 2018

Contents

1 Introduction

This document describes the syntax and semantics of BAP Instruction Language. The language is used to represent a semantics of machine instructions. Each machine instruction is represented by a BIL program that captures side effect of the instruction.

2 Syntax

2.1 Metavariables

We define a small set of metavariables that are used to denote subscripts, numerals and string literals:

index, *m*, *n*	subscripts
id	a literal for variable
num	number literal
string, *str*	quoted string literal

2.2 BIL syntax

BIL program is reperesented as a sequence of statements. Each statement performs some side-effectful computation.

bil, *seq*	::=			
	|	$\{s_1; \ldots; s_n\}$	S	

stmt, *s*	::=			
	|	$var := exp$		– assign *exp* to *var*
	|	**jmp** e		– transfer control to a given address e
	|	**cpuexn** (num)		– interrupt CPU with a given interrupt *num*
	|	**special** $(string)$		– instruction with unknown semantics
	|	**while** $(exp) seq$		– eval *seq* while *exp* is true
	|	**if** $(e) seq$	S	– eval *seq* if e is true
	|	**if** $(e) seq_1$ **else** seq_2		– if e is true then eval seq_1 else seq_2

BIL expressions are side-effect free. Expressions include a usual set of operations on bitvectors, like arithmetic operations and converting bitvectors of one size to bitvectors of another size (casting in BIL parlance). We write $[e_1/var]e_2$ for the capture-avoiding substitution of e_1 for free occurances of *var* in e_2

exp, *e*	::=			
	|	(exp)	S	
	|	var		– a variable
	|	$word$		– an immediate value
	|	$v[w \leftarrow v' : sz]$		– a memory value
	|	$e_1[e_2, endian] : nat$		– load a value from address e_2 at storage e_1
	|	e_1 **with** $[e_2, endian] : nat \leftarrow e_3$		– update a storage e_1 with binding $e_2 \leftarrow e_3$
	|	$e_1\ bop\ e_2$		– perform binary operation on e_1 and e_2
	|	$uop\ e_1$		– perform an unary operation on e_1
	|	$cast : nat[e]$		– extract or extend bitvector e
	|	**let** $var = e_1$ **in** e_2		– bind e_1 to *var* in expression e_2
	|	**unknown** $[string] : type$		– unknown or undefined value of a given *type*
	|	**ite** $e_1\ e_2\ e_3$		– evaluates to e_2 if e_1 is true else to e_3

	|	**extract** : nat_1 : $nat_2[e]$		– extract or extend bitvector e
	|	$e_1 @ e_2$		– concatenate two bitvector e_1 to e_2
	|	$[e_1/var]e_2$	M	– the (capture avoiding) substitution of e_1 for var in e_2

var	::=		
	|	id : $type$	S

bop	::=		
	|	aop	– arithmetic operators
	|	lop	– logical operators
aop	::=		
	|	$+$	– plus
	|	$-$	– minus
	|	$*$	– times
	|	$/$	– divide
	|	$\overset{signed}{/}$	– signed divide
	|	$\%$	– modulo
	|	$\overset{signed}{\%}$	– signed modulo
	|	$\&$	– bitwise and
	|	$\vert$	– bitwise or
	|	**xor**	– bitwise xor
	|	$\ll$	– logical shift left
	|	$\gg$	– logical shift right
	|	$\ggg$	– arithmetic shift right
lop	::=		
	|	$=$	– equality
	|	$\neq$	– non-equality
	|	$<$	– less than
	|	$\leq$	– less than or equal
	|	$\overset{signed}{<}$	– signed less than
	|	$\overset{signed}{\leq}$	– signed less than or equal
uop	::=		
	|	$-$	– unary negation
	|	$\neg$	– bitwise complement
$endian$, ed	::=		
	|	**el**	– little endian
	|	**be**	– big endian
$cast$	::=		
	|	**low**	– extract lower bits
	|	**high**	– extract high bits
	|	**signed**	– extend with sign bit
	|	**unsigned**	– extend with zero

The type system of BIL consists of two type families - immediate values, indexed by a bitwidth, and storagies (aka memories), indexed with address bitwidth and values bitwidth.

type, t	::=			
	|	**imm** $< sz >$		– immediate of size sz
	|	**mem** $< sz_1, sz_2 >$		– memory with address size sz_1 and element size sz_2
	|	$type(v)$	M	a function that computes the type of a value

2.3 Bitvector syntax

We define a type of *words*, which are concrete bitvectors represented by a pair of value and size. Many operations on words are required by the semantics and are listed below.

Operations marked with `sbv` are signed. All other operations are unsigned (if it does matter). Operations `ext` and `exts` performs extract/extend operation. The former is unsigned (i.e., it extends with zeros), the latter is signed. This operation extracts bits from a bitvector starting from *hi* and ending with *lo* bit (both ends included). If *hi* is greater than the bitwidth of the bitvector, then it is extended with zeros (for `ext` operation) or with a sign bit (for `exts`) operation.

word, w	::=			
	|	$num : nat$	M	
	|	(w)	S	
	|	$1 : nat$	S	
	|	**true**	S	– sugar for 1:1
	|	**false**	S	– sugar for 0:1
	|	$w_1 \overset{bv}{+} w_2$	S	– plus
	|	$w_1 \overset{bv}{-} w_2$	S	– minus
	|	$w_1 \overset{bv}{*} w_2$	S	– times
	|	$w_1 \overset{bv}{/} w_2$	S	– division
	|	$w_1 \overset{sbv}{/} w_2$	S	– signed division
	|	$w_1 \overset{bv}{\%} w_2$	S	– modulo
	|	$w_1 \overset{sbv}{\%} w_2$	S	– signed modulo
	|	$w_1 \overset{bv}{\ll} w_2$	S	– logical shift left
	|	$w_1 \overset{bv}{\gg} w_2$	S	– logical shift right
	|	$w_1 \overset{bv}{\ggg} w_2$	S	– arithmetic shift right
	|	$w_1 \overset{bv}{\&} w_2$	S	– bitwise and
	|	$w_1 \overset{bv}{\mid} w_2$	S	– bitwise or
	|	$w_1 \overset{bv}{xor} w_2$	S	– bitwise xor
	|	$w_1 \overset{bv}{<} w_2$	S	– less than
	|	$w_1 \overset{sbv}{<} w_2$	S	– signed less than
	|	$\overset{bv}{-} w$	S	– integer negation
	|	$\overset{bv}{\sim} w$	S	– logical negation
	|	$w_1 \overset{bv}{\cdot} w_2$	S	– concatenation
	|	**ext** $w \sim$ **hi** $: sz_1 \sim$ **lo** $: sz_2$	S	– extract/extend
	|	**exts** $w \sim$ **hi** $: sz_1 \sim$ **lo** $: sz_2$	S	– signed extract/extend

2.4 Value syntax

Values are syntactic subset of expressions. They are used to represent expressions that are not reducible.

We have three kinds of values — immediates, represented as bitvectors; unknown values and storages (memories in BIL parlance), represented symbolically as a list of assignments:

$\mathit{val},\ v$	$::=$		
	$\mid$	word	
	$\mid$	$v[w \leftarrow v' : \mathit{sz}]$	
	$\mid$	$\textbf{unknown}\,[\mathit{string}] : \mathit{type}$	

2.5 Context syntax

Contexts are used in the typing judgments to specify the types of all variables. While each variable is annotated with its type, the context ensures that all uses of a given variable have the same type.

Γ	$::=$			
	$\mid$	$[]$		– empty
	$\mid$	Γ, var		– extend
	$\mid$	$[\mathit{var}]$	S	– singleton list
	$\mid$	(Γ)	S	
	$\mid$	$\mathsf{dom}(\Delta)$	M	– domain of a runtime binding context

2.6 Formula syntax

The following syntax is used to specify symbolic formulas in premises of judgments.

We use Δ to denote set of bindings of variables to values. The Δ context is represented as list of pairs. We write $(\mathit{var}, v) \in \Delta$ to indicate that the value v is the right-most binding of var in Δ. Additionally, we write $\mathsf{dom}(\Delta)$ for Δ's *domain* (the set of variables for which it contains values).

We also add a small set of operations over natural numbers, like comparison and arithmetics. Natural numbers are mostly used to reason about sizes of bitvectors, that's why they are often referred as sz.

We also add syntax for equality comparison for values and variables.

Δ	$::=$		
	$\mid$	$[]$	– empty
	$\mid$	$\Delta[\mathit{var} \leftarrow \mathit{val}]$	– extend

$\mathit{formula}$	$::=$		
	$\mid$	$\mathit{judgement}$	
	$\mid$	$(\mathit{formula})$	M
	$\mid$	$v_1 \neq v_2$	M
	$\mid$	$\mathit{var}_1 \neq \mathit{var}_2$	M
	$\mid$	$w_1 <> w_2$	M
	$\mid$	$\mathit{nat}_1 > \mathit{nat}_2$	M
	$\mid$	$\mathit{nat}_1 = \mathit{nat}_2$	M
	$\mid$	$\mathit{nat}_1 >= \mathit{nat}_2$	M
	$\mid$	$\mathit{nat}_1 \% \mathit{sz} = 0$	M
	$\mid$	$t_1 = t_2$	M
	$\mid$	$e_1 \overset{\text{def}}{:=} e_2$	M

$\vert$	$(var, val) \in \Delta$	M
$\vert$	$var \notin \mathsf{dom}(\Delta)$	M
$\vert$	$var \in \Gamma$	M
$\vert$	$id \notin \mathsf{dom}(\Gamma)$	M

$nat,\ sz$	$::=$		
	$\vert$	0	M
	$\vert$	1	M
	$\vert$	8	M
	$\vert$	$nat_1 + nat_2$	M
	$\vert$	$nat_1 - nat_2$	M
	$\vert$	(nat)	M

2.7 Instruction syntax

To reason about the whole program we introduce a syntax for instruction. An instruction is a binary sequence of w_2 bytes, that was read by a decoder from an address w_1. The semantics of an instruction is described by the *bil* program.

$insn$	$::=$		
	$\vert$	$\{\mathbf{addr} = w_1;\ \mathbf{size} = w2;\ \mathbf{code} = bil\}$	S

3 Typing

This section defines typing rules for BIL programs. We define four judgements: $\Gamma \vdash \mathit{bil}\ \textbf{is ok}$ for programs, $\Gamma \vdash s\ \textbf{is ok}$ for statements, $\Gamma \vdash e :: t$ for expressions, and $\Gamma\ \textbf{is ok}$ for contexts.

BIL statement-level variables represent global state, and are implicitly declared at their first use. While variables carry their type with them, it is still necessary to track them in a context during type checking. This is required to rule out programs like:

```
if (foo) then {x:imm<1> = 0} else {x:imm<32> = 42};
bar
```

Such a program would leave the type associated with `x` unclear in `bar`. This is ruled out because the typing rules are set up such that $\Gamma \vdash \mathit{bil}\ \textbf{is ok}$ implies $\Gamma\ \textbf{is ok}$, which requires that each variable has a single type.

$\boxed{\Gamma \vdash \mathit{bil}\ \textbf{is ok}}$

$$\frac{\Gamma \vdash \mathit{stmt}\ \textbf{is ok}}{\Gamma \vdash \{\mathit{stmt}\}\ \textbf{is ok}}$$ ~~T_SEQ_ONE~~

$$\frac{\begin{array}{c}\Gamma \vdash s_1\ \textbf{is ok}\\ \Gamma \vdash \{s_2;\ ..;\ s_n\}\ \textbf{is ok}\end{array}}{\Gamma \vdash \{s_1;\ s_2;\ ..;\ s_n\}\ \textbf{is ok}}$$ T_SEQ_REC

$\boxed{\Gamma \vdash \mathit{stmt}\ \textbf{is ok}}$

$$\frac{\begin{array}{c}\Gamma \vdash \mathit{var} :: t\\ \Gamma \vdash \mathit{exp} :: t\end{array}}{\Gamma \vdash \mathit{var} := \mathit{exp}\ \textbf{is ok}}$$ T_MOVE

$$\frac{\Gamma \vdash \mathit{exp} :: \textbf{imm}\ < \mathit{nat} >}{\Gamma \vdash \textbf{jmp}\ \mathit{exp}\ \textbf{is ok}}$$ T_JMP

$$\frac{\Gamma\ \textbf{is ok}}{\Gamma \vdash \textbf{cpuexn}\,(\mathit{num})\ \textbf{is ok}}$$ T_CPUEXN

$$\frac{\Gamma\ \textbf{is ok}}{\Gamma \vdash \textbf{special}\,(\mathit{str})\ \textbf{is ok}}$$ T_SPECIAL

$$\frac{\begin{array}{c}\Gamma \vdash e :: \textbf{imm}\ < 1 >\\ \Gamma \vdash \mathit{seq}\ \textbf{is ok}\end{array}}{\Gamma \vdash \textbf{while}\,(e)\mathit{seq}\ \textbf{is ok}}$$ T_WHILE

$$\frac{\begin{array}{c}\Gamma \vdash e :: \textbf{imm}\ < 1 >\\ \Gamma \vdash \mathit{seq}\ \textbf{is ok}\end{array}}{\Gamma \vdash \textbf{if}\ (e)\mathit{seq}\ \textbf{is ok}}$$ T_IFTHEN

$$\frac{\begin{array}{c}\Gamma \vdash e :: \textbf{imm}\ < 1 >\\ \Gamma \vdash \mathit{seq}_1\ \textbf{is ok}\\ \Gamma \vdash \mathit{seq}_2\ \textbf{is ok}\end{array}}{\Gamma \vdash \textbf{if}\ (e)\mathit{seq}_1\ \textbf{else}\ \mathit{seq}_2\ \textbf{is ok}}$$ T_IF

$\boxed{\Gamma \vdash \mathit{exp} :: \mathit{type}}$

$$\frac{\begin{array}{c}\mathit{id} : t\ \in\ \Gamma\\ \Gamma\ \textbf{is ok}\end{array}}{\Gamma \vdash \mathit{id} : t :: t}$$ T_VAR

$$\frac{\begin{array}{c} sz > 0 \\ \Gamma\ \mathbf{is\ ok} \end{array}}{\Gamma \vdash num : sz :: \mathbf{imm}\ < sz >}\ \text{T\_INT}$$

$$\frac{\begin{array}{l} sz > 0 \\ nat > 0 \\ \Gamma \vdash v :: \mathbf{mem}\ < nat, sz > \\ \Gamma \vdash v' :: \mathbf{imm}\ < sz > \end{array}}{\Gamma \vdash v[num_1 : nat \leftarrow v' : sz] :: \mathbf{mem}\ < nat, sz >}\ \text{T\_MEM}$$

$$\frac{\begin{array}{l} sz' \% sz = 0 \\ sz' > 0 \\ \Gamma \vdash e_1 :: \mathbf{mem}\ < nat, sz > \\ \Gamma \vdash e_2 :: \mathbf{imm}\ < nat > \end{array}}{\Gamma \vdash e_1[e_2, ed] : sz' :: \mathbf{imm}\ < sz' >}\ \text{T\_LOAD}$$

$$\frac{\begin{array}{l} sz' \% sz = 0 \\ sz' > 0 \\ \Gamma \vdash e_1 :: \mathbf{mem}\ < nat, sz > \\ \Gamma \vdash e_2 :: \mathbf{imm}\ < nat > \\ \Gamma \vdash e_3 :: \mathbf{imm}\ < sz' > \end{array}}{\Gamma \vdash e_1\ \mathbf{with}\ [e_2, ed] : sz' \leftarrow e_3 :: \mathbf{mem}\ < nat, sz >}\ \text{T\_STORE}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 :: \mathbf{imm}\ < sz > \\ \Gamma \vdash e_2 :: \mathbf{imm}\ < sz > \end{array}}{\Gamma \vdash e_1\ aop\ e_2 :: \mathbf{imm}\ < sz >}\ \text{T\_AOP}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 :: \mathbf{imm}\ < sz > \\ \Gamma \vdash e_2 :: \mathbf{imm}\ < sz > \end{array}}{\Gamma \vdash e_1\ lop\ e_2 :: \mathbf{imm}\ < 1 >}\ \text{T\_LOP}$$

$$\frac{\Gamma \vdash e_1 :: \mathbf{imm}\ < sz >}{\Gamma \vdash uop\ e_1 :: \mathbf{imm}\ < sz >}\ \text{T\_UOP}$$

$$\frac{\begin{array}{l} sz > 0 \\ sz >= nat \\ \Gamma \vdash e :: \mathbf{imm}\ < nat > \end{array}}{\Gamma \vdash widen\_cast : sz[e] :: \mathbf{imm}\ < sz >}\ \text{T\_CAST\_WIDEN}$$

$$\frac{\begin{array}{l} sz > 0 \\ nat >= sz \\ \Gamma \vdash e :: \mathbf{imm}\ < nat > \end{array}}{\Gamma \vdash narrow\_cast : sz[e] :: \mathbf{imm}\ < sz >}\ \text{T\_CAST\_NARROW}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 :: t \\ \Gamma, id : t \vdash e_2 :: t' \end{array}}{\Gamma \vdash \mathbf{let}\ id : t = e_1\ \mathbf{in}\ e_2 :: t'}\ \text{T\_LET}$$

$$\frac{\begin{array}{c} t\ \mathbf{is\ ok} \\ \Gamma\ \mathbf{is\ ok} \end{array}}{\Gamma \vdash \mathbf{unknown}\ [str] : t :: t}\ \text{T\_UNKNOWN}$$

$$\frac{\begin{array}{l} \Gamma \vdash e_1 :: \mathbf{imm}\ < 1 > \\ \Gamma \vdash e_2 :: t \\ \Gamma \vdash e_3 :: t \end{array}}{\Gamma \vdash \mathbf{ite}\ e_1\ e_2\ e_3 :: t}\ \text{T\_ITE}$$

$$\frac{\begin{array}{c}\Gamma \vdash e :: \mathbf{imm}\ < sz > \\ sz_1 >= sz_2\end{array}}{\Gamma \vdash \mathbf{extract}\ : sz_1 : sz_2[e] :: \mathbf{imm}\ < sz_1 - sz_2 + 1 >} \quad \text{T\_EXTRACT}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 :: \mathbf{imm}\ < sz_1 > \\ \Gamma \vdash e_2 :: \mathbf{imm}\ < sz_2 >\end{array}}{\Gamma \vdash e_1 @ e_2 :: \mathbf{imm}\ < sz_1 + sz_2 >} \quad \text{T\_CONCAT}$$

$\boxed{t\ \textbf{is ok}}$

$$\frac{sz > 0}{\mathbf{imm}\ < sz >\ \textbf{is ok}} \quad \text{TWF\_IMM}$$

$$\frac{\begin{array}{c}nat > 0 \\ sz > 0\end{array}}{\mathbf{mem}\ < nat, sz >\ \textbf{is ok}} \quad \text{TWF\_MEM}$$

$\boxed{\Gamma\ \textbf{is ok}}$

$$\frac{}{[]\ \textbf{is ok}} \quad \text{TG\_NIL}$$

$$\frac{\begin{array}{l}id \notin \mathsf{dom}(\Gamma) \\ t\ \textbf{is ok} \\ \Gamma\ \textbf{is ok}\end{array}}{(\Gamma, id : t)\ \textbf{is ok}} \quad \text{TG\_CONS}$$

4 Operational semantics

4.1 Model of a program

Program is coinductively defined as an infinite stream of program states, produced by a step rule. Each state is represented with a triplet (Δ, w, var), where Δ is a mapping from variables to values, w is a program counter, and var is a variable denoting currently active memory.

The `step` rule defines how a machine instruction is evaluated. We use "magic" rule `decode` that fetches instructions from the memory and decodes them to a BIL program.

The BIL code is evaluated using reduction rules of statements (see section 5). Then the program counter is updated with the w_3, that initially points to a byte following current instruction.

$\boxed{\Delta, w, var \rightsquigarrow \Delta', w', var'}$

$$\frac{\begin{array}{c} delta, w, var \mapsto \{\,\mathbf{addr} = w_1;\ \mathbf{size} = w2;\ \mathbf{code} = bil\} \\ \Delta, w_1 \overset{bv}{+} w_2 \vdash bil \rightsquigarrow \Delta', w_3, \{\,\} \end{array}}{\Delta, w, var \rightsquigarrow \Delta', w_3, var} \quad \text{STEP}$$

$\boxed{delta, w, var \mapsto insn}$

$$\frac{}{delta, w, var \mapsto insn} \quad \text{DECODE}$$

5 Semantics of statements

The reduction rule defines transformation of a state for each statement. The state of the reduction rule consists of a pair (Δ, w), where Δ is a mapping from variables to values and w is an address of a next instruction.

Two statements affect the state: `Move` statement introduces new $var \leftarrow v$ binding in Δ, and `Jmp` affects program counter.

The `if` and `while` instructions introduce local control flow.

There is no special semantics associated with `special` and `cpuexn` statements.

$\boxed{\Delta, word \vdash stmt \rightsquigarrow \Delta', word'}$

$$\frac{\Delta \vdash e \rightsquigarrow^* v}{\Delta, w \vdash var := e \rightsquigarrow \Delta[var \leftarrow v], w} \quad \text{MOVE}$$

$$\frac{\Delta \vdash e \rightsquigarrow^* w'}{\Delta, w \vdash \mathbf{jmp}\ e \rightsquigarrow \Delta, w'} \quad \text{JMP}$$

$$\frac{}{\Delta, w \vdash \mathbf{cpuexn}\,(num) \rightsquigarrow \Delta, w} \quad \text{CPUEXN}$$

$$\frac{}{\Delta, w \vdash \mathbf{special}\,(str) \rightsquigarrow \Delta, w} \quad \text{SPECIAL}$$

$$\frac{\begin{array}{c} \Delta \vdash e \rightsquigarrow^* \mathbf{true} \\ \Delta, word \vdash seq \rightsquigarrow \Delta', word', \{\,\} \end{array}}{\Delta, word \vdash \mathbf{if}\ (e) seq \rightsquigarrow \Delta', word'} \quad \text{IFTHEN\_TRUE}$$

$$\frac{\begin{array}{c} \Delta \vdash e \rightsquigarrow^* \mathbf{true} \\ \Delta, word \vdash seq \rightsquigarrow \Delta', word', \{\,\} \end{array}}{\Delta, word \vdash \mathbf{if}\ (e) seq\ \mathbf{else}\ seq_1 \rightsquigarrow \Delta', word'} \quad \text{IF\_TRUE}$$

$$\frac{\begin{array}{c} \Delta \vdash e \rightsquigarrow^* \mathbf{false} \\ \Delta, word \vdash seq \rightsquigarrow \Delta', word', \{\,\} \end{array}}{\Delta, word \vdash \mathbf{if}\ (e) seq_1\ \mathbf{else}\ seq \rightsquigarrow \Delta', word'} \quad \text{IF\_FALSE}$$

$$\frac{\begin{array}{l}\Delta_1 \vdash e \leadsto^* \textbf{true}\\ \Delta_1, \mathit{word}_1 \vdash \mathit{seq} \leadsto \Delta_2, \mathit{word}_2, \{\,\}\\ \Delta_2, \mathit{word}_2 \vdash \textbf{while}\,(e)\mathit{seq} \leadsto \Delta_3, \mathit{word}_3\end{array}}{\Delta_1, \mathit{word}_1 \vdash \textbf{while}\,(e)\mathit{seq} \leadsto \Delta_3, \mathit{word}_3} \quad \text{WHILE}$$

$$\frac{\Delta \vdash e \leadsto^* \textbf{false}}{\Delta, \mathit{word} \vdash \textbf{while}\,(e)\mathit{seq} \leadsto \Delta, \mathit{word}} \quad \text{WHILE\_FALSE}$$

$\boxed{\Delta, \mathit{word} \vdash \mathit{seq} \leadsto \Delta', \mathit{word}', \mathit{seq}'}$

$$\frac{\Delta, \mathit{word} \vdash s_1 \leadsto \Delta', \mathit{word}'}{\Delta, \mathit{word} \vdash \{s_1; s_2; \,..\,; s_n\} \leadsto \Delta', \mathit{word}', \{s_2; \,..\,; s_n\}} \quad \text{SEQ\_REC}$$

$$\frac{\Delta, \mathit{word} \vdash s_1 \leadsto \Delta', \mathit{word}'}{\Delta, \mathit{word} \vdash \{s_1; s_2\} \leadsto \Delta', \mathit{word}', \{s_2\}} \quad \text{SEQ\_LAST}$$

$$\frac{\Delta, \mathit{word} \vdash s_1 \leadsto \Delta', \mathit{word}'}{\Delta, \mathit{word} \vdash \{s_1\} \leadsto \Delta', \mathit{word}', \{\,\}} \quad \text{SEQ\_ONE}$$

$$\frac{}{\Delta, \mathit{word} \vdash \{\,\} \leadsto \Delta, \mathit{word}, \{\,\}} \quad \text{SEQ\_NIL}$$

6 Semantics of expressions

This section describes a small step operational semantics for expressions. A symbolic formula $\Delta \vdash e \to e'$ defines a step of transformation from expression e to an expression e' under given context Δ.

A well formed (well typed) expression evaluates to a value expression, that is syntactic subset of expression grammar (see section 2.4).

A value can be either an immediate, represented by a bitvector, a unknown value, or a memory storage.

A memory storage is represented symbolically as a sequence of storages to the originally undefined memory. Each storage operation of size greater than 8 bits is desugared into a sequence of 8 bit storages in a big endian order.

A load operation will first reduce all sub expressions of a memory object to values and then recursively destruct the object until one of the following conditions is met:

load-byte: if the memory object is a storage of a `value` to an immediate (known) address that we're trying to load then the load expression is reduced to `value`.

load-un-memory: if the memory object is an `unknown` value, then the load expression evaluates to `unknown`.

load-un-addr: if the memory object is a storage to `unknown` value address then load expression evaluates to `unknown`.

This section also defines $\Delta \vdash e_1 \leadsto^* e_2$. This relation is the reflexive, transitive closure of $\leadsto$. It is useful to describe reductions that may take many steps. For example, in the evaluation rules for statements, it is often necessary to evaluate an expression completely to a value. The $\leadsto^*$ relation is used to allow reductions that take an indeterminate number of individual $\leadsto$ steps. Such a derivation can be built with repeated use of the REDUCE rule.

Some evaluation rules depend on the type of a value. Since there are two canonical forms for each type, we avoid duplicating each rule by defining the following metafunction:

$\boxed{\mathit{type}(v)}$ a function that computes the type of a value

$$\begin{array}{l} \mathit{type}(v[\mathit{num}_1 : \mathit{nat} \leftarrow v' : \mathit{sz}]) \equiv \textbf{mem} < \mathit{nat}, \mathit{sz} > \\ \mathit{type}(\mathit{num} : \mathit{nat}) \equiv \textbf{imm} < \mathit{nat} > \\ \mathit{type}(\textbf{unknown}\,[\mathit{str}] : t) \equiv t \end{array}$$

$$\boxed{\Delta \vdash \mathit{exp} \leadsto \mathit{exp}'}$$

$$\frac{(\mathit{var}, v) \in \Delta}{\Delta \vdash \mathit{var} \leadsto v} \quad \text{VAR\_IN}$$

$$\frac{\mathit{id} : \mathit{type} \notin \mathsf{dom}(\Delta)}{\Delta \vdash \mathit{id} : \mathit{type} \leadsto \textbf{unknown}\,[\mathit{str}] : \mathit{type}} \quad \text{VAR\_UNKNOWN}$$

$$\frac{\Delta \vdash e_2 \leadsto e_2'}{\Delta \vdash e_1[e_2, \mathit{ed}] : \mathit{sz} \leadsto e_1[e_2', \mathit{ed}] : \mathit{sz}} \quad \text{LOAD\_STEP\_ADDR}$$

$$\frac{\Delta \vdash e_1 \leadsto e_1'}{\Delta \vdash e_1[v_2, \mathit{ed}] : \mathit{sz} \leadsto e_1'[v_2, \mathit{ed}] : \mathit{sz}} \quad \text{LOAD\_STEP\_MEM}$$

$$\frac{}{\Delta \vdash v[w \leftarrow v' : \mathit{sz}][w, \mathit{ed}'] : \mathit{sz} \leadsto v'} \quad \text{LOAD\_BYTE}$$

$$\frac{w_1 \neq w_2}{\Delta \vdash v[w_1 \leftarrow v' : \mathit{sz}][w_2, \mathit{ed}] : \mathit{sz} \leadsto v[w_2, \mathit{ed}] : \mathit{sz}} \quad \text{LOAD\_BYTE\_FROM\_NEXT}$$

$$\frac{}{\Delta \vdash (\textbf{unknown}\,[\mathit{str}] : t)[v, \mathit{ed}] : \mathit{sz} \leadsto \textbf{unknown}\,[\mathit{str}] : \textbf{imm} < \mathit{sz} >} \quad \text{LOAD\_UN\_MEM}$$

$$\frac{}{\Delta \vdash (v[w_1 \leftarrow v' : \mathit{sz}])[\textbf{unknown}\,[\mathit{str}] : t, \mathit{ed}] : \mathit{sz}' \leadsto \textbf{unknown}\,[\mathit{str}] : \textbf{imm} < \mathit{sz}' >} \quad \text{LOAD\_UN\_ADDR}$$

$$\frac{\begin{array}{l} \mathit{sz}' > \mathit{sz} \\ \textbf{succ}\, w = w' \\ \mathit{type}(v) = \textbf{mem} < \mathit{nat}, \mathit{sz} > \end{array}}{\Delta \vdash v[w, \textbf{be}] : \mathit{sz}' \leadsto v[w, \textbf{be}] : \mathit{sz}@(v[w', \textbf{be}] : (\mathit{sz}' - \mathit{sz}))} \quad \text{LOAD\_WORD\_BE}$$

$$\frac{\begin{array}{l} \mathit{sz}' > \mathit{sz} \\ \textbf{succ}\, w = w' \\ \mathit{type}(v) = \textbf{mem} < \mathit{nat}, \mathit{sz} > \end{array}}{\Delta \vdash v[w, \textbf{el}] : \mathit{sz}' \leadsto v[w', \textbf{el}] : (\mathit{sz}' - \mathit{sz})@(v[w, \textbf{be}] : \mathit{sz})} \quad \text{LOAD\_WORD\_EL}$$

$$\frac{\Delta \vdash e_3 \leadsto e_3'}{\Delta \vdash e_1 \,\textbf{with}\, [e_2, \mathit{ed}] : \mathit{sz} \leftarrow e_3 \leadsto e_1 \,\textbf{with}\, [e_2, \mathit{ed}] : \mathit{sz} \leftarrow e_3'} \quad \text{STORE\_STEP\_VAL}$$

$$\frac{\Delta \vdash e_2 \leadsto e_2'}{\Delta \vdash e_1 \,\textbf{with}\, [e_2, \mathit{ed}] : \mathit{sz} \leftarrow v_3 \leadsto e_1 \,\textbf{with}\, [e_2', \mathit{ed}] : \mathit{sz} \leftarrow v_3} \quad \text{STORE\_STEP\_ADDR}$$

$$\frac{\Delta \vdash e_1 \leadsto e_1'}{\Delta \vdash e_1 \,\textbf{with}\, [v_2, \mathit{ed}] : \mathit{sz} \leftarrow v_3 \leadsto e_1' \,\textbf{with}\, [v_2, \mathit{ed}] : \mathit{sz} \leftarrow v_3} \quad \text{STORE\_STEP\_MEM}$$

$$\frac{\begin{array}{l} \mathit{sz}' > \mathit{sz} \\ \textbf{succ}\, w = w' \\ \mathit{type}(v) = \textbf{mem} < \mathit{nat}, \mathit{sz} > \\ e_1 \overset{\text{def}}{:=} (v \,\textbf{with}\, [w, \textbf{be}] : \mathit{sz} \leftarrow \textbf{high} : \mathit{sz}[\mathit{val}]) \end{array}}{\Delta \vdash v \,\textbf{with}\, [w, \textbf{be}] : \mathit{sz}' \leftarrow \mathit{val} \leadsto e_1 \,\textbf{with}\, [w', \textbf{be}] : (\mathit{sz}' - \mathit{sz}) \leftarrow \textbf{low} : (\mathit{sz}' - \mathit{sz})[\mathit{val}]} \quad \text{STORE\_WORD\_BE}$$

$$\frac{\begin{array}{l} \mathit{sz}' > \mathit{sz} \\ \textbf{succ}\, w = w' \\ \mathit{type}(v) = \textbf{mem} < \mathit{nat}, \mathit{sz} > \\ e_1 \overset{\text{def}}{:=} (v \,\textbf{with}\, [w, \textbf{el}] : \mathit{sz} \leftarrow \textbf{low} : \mathit{sz}[\mathit{val}]) \end{array}}{\Delta \vdash v \,\textbf{with}\, [w, \textbf{el}] : \mathit{sz}' \leftarrow \mathit{val} \leadsto e_1 \,\textbf{with}\, [w', \textbf{el}] : (\mathit{sz}' - \mathit{sz}) \leftarrow \textbf{high} : (\mathit{sz}' - \mathit{sz})[\mathit{val}]} \quad \text{STORE\_WORD\_EL}$$

$$\frac{\mathit{type}(v) = \mathbf{mem}\ < \mathit{nat}, \mathit{sz} >}{\Delta \vdash v\ \mathbf{with}\,[w, \mathit{ed}] : \mathit{sz} \leftarrow v' \rightsquigarrow v[w \leftarrow v' : \mathit{sz}]} \quad \text{STORE\_VAL}$$

$$\frac{\mathit{type}(v) = t}{\Delta \vdash v\ \mathbf{with}\,[\mathbf{unknown}\,[\mathit{str}] : t', \mathit{ed}] : \mathit{sz}' \leftarrow v_2 \rightsquigarrow \mathbf{unknown}\,[\mathit{str}] : t} \quad \text{STORE\_UN\_ADDR}$$

$$\frac{\Delta \vdash e_1 \rightsquigarrow e_1'}{\Delta \vdash \mathbf{let}\ \mathit{var} = e_1\ \mathbf{in}\ e_2 \rightsquigarrow \mathbf{let}\ \mathit{var} = e_1'\ \mathbf{in}\ e_2} \quad \text{LET\_STEP}$$

$$\frac{}{\Delta \vdash \mathbf{let}\ \mathit{var} = v\ \mathbf{in}\ e \rightsquigarrow [v/\mathit{var}]e} \quad \text{LET}$$

$$\frac{\Delta \vdash e_1 \rightsquigarrow e_1'}{\Delta \vdash \mathbf{ite}\ e_1\ v_2\ v_3 \rightsquigarrow \mathbf{ite}\ e_1'\ v_2\ v_3} \quad \text{ITE\_STEP\_COND}$$

$$\frac{\Delta \vdash e_2 \rightsquigarrow e_2'}{\Delta \vdash \mathbf{ite}\ e_1\ e_2\ v_3 \rightsquigarrow \mathbf{ite}\ e_1\ e_2'\ v_3} \quad \text{ITE\_STEP\_THEN}$$

$$\frac{\Delta \vdash e_3 \rightsquigarrow e_3'}{\Delta \vdash \mathbf{ite}\ e_1\ e_2\ e_3 \rightsquigarrow \mathbf{ite}\ e_1\ e_2\ e_3'} \quad \text{ITE\_STEP\_ELSE}$$

$$\frac{}{\Delta \vdash \mathbf{ite}\,\mathbf{true}\ v_2\ v_3 \rightsquigarrow v_2} \quad \text{ITE\_TRUE}$$

$$\frac{}{\Delta \vdash \mathbf{ite}\,\mathbf{false}\ v_2\ v_3 \rightsquigarrow v_3} \quad \text{ITE\_FALSE}$$

$$\frac{\mathit{type}(v_2) = t'}{\Delta \vdash \mathbf{ite}\,\mathbf{unknown}\,[\mathit{str}] : t\ v_2\ v_3 \rightsquigarrow \mathbf{unknown}\,[\mathit{str}] : t'} \quad \text{ITE\_UNK}$$

$$\frac{\Delta \vdash e_2 \rightsquigarrow e_2'}{\Delta \vdash v_1\ \mathit{bop}\ e_2 \rightsquigarrow v_1\ \mathit{bop}\ e_2'} \quad \text{BOP\_RHS}$$

$$\frac{\Delta \vdash e_1 \rightsquigarrow e_1'}{\Delta \vdash e_1\ \mathit{bop}\ e_2 \rightsquigarrow e_1'\ \mathit{bop}\ e_2} \quad \text{BOP\_LHS}$$

$$\frac{}{\Delta \vdash e\ \mathit{aop}\ \mathbf{unknown}\,[\mathit{str}] : t \rightsquigarrow \mathbf{unknown}\,[\mathit{str}] : t} \quad \text{AOP\_UNK\_RHS}$$

$$\frac{}{\Delta \vdash \mathbf{unknown}\,[\mathit{str}] : t\ \mathit{aop}\ e \rightsquigarrow \mathbf{unknown}\,[\mathit{str}] : t} \quad \text{AOP\_UNK\_LHS}$$

$$\frac{}{\Delta \vdash e\ \mathit{lop}\ \mathbf{unknown}\,[\mathit{str}] : t \rightsquigarrow \mathbf{unknown}\,[\mathit{str}] : \mathbf{imm}\ < 1 >} \quad \text{LOP\_UNK\_RHS}$$

$$\frac{}{\Delta \vdash \mathbf{unknown}\,[\mathit{str}] : t\ \mathit{lop}\ e \rightsquigarrow \mathbf{unknown}\,[\mathit{str}] : \mathbf{imm}\ < 1 >} \quad \text{LOP\_UNK\_LHS}$$

$$\frac{}{\Delta \vdash w_1 + w_2 \rightsquigarrow w_1 \overset{bv}{+} w_2} \quad \text{PLUS}$$

$$\frac{}{\Delta \vdash w_1 - w_2 \rightsquigarrow w_1 \overset{bv}{-} w_2} \quad \text{MINUS}$$

$$\frac{}{\Delta \vdash w_1 * w_2 \rightsquigarrow w_1 \overset{bv}{*} w_2} \quad \text{TIMES}$$

$$\frac{}{\Delta \vdash w_1 / w_2 \rightsquigarrow w_1 \overset{bv}{/} w_2} \quad \text{DIV}$$

$$\frac{}{\Delta \vdash w_1 \overset{signed}{/} w_2 \rightsquigarrow w_1 \overset{sbv}{/} w_2} \quad \text{SDIV}$$

$$\frac{}{\Delta \vdash w_1 \,\%\, w_2 \rightsquigarrow w_1 \overset{bv}{\%} w_2} \quad \text{MOD}$$

$$\frac{}{\Delta \vdash w_1 \overset{signed}{\%} w_2 \rightsquigarrow w_1 \overset{sbv}{\%} w_2}\ \text{SMOD}$$

$$\frac{}{\Delta \vdash w_1 \ll w_2 \rightsquigarrow w_1 \overset{bv}{\ll} w_2}\ \text{LSL}$$

$$\frac{}{\Delta \vdash w_1 \gg w_2 \rightsquigarrow w_1 \overset{bv}{\gg} w_2}\ \text{LSR}$$

$$\frac{}{\Delta \vdash w_1 \ggg w_2 \rightsquigarrow w_1 \overset{bv}{\ggg} w_2}\ \text{ASR}$$

$$\frac{}{\Delta \vdash w_1 \,\&\, w_2 \rightsquigarrow w_1 \overset{bv}{\&} w_2}\ \text{LAND}$$

$$\frac{}{\Delta \vdash w_1 \,|\, w_2 \rightsquigarrow w_1 \overset{bv}{|} w_2}\ \text{LOR}$$

$$\frac{}{\Delta \vdash w_1 \,\mathbf{xor}\, w_2 \rightsquigarrow w_1 \overset{bv}{xor} w_2}\ \text{XOR}$$

$$\frac{}{\Delta \vdash w = w \rightsquigarrow \mathbf{true}}\ \text{EQ\_SAME}$$

$$\frac{w_1 <> w_2}{\Delta \vdash w_1 = w_2 \rightsquigarrow \mathbf{false}}\ \text{EQ\_DIFF}$$

$$\frac{}{\Delta \vdash w \neq w \rightsquigarrow \mathbf{false}}\ \text{NEQ\_SAME}$$

$$\frac{w_1 <> w_2}{\Delta \vdash w_1 \neq w_2 \rightsquigarrow \mathbf{true}}\ \text{NEQ\_DIFF}$$

$$\frac{}{\Delta \vdash w_1 < w_2 \rightsquigarrow w_1 \overset{bv}{<} w_2}\ \text{LESS}$$

$$\frac{}{\Delta \vdash w_1 \leq w_2 \rightsquigarrow (w_1 < w_2) \,|\, (w_1 = w_2)}\ \text{LESS\_EQ}$$

$$\frac{}{\Delta \vdash w_1 \overset{signed}{<} w_2 \rightsquigarrow w_1 \overset{sbv}{<} w_2}\ \text{SIGNED\_LESS}$$

$$\frac{}{\Delta \vdash w_1 \overset{signed}{\leq} w_2 \rightsquigarrow (w_1 = w_2) \,\&\, (w_1 \overset{signed}{<} w_2)}\ \text{SIGNED\_LESS\_EQ}$$

$$\frac{\Delta \vdash e \rightsquigarrow e'}{\Delta \vdash uop\ e \rightsquigarrow uop\ e'}\ \text{UOP}$$

$$\frac{}{\Delta \vdash uop\, \mathbf{unknown}\,[str] : t \rightsquigarrow \mathbf{unknown}\,[str] : t}\ \text{UOP\_UNK}$$

$$\frac{}{\Delta \vdash \neg\, w \rightsquigarrow \overset{bv}{\sim} w}\ \text{NOT}$$

$$\frac{}{\Delta \vdash -\, w \rightsquigarrow \overset{bv}{-} w}\ \text{NEG}$$

$$\frac{\Delta \vdash e_2 \rightsquigarrow e_2'}{\Delta \vdash e_1 @ e_2 \rightsquigarrow e_1 @ e_2'}\ \text{CONCAT\_RHS}$$

$$\frac{\Delta \vdash e_1 \rightsquigarrow e_1'}{\Delta \vdash e_1 @ v_2 \rightsquigarrow e_1' @ v_2}\ \text{CONCAT\_LHS}$$

$$\frac{type(v_2) = \mathbf{imm} < sz_2 >}{\Delta \vdash \mathbf{unknown}\,[str] : \mathbf{imm} < sz_1 > @ v_2 \rightsquigarrow \mathbf{unknown}\,[str] : \mathbf{imm} < sz_1 + sz_2 >}\ \text{CONCAT\_LHS\_UN}$$

$$\frac{type(v_1) = \mathbf{imm}\ < sz_1 >}{\Delta \vdash v_1 @ \mathbf{unknown}\,[str] : \mathbf{imm}\ < sz_2 > \rightsquigarrow \mathbf{unknown}\,[str] : \mathbf{imm}\ < sz_1 + sz_2 >} \quad \text{CONCAT\_RHS\_UN}$$

$$\frac{}{\Delta \vdash w_1 @ w_2 \rightsquigarrow w_1 \overset{bv}{\cdot} w_2} \quad \text{CONCAT}$$

$$\frac{\Delta \vdash e \rightsquigarrow e'}{\Delta \vdash \mathbf{extract}\ : sz_1 : sz_2[e] \rightsquigarrow \mathbf{extract}\ : sz_1 : sz_2[e']} \quad \text{EXTRACT\_REDUCE}$$

$$\frac{}{\Delta \vdash \mathbf{extract}\ : sz_1 : sz_2[\mathbf{unknown}\,[str] : t] \rightsquigarrow \mathbf{unknown}\,[str] : \mathbf{imm}\ < (sz_1 - sz_2) + 1 >} \quad \text{EXTRACT\_UN}$$

$$\frac{}{\Delta \vdash \mathbf{extract}\ : sz_1 : sz_2[w] \rightsquigarrow \mathbf{ext}\ w \sim \mathbf{hi} : sz_1 \sim \mathbf{lo} : sz_2} \quad \text{EXTRACT}$$

$$\frac{\Delta \vdash e \rightsquigarrow e'}{\Delta \vdash cast : sz[e] \rightsquigarrow cast : sz[e']} \quad \text{CAST\_REDUCE}$$

$$\frac{}{\Delta \vdash cast : sz[\mathbf{unknown}\,[str] : t] \rightsquigarrow \mathbf{unknown}\,[str] : \mathbf{imm}\ < sz >} \quad \text{CAST\_UNK}$$

$$\frac{}{\Delta \vdash \mathbf{low} : sz[w] \rightsquigarrow \mathbf{ext}\ w \sim \mathbf{hi} : (sz - 1) \sim \mathbf{lo} : 0} \quad \text{CAST\_LOW}$$

$$\frac{}{\Delta \vdash \mathbf{high} : sz[num : sz'] \rightsquigarrow \mathbf{ext}\ num : sz' \sim \mathbf{hi} : (sz' - 1) \sim \mathbf{lo} : (sz' - sz)} \quad \text{CAST\_HIGH}$$

$$\frac{}{\Delta \vdash \mathbf{signed} : sz[w] \rightsquigarrow \mathbf{exts}\ w \sim \mathbf{hi} : (sz - 1) \sim \mathbf{lo} : 0} \quad \text{CAST\_SIGNED}$$

$$\frac{}{\Delta \vdash \mathbf{unsigned} : sz[w] \rightsquigarrow \mathbf{ext}\ w \sim \mathbf{hi} : (sz - 1) \sim \mathbf{lo} : 0} \quad \text{CAST\_UNSIGNED}$$

$\boxed{\mathbf{succ}\, w_1 = exp}$

$$\frac{}{\mathbf{succ}\, num : sz = num : sz \overset{bv}{+} 1 : sz} \quad \text{SUCC}$$

$\boxed{\Delta \vdash exp \rightsquigarrow^* exp'}$

$$\frac{}{\Delta \vdash e \rightsquigarrow^* e} \quad \text{REFL}$$

$$\frac{\begin{array}{c}\Delta \vdash e_1 \rightsquigarrow e_2 \\ \Delta \vdash e_2 \rightsquigarrow^* e_3\end{array}}{\Delta \vdash e_1 \rightsquigarrow^* e_3} \quad \text{REDUCE}$$