1 Abstract Syntax and Semantics

1.1 Abstract Syntax

1.1.1 Expressions

Programs in V are expressions. Each expression is a member of the abstract syntax tree defined below. The syntax tree will be constructed in parts, with an explanation of what each expression means and their uses. The full syntax tree can be obtained by simply joining all the separate sections.

Functions V, as a functional language, treats functions as first class constructions. This means that functions are regular expressions, and can be passed as arguments, bound to identifiers, etc. Below are both of the function expressions available in V, along with function application.

$$\begin{array}{lcl} e & ::= & \cdots \\ & | & func \\ & | & e_1\ e_2 \\ & | & x \\ \\ func & ::= & \texttt{fn}\ x \Rightarrow e \\ & | & \texttt{rec}\ f\ x \Rightarrow e \\ & | & \texttt{rec}\ f : T\quad x \Rightarrow e \\ \\ x & ::= & \{x_0, x_1, \ldots\} \end{array}$$

All functions in V take exactly one parameter, and so function application evaluates the function by providing a single argument to it.

The first type of function ($\texttt{fn}\ x \Rightarrow e$) defines a simple unnamed function with a parameter x. x is an identifier from a set of name identifiers.

The body of the function is the expression e, which may or may not contain occurrences of the value passed as the argument of the function.

The other two types of functions are both variants of recursive functions available in V. These functions have a name, f, which is also a member of the set of name identifiers, and this name is used to allow recursive calls from withing their body (e). Like unnamed functions, they take exactly one parameter, x, which may or may not be present in their bodies.

The difference between these variants is in their type declaration: the first variant is implicitly typed, while the second is explicitly typed. In the second variant, the programmer specifies the return type of the function as T (types will be shown later).

There are also expressions to use functions.

The first of them is function application: the first of the expressions is a function, and the second is the argument being passed to the function.

The last expression is simply to allow the use of the parameter defined in a function. An identifier x is only considered valid if it has been bound before (either by a function or, as will be seen later, by let declarations of `match` expressions).

Built-in Functions V has a few built-in functions that provide basic behavior.
These are:

$$\begin{array}{lcll}
e & ::= & \cdots & \\
 & | & \mathit{Builtin} & \\
 & & & \\
\mathit{Builtin} & ::= & + & \text{(add, binary)} \\
 & | & - & \text{(subtract, binary)} \\
 & | & * & \text{(multiply, binary)} \\
 & | & \div & \text{(divide, binary)} \\
 & | & - & \text{(negate, unary)} \\
 & & & \\
 & | & < & \text{(less than, binary)} \\
 & | & \leq & \text{(less than or equal, binary)} \\
 & | & > & \text{(more than, binary)} \\
 & | & \geq & \text{(more than or equal, binary)} \\
 & | & = & \text{(equal, binary)} \\
 & | & \neq & \text{(not equal, binary)} \\
 & & & \\
 & | & \vee & \text{(Or, binary)} \\
 & | & \wedge & \text{(And, binary)} \\
 & & & \\
 & | & \text{get} & \text{(binary)} \\
 & | & \text{set} & \text{(ternary)} \\
 & | & \text{stack} & \text{(binary)} \\
 & | & \text{distort} & \text{(ternary)}
\end{array}$$

Every function has its arity declared. The arity of a function defines how many parameters it needs before it can be fully evaluated.

This behavior does not change the fact that functions receive only one parameter. These particular cases can be thought of as nested functions, each taking a single parameter, until all the necessary parameters have been received.

This means that partially applied built-in functions are also treated as functions, and, therefore, can be passed as arguments, bound to identifiers, etc.

Most of these functions are basic and require no explanation. The last 4, however, will only be explained later, after records and accessors have been introduced.

The boolean functions $\vee$ and $\wedge$ are treated differently than others. Even though they are binary, they have a short-circuit mechanism. This means that, if the result of the application can be known by the first parameter (True in the case of $\vee$ or False in the case of $\wedge$), the second parameter is not evaluated.

This is in contrast to all other functions in V, which evaluate their arguments before trying to evaluate themselves (This will be explained in more detail in 1.2)

Constructors V has another type of special function: Data Constructors. Data Constructors are, as their name suggests, functions that construct values (data).

When fully evaluated, constructors define a form of structured data, storing the values passed to them as arguments. To access these values, it is possible to pattern match (see 1.1.1) on the constructor name when fully evaluated.

Like built-in functions, constructors can be in a partially evaluated state. Partially evaluated constructors are treated as normal functions and, therefore, do not allow matching on their name.

$$\begin{array}{lcll}
e & ::= & con & \\
 & & & \\
con & ::= & n & \text{(arity 0)} \\
 & | & b & \text{(arity 0)} \\
 & | & c & \text{(arity 0)} \\
 & | & nil & \text{(arity 0)} \\
 & | & :: & \text{(arity 2)} \\
 & | & \text{Tuple } n & \text{(arity } n,\ n \geq 2\text{)} \\
 & & & \\
b & ::= & true \mid false & \\
n & ::= & \mathbb{Z} & \\
c & ::= & `char' & \\
char & ::= & \texttt{ASCII characters} &
\end{array}$$

These functions are *extra* special, however, because they can have arity zero. This means that they "construct" values as soon as they are declared. The basic zero-arity constructors defined in the language are integers, booleans and characters.

The following two constructors (*nil* and ::) are related to lists. The first (with arity 0) is the empty list. The second constructor has arity 2, and extends a list (its second argument) by adding a new value (its first argument) to its head.

The last constructor is actually a family of constructors describing tuples. A constructor Tuple$n$ defines a constructor that has n parameters and evaluates to a tuple with n elements. It is also important to note that $n \geq 2$. This means that only tuple constructors with 2 or more parameters are valid.

Records and Accessors The record system in V is composed of two parts: records and accessors.

Records are a type of structured data composed of associations between labels and values, called fields. Each label is part of an ordered set of labels l, and can only appear once in every record. For a record to be valid, the order in which its labels are declared must respect the order in the set of labels.

Accessors are terms that allow access to fields within a record. Accessors view records as trees, where each non-leaf node is a record, and each edge has a name (the label of the field). Accessors, then, define a path on this tree, extracting the node at the end of the path.

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
  & | & \{l_1 : e_1,\ \ldots\ l_n : e_n\} & (n \geq 1) \\
  & | & \#l & \\
  & | & \#(e_1,\ \ldots\ e_n) & (n \geq 2) \\
  & & & \\
l & ::= & \{l_1, l_2, ...\} &
\end{array}
$$

The most basic type of path is a simple label $\#l$. An accessor can also be made by combining multiple accesors. The $\#(e_1,\ \ldots\ e_n)$ expression creates an accessor for multiple fields of the same record. Another type of composition is vertical, and is obtained by using the built-in function "stack".

Furthermore, paths can be distorted with the "distort" built-in function, specifying a pair of functions to be applied when extracting or inserting values in the field.

More details about how accessors (and paths) work will be provided in a later section.

Let and Patterns The `let` expression is used to bind values to identifiers, allowing them to be reused in a sub-expression. A `let` expression is divided into 2 parts: the binding and the sub-expression. The binding itself also has 2 parts: the left-hand side, which is a pattern; and the right-hand side, which will be the value to be bound.

Patterns are used to "unpack" values, and can be either explicitly or implicitly typed.

$$
\begin{array}{lcll}
e & ::= & \cdots & \\
  & | & \texttt{let}\ p = e_1\ \texttt{in}\ e_2 & \\
  & & & \\
p & ::= & patt & \\
  & | & patt : T & \\
  & & & \\
patt & ::= & x & \\
  & | & \_ & \\
  & | & con\ p_1,\ \ldots\ p_n & \text{(constructor pattern, } n = \text{arity } con) \\
  & | & \{l_1 : p_1,\ \ldots\ l_n : p_n\} & (n \geq 1) \\
  & | & \{l_1 : p_1,\ \ldots\ l_n : p_n, \ldots\} & (\text{partial record}, n \geq 1)
\end{array}
$$

Much like functions, `let` expressions allow the use of identifiers in an expression by attaching values to the identifiers. Differently from functions, however, a single `let` expression can bind multiple identifiers to values by using patterns. Patterns are matched against the values in the right-hand side of the binding, and can, depending on their structure, create any number of identifier bindings.

The pattern x is a simple identifier pattern. It matches any value in the right-hand side, and binds it to the identifier.

The _ also matches any value, but it does not create any new bindings (this is called the ignore pattern).

The *con* pattern matches a completely applied constructor. This pattern is a compound pattern, with the same number of components as the arity of the constructor.

The *con* pattern itself does not create any bindings, but its components might, since they are themselves patterns and, as such, will be matched against the components of the constructor.

The next pattern is a record pattern. This matches a record with *exactly* the same fields as the pattern. Since all labels in a record are ordered, the fields do not need to be reordered for the matching.

For matching any record with *at least* the fields $l_1, \ldots\ l_n$, one can use the pattern $\{l_1 : p_1, \ldots\ l_n : p_n, \ldots\}$. This pattern will match any record whose set of labels is a superset of $l_1, \ldots\ l_n$.

Match Expression A `match` expression attempts to match a value against a list of patterns. Every pattern is paired with a resulting expression to be evaluated if the pattern matches. Furthermore, it is possible to specify a boolean condition to be tested alongside the pattern matching. This condition will only be tested if the match succeeds, so it can use any identifier bound by the pattern. The matching stops at the first pattern that successfully matches (and any condition is satisfied), and its paired expression is then evaluated.

$$\begin{array}{lcll} e & ::= & \cdots & \\ & \mid & \texttt{match}\ e\ \texttt{with}\ match_1, \ldots\ match_n & (n \geq 1) \\ \\ match & ::= & p \rightarrow e & \\ & \mid & p\ \texttt{when}\ e_1 \rightarrow e_2 & \end{array}$$

Exceptions This expression always evaluates to a runtime error.

$$\begin{array}{lcl} e & ::= & \cdots \\ & \mid & raise \end{array}$$

Runtime errors usually happen when an expression cannot be correctly evaluated, such as division by zero, accessing an empty list, etc.

Sometimes, however, it can be necessary to directly cause an error. The *raise* expression serves this purpose.

1.1.2 Types

Since V is strongly typed, every (valid) expression has exactly one type associated with it. Some expressions allow the programmer to explicitly declare types, such as patterns and recursive functions. Other expressions, such as $e_1 = e_2$, or even constants, such as 1 or *true*, have types implicitly associated with them. These types are used by the type system (see 1.3) to check whether an expression is valid or not, avoiding run-time errors that can be detected at compile time.

Types Below are all the types available in V. The first type is a fully applied constructor type. The second type is a function type. The third type is a record type and, finally, the last type is an accessor type.

$$
\begin{array}{lcll}
T & ::= & \cdots & \\
& | & conT\ T_1,\ \ldots\ T_n & (n = \text{arity } conT) \\
& | & T_1 \rightarrow T_2 & \\
& | & \{l_1 : T_1,\ \ldots\ l_n : T_n\} & (n \geq 1) \\
& | & T_1\#T_2 & \mathit{Accessor}
\end{array}
$$

Most of the types are compound types, and the only scenario in which a type is not compound is for constructor types with arity 0. Function types specify the type of the single parameter (T_1) and the type of output (T_2).

Record types are also compound types, but they associate every component to its corresponding label. Just like record expressions, the labels must be ordered.

Finally, accessor types define the types of accessor expressions. They have two components: T_1, which is the type of the record being accessed; and T_2, the type of the value being accessed. It is read as T_1 *accesses* T_2.

Constructor Types These are types associated with constructors. Much like constructors, they can take any amount of arguments to be fully applied, and the number of arguments they take is described by their arity. Instead of taking values as arguments, however, constructor types take types as arguments.

$$
\begin{array}{lcll}
conT & ::= & \text{Int} & (\text{arity } 0) \\
& | & \text{Bool} & (\text{arity } 0) \\
& | & \text{Char} & (\text{arity } 0) \\
& | & \text{List} & (\text{arity } 1) \\
& | & \text{TupleT } n & (\text{arity } n,\ n \geq 2)
\end{array}
$$

Variable Types These types represent an unknown constant type. Explicitly typed expressions cannot be given variable types, but they are used by the type system for implicitly typed expressions. In the course of the type inference, the type system can replace variable types for their type.

It is important to realize that variable types already represent a unique type with an unknown identity. This means that a variable type may only be replaced by the specific type which it represents and not any other type. This distinction becomes important when talking about polymorphism, which uses variable types, along with universal quantifiers, to represent a placeholder for any possible type (this is discussed in greater detail in 1.3.1).

$$
\begin{array}{lcl}
T & ::= & \cdots \\
& | & X^{Traits}
\end{array}
$$

$$
X \quad ::= \quad X_1, X_2, \ldots
$$

1.1.3 Traits

Types can conform to traits, which define certain behaviors that are expected of said type. Regular types always have their trait information implicitly defined, since this information is included in the language. Variable types, on the other hand, can explicitly state which traits they possess, restricting the set of possible types they can represent (this is represented by the superscript $Traits$ in a variable type X).

$$\begin{array}{lll} Traits & ::= & \varnothing \\ & \mid & \{Trait\} \cup Traits \end{array}$$

$$\begin{array}{llll} Trait & ::= & Equatable & \\ & \mid & Orderable & \\ & \mid & \{l : Type\} & \text{(Record Label)} \end{array}$$

The information on which types conform to which traits is defined inside the type inference environment. When a type T conforms to a trait $Trait$, the notation used is $T \in Trait$.

By default, the following rules hold for conformance:

Equatable If a type T is *Equatable*, expressions of type T can use the equality operators $(=, \neq)$.

To define the set of types that belong to *Equatable*, the following rules are used:

$$\begin{array}{l} \{\text{Int}, \text{Bool}, \text{Char}\} \subset Equatable \\ T \in Equatable \implies List\ T \in Equatable \\ X^{Traits} \in Equatable \implies Equatable \in Traits \end{array}$$

Orderable If a type T is *Orderable*, expressions of type T can use the inequality operators $(<, \leq, >, \geq)$. Any type that is *Orderable* is also *Equatable*.

To define the set of types that belong to *Orderable*, the following rules are used:

$$\begin{array}{l} \{\text{Int}, \text{Char}\} \subset Orderable \\ T \in Orderable \implies List\ T \in Orderable \\ X^{Traits} \in Orderable \implies Orderable \in Traits \end{array}$$

Record Label A type T_1 conforms to a Record Label Trait $\{l : T_2\}$ if it is a record that contains a field with the label l and the type T_2.

If the type conforms to the trait $\{l : T_2\}$, it can then use the accessor #l. This ensures that accessor for a field can only be used on records that have that field.

To define the set of types that belong to a record label $\{l : T\}$, the following rules are used:

$$\begin{array}{l} \{l_1 : T_1, \ \dots \ l_n : T_n, \ \dots \ T_k\} \in \{l : T\} \iff l_n = l \wedge T_n = T \qquad (1 \leq n \leq k) \\ X^{Traits} \in \{l : T\} \implies \{l : T\} \in Traits \end{array}$$

1.2 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:

$$
\begin{array}{lcll}
env & ::= & (arities, vars) & \\
 & & & \\
arities & ::= & \{\} \mid \{con \rightarrow n\} \cup arities & (n \in \mathbb{N}) \\
vars & ::= & \{\} \mid \{x \rightarrow v\} \cup vars & \\
 & & & \\
v & ::= & con\ v_1,\ \ldots\ v_n & (n = \text{arity } con) \\
 & \mid & raise & \\
 & \mid & \{l_1 : v_1,\ \ldots\ l_n : v_n\} & (n \geq 1) \\
 & \mid & \#path & \\
 & \mid & \langle func, env \rangle & \\
 & \mid & \ll Builtin\ .\ v_1,\ \ldots v_n \gg & (n < \text{arity } Builtin) \\
 & \mid & \ll con\ .\ v_1,\ \ldots v_n \gg & (n < \text{arity } con) \\
 & & & \\
path & ::= & l & \\
 & \mid & path\ .\ path & \\
 & \mid & (path_1,\ \ldots\ path_n) & (n \geq 2) \\
 & \mid & path\ [v_1,\ v_2] &
\end{array}
$$

The value $\langle func, env \rangle$ 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 $\ll \mathit{Builtin}\ .\ v_1,\ \ldots v_n \gg$ and $\ll \mathit{con}\ .\ v_1,\ \ldots v_n \gg$ 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 $\mathit{con}\ v_1,\ \ldots\ v_n$.

1.2.1 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.

$$\frac{\begin{array}{c}1 \leq ||k|| \leq ||n|| \\ r = \{l_1 : v_1,\ \ldots\ l_k : v,\ \ldots\ l_n : v_n\}\end{array}}{\mathit{traverse}(l_k, \{l_1 : v_1, \ldots l_n : v_n\}, v) = v_k, r}$$

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.

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

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 the evaluation of two applications, one before and one after updating.

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

Initially, the provided update value is applied 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 an old value and the updated record. This old value is then provided as argument to the first component (v_1) of the distorted accessor, and the result of this is the output of the whole *traverse* (along with the updated record).

1.2.2 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:

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

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

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

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

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

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

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

1.2.3 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.

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

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.

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

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

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.

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

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

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

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 1.2.3) 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.

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

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

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

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

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

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

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

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

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

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

(BS-=ConFalse2)

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

(BS-=RecordTrue)

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \ll =\ .\ \{l^1_1 : v^1_1,\ \ldots\ l^1_n : v^1_n\} \gg \qquad \text{env} \vdash e_2 \Downarrow \{l^2_1 : v^2_1,\ \ldots\ l^2_n : v^2_n\} \\ \exists\, k \in [1, n]\ \ l^1_k = l^2_k \land \text{env} \vdash (=\ v^1_k)\ v^2_k \Downarrow \mathit{false} \\ \forall\ j \in [1, k)\ \ \text{env} \vdash v^1_j = v^2_j \Downarrow \mathit{true}\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow \mathit{false}} \quad \text{(BS-=RecordFalse)}$$

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Boolean Operations The built-in functions ∨ (OR) and ∧ (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.

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

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

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

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

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.

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

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

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

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

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*.

$$\frac{\begin{array}{c}\text{env} \vdash e \Downarrow v \\ \exists j \in [1..n]\, \mathit{multiMatch}(v, \text{env}, \mathit{match}_j) = v_j \\ \forall k \in [1..j) \neg\, \mathit{multiMatch}(v, \text{env}, \mathit{match}_k)\end{array}}{\text{env} \vdash \texttt{match}\ e\ \texttt{with}\ \mathit{match}_1, ...\ \mathit{match}_n \Downarrow v_j} \quad \text{(BS-MATCH)}$$

$$\frac{\begin{array}{c}\text{env} \vdash e \Downarrow v \\ \forall j \in [1..n] \neg\, \mathit{multiMatch}(v, \text{env}, \mathit{match}_j)\end{array}}{\text{env} \vdash \texttt{match}\ e\ \texttt{with}\ \mathit{match}_1, ...\ \mathit{match}_n \Downarrow \mathit{raise}} \quad \text{(BS-MATCH2)}$$

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.

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

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

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

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

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

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

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:

$$\text{env} \vdash \mathit{raise} \Downarrow \mathit{raise} \qquad \text{(BS-R}\textsc{aise}\text{)}$$

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*.

1.3 Type System

V has a strong and static type inference system that checks a program to decide whether or not it is "'well-typed"'. If a program is considered to be well-typed, the type system guarantees that the program will be able to be properly evaluated according to the operational semantics of *V*. As a side-effect of checking the validity of a program, the type system can also provide the actual type of any implicitly typed expression down to its basic types, be those concrete types or variable types.

1.3.1 Polymorphism

V has support for parametric Damas-Milner polymorphism. This means that functions can have their types be defined with universal quantifiers, allowing their use with any type.

For instance, take the function *count*, which counts the number of elements in a list. This function can be defined as follows:

```
let count = rec count x ⇒ if isempty x then 0 else 1 + count (tl x) in
count (3::4::nil)
```

In this situation, *count* can be used with a list of any type, not only Int. To allow this, its identifier (*count*) must have a universal association in the environment, defined as so:

$\forall x.\ x\ list \rightarrow \text{Int}$

The universal quantifier $\forall x$ allows the type variable x to be substituted for any concrete type when the function is called. When creating a polymorphic type, the type system must identify which type variables are free in the function type and which are bound in the environment. This process guarantees that a polymorphic type only universally quantifies those type variables that are not already bound, while still allowing all free variables to be instantiated when the function is called.

1.3.2 Traits

Traits are characteristics that a type can have, defining behaviors expected of that type. Some expressions are polymorphic in a sense that they accept certain types for their operators, but not any type.

1.3.3 Type Inference System

The type inference system is composed of two basic parts:

- Constraint Collection
- Constraint Unification

Constraints are equations between type expressions, which can have both constant types and variable types. To infer the type of a program, the type system recursively collects a set of constraints for every subexpression in that program. This is done in a static way across the expression tree from the nodes to the root, without having to evaluate any of the expressions. To create a valid set of constraints, the system must contain an environment, built from the root to the nodes, to ensure identifiers are properly typed.

Environment Just like the operational semantics, the type system also uses an environment to store information about identifiers. In this case, the environment maps identifiers to type associations. These can be either simple associations or universal associations, which are used for polymorphic functions.

Simple Associations These associate an identifier with a unique type, which can be either constant or a variable type. When the association is called, the type is returned as-is, even if it is a variable type.

Universal Associations This association, also called a type scheme, stores a type which contains at least one variable type bound by a "for all" quantifier ($\forall$). When called, this association creates a new variable type for each bound variable and returns a new instance of the type scheme. Universal associations are used exclusively for polymorphic functions.

To create this type of association, the type system must generate a list of "free variables" present in the type that is to be universalized. These are the variable types that are not present in the environment when the identifier is declared. When these free variables are found, they are universally bound. This ensures that only those variable types that are unbound in the environment become universally bound in the resulting association.

Constraint Unification After collecting every type constraint for the program, the type inference system attempts to unify these constraints and find a solution for them. This solution comes in the form of type substitutions, which associate variable types to other types, and type traits, which associate variable types to sets of traits.

If the constraints cannot be unified - that is, if a conflict is found -, the program is deemed not well-typed. Because of how the collection and unification process works, little information is given about where the problem ocurred.

Unification Application After obtaining a valid solution to the set of constraints, the type inference system applies the substitution to the type of the program. This is done recursively until no more substitutions are found, resulting in what is called the principal type. If there are any variable types in the principal type, the traits are applied to them, restricting the set of types that the variable types can represent.

Pattern Matching When a pattern is encountered (such as a `let` expression or function declaration), it is necessary to match the type of the pattern with the value.

To do this, a "match" function is defined. It takes a pattern p and a type T, returning a list of constraints and a mapping of identifiers to associations.

The following are the rules for the "match" function:

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

$$match(x : T_1, T_2) = \{T_1 = T_2\}, \{x \rightarrow T_1\}$$

$$match(n, T) = \{T = Int\}, \{\}$$

$$match(n : T_1, T_2) = \{T_1 = Int, T_2 = Int\}, \{\}$$

$$match(b, T) = \{T = Bool\}, \{\}$$

$$match(b : T_1, T_2) = \{T_1 = Bool, T_2 = Bool\}, \{\}$$

$$match(c, T) = \{T = Char\}, \{\}$$

$$match(c : T_1, T_2) = \{T_1 = Char, T_2 = Char\}, \{\}$$

$$match(,T) = \{\}, \{\}$$

$$match(:T_1, T_2) = \{T_1 = T_2\}, \{\}$$

$$\frac{X_1 \ is \ new}{match(nil, T) = \{X_1 list = T\}, \{\}}$$

$$\frac{X_1 \ is \ new}{match(nil : T_1, T_2) = \{X_1 list = T_1, T_1 = T_2\}, \{\}}$$

$$\frac{X_1 \ is \ new \qquad match(p_1, X_1) = c_1, env_1 \qquad match(p_2, X_1 list) = c_2, env_2}{match(p_1 :: p_2, T) = \{X_1 list = T\} \cup c_1 \cup c_2, env_1 \cup env_2}$$

$$\frac{X_1 \textit{ is new} \qquad \textit{match}(p_1, X_1) = c_1, \textit{env}_1 \qquad \textit{match}(p_2, X_1 \textit{list}) = c_2, \textit{env}_2}{\textit{match}(p_1 :: p_2 : T_1, T_2) = \{X_1\textit{list} = T_1, T_1 = T_2\} \cup c_1 \cup c_2, \textit{env}_1 \cup \textit{env}_2}$$

$$\frac{\forall i \in [1,n] \quad X_i \textit{ is new} \wedge \textit{match}(p_i, X_i) = c_i, \textit{env}_i}{\textit{match}((p_1, ...p_n), T) = \{(X_i, ...X_n) = T\} \cup \bigcup_{i=1}^{n} c_i, \bigcup_{i=1}^{n} \textit{env}_i}$$

$$\frac{\forall i \in [1,n] \quad X_i \textit{ is new} \wedge \textit{match}(p_i, X_i) = c_i, \textit{env}_i}{\textit{match}((p_1, ...p_n) : T_1, T_2) = \{(X_i, ...X_n) = T_1, T_1 = T_2\} \cup \bigcup_{i=1}^{n} c_i, \bigcup_{i=1}^{n} \textit{env}_i}$$

$$\frac{\forall i \in [1,n] \quad X_i \textit{ is new} \wedge \textit{match}(p_i, X_i) = c_i, \textit{env}_i \qquad X_0^{\{\{l_i:X_i\}\}\forall i\in[1,n]}}{\textit{match}(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\}, T) = \{X_0 = T\} \cup \bigcup_{i=1}^{n} c_i, \bigcup_{i=1}^{n} \textit{env}_i}$$

$$\frac{\forall i \in [1,n] \quad X_i \textit{ is new} \wedge \textit{match}(p_i, X_i) = c_i, \textit{env}_i \qquad X_0^{\{\{l_i:X_i\}\}\forall i\in[1,n]}}{\textit{match}(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\} : T_1, T_2) = \{X_0 = T_1, T_1 = T_2\} \cup \bigcup_{i=1}^{n} c_i, \bigcup_{i=1}^{n} \textit{env}_i}$$

$$\frac{\forall i \in [1,n] \quad X_i \textit{ is new} \wedge \textit{match}(p_i, X_i) = c_i, \textit{env}_i}{\textit{match}(\{l_1 : p_1, \ldots, l_n : p_n\}, T) = \{\{l_1 : X_1, \ldots, l_n : X_n\} = T\} \cup \bigcup_{i=1}^{n} c_i, \bigcup_{i=1}^{n} \textit{env}_i}$$

$$\frac{\forall i \in [1,n] \quad X_i \textit{ is new} \wedge \textit{match}(p_i, X_i) = c_i, \textit{env}_i}{\textit{match}(\{l_1 : p_1, \ldots, l_n : p_n\} : T_1, T_2) = \{\{l_1 : X_1, \ldots, l_n : X_n\} = T_1, T_1 = T_2\} \cup \bigcup_{i=1}^{n} c_i, \bigcup_{i=1}^{n} \textit{env}_i}$$

Constraint Collection Rules Every expression in V has a rule for constraint collection, similar to how every expression has a rule for its semantic evaluation.

$$\Gamma \vdash n : \text{Int} \mid \{\} \qquad \text{(T-NUM)}$$

$$\Gamma \vdash b : \text{Bool} \mid \{\} \qquad \text{(T-BOOL)}$$

$$\Gamma \vdash c : \text{Char} \mid \{\} \qquad \text{(T-CHAR)}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T \mid \{\}} \qquad \text{(T-IDENT)}$$

$$\frac{\forall\, k \in [1,n] \quad \Gamma \vdash e_k : T_k \mid C_k}{\Gamma \vdash (e_1,\ \ldots\ e_n) : (T_1,\ \ldots\ T_n) \mid C_1 \cup \cdots C_n} \quad \text{(T-Tuple)}$$

$$\frac{\forall\, k \in [1,n] \quad \Gamma \vdash e_k : T_k \mid C_k}{\Gamma \vdash \{l_1 : e_1,\ \ldots\ l_n : e_n\} : \{l_1 : T_1,\ \ldots\ l_n : T_n\} \mid C_1 \cup \cdots C_n} \quad \text{(T-Record)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{\{l:T_1\}\}}\ \textit{is new}}{\Gamma \vdash \#l\ e_1\ e_2 : (T_1, T_2) \mid C_1 \cup C_2 \cup \{X_1 = T_2\}} \quad \text{(T-RecordAccess)}$$

$$\frac{X_1\ \textit{is new}}{\Gamma \vdash \textit{nil} : X_1\ \textit{list} \mid \{\}} \quad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 :: e_2 : T_1\ \textit{list} \mid C_1 \cup C_2 \cup \{T_1\ \textit{list} = T_2\}} \quad \text{(T-List)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 + e_2 : \text{Int} \mid C_1 \cup C_2 \cup \{T_1 = \text{Int}; T_2 = \text{Int}\}} \quad \text{(T-+)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 - e_2 : \text{Int} \mid C_1 \cup C_2 \cup \{T_1 = \text{Int}; T_2 = \text{Int}\}} \quad \text{(T–)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 * e_2 : \text{Int} \mid C_1 \cup C_2 \cup \{T_1 = \text{Int}; T_2 = \text{Int}\}} \quad \text{(T-*)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2}{\Gamma \vdash e_1 \div e_2 : \text{Int} \mid C_1 \cup C_2 \cup \{T_1 = \text{Int}; T_2 = \text{Int}\}} \quad \text{(T-÷)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{Equatable\}}\ \textit{is new}}{\Gamma \vdash e_1 = e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{Equatable\}} = T_2\}} \quad \text{(T-=)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{Equatable\}}\ \textit{is new}}{\Gamma \vdash e_1 \neq e_2 : \text{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{Equatable\}} = T_2\}} \quad \text{(T-≠)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{Orderable\}}\ \textit{is new}}{\Gamma \vdash e_1 < e_2 : \mathrm{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{Orderable\}} = T_2\}} \quad \text{(T-<)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{Orderable\}}\ \textit{is new}}{\Gamma \vdash e_1 \leq e_2 : \mathrm{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{Orderable\}} = T_2\}} \quad \text{(T-}\leq\text{)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{Orderable\}}\ \textit{is new}}{\Gamma \vdash e_1 > e_2 : \mathrm{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{Orderable\}} = T_2\}} \quad \text{(T->)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1^{\{Orderable\}}\ \textit{is new}}{\Gamma \vdash e_1 \geq e_2 : \mathrm{Bool} \mid C_1 \cup C_2 \cup \{T_1 = T_2; X_1^{\{Orderable\}} = T_2\}} \quad \text{(T-}\geq\text{)}$$

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \qquad X_1\ \textit{is new}}{\Gamma \vdash e_1\ e_2 : X \mid C_1 \cup C_2 \cup \{T1 = T_2 \rightarrow X_1} \quad \text{(T-App)}$$

$$\frac{X\ \textit{is new} \qquad \textit{match}(p, X) = C, \textit{env} \qquad \textit{env} \cup \Gamma \vdash e : T_1 \mid C_1}{\Gamma \vdash \texttt{fn}\ p \Rightarrow e : X \rightarrow T_1 \mid C \cup C_1} \quad \text{(T-Fn)}$$

$$\frac{\begin{array}{c} X\ \textit{is new} \qquad \textit{match}(p, X) = C, \textit{env} \\ \{x \rightarrow (X \rightarrow T)\} \cup \textit{env} \cup \Gamma \vdash e : T_1 \mid C_1 \end{array}}{\Gamma \vdash \texttt{rec}\ x : T \quad p \Rightarrow e : X \rightarrow T_1 \mid C \cup C_1 \cup \{T_1 = T\}} \quad \text{(T-Rec)}$$

$$\frac{\begin{array}{c} X_1\ \textit{is new} \qquad X_2\ \textit{is new} \qquad \textit{match}(p, X_1) = C, \textit{env} \\ \{x \rightarrow X_2\} \cup \textit{env} \cup \Gamma \vdash e : T_1 | C_1 \end{array}}{\Gamma \vdash \texttt{rec}\ x\ p \Rightarrow e : X_1 \rightarrow T_1 \mid C \cup C_1 \cup \{X_2 = X_1 \rightarrow T_1\}} \quad \text{(T-Rec2)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : T_1 \mid C_1 \qquad \textit{match}(p, T_1) = C, \textit{env} \\ \textit{env} \cup \Gamma \vdash e_2 : T_2 \mid C_2 \end{array}}{\Gamma \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 : T_2 \mid C \cup C_1 \cup C_2} \quad \text{(T-Let)}$$

$$\frac{\begin{array}{c} \Gamma \vdash e : T \mid C \qquad X_1\ \textit{is new} \\ \forall j \in [1..n]\ \textit{multiMatch}(T, X_1, \Gamma, \textit{match}_j) = C_j \end{array}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ \textit{match}_1, \ldots\ \textit{match}_n : X_1 \mid C \cup \bigcup_{i=1}^{n} C_i} \quad \text{(T-Match)}$$

$$\frac{match(p, T_1) = C, \Gamma_1 \qquad \Gamma_1 \cup \Gamma \vdash e : T_3 \mid C_3}{multiMatch(T_1, T_2, \Gamma, p \to e) = C \cup C_3 \cup \{T_3 = T_2\}}$$

$$\frac{match(p, T_1) = C, \Gamma_1 \qquad \Gamma_1 \cup \Gamma \vdash e_1 : T_3 \mid C_3 \qquad \Gamma_1 \cup \Gamma \vdash e_2 : T_4 \mid C_4}{multiMatch(T_1, T_2, \Gamma, p \text{ when } e_1 \to e_2) = C \cup C_3 \cup C_4 \cup \{T_3 = Bool, T_4 = T_2}$$

$$\frac{X_1 \ \ is\ new}{\Gamma \vdash raise : X_1 \mid \{\}} \qquad \text{(T-R{\scriptsize AISE})}$$