0.1 Operational Semantics

The *V* language is evaluated using a big-step evaluation with environments. This evaluation reduces an expression into a value directly, not necessarily having a rule of evaluation for every possible expression. To stop programmers from creating programs that cannot be evaluated, a type inference system will be specified later.

Value A value is the result of the evaluation of an expression in big-step. This set of values is different from the set of expressions of *V*, even though they share many similarities.

Environment An evaluation environment is a 2-tuple which contains the following information:

1. Arity of constructors

   If a constructor has arity n, it requires n arguments to become fully evaluated.

2. Associations between identifiers and values

   A new association is created every time a value is bound. This happens in `let` declarations, function application and `match` expressions.

Below are the definitions of both values and environments:

$$
\begin{array}{lcll}
\mathit{env} & ::= & (\mathit{arities}, \mathit{vars}) & \\
 & & & \\
\mathit{arities} & ::= & \{\} \mid \{\mathit{con} \to n\} \cup \mathit{arities} & (n \in \mathbb{N}) \\
\mathit{vars} & ::= & \{\} \mid \{x \to v\} \cup \mathit{vars} & \\
 & & & \\
v & ::= & \mathit{con}\ v_1,\ \ldots\ v_n & (n = \text{arity } \mathit{con}) \\
 & \mid & \mathit{raise} & \\
 & \mid & \{l_1 : v_1,\ \ldots\ l_n : v_n\} & (n \geq 1) \\
 & \mid & \#\mathit{path} & \\
 & \mid & \langle \mathit{func}, \mathit{env} \rangle & \\
 & \mid & \ll \mathit{Builtin}\ .\ v_1,\ \ldots v_n \gg & (n < \text{arity } \mathit{Builtin}) \\
 & \mid & \ll \mathit{con}\ .\ v_1,\ \ldots v_n \gg & (n < \text{arity } \mathit{con}) \\
 & & & \\
\mathit{path} & ::= & l & \\
 & \mid & \mathit{path}\ .\ \mathit{path} & \\
 & \mid & (\mathit{path}_1,\ \ldots\ \mathit{path}_n) & (n \geq 2) \\
 & \mid & \mathit{path}\ [v_1,\ v_2] &
\end{array}
$$

The value $\langle \mathit{func}, \mathit{env} \rangle$ defines closures. They represent the result of evaluating functions (and recursive functions) and store the environment at the moment of evaluation. This means that *V* has static scope, since closures capture the environment at the moment of evaluation and *V* has eager evaluation.

The values $\ll$ *Builtin* . $v_1, \ldots v_n \gg$ and $\ll$ *con* . $v_1, \ldots v_n \gg$ are partial applications of built-in functions and constructors, respectively.

Once all the parameters have been defined, they evaluate either to the result of the function (in the case of *Builtin*) or to a fully applied constructor *con* $v_1, \ldots v_n$.

0.1.1 Paths

Accessors possess structure when treated as values. This structure is built through use of the operations available on accessors, such as the compose expression or the built-in functions. Since accessors view records as trees, this structure is a *path* along a tree.

The most basic structure of a path is a single label. This path describes a field immediately below the root of the tree (the root is viewed as the record itself, and every child is a field of the record). This path is created by using the simple accessor expression #*l*.

Two paths can be composed vertically, allowing access to subfields of a record. In this scenario, the tree must have at least the same depth as the path (and along the correct field names). Vertical composition is achieved by using the "stack" built-in function, and always combines two paths.

Paths can also be composed horizontally, described as a tuple of paths. When composed horizontally, all paths are used on the root of the record, and the end points of the paths are joined in a tuple for extraction or updating.

Finally, paths can be distorted. This means that the path has two values (functions) associated with it. These functions are then used to transform the values stored in the field of the record (one for extraction, another for updating).

Path Traversal Rules As previously stated, accessors describe paths along a record tree. To use these paths, an auxiliary *traverse* function is used. This functions receives 3 arguments: a path, a record and an update value. The function returns 2 values: the old value associated with the field specified by the path; and an updated record.

This updated record uses the value provided as input to the function to update the field specified by the path. This last argument of the *traverse* function can be omitted and, in such a case, no update is done (that is, the updated record is the same as the input record).

The first rule is for a simple label path. The label must be present in the provided record. A new record is created, associating the provided value with the label specified by the path.

$$\frac{1 \leq \|k\| \leq \|n\| \qquad r = \{l_1 : v_1, \ \ldots \ l_k : v, \ \ldots \ l_n : v_n\}}{traverse(l_k, \{l_1 : v_1, ...l_n : v_n\}, v) = v_k, r}$$

For vertically composed paths, three calls to *traverse* are needed.

$$\frac{\begin{array}{c} traverse(path_1, \{l_1 : v_1, ...l_n : v_n\}) = rec, r \\ traverse(path_2, rec, v) = v', rec' \\ traverse(path_1, \{l_1 : v_1, ...l_n : v_n\}, rec') = rec, r' \end{array}}{traverse(path_1\ .\ path_2, \{l_1 : v_1, ...l_n : v_n\}, v) = v', r'}$$

The first call omits the update value, and is used to extract a record associated with the first component of the path. This record is then passed, along with the second component of the path and the update value, to the second call of *traverse*. Finally, the third call uses the return of the second call to update the internal record, returning a new updated record.

Joined paths also require multiple calls to *traverse*, but the exact number depends on the amount of paths joined.

$$\frac{\begin{array}{c} r_0 = \{l_1 : v_1, ...l_n : v_n\} \\ \forall i \in [1,\ n]\ .\ traverse(path_i, r_{i-1}, v_i) = v'_i, r_i \end{array}}{traverse((path_1,\ \ldots\ path_n), \{l_1 : v_1, ...l_n : v_n\}, (v_1,\ \ldots\ v_n)) = (v'_1,\ \ldots\ v'_n), r_n}$$

Pairing the paths with the components of the tuple provided as the update value, each pair is passed as input to a call to *traverse*. This happens from left to right, and each call returns a part of the old value and a partially updated record. Every call uses the partially updated record provided, and the last call to *traverse* returns the fully updated record.

Distorted paths require two calls to *traverse*, one before and one after updating.

$$\frac{\begin{array}{c} traverse(path, \{l_1 : v_1, ...l_n : v_n\}) = v_{old}, r \\ \{\} \vdash v_2\ v\ v_{old} \Downarrow v' \qquad \{\} \vdash v_1\ v_{old} \Downarrow v'_{old} \\ traverse(path, \{l_1 : v_1, ...l_n : v_n\}, v') = v_{old}, r \end{array}}{traverse(path\ [v_1,\ v_2], \{l_1 : v_1, ...l_n : v_n\}, v) = v'_{old}, r}$$

First, the current value of the field is extracted. This value is then passed to the first component (v_1) of the accessor, returning the distorted current value. Then, the new distorted value v, along with the current value of the field, is passed to the second component (v_2) of the distorted accessor. This value is then provided as the new update value for a call to *traverse*, returning the updated record.

0.1.2 Pattern Matching

For `let` and `match` expressions, it is necessary to match a pattern p to a value v. This process, if successful, creates a mapping of identifiers to their corresponding elements of v. If v does not match the pattern p, the process fails.

In the case of a `let` expression, failing to match means the whole expression evaluates to *raise*. For `match` expressions, a failed pattern causes the next pattern to be attempted. If there are no more patterns, the expression evaluates to *raise*.

To aid in this matching, a auxiliary "match" function is defined. The function takes a pattern p and a value v, returning a mapping of identifiers to values (the $vars$ of an environment). If the matching fails, the function will return nothing.

The following are the rules for the match function:

$$match(x, v) = \{x \rightarrow v\}$$

$$match(\_, v) = \{\}$$

$$\frac{con_1 = con_2 \qquad \forall\ i \in [1, n] \quad match(p_i, v_i) = vars_i}{match(con_1\ v_1,\ \ldots v_n, con_2\ p_1,\ \ldots p_n) = \bigcup_{i=1}^{n} vars_i}$$

$$\frac{k \geq n \qquad \forall\ i \in [1, n] \quad \exists\ j \in [1, k] \quad l_i^1 = l_j^2 \wedge match(p_i, v_j) = vars_i}{match(\{l_1^1 : p_1, \ldots, l_n^1 : p_n, \ldots\}, \{l_1^2 : v_1, \ldots l_k^2 : v_k\}) = \bigcup_{i=1}^{n} vars_i}$$

$$\frac{\forall\ i \in [1, n] \quad l_i^1 = l_i^2 \wedge match(p_i, v_i) = vars_i}{match(\{l_1^1 : p_1, \ldots, l_n^1 : p_n\}, \{l_1^2 : v_1, \ldots l_n^2 : v_n\}) = \bigcup_{i=1}^{n} vars_i}$$

Any other inputs provided to the match function will result in a failed matching. This is represented by:

$$\neg\ match(p, v)$$

0.1.3 Big-Step Rules

Function Expressions Every function evaluates to a closure. This basically stores the function definition and the current environment in a value, allowing the evaluation environment to be restored on function application.

$$\text{env} \vdash func \Downarrow \langle func, env \rangle \qquad \text{(BS-Fn)}$$

Built-in functions evaluate to a partially applied built-in without any arguments.

$$\text{env} \vdash Builtin \Downarrow \ll Builtin\ .\ \gg \qquad \text{(BS-Builtin)}$$

Similarly, constructors, if they need at least one argument, evaluate to a partially applied constructor. If, however, they do not take any arguments, they immediately evaluate to a fully applied constructor.

$$\frac{\text{env.arities}(con) > 0}{\text{env} \vdash con \Downarrow\!\ll con\ .\ \gg} \quad \text{(BS-Con)}$$

$$\frac{\text{env.arities}(con) = 0}{\text{env} \vdash con \Downarrow con} \quad \text{(BS-Con0)}$$

Application An application expression requires either a closure, a partially applied built-in function or a partially applied constructor for its left-hand operand.

In the case of a closure, there are two different behaviors, depending on whether the function is recursive or not.

When applying non recursive functions, a new association between the parameter identifier (x) and the argument (v_2) is added to the environment stored in the closure (env_1). The body of the function (e) is then evaluated using this new environment.

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \langle \texttt{fn}\ x \Rightarrow e, \text{env}_1 \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2 \\ \{x \rightarrow v_2\} \cup \text{env}_1 \vdash e \Downarrow v\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow v} \quad \text{(BS-AppFn)}$$

Recursive functions, besides associating the identifier to the argument, also create an association between the function name and its value (i.e the closure itself). This allows the body of the function to call itself, creating a recursive structure.

For operational semantics, there is no difference between the typed and untyped versions of recursive functions, so both have the same evaluation rules.

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \langle \texttt{rec}\ f\ x \Rightarrow e, \text{env}_1 \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2 \\ \{x \rightarrow v_2, f \rightarrow \langle \texttt{rec}\ f\ x \Rightarrow e, \text{env}_1 \rangle\} \cup \text{env}_1 \vdash e \Downarrow v\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow v} \quad \text{(BS-AppFnRec)}$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \langle \texttt{rec}\ f : T\ x \Rightarrow e, \text{env}_1 \rangle \qquad \text{env} \vdash e_2 \Downarrow v_2 \\ \{x \rightarrow v_2, f \rightarrow \langle \texttt{rec}\ f : T\ x \Rightarrow e, \text{env}_1 \rangle\} \cup \text{env}_1 \vdash e \Downarrow v\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow v} \quad \text{(BS-AppFnRec2)}$$

Application on partially applied constructors can behave in two different ways, depending on how many arguments have been already applied.

If the arity of the constructor is larger than the number of arguments already applied (plus the new one being applied), the result of the application is a partially applied constructor with the new value added to the end.

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow\!\ll con\ .\ v_1,\ \ldots v_n \gg \\ n + 1 < \text{env.arities}(con) \qquad \text{env} \vdash e_2 \Downarrow v\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow\!\ll con\ .\ v_1,\ \ldots v_n,\ v \gg} \quad \text{(BS-AppCon)}$$

If the arity of the constructor is equal to 1 more than the number of already applied arguments, the application results in a completely applied constructor.

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll con\ .\ v_1,\ \ldots v_n \gg \quad n+1 = \text{env.arities}(con) \qquad \text{env} \vdash e_2 \Downarrow v}{\text{env} \vdash e_1\ e_2 \Downarrow con\ v_1,\ \ldots v_n,\ v} \quad \text{(BS-AppConTotal)}$$

Application on partially applied built-in functions works similarly, having different rules depending on the number of arguments.

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll Builtin\ .\ v_1,\ \ldots v_n \gg \quad n+1 < \text{arity}Builtin \qquad \text{env} \vdash e_2 \Downarrow v}{\text{env} \vdash e_1\ e_2 \Downarrow\!\ll Builtin\ .\ v_1,\ \ldots v_n,\ v \gg} \quad \text{(BS-AppBuiltin)}$$

The result of applying the last argument of a built-in function varies depending on what the function does (and what kind of arguments it accepts). These rules will be provided later.

Application propagates exceptions (*raise*). If the first sub-expression of an application evaluates to *raise*, the whole expression evaluates to *raise*. This is true for the second expression in most scenarios, but there are a couple of exceptions (see 0.1.3) that do not necessarily evaluate this sub-expression for complete evaluation.

$$\frac{\text{env} \vdash e_1 \Downarrow raise}{\text{env} \vdash e_1\ e_2 \Downarrow raise} \quad \text{(BS-AppRaise)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad \text{env} \vdash e_2 \Downarrow raise}{\text{env} \vdash e_1\ e_2 \Downarrow raise} \quad \text{(BS-AppRaise2)}$$

Identifier The evaluation of an identifier depends on the environment in which it is evaluated. If the environment has an association between the identifier and a value, the value is returned. If it does not, the program is malformed and cannot be evaluated (this will be caught in the type system).

$$\frac{\text{env.vars}(x) = v}{\text{env} \vdash x \Downarrow v} \quad \text{(BS-Ident)}$$

Records A record construction expression $\{l_1 : e_1,\ \ldots\ l_n : e_n\}$ evaluates each of its sub-expressions individually, resulting in a record value. The order of evaluation is defined by the order of the labels and is done from smallest to largest.

$$\frac{\forall\ k \in [1, n] \quad \text{env} \vdash e_k \Downarrow v_k}{\text{env} \vdash \{l_1 : e_1,\ \ldots\ l_n : e_n\} \Downarrow \{l_1 : v_1,\ \ldots\ l_n : v_n\}} \quad \text{(BS-Record)}$$

If any of the sub-expressions evaluate to raise, the whole record also evaluates to raise.

$$\frac{\exists\ k \in [1, n] \quad \text{env} \vdash e_k \Downarrow raise}{\text{env} \vdash \{l_1 : e_1,\ \ldots\ l_n : e_n\} \Downarrow raise} \quad \text{(BS-RecordRaise)}$$

Accessors There is a different evaluation rule for each type of path available to accessors.

The simplest rule is for a label accessor, which is in itself a value.

$$\text{env} \vdash \#l \Downarrow \#l \qquad \text{(BS-Label)}$$

Joined accessors evaluate each of their sub-expressions, expecting an accessor value as a result.

$$\frac{\forall\ k \in [1, n] \quad \text{env} \vdash e_k \Downarrow \#path_k}{\text{env} \vdash \#(e_1,\ \ldots\ e_n) \Downarrow \#(path_1,\ \ldots\ path_n)} \qquad \text{(BS-Joined)}$$

To create a stacked accessor, the built-in function "stack" must be used. This function has arity 2, and requires both arguments to be accessors. The paths of the accessors are then composed in a stacked accessor, which is the result of the evaluation.

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll \text{stack}\ .\ \#path_1 \gg \qquad \text{env} \vdash e_2 \Downarrow \#path_2}{\text{env} \vdash e_1\ e_2 \Downarrow \#path_1\ .\ path_2} \qquad \text{(BS-Stacked)}$$

Similarly, creating distorted accessors requires the built-in function "distort". This function takes 3 arguments, the first being an accessor, and the remaining two being functions. When fully evaluated, the path of the accessor is combined with the function values, creating a distorted accessor.

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll \text{distort}\ .\ \#path,\ v_1 \gg \qquad \text{env} \vdash e_2 \Downarrow v_2}{\text{env} \vdash e_1\ e_2 \Downarrow \#path\ [v_1,\ v_2]} \qquad \text{(BS-Distorted)}$$

Using Accessors There are two built-in functions that take accessors as arguments.

Get takes 2 arguments: an accessor and a record. The *traverse* function is then called with the accessor's path and the record (the third argument is omitted), and the first return (i.e. the value associated with the path) is used as the result of the evaluation.

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow\!\ll \text{get}\ .\ \#path \gg \qquad \text{env} \vdash e_2 \Downarrow \{l_1 : v_1,\ \ldots\ l_n : v_n\} \\ traverse(path, \{l_1 : v_1,\ \ldots\ l_n : v_n\}) = v', r'\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow v'} \qquad \text{(BS-Get)}$$

Set takes 3 arguments: an accessor, a generic value and a record. The *traverse* function is then called with the arguments, using the generic value as the update value of the call. The result of the evaluation is the second return of the *traverse* function (i.e. the updated record).

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow\!\ll \text{set}\ .\ \#path,\ v \gg \qquad \text{env} \vdash e_2 \Downarrow \{l_1 : v_1,\ \ldots\ l_n : v_n\} \\ traverse(path, \{l_1 : v_1,\ \ldots\ l_n : v_n\}, v) = v', r'\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow r'} \qquad \text{(BS-Set)}$$

Numerical Operations The *V* language only supports integers, so all operations are done on integer numbers. This means that the division always results in a whole number, truncated towards zero.

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll\ +\ .\ n_1 \gg \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n\| = \|n_1\| + \|n_2\|}{\text{env} \vdash e_1\ e_2 \Downarrow n} \quad \text{(BS-+)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll\ -\ .\ n_1 \gg \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n\| = \|n_1\| - \|n_2\|}{\text{env} \vdash e_1\ e_2 \Downarrow n} \quad \text{(BS–)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll\ -\ .\ \gg \qquad \text{env} \vdash e_2 \Downarrow n_1 \qquad \|n\| = -\|n_1\|}{\text{env} \vdash e_1\ e_2 \Downarrow n} \quad \text{(BS– (UNARY))}$$

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll\ *\ .\ n_1 \gg \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n\| = \|n_1\| * \|n_2\|}{\text{env} \vdash e_1\ e_2 \Downarrow n} \quad \text{(BS-*)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll\ \div\ .\ n_1 \gg \qquad \text{env} \vdash e_2 \Downarrow 0}{\text{env} \vdash e_1\ e_2 \Downarrow \mathit{raise}} \quad \text{(BS-÷ZERO)}$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow\!\ll\ \div\ .\ n_1 \gg \qquad \text{env} \vdash e_2 \Downarrow n_2 \\ \|n_2\| \neq 0 \qquad \|n\| = \|n_1\| \div \|n_2\|\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow n} \quad \text{(BS-÷)}$$

Equality Operations The equality operators (= and ≠) test the equality of fully applied constructors and records.

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow\!\ll =\ .\ (\mathit{con}\ v_1^1,\ \ldots v_n^1) \gg \qquad \text{env} \vdash e_2 \Downarrow (\mathit{con}\ v_1^2,\ \ldots v_n^2) \\ \forall\ k \in [1,n] \quad \text{env} \vdash (=\ v_k^1)\ v_k^2 \Downarrow \mathit{true}\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow \mathit{true}} \quad \text{(BS-=CONTRUE)}$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow\!\ll =\ .\ (\mathit{con}\ v_1^1,\ \ldots v_n^1) \gg \qquad \text{env} \vdash e_2 \Downarrow (\mathit{con}\ v_1^2,\ \ldots v_n^2) \\ \exists\ k \in [1,n] \quad \text{env} \vdash (=\ v_k^1)\ v_k^2 \Downarrow \mathit{false}\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow \mathit{false}} \quad \text{(BS-=CONFALSE)}$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow\!\ll =\ .\ (\mathit{con}\ v_1^1,\ \ldots v_n^1) \gg \qquad \text{env} \vdash e_2 \Downarrow (\mathit{con}'\ v_1^2,\ \ldots v_k^2) \\ \mathit{con}' \neq \mathit{con}\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow \mathit{false}}$$

(BS-=CONFALSE2)

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow\!\ll =\ .\ \{l_1^1 : v_1^1,\ \ldots\ l_n^1 : v_n^1\} \gg \qquad \text{env} \vdash e_2 \Downarrow \{l_1^2 : v_1^2,\ \ldots\ l_n^2 : v_n^2\} \\ \forall\ k \in [1,n] \quad l_k^1 = l_k^2 \wedge \text{env} \vdash (=\ v_k^1)\ v_k^2 \Downarrow \mathit{true}\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow \mathit{true}}$$

(BS-=RECORDTRUE)

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \ll = \ . \{l_1^1 : v_1^1,\ \ldots\ l_n^1 : v_n^1\} \gg \qquad \text{env} \vdash e_2 \Downarrow \{l_1^2 : v_1^2,\ \ldots\ l_n^2 : v_n^2\} \\ \exists\, k \in [1,n]\ \ l_k^1 = l_k^2 \wedge \text{env} \vdash (= \ v_k^1)\ v_k^2 \Downarrow \mathit{false} \\ \forall\ j \in [1,k)\ \ \text{env} \vdash v_j^1 = v_j^2 \Downarrow \mathit{true}\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow \mathit{false}} \text{(BS-=RecordFalse)}$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \ll \neq \ . v_1 \gg \qquad \text{env} \vdash e_2 \Downarrow v_2 \\ \text{env} \vdash (= \ v_1)\ v_2 \Downarrow \mathit{false}\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow \mathit{true}} \text{(BS-}\neq\text{True)}$$

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \ll \neq \ . v_1 \gg \qquad \text{env} \vdash e_2 \Downarrow v_2 \\ \text{env} \vdash (= \ v_1)\ v_2 \Downarrow \mathit{true}\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow \mathit{false}} \text{(BS-}\neq\text{False)}$$

Inequality Operations The inequality operators function much in the same way as the equality operators. The only difference is that they do not allow comparison of certain kinds of expressions (such as booleans) when such expressions do not have a clear ordering to them.

To reduce the number of rules, some rules are condensed for all inequality operators ($<, \leq, >, \geq$). The comparison done on numbers is the ordinary numerical comparison. For characters, the ASCII values are compared numerically.

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \|n_1\|\, \mathit{opIneq}\, \|n_2\|}{\text{env} \vdash e_1\ \mathit{opIneq}\ e_2 \Downarrow \mathit{true}} \text{(BS-IneqNumTrue)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow n_1 \qquad \text{env} \vdash e_2 \Downarrow n_2 \qquad \neg\, \|n_1\|\, \mathit{opIneq}\, \|n_2\|}{\text{env} \vdash e_1\ \mathit{opIneq}\ e_2 \Downarrow \mathit{true}} \text{(BS-IneqNumFalse)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow c_1 \qquad \text{env} \vdash e_2 \Downarrow c_2 \qquad \|c_1\|\, \mathit{opIneq}\, \|c_2\|}{\text{env} \vdash e_1\ \mathit{opIneq}\ e_2 \Downarrow \mathit{true}} \text{(BS-IneqCharTrue)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow c_1 \qquad \text{env} \vdash e_2 \Downarrow c_2 \qquad \neg\, \|c_1\|\, \mathit{opIneq}\, \|c_2\|}{\text{env} \vdash e_1\ \mathit{opIneq}\ e_2 \Downarrow \mathit{true}} \text{(BS-IneqCharFalse)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \mathit{nil} \qquad \text{env} \vdash e_2 \Downarrow \mathit{nil}}{\text{env} \vdash e_1 < e_2 \Downarrow \mathit{false}} \text{(BS-<Nil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \leq e_2 \Downarrow true} \quad \text{(BS-}\leq\text{Nil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 > e_2 \Downarrow false} \quad \text{(BS->Nil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \geq e_2 \Downarrow true} \quad \text{(BS-}\geq\text{Nil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 < e_2 \Downarrow false} \quad \text{(BS-<ListNil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \leq e_2 \Downarrow false} \quad \text{(BS-}\leq\text{ListNil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 > e_2 \Downarrow true} \quad \text{(BS->ListNil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \qquad \text{env} \vdash e_2 \Downarrow nil}{\text{env} \vdash e_1 \geq e_2 \Downarrow true} \quad \text{(BS-}\geq\text{ListNil)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 < e_2 \Downarrow true} \quad \text{(BS-<NilList)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 \leq e_2 \Downarrow true} \quad \text{(BS-}\leq\text{NilList)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 > e_2 \Downarrow false} \quad \text{(BS->NilList)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow nil \qquad \text{env} \vdash e_2 \Downarrow v_1 :: v_2}{\text{env} \vdash e_1 \geq e_2 \Downarrow false} \quad \text{(BS-}\geq\text{NilList)}$$

$$\frac{\begin{array}{cc}\text{env} \vdash e_1 \Downarrow v_1 :: v_2 & \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \\ \text{env} \vdash v_1 = v_3 \Downarrow false & \text{env} \vdash v_1 \; opIneq \; v_3 \Downarrow b\end{array}}{\text{env} \vdash e_1 \; opIneq \; e_2 \Downarrow b} \quad \text{(BS-IneqListHead)}$$

$$\frac{\begin{array}{cc}\text{env} \vdash e_1 \Downarrow v_1 :: v_2 & \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \\ \text{env} \vdash v_1 = v_3 \Downarrow true & \text{env} \vdash v_2 \; opIneq \; v_4 \Downarrow b\end{array}}{\text{env} \vdash e_1 \; opIneq \; e_2 \Downarrow b} \quad \text{(BS-IneqListTail)}$$

Boolean Operations The built-in functions $\vee$ (OR) and $\wedge$ (AND) are treated differently from all other functions in V. They are binary functions, but they only evaluate their second argument if strictly necessary. This is done to provide them a short-circuit behavior, keeping in line with expectations from other programming languages.

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll \ \wedge \ . \ false \gg}{\text{env} \vdash e_1 \ e_2 \Downarrow false} \quad \text{(BS-}\wedge\text{False)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll \ \wedge \ . \ true \gg \qquad \text{env} \vdash e_2 \Downarrow b}{\text{env} \vdash e_1 \ e_2 \Downarrow b} \quad \text{(BS-}\wedge\text{True)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll \ \vee \ . \ true \gg}{\text{env} \vdash e_1 \ e_2 \Downarrow true} \quad \text{(BS-}\vee\text{True)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll \ \vee \ . \ false \gg \qquad \text{env} \vdash e_2 \Downarrow b}{\text{env} \vdash e_1 \ e_2 \Downarrow b} \quad \text{(BS-}\vee\text{False)}$$

Let Expressions These expressions are used to associate an identifier with a specific value, allowing the value to be reused throughout the program. Since V is a functional language, these are not variables, and the values assigned to an identifier will be constant (unless the same identifier is used in a new *let* expression).

After evaluating the expression that is to be associated to the identifier (that is, e_1), resulting in v, the *let* expression evaluates e_2. For this evaluation, the association of p to v is added to the environment. The result of this evaluation (that is, v_2) is the final result of the evaluation of the entire *let* expression.

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow v \qquad match(p, v) = \text{env}_1 \\ \text{env}_1 \cup \text{env} \vdash e_2 \Downarrow v_2\end{array}}{\text{env} \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 \Downarrow v_2} \quad \text{(BS-Let)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow v \qquad \neg match(p, v)}{\text{env} \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 \Downarrow raise} \quad \text{(BS-Let2)}$$

If the sub-expression e_1 evaluates to *raise*, the whole expression also evaluates to *raise*.

$$\frac{\text{env} \vdash e_1 \Downarrow raise}{\text{env} \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 \Downarrow raise} \quad \text{(BS-LetRaise)}$$

Match Expression The match expression receives a input value and a list of *matches*, attempting to pattern match against each one. The first *match* which correctly matches terminates the processing, and its corresponding expression is evaluated as the result of the whole expression.

If no *match* returns a valid result, the whole expression evaluates to *raise*.

$$\frac{\begin{array}{c}\text{env} \vdash e \Downarrow v \\ \exists j \in [1..n]\, multiMatch(v, \text{env}, match_j) = v_j \\ \forall k \in [1..j)\, \neg\, multiMatch(v, \text{env}, match_k)\end{array}}{\text{env} \vdash \texttt{match}\ e\ \texttt{with}\ match_1, ...\ match_n \Downarrow v_j} \quad \text{(BS-Match)}$$

$$\frac{\begin{array}{c}\text{env} \vdash e \Downarrow v \\ \forall j \in [1..n]\, \neg\, multiMatch(v, \text{env}, match_j)\end{array}}{\text{env} \vdash \texttt{match}\ e\ \texttt{with}\ match_1, ...\ match_n \Downarrow raise} \quad \text{(BS-Match2)}$$

In order to properly evaluate a match expression, it is necessary to define an auxiliary function, here called *multiMatch*. This function receives an input value, an environment and a *match*.

If the *match* has a conditional expression, it must evaluate to *true* for the match to be considered valid.

$$\frac{\neg\, match(p, v)}{\neg\, multiMatch(v, \text{env}, p \rightarrow e)}$$

$$\frac{\neg\, match(p, v)}{\neg\, multiMatch(v, \text{env}, p\ \text{when}\ e_1 \rightarrow e_2)}$$

$$\frac{match(p, v) = \text{env}_1 \qquad \text{env} \cup \text{env}_1 \vdash e_1 \Downarrow false}{\neg\, multiMatch(v, \text{env}, p\ \text{when}\ e_1 \rightarrow e_2)}$$

$$\frac{match(p, v) = \text{env}_1 \qquad \text{env} \cup \text{env}_1 \vdash e_1 \Downarrow raise}{\neg\, multiMatch(v, \text{env}, p\ \text{when}\ e_1 \rightarrow e_2)}$$

$$\frac{match(p, v) = \text{env}_1 \qquad \text{env} \cup \text{env}_1 \vdash e \Downarrow v_2}{multiMatch(v, \text{env}, p \rightarrow e) = v_2}$$

$$\frac{\begin{array}{c}match(p, v) = \text{env}_1 \qquad \text{env} \cup \text{env}_1 \vdash e_1 \Downarrow true \\ \text{env} \cup env_1 \vdash e_2 \Downarrow v_2\end{array}}{multiMatch(v, \text{env}, p\ \text{when}\ e_1 \rightarrow e_2) = v_2}$$

Exceptions Some programs can be syntactically correct but still violate the semantics of the *V* language, such as a dividing by zero or trying to access the head of an empty list. In these scenarios, the expression is evaluated as the *raise* value.

Besides violation of semantic rules, the only other expression that evaluates to the *raise* value is the *raise* expression, using the following rule:

$$\text{env} \vdash \mathit{raise} \Downarrow \mathit{raise} \qquad \text{(BS-RAISE)}$$

This value propagates upwards through the evaluation tree if a "regular" value is expected. This means that expressions that need well-defined sub-expressions, such as numerical and equality operations, evaluate to *raise* if any of these sub-expressions evaluate to *raise*.