V Syntax and Documentation

Arthur Giesel Vedana

October 14, 2019

Contents

Introduction

The *V* programming language is a functional language with eager left-to-right evaluation. It has a simple I/O system supporting only direct string operations. It is a trait based strongly and statically typed language supporting both explicit and implicit typing.

This document both specifies the *V* language and shows its implementation in F#. It is divided into 6 categories:

1. Abstract Syntax and Semantics

   This defines the abstract syntax and semantics of the language. It only contains the bare minimum for the language to function, without any syntactic sugar.

2. Extended Language

   Defines the extended abstract syntax tree and its translation into the core language.

3. Concrete Syntax

   Defines the concrete syntax for the language, describing what are valid expressions and programs.

4. Language Guide

   A guide for programming in *V*. This defines all operators, syntactic sugar and other aspects of the language, along with short user-friendly explanations of each language feature (and limitation).

5. Standard Library

   Describes all functions provided in the *V* standard library.

6. Changelog

   Lists the changes done to the language in each version.

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.

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

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:

e	::=	⋯	
	|	*Builtin*	
Builtin	::=	+	(add, binary)
	|	−	(subtract, binary)
	|	∗	(multiply, binary)
	|	÷	(divide, binary)
	|	−	(negate, unary)
	|	<	(less than, binary)
	|	≤	(less than or equal, binary)
	|	>	(more than, binary)
	|	≥	(more than or equal, binary)
	|	=	(equal, binary)
	|	≠	(not equal, binary)
	|	∨	(Or, binary)
	|	∧	(And, binary)
	|	get	(binary)
	|	set	(ternary)
	|	stack	(binary)
	|	distort	(ternary)

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 ∨ and ∧ 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 ∨ or False in the case of ∧), 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.

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

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.

e	::=	$\cdots$	
	|	`match` e `with` $match_1, \ldots match_n$	$(n \geq 1)$

match	::=	$p \rightarrow e$	
	|	p `when` $e_1 \rightarrow e_2$	

IO The language supports simple input and output through the `read` and `write` built-in functions. These functions operate on values constructed on the `IO` constructor, and a new empty value (`Void`) is introduced to represent the result of evaluating the `write` function.

Builtin	::=	$\cdots$	
	|	read	(unary)
	|	write	(unary)

con	::=	$\cdots$	
	|	IO	(arity 1)
	|	Void	(arity 0)

Both `IO` functions operate on single characters; `read` receives one character from the standard system IO (console), while `write` prints a single character to the system IO.

To allow composition of these functions, two monadic operations were introduced to the language: `return` and `bind`.

Builtin	::=	$\cdots$	
	|	return	(unary)
	|	bind	(binary)

`return` is a function that takes any value and returns that value encapsulated in the `IO` constructor. `bind` takes an `IO` value and a function, returning the result of applying the function to the value that was encapsulated in the `IO`.

Exceptions This expression always evaluates to a runtime error.

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

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.

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

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.

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

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 **??**).

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

$$
X ::= X_1, X_2, ...
$$

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).

$$
\begin{array}{lcll}
Traits & ::= & \varnothing & \\
 & | & \{Trait\} \cup Traits &
\end{array}
$$

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

The information on which types conform to which traits is defined in the unification environment (see 1.3.2). When a type T conforms to a trait $Trait$, the notation used is $T \in Trait$. The same notation can be used to describe when a type conforms to a set of traits $Traits$ (i.e. $T \in Traits$).

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:

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

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:

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

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:

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

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:

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

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 $\ll \mathit{Builtin}\ .\ v_1,\ \ldots v_n \gg$ 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.

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

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

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

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.

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

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 applying the distortions.

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

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.

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:

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

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

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

$$\frac{k \geq n \qquad \forall\ i \in [1, n] \quad \exists\ j \in [1, k] \quad l^1_i = l^2_j \wedge match(p_i, v_j) = vars_i}{match(\{l^1_1 : p_1, \ldots, l^1_n : p_n, \ldots\}, \{l^2_1 : v_1, ...l^2_k : v_k\}) = \bigcup_{i=1}^{n} vars_i}$$

$$\frac{\forall\ i \in [1, n] \quad l^1_i = l^2_i \wedge match(p_i, v_i) = vars_i}{match(\{l^1_1 : p_1, \ldots, l^1_n : p_n\}, \{l^2_1 : v_1, ...l^2_n : v_n\}) = \bigcup_{i=1}^{n} vars_i}$$

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.

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

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

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.

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

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.

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

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

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.

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

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.

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

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

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

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 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.

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

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.

$$\text{env} \vdash \#l \Downarrow \#l \quad \text{(BS-LABEL)}$$

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

$$\frac{\forall\ k \in [1, n] \quad \text{env} \vdash e_k \Downarrow \#\mathit{path}_k}{\text{env} \vdash \#(e_1,\ \ldots\ e_n) \Downarrow \#(\mathit{path}_1,\ \ldots\ \mathit{path}_n)} \quad \text{(BS-JOINED)}$$

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.

$$\frac{\text{env} \vdash e_1 \Downarrow\!\ll \text{stack}\ .\ \#\mathit{path}_1 \gg \qquad \text{env} \vdash e_2 \Downarrow \#\mathit{path}_2}{\text{env} \vdash e_1\ e_2 \Downarrow \#\mathit{path}_1\ .\ \mathit{path}_2} \quad \text{(BS-STACKED)}$$

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.

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

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.

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

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).

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

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.

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c}\text{env} \vdash e_1 \Downarrow \ll = \ .\ (con\ v_1^1,\ \ldots v_n^1) \gg \qquad \text{env} \vdash e_2 \Downarrow (con'\ v_1^2,\ \ldots v_k^2) \\ con' \neq con\end{array}}{\text{env} \vdash e_1\ e_2 \Downarrow false} \text{ (BS-=ConFalse2)}$$

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

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

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

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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

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

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

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

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*.

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

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

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.

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

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

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

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

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

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

IO Expressions Evaluating any IO expression requires interaction with an operating system, and how this is done is left as a matter of implementation. The only requirement imposed by the operational rules is that reading and writing be done one character at a time.

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \text{read} \ldots \gg \qquad \text{env} \vdash e_1 \Downarrow \text{Void} \qquad c \text{ is the next character in the standard input}}{\text{env} \vdash e_1\ e_2 \Downarrow \text{IO}\ c} \quad \text{(BS-Read)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \text{write} \ldots \gg \qquad \text{env} \vdash e_1 \Downarrow c \qquad c \text{ is written to the standard input}}{\text{env} \vdash e_1\ e_2 \Downarrow \text{IO}\ Void} \quad \text{(BS-Write)}$$

Composing IO In order to compose IO operations, two monadic functions were introduced in the language: return and bind. Currently, the only kind of value that can be manipulated with these functions is IO, but this system can be extended to support any monadic type in the future.

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \text{return} \ldots \gg \qquad \text{env} \vdash e_1 \Downarrow v}{\text{env} \vdash e_1\ e_2 \Downarrow \text{IO}\ v} \quad \text{(BS-Return)}$$

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \text{bind} \ldots \text{IO}\ v_1 \gg \qquad \text{env} \vdash e_1 \Downarrow v_2 \qquad \text{env} \vdash v_2\ v_1 \Downarrow v_3}{\text{env} \vdash e_1\ e_2 \Downarrow v_3} \quad \text{(BS-Bind)}$$

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*.

1.3 Type System

Like many programming languages, *V* has a type system. A type system is a way of statically analyzing programs to decide whether they are well-formed or not. To do this, every expression in the abstract syntax tree has, associated with it, type information.

Typing rules are then used to check that a program is correctly constructed. If a program passes the type system, guarantees are made about its execution. *V*'s type system is not *secure* in the sense that correctly typed programs will always have correct execution.

Examples of programs that pass the type system but fail to execute are division by zero and accessing the head of an empty list. In general, however, these errors are caused by incomplete pattern matching on algebraic data types, and most errors are still caught by the type system.

V's type system is a Hindley-Milner style type system, with support for implicit type annotations and let polymorphism. Furthermore, *traits* allow ad-hoc polymorphism and, used with records, a kind of structural subtyping (more details will be provided later).

The Type Inference Algorithm Since *V* allows implicitly typed expressions (that is, expressions without any type annotations provided by the programmer), it is necessary to infer, and not only check, the type of expressions. A constraint-based inference system is used, which divides the algorithm into three parts: constraint collection, in which the abstract syntax tree is traversed and both a type and a list of type equality constraints is generated; constraint unification, in which the list of constraints is condensed into a type substitution; and substitution application, which applies the substitution to the type to obtain a principle type.

1.3.1 Constraint Collection

The first step of the type inference algorithm is the collection of type constraints. A type constraint is an equality between two types, and so the result of the constraint collection is a system of equations.

The constraint collection algorithm takes, as input, an expression e and a typing environment Γ (defined below), and produces, as an output, a type T, a set of constraints C and a unification environment γ (defined in 1.3.2). The algorithm is given as a set of rules of the form:

$$\Gamma \vdash e : T \mid C \mid \gamma \tag{T}$$

Constraints As described above, a set of constraints C is composed of equalities between two types.

$$C \quad ::= \quad \emptyset \mid \{T_1 = T_2\} \cup C$$

Environment The type inference environment is a 2-tuple with the following components:

1. Mapping between constructors and their type

   Every constructor has a type associated with it. This type will be a function if the constructor has arity greater than 0, and can contain variable types.

2. Mapping between identifiers and type associations

   An identifier can be either simply or universally bound to a type. The difference between these associations will be explained later.

$$\begin{array}{lll} \Gamma & ::= & (constructors, vars) \\ \\ constructors & ::= & \{\} \mid \{con \rightarrow T\} \cup constructors \\ vars & ::= & \{\} \mid \{x \rightarrow assoc\} \cup vars \\ \\ assoc & ::= & T \qquad \text{(Simple Association)} \\ & \mid & \forall X_1, \ldots X_n \,.\, T \qquad \text{(Universal Association)} \end{array}$$

Type Associations When identifiers are bound in a program (with pattern matching, for example), an association between the identifier and its type is added to the typing environment. Depending on the type of binding, however, this association can be one of two types: simple or universal.

A simple association binds a name to a monomorphic type. This type can "simple", such as Int, or it can contain variable types, such as $X \rightarrow$ Bool. In either case, however, the type is "constant", and it is returned unchanged from the environment.

A universal association binds a name to a type scheme. A type scheme is composed of a type T and any number of type variables $X_1, \cdots X_n$. These variables are free in the type T, and a new instance of them is generated every time the association is returned from the environment.

Universal associations allow for polymorphic functions in the language, so each use of the function does not add constraints to other uses. The only expressions that create universal associations are let-expressions. This means that function parameters cannot be polymorphic, since function parameters are bound to simple type associations.

Free Variables Type variables can either be bound or free relative to an environment. Bound variable types are those that are associated to an identifier. This association must be a simple association, but the variable type can occur anywhere in the type tree. As an example, the type variable X_1 is bound in the environment below:

$$\Gamma = \{x \to (Int, X_1, Char)\}$$

Inversely, free type variables are all those that do not occur bound in the environment.

A helper function, $\Gamma \vdash free(T)$, returns the set of all free type variables in the type T. Another function, $\Gamma \vdash fresh(T)$, returns a type T' in which all free type variables in T are replaced by new, unbound type variables. Both of these functions require an environment Γ with which to judge whether type variables are free or not.

Pattern Matching One way to bind identifiers is by pattern matching. When a pattern is encountered (such as a `let` expression), it is necessary to match the type of the pattern with the type of the value.

To do this, two auxiliary *match* functions are defined. Both take, as input, a pattern p and a type T, returning a list of constraints and a modified typing environment.

The first of the functions, *match*, only creates simple type associations and is used in `match` expressions. The second, $match_U$, can create both simple and universal associations, being used in `let` expressions.

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

$$\Gamma \vdash match(x, T) = \{\}, \{x \to T\} \cup \Gamma$$

$$\Gamma \vdash match(x : T_{pat}, T) = \{T_{pat} = T\}, \{x \to T\} \cup \Gamma$$

$$\Gamma \vdash match(\_, T) = \{\}, \Gamma$$

$$\Gamma \vdash match(\_ : T_{pat}, T) = \{T_{pat} = T\}, \Gamma$$

$$\frac{\begin{array}{c}\Gamma(con) = T' \qquad \Gamma \vdash fresh(T') = T'' \\ T'' = T_1 \to \cdots \to T_n \qquad \Gamma_0 = \Gamma \\ \forall i \in [1, n] \quad \Gamma_{i-1} \vdash match(p_i, T_i) = C_i, \Gamma_i\end{array}}{\Gamma \vdash match(con\ p_1,\ \ldots\ p_n, T) = \{T_n = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\begin{array}{c}\Gamma(con) = T' \qquad \Gamma \vdash fresh(T') = T'' \\ T'' = T_1 \to \cdots \to T_n \qquad \Gamma_0 = \Gamma \\ \forall i \in [1, n] \quad \Gamma_{i-1} \vdash match(p_i, T_i) = C_i, \Gamma_i\end{array}}{\Gamma \vdash match(con\ p_1,\ \ldots\ p_n : T_{pat}, T) = \{T_{pat} = T, T_n = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\forall i \in [1,n] \quad \Gamma_{i-1} \vdash fresh(X) = X_i \wedge \Gamma_{i-1} \vdash match(p_i, X_i) = C_i, \Gamma_i}{match(\{l_1 : p_1, \ldots, l_n : p_n\}, T) = \{\{l_1 : X_1, \ldots, l_n : X_n\} = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\forall i \in [1,n] \quad \Gamma_{i-1} \vdash fresh(X) = X_i \wedge \Gamma_{i-1} \vdash match(p_i, X_i) = C_i, \Gamma_i}{match(\{l_1 : p_1, \ldots, l_n : p_n\} : T_{pat}, T) = \{\{l_1 : X_1, \ldots, l_n : X_n\} = T, T_{pat} = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\forall i \in [1,n] \quad \Gamma_{i-1} \vdash fresh(X) = X_i \wedge \Gamma_{i-1} \vdash match(p_i, X_i) = C_i, \Gamma_i \qquad X_0^{\{\{l_i:X_i\},\ldots\{l_n:X_n\}\}}}{match(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\}, T) = \{X_0 = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\forall i \in [1,n] \quad \Gamma_{i-1} \vdash fresh(X) = X_i \wedge \Gamma_{i-1} \vdash match(p_i, X_i) = C_i, \Gamma_i \qquad X_0^{\{\{l_i:X_i\},\ldots\{l_n:X_n\}\}}}{match(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\} : T_{pat}, T) = \{X_0 = T, T_{pat} = T\} \cup \bigcup_{i=1}^{n} c_i, \Gamma_n}$$

The rules for the $match_U$ are similar, but with a few key differences. Since universal matching is always called with a completely unified type in a `let` expression, certain concessions can be made. Furthermore, other differences arise due to the ability for the $match_U$ function to create universal type associations.

Because $match_U$ can create universal associations, it checks for any free variable types in the type T. If free variable types are found, then a universal association is made based on these variable types. If there are no free variable types, a simple association is created instead.

$$\frac{\Gamma \vdash free(T) = \{\}}{\Gamma \vdash match_U(x, T) = \{\}, \{x \to T\} \cup \Gamma}$$

$$\frac{\Gamma \vdash free(T) = \{X_1, \ldots, X_n\}}{\Gamma \vdash match_U(x, T) = \{\}, \{x \to \forall X_1, \ldots X_n \,.\, T\} \cup \Gamma}$$

$$\frac{\Gamma \vdash free(T) = \{\}}{\Gamma \vdash match_U(x : T_{pat}, T) = \{T_{pat} = T\}, \{x \to T\} \cup \Gamma}$$

$$\frac{\Gamma \vdash free(T) = \{X_1, \ldots, X_n\}}{\Gamma \vdash match_U(x : T_{pat}, T) = \{T_{pat} = T\}, \{x \to \forall X_1, \ldots X_n \,.\, T\} \cup \Gamma}$$

$$\Gamma \vdash match_U(\_, T) = \{\}, \Gamma$$

$$\Gamma \vdash match_U(\_ : T_{pat}, T) = \{T_{pat} = T\}, \Gamma$$

The next difference comes when matching against constructor patterns. Instead of creating a fresh instance of the type associated with the constructor by replacing all variable types with fresh variable types, the type T (passed as paramenter to $match_U$) is used.

This function is called $\Gamma \vdash rebase(T_1, T_2)$, creating a new instance of T_1 based on T_2. Informally, this means that both the type T_1 and T_2 are traversed simultaneously and, when a variable type is encountered in T_1, it is replaced by the equivalent type in T_2.

$$\frac{\begin{array}{c}\Gamma(con) = T' \qquad \Gamma \vdash rebase(T', T) = T'' \\ T'' = T_1 \to \cdots \to T_n \qquad \Gamma_0 = \Gamma \\ \forall i \in [1, n] \quad \Gamma_{i-1} \vdash match_U(p_i, T_i) = C_i, \Gamma_i\end{array}}{\Gamma \vdash match_U(con\ p_1,\ \ldots\ p_n, T) = \{T_n = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\begin{array}{c}\Gamma(con) = T' \qquad \Gamma \vdash rebase(T', T) = T'' \\ T'' = T_1 \to \cdots \to T_n \qquad \Gamma_0 = \Gamma \\ \forall i \in [1, n] \quad \Gamma_{i-1} \vdash match_U(p_i, T_i) = C_i, \Gamma_i\end{array}}{\Gamma \vdash match_U(con\ p_1,\ \ldots\ p_n : T_{pat}, T) = \{T_{pat} = T, T_n = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

Finally, pattern matching on records in also slightly different. For complete record patterns, we know that the type T is a record type with the necessary fields, and so the matching rule becomes much smaller.

$$\frac{\begin{array}{c}T = \{l_1 : T_1, \ldots, l_n : T_n\} \\ \forall i \in [1, n] \quad \Gamma_{i-1} \vdash match_U(p_i, T_i) = C_i, \Gamma_i\end{array}}{match(\{l_1 : p_1, \ldots, l_n : p_n\}, T) = \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\begin{array}{c}T = \{l_1 : T_1, \ldots, l_n : T_n\} \\ \forall i \in [1, n] \quad \Gamma_{i-1} \vdash match(p_i, T_i) = C_i, \Gamma_i\end{array}}{match(\{l_1 : p_1, \ldots, l_n : p_n\} : T_{pat}, T) = \{T_{pat} = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

For partial record patterns, however, a little more care must be taken. Since there may be fewer fields in the pattern than the type, a search must be done to match the correct sub-patterns with the sub-types.

$$\frac{\begin{array}{c}T = \{l'_1 : T_1, \ldots, l'_k : T_k\} \\ k \geq n \qquad \forall\ i \in [1, n] \quad \exists\ j \in [1, k] \quad l_i = l'_j \wedge \Gamma_{i-1} \vdash match(p_i, T_j) = C_i, \Gamma_i\end{array}}{match(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\}, T) = \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

$$\frac{\begin{array}{c} T = \{l'_1 : T_1, \ldots, l'_k : T_k\} \\ k \geq n \qquad \forall\, i \in [1,n] \;\; \exists\, j \in [1,k] \;\; l_i = l'_j \wedge \Gamma_{i-1} \vdash match(p_i, T_j) = C_i, \Gamma_i \end{array}}{match(\{l_1 : p_1, \ldots, l_n : p_n, \ldots\} : T_{pat}, T) = \{T_{pat} = T\} \cup \bigcup_{i=1}^{n} C_i, \Gamma_n}$$

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.

If a rule does not create any constraints or unification environment (i.e. they are both empty), then these will be omited to improve readability. As an example, the following rule:

$$\Gamma \vdash + : \text{Int} \to \text{Int} \to \text{Int} \mid \{\} \mid \{\} \tag{T-+}$$

will be written as:

$$\Gamma \vdash + : \text{Int} \to \text{Int} \to \text{Int} \tag{T-+}$$

Functions The rules for function expressions are all similar, though with a few differences between them. All of them create a fresh type variable X_1 to represent the type of their argument, and the resulting type is always $X_1 \to T_1$, where T_1 is the type of the body of the function.

When calling the collection algorithm on the body of the function (i.e. e), the typing environment Γ is modified by addind a new association between the identifier x and the type X_1.

$$\frac{\Gamma \vdash fresh(X) = X_1 \qquad \{x \to X_1\} \cup \Gamma \vdash e : T_1 \mid C_1 \mid \gamma_1}{\Gamma \vdash \texttt{fn}\ x \Rightarrow e : X_1 \to T_1 \mid C_1 \mid \gamma_1} \tag{T-Fn}$$

Recursive functions add the same association between x and X_1, but they also create a new association for the name of the function, f. If the function is implicitly typed, a new type variable, X_2, is used to represent the type of the function. Thus, f is associated to X_2, and a new constraint between X_2 and $X_1 \to T_1$ is created.

$$\frac{\begin{array}{c} \Gamma \vdash fresh(X) = X_1 \qquad \Gamma \vdash fresh(X) = X_2 \\ \{f \to X_2, x \to X_1\} \cup \Gamma \vdash e : T_1 | C_1 \mid \gamma_1 \end{array}}{\Gamma \vdash \texttt{rec}\ f\ x \Rightarrow e : X_1 \to T_1 \mid C_1 \cup \{X_2 = X_1 \to T_1\} \mid \gamma_1} \tag{T-Rec}$$

If the function is explicitly typed, however, no new type variables are created. Instead, f is associated directly to $X_1 \to T$, and a constraint to guarantee that the provided type is correct is created (that is, that T is equal to T_1).

$$\frac{\Gamma \vdash fresh(X) = X_1 \qquad \{f \to (X_1 \to T, x \to X_1)\} \cup \Gamma \vdash e : T_1 \mid C_1 \mid \gamma_1}{\Gamma \vdash \texttt{rec}\ f : T \;\; x \Rightarrow e : X \to T_1 \mid C_1 \cup \{T_1 = T\} \mid \gamma_1} \tag{T-Rec2}$$

Built-in Functions None of the built-in functions create any constraints or unification environment, nor do their types depend on a typing environment. Some functions, because of their polymorphic nature, require creation of fresh type variables.

Numerical Functions These functions all manipulate Int values, with negate ($-$) being the only function with a single argument.

$$\Gamma \vdash + : \text{Int} \to \text{Int} \to \text{Int} \tag{T-+}$$

$$\Gamma \vdash - : \text{Int} \to \text{Int} \to \text{Int} \tag{T–}$$

$$\Gamma \vdash * : \text{Int} \to \text{Int} \to \text{Int} \tag{T-*}$$

$$\Gamma \vdash \div : \text{Int} \to \text{Int} \to \text{Int} \tag{T-÷}$$

$$\Gamma \vdash - : \text{Int} \to \text{Int} \tag{T-Negate}$$

Equality Functions These functions do not require a specific type for their arguments, but they both must be equal and conform to the *Equatable* trait.

$$\frac{\Gamma \vdash fresh(X^{\{Equatable\}}) = T}{\Gamma \vdash =: T \to T \to \text{Bool}} \tag{T-=}$$

$$\frac{\Gamma \vdash fresh(X^{\{Equatable\}}) = T}{\Gamma \vdash \neq: T \to T \to \text{Bool}} \tag{T-≠}$$

Inequality Functions Similar to equality, both arguments must have the same type and conform to the *Orderable* trait.

$$\frac{\Gamma \vdash fresh(X^{\{Orderable\}}) = T}{\Gamma \vdash <: T \to T \to \text{Bool}} \tag{T-<}$$

$$\frac{\Gamma \vdash fresh(X^{\{Orderable\}}) = T}{\Gamma \vdash \leq: T \to T \to \text{Bool}} \tag{T-≤}$$

$$\frac{\Gamma \vdash fresh(X^{\{Orderable\}}) = T}{\Gamma \vdash >: T \to T \to \text{Bool}} \tag{T->}$$

$$\frac{\Gamma \vdash fresh(X^{\{Orderable\}}) = T}{\Gamma \vdash \geq: T \to T \to \text{Bool}} \tag{T-≥}$$

Boolean Functions Both functions require two Bool arguments, returning another Bool.

$$\Gamma \vdash \vee : \text{Bool} \to \text{Bool} \to \text{Bool} \tag{T-$\vee$}$$

$$\Gamma \vdash \wedge : \text{Bool} \to \text{Bool} \to \text{Bool} \tag{T-$\wedge$}$$

Accessor Functions These functions manipulate accessors, creating fresh type variables to represent all necessary types.

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2}{\Gamma \vdash \text{get} : T_2 \# T_1 \to T_2 \to T_1} \tag{T-GET}$$

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2}{\Gamma \vdash \text{set} : T_2 \# T_1 \to T_1 \to T_2 \to T_2} \tag{T-SET}$$

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2 \qquad \Gamma \vdash fresh(X) = T_3}{\Gamma \vdash \text{stack} : T_2 \# T_1 \to T_1 \# T_3 \to T_2 \# T_3} \tag{T-STACK}$$

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2 \qquad \Gamma \vdash fresh(X) = T_3}{\Gamma \vdash \text{distort} : T_2 \# T_1 \to (T_1 \to T_3) \to (T_3 \to T_1 \to T_1) \to T_2 \# T_3} \tag{T-DISTORT}$$

IO Functions The typing rules for both IO related functions are very simple, relying on Void_T and IO_T to define their inputs and outputs.

$$\Gamma \vdash \text{read} : \text{Void}_T \to \text{IO}_T\ \text{Char} \tag{T-READ}$$

$$\Gamma \vdash \text{write} : \text{Char} \to \text{IO}_T\ \text{Void}_T \tag{T-GET}$$

For IO composition, the return and bind functions also have straightforward type signatures.

$$\frac{\Gamma \vdash fresh(X) = T_1}{\Gamma \vdash \text{return} : T_1 \to \text{IO}_T\ T_1} \tag{T-RETURN}$$

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X) = T_2}{\Gamma \vdash \text{write} : \text{IO}_T\ T_1 \to (T_1 \to \text{IO}_T\ T_2) \to \text{IO}_T\ T_2} \tag{T-BIND}$$

Constructors The rule for typing constructors is very simple. The type is extracted from the environment, and then a fresh instance is generated from that type.

$$\frac{\Gamma(con) = T \qquad \Gamma \vdash fresh(T) = T'}{\Gamma \vdash con : T'} \tag{T-Con}$$

Application The constraint collection rule for an application is simple, creating just one fresh type variable and one new constraint. The type variable X_1, represents the type of the result of the application, and, therefore, is the return type of the collection. Furthermore, the type of e_1, T_1, must be equal to a function that takes T_2 (the type of e_2) as an argument and returns X_1.

$$\frac{\Gamma \vdash e_1 : T_1 \mid C_1 \mid \gamma_1 \qquad \Gamma \vdash e_2 : T_2 \mid C_2 \mid \gamma_2 \qquad \Gamma \vdash fresh(X) = X_1}{\Gamma \vdash e_1\ e_2 : X_1 \mid C_1 \cup C_2 \cup \{T_1 = T_2 \to X_1\} \mid \gamma_1 \cup \gamma_2} \text{ (T-App)}$$

Identifiers The type of an identifier is, like for constructors, completely defined by its typing association in the environment. The typing rule does not create a fresh instance of this type, since the environment already does this when returning types that are universally bound.

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \text{ (T-Ident)}$$

Records The constraint collection rule for a record is relatively straightforward. Each field of the record is passed through the collection algorithm, and the resulting types are combined into a single record type with their matching labels. Similarly, the resulting constraints and unification environments are combined by union.

$$\frac{\forall\ k \in [1, n] \quad \Gamma \vdash e_k : T_k \mid C_k \mid \gamma_k}{\Gamma \vdash \{l_1 : e_1,\ \ldots\ l_n : e_n\} : \{l_1 : T_1,\ \ldots\ l_n : T_n\} \mid \bigcup_{i=1}^{n} C_i \mid \bigcup_{i=1}^{n} \gamma_i} \text{ (T-Record)}$$

Accessors The constraint rules for simple label accessors relies on type variables and record label traits. A new fresh type variable, T_1, is generated, representing the type of the field being accessed. Another new fresh type variable, T_2, which must conform to the record label trait associating the label l to the type T_1, is generated, representing the type of the record that is being accessed.

$$\frac{\Gamma \vdash fresh(X) = T_1 \qquad \Gamma \vdash fresh(X^{\{l:T_1\}}) = T_2}{\Gamma \vdash \#l : T_2 \# T_1} \text{ (T-Label)}$$

A joined accessor does not use record label traits, but instead relies on type variables and constraints to guarantee the correct type information.

A single type variable X_0, represents the record being accessed. For every component e_i of the accessor, a new type variable X_i is generated, along with the resulting type T_i of calling the constraint collection algorithm. The type T_i is then constrained to be equal to $X_0 \# X_i$, indicating that all components refer the same record, but access fields with (possibly) different types.

Finally, the resulting type is an accessor that returns a tuple composed of all X_i when accessing a record of type X_0.

$$\frac{fresh(X) = X_0 \qquad \forall\ i \in [1,n] \quad \Gamma \vdash fresh(X) = X_i \wedge \Gamma \vdash e_i : T_i \mid C_i \mid \gamma_i}{\Gamma \vdash \#(e_1,\ \ldots\ e_n) : X_0\#(X_i,\ \ldots\ X_n) \mid \bigcup_{i=1}^{n} C_i \cup \{T_i = X_0\#X_i\} \mid \bigcup_{i=1}^{n} \gamma_i} \text{ (T-Joined)}$$

Let Expression The constraint collection rule for a `let` expression depends on both the unification and the application algorithms.

The expression e_1 is passed through the constraint collection algorithm, resulting in a type T_1, a set of constraints C_1 and a unification environment γ_1. The constraints C_1 are then unified (see 1.3.2) under the unification environment γ_1, resulting in a substitution σ.

The substitution σ is then applied (see 1.3.3) to the type T_1, resulting in a principle type T_1'. The substitution is also applied to the environment Γ, and the result of this application is used to evaluate a universal match between p and T_1', resulting in a new set of constraints C_1' and a new typing environment Γ'.

Finally, the type of the expression e_2 is obtained under the environment Γ'.

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : T_1 \mid C_1 \mid \gamma_1 \qquad \gamma_1 \vdash \mathcal{U}(C_1) = \sigma \qquad \sigma(T_1) = T_1' \\ \sigma(\Gamma) \vdash match_U(p, T_1') = C_1', \Gamma' \qquad \Gamma' \vdash e_2 : T_2 \mid C_2 \mid \gamma_2\end{array}}{\Gamma \vdash \texttt{let}\ p = e_1\ \texttt{in}\ e_2 : T_2 \mid C_1' \cup C_1 \cup C_2 \mid \gamma_2 \cup \gamma_1} \text{ (T-Let)}$$

Match Expression The constraint collection rule for a match expression requires an auxiliary function, much like its operational semantic rule. A fresh type variable X_1 is created, representing the output type of the expression and, along with the type T of the expression e, is used to validate every $match_i$ in the expression.

$$\frac{\begin{array}{c}\Gamma \vdash e : T \mid C \mid \gamma \qquad \Gamma \vdash fresh(X) = X_1 \\ \forall i \in [1..n]\, \Gamma \vdash validate(match_i, T, X_1) = C_i \mid \gamma_i\end{array}}{\Gamma \vdash \texttt{match}\ e\ \texttt{with}\ match_1, \ldots match_n : X_1 \mid C \cup \bigcup_{i=1}^{n} C_i \mid \gamma \cup \bigcup_{i=1}^{n} \gamma_i} \text{ (T-Match)}$$

The *validate* function takes a *match* expression, a type T_{in}, representing the type of the pattern, and a T_{out}, representing the result of evaluating the *match* expression, as inputs. The function outputs a set of constraints and a unification environment if successful.

For an unconditional *match*, the pattern in matched against the provided input type T_{in} and the type of the expression e is constrained to equal the provided type T_{out}. It is important to realize that the typing environment returned by the match (i.e. Γ') is used only to obtain the type of e, since any identifiers bound in the pattern p can only be used inside a single *match* expression.

$$\frac{\Gamma \vdash match(p, T_{in}) = C, \Gamma' \qquad \Gamma' \vdash e : T_1 \mid C_1 \mid \gamma_1}{\Gamma \vdash validate(p \rightarrow e, T_{in}, T_{out}) = C \cup C_1 \cup \{T_1 = T_{out}\} \mid \gamma_1}$$

The same holds true for a conditional *match*, with the added verification that the type of e_1 must be equal to *Bool*.

$$\frac{\Gamma \vdash match(p, T_{in}) = C, \Gamma' \qquad \Gamma' \vdash e_1 : T_1 \mid C_1 \mid \gamma_1 \qquad \Gamma' \vdash e_2 : T_2 \mid C_2 \mid \gamma_2}{\Gamma \vdash validate(p \text{ when } e_1 \rightarrow e_2, T_{in}, T_{out}) = C \cup C_1 \cup C_2 \cup \{T_1 = Bool, T_2 = T_{out}\} \mid \gamma_1 \cup \gamma_2}$$

Exception The *raise* expression simply creates and returns a new fresh type variable.

$$\frac{\Gamma \vdash fresh(X) = X_1}{\Gamma \vdash raise : X_1} \qquad \text{(T-Raise)}$$

1.3.2 Unification

After constraint collection, the second step in the type inference algorithm is unification. Unification attempts to solve the set of equalities defined by the constraints collected in the previous step.

The algorithm takes, as input, a set of constraints C and a unification environment γ, and produces a substitution σ as output. The algorithm is given as a set of rules following the form:

$$\gamma \vdash \mathcal{U}(C) = \sigma \qquad \text{(U)}$$

Unification Environment The unification environment is a set of trait specifications. Trait specifications are 3-tuples that define the requirements for a specific type to conform to a specific trait.

$$\gamma \quad ::= \quad \{\} \mid \{\text{trtSpec}\} \cup \gamma$$

$$\text{trtSpec} \quad ::= \quad (conT, Trait, [Traits_1, \ldots Traits_n]) \qquad (n = \text{arity } conT)$$

A trait specification describes conformance to a trait. Some types, such as records and functions, have their trait conformance built into the language, and it is not necessary to use the unification environment to decide conformance. For a constructor type $conT\ T_1, \ldots\ T_n$ to conform to a trait $Trait$, however, the following criteria must hold:

1. There exists a trait specification $(conT, Trait, [Traits_1, \ldots Traits_n])$ in the unification environment
2. For all $i \in [1, n],\ T_i \in Traits_i$

This will be formally defined in the unification algorithm itself, but these are the general rules that govern trait conformance.

Unification Rules Unification iterates through the list of constraints, always operating on the first constraint in the list and recursing on the remaining constraints. Because of this format, it can be defined as a base case and recursion rules.

Base Cases The basic case, with an empty constraint list, returns an empty substitution.

$$\gamma \vdash \mathcal{U}(\emptyset) = \emptyset \qquad \text{(U-Empty)}$$

If both types of a constraint are equal, they are discarded and the recursion is called.

$$\frac{T_1 = T_2}{\gamma \vdash \mathcal{U}(\{T_1 = T_2\} \cup C) = \gamma \vdash \mathcal{U}(C) = \sigma} \qquad \text{(U-Equals)}$$

Compound Types When encountered, compound types are destructured and equality constraints with their corresponding components are added to the end of the constraint list.

$$\frac{\begin{array}{c} T_1 = T_1^1 \# T_1^2 \qquad T_2 = T_2^1 \# T_2^2 \\ C' = \{T_1^1 = T_2^1, T_1^2 = T_2^2\} \end{array}}{\gamma \vdash \mathcal{U}(\{T_1 = T_2\} \cup C) = \gamma \vdash \mathcal{U}(C \cup C')} \qquad \text{(U-Accessor)}$$

$$\frac{\begin{array}{c} T_1 = T_1^1 \to T_1^2 \qquad T_2 = T_2^1 \to T_2^2 \\ C' = \{T_1^1 = T_2^1, T_1^2 = T_2^2\} \end{array}}{\gamma \vdash \mathcal{U}(\{T_1 = T_2\} \cup C) = \gamma \vdash \mathcal{U}(C \cup C')} \qquad \text{(U-Fn)}$$

$$\frac{\begin{array}{c} T_1 = \{l_1 : T_1^1, \ \ldots \ l_n : T_1^n\} \qquad T_2 = \{l_1 : T_2^1, \ \ldots \ l_n : T_2^n\} \\ C' = \{T_1^1 = T_2^1, \ \ldots \ T_1^n = T_2^n\} \end{array}}{\gamma \vdash \mathcal{U}(\{T_1 = T_2\} \cup C) = \gamma \vdash \mathcal{U}(C \cup C')} \qquad \text{(U-Record)}$$

For applied constructor types, it is also necessary to verify that the constructor types themselves are equal. If not, the unification fails.

$$\frac{\begin{array}{c} T_1 = conT_1 \ T_1^1, \ \ldots \ T_1^n \qquad T_2 = conT_2 \ T_2^1, \ \ldots \ T_2^n \\ conT_1 = conT_2 \qquad C' = \{T_1^1 = T_2^1, \ \ldots \ T_1^n = T_2^n\} \end{array}}{\gamma \vdash \mathcal{U}(\{T_1 = T_2\} \cup C) = \gamma \vdash \mathcal{U}(C \cup C')} \qquad \text{(U-Cons)}$$

Type Variables When unifying a constraint that contains a type variable, a more complicated process of unification is necessary.

First, the type variable must not be contained in the free variables of its paired type, ensuring that the types are not circular. Then, the type T_2 must conform to the type variable's traits, and this process might create additional constraints that must be unified. These newly created constraints, along with the remaining constraints, are then unified after having every occurrence of X replaced with the type T_2.

$$\frac{\begin{array}{c} T_1 = X^{Traits} \qquad X \notin free(T_2) \\ T_2 \in Traits \to C_T \qquad C' = C \cup C_T \\ \gamma \vdash \mathcal{U}(C'[X \to T_2]) = \sigma \end{array}}{\gamma \vdash \mathcal{U}(\{T_1 = T_2\} \cup C) = \{X \to T_2\} \cup \sigma} \qquad \text{(U-Var)}$$

Substitution The result of applying the unification algorithm is a substitution σ. A substitution is a mapping from type variables to types.

$$\sigma \quad ::= \quad \emptyset \mid \{X \to T\} \cup \sigma$$

1.3.3 Application

The last component of the type inference algorithm is the application of a type substitution. This replaces all type variables that are specified by the substitution, resulting in a new instance of the input type.

Application takes, as input, a type T and a substitution σ, producing another type T' as output. It is defined with rules of the form:

$$\sigma(T) = T' \tag{A}$$

2 Extended Language

In order to facilitate programming, it is useful to define an extended language. A program is first parsed into this language, and the resulting tree is translated into the regular abstract syntax.

This allows the core language to be concise, reducing the complexity of type inference and evaluation. Complex constructs (such as comprehensions and multi-parameter functions) can be included only in the extended language, and it suffices to provide a translation into the core language.

This translation does have the drawback of reducing the formality of evaluation. Since there are no evaluation rules for the additional constructs, it is impossible to prove the correctness of the translation rules. This does not in any way affect the correctness of the core language type inference and evaluation, and the advantages gained by this method far outweigh the drawbacks, so it is still a net positive to the language.

The following two sections will describe the abstract syntax tree for the extended language and how it translates to a syntax tree in the core language.

2.1 Abstract Syntax

The extended language has terms which are similar to (if not exactly the same as) ones existing in the core language. These terms are presented in their entirety here, and, since they are directly extracted from the core language, no explanation will be given for them.

$$
\begin{array}{lcll}
e' & ::= & func' & \\
 & | & e'_1\ e'_2 & \\
 & | & x & \\
 & | & Builtin & \\
 & | & con & \\
 & | & \{l_1 : e'_1,\ \ldots\ l_n : e'_n\} & (n \geq 1) \\
 & | & \#l & \\
 & | & \#(e'_1,\ \ldots\ e'_n) & (n \geq 2) \\
 & | & raise & \\
 & | & \texttt{match}\ e'\ \texttt{with}\ match'_1, \ldots\ match'_n & (n \geq 1) \\
 & & & \\
match' & ::= & p' \rightarrow e' & \\
 & | & p'\ \texttt{when}\ e'_1 \rightarrow e'_2 &
\end{array}
$$

Most patterns are extracted from the core language without any differences, and are defined below.

$$
\begin{array}{lcll}
p' & ::= & patt' & \\
 & | & patt' : T' & \\
 & & & \\
patt' & ::= & x & \\
 & | & \_ & \\
 & | & con\ p'_1,\ \ldots\ p'_n & \text{(constructor pattern, } n = \text{arity } con\text{)} \\
 & | & \{l_1 : p'_1,\ \ldots\ l_n : p'_n\} & (n \geq 1) \\
 & | & \{l_1 : p'_1,\ \ldots\ l_n : p'_n, \ldots\} & \text{(partial record, } n \geq 1\text{)}
\end{array}
$$

Most types are, like patterns, extracted from the language.

$$
\begin{array}{lcll}
T' & ::= & X^{Traits} & \\
 & | & conT\ T'_1,\ \ldots\ T'_n & (n = \text{arity } conT) \\
 & | & T'_1 \rightarrow T'_2 & \\
 & | & \{l_1 : T'_1,\ \ldots\ l_n : T'_n\} & (n \geq 1) \\
 & | & T'_1 \# T'_2 & Accessor
\end{array}
$$

2.1.1 Additions

Type Aliases The first addition to the extended language is the concept of *type aliases*. These types are simple renames of existing types, and can be used in programs as a way to simplify type declarations.

$$
\begin{array}{lcl}
T' & ::= & \cdots \\
 & | & \tau \\
 & & \\
\tau & ::= & \{\tau_0, \tau_1, \ldots\}
\end{array}
$$

Conditional Expression A conditional expression, which translates to a `match` expression on the patterns *true* and $false$, has been added.

$$
\begin{array}{lcl}
e' & ::= & \cdots \\
 & | & \texttt{if}\ e'_1\ \texttt{then}\ e'_2\ \texttt{else}\ e'_3
\end{array}
$$

Multi-Parameter and Pattern Matching Functions Functions have been extended to allow multiple parameters, removing the necessity of declaring nested functions. These functions still require at least one parameter.

Furthermore, patterns are allowed as the definition of parameters.

$$
\begin{array}{lcll}
func' & ::= & \cdots & \\
 & | & \texttt{fn}\ p'_1,\ \ldots\ p'_n \Rightarrow e' & (n \geq 1) \\
 & | & \texttt{rec}\ f : T\ \ p'_1,\ \ldots\ p'_n \Rightarrow e' & (n \geq 1) \\
 & | & \texttt{rec}\ f\ p'_1,\ \ldots\ p'_n \Rightarrow e' & (n \geq 1)
\end{array}
$$

Declarations The `let` expression is also extended, and a new construction ($decl'$) is needed. Besides the basic value binding, 4 new function bindings are allowed. These correspond to all combinations of typed, untyped, recursive and non-recursive functions, with at least one parameter.

Along with value and function bindings, a new type alias binding was added. This binding creates a new type alias that can be used further down in the syntax tree.

$$
\begin{array}{lcll}
e' & ::= & \cdots & \\
 & | & decl' \texttt{ in } e & \\
 & & & \\
decl' & ::= & \texttt{let } p' = e' & \\
 & | & \texttt{let } f\ p'_1,\ \ldots\ p'_n = e' & (n \geq 1) \\
 & | & \texttt{let rec } f\ p'_1,\ \ldots\ p'_n = e' & (n \geq 1) \\
 & | & \texttt{let } f : T'\ p'_1,\ \ldots\ p'_n = e' & (n \geq 1) \\
 & | & \texttt{let rec } f : T'\ p'_1,\ \ldots\ p'_n = e' & (n \geq 1) \\
 & | & \texttt{type alias } \tau = T' &
\end{array}
$$

Lists Although lists are supported by the base language with the :: (cons) and *nil* data constructors, they are not easy to use with only these terms.

The extended language provides a term to implicitly define a list, specifying all of its components between square brackets.

$$
\begin{array}{lcll}
e' & ::= & \cdots & \\
 & | & [e'_1,\ \ldots\ e'_n] & (n \geq 0)
\end{array}
$$

In a similar fashion, a specific pattern for lists is added.

$$
\begin{array}{lcll}
patt' & ::= & \cdots & \\
 & | & [p'_1,\ \ldots\ p'_n] & (n \geq 0)
\end{array}
$$

Range Using a similar construction to basic lists, *ranges* allow the programmer to specify a list of numbers without having to declare all of them explicitly.

There are two variations on ranges. The first is a simple range, providing the start and end values. This range creates a list with all integers starting from the first value, incrementing by one until the last value.

The second variation provides, along with the start and end values, the second value of the list. This allows the language to know what the increment of the range is. Besides allowing increments greater than 1, this also allows ranges that decrement from the start value until the end value.

$$
\begin{array}{lcl}
e' & ::= & \cdots \\
 & | & [e'_1\ ..\ e'_2] \\
 & | & [e'_1,\ e'_2\ ..\ e'_3]
\end{array}
$$

Comprehension *V* provides a very basic list comprehension syntax. This allows evaluating an expression for every value in an existing list, returning a list with the results of every evaluation.

$$
\begin{array}{lcl}
e' & ::= & \cdots \\
 & | & [e'_1 \text{ for } p' \text{ in } e'_2]
\end{array}
$$

Tuple Like lists, tuples are supported by the language through the Tuple *n* constructor. To allow easier creation of tuples, a new term is added to the extended language.

$$
\begin{array}{lcll}
e' & ::= & \cdots & \\
 & | & (e'_1, \ \ldots \ e'_n) & (n \geq 2)
\end{array}
$$

Do Notation For composing multiple IO operations, using the basic bind and return operations is cumbersome. Because of this, the following notation was introduced:

$$
\begin{array}{lcl}
e' & ::= & \cdots \\
 & | & \texttt{do}\ doTerms \\
 & & \\
doTerms & ::= & e' \\
 & | & doTerm :: doTerms \\
 & & \\
doTerm & ::= & p' \leftarrow e' \\
 & | & e' \\
 & | & decl'
\end{array}
$$

2.1.2 Accessors and Records

The extended language provides easier syntax for creating and using accessors, both in accessing and updating fields of records.

The basis for this syntax is the *dot*, which simplifies the construction of the most common use cases for accessors. A *dot* is composed of a stack of *acc*, which are individual components in an accessor. *acc*s can be field labels, arbitrary identifiers or joined *dot*s.

$$
\begin{array}{lcl}
e' & ::= & \cdots \\
 & | & \#dot \\
 & & \\
dot & ::= & acc \ . \ dot \\
 & | & acc \\
 & & \\
acc & ::= & l \\
 & | & 'x \\
 & | & (dot_1, \ \ldots \ dot_n) \qquad (n \geq 2)
\end{array}
$$

Dot Access Another use of the *dot* is to allow a simple syntax for getting a field of a record, being very similar to many object-oriented languages.

$$\begin{array}{lcl} e' & ::= & \cdots \\ & | & x \; . \; dot \end{array}$$

Record Update Finally, the dot syntax is used to allow the easy update of a record.

$$\begin{array}{lcl} e' & ::= & \cdots \\ & | & \texttt{update}\ updates \\ \\ updates & ::= & \varnothing \\ & | & update :: updates \\ \\ update & ::= & dot \leftarrow e' \\ & | & dot \leftsquigarrow e' \\ & | & decl' \end{array}$$

2.2 Translation

To actually evaluate or type check a program in the extended language, it must first be translated into the core language. This is done by a translation algorithm which, besides converting extended terms into core terms, also performs some additional safety checks.

A translation rule is of the form:

$$\gamma \vdash e' \Rightarrow e$$

where γ is the translation environment.

Besides translating expressions, the translation algorithm also translates types (T'), functions ($func'$), etc. All these translations will be described using the same format, and also use the same environment.

Enviroment Like evaluation and type inference, the translation algorithm requires an environment to properly function. This environment contains the following information:

1. Type aliases

   A mapping of type aliases to core types

2. Mapping of generated identifiers

   A mapping of identifiers to other identifiers. This is used because the translation algorithm can create new identifiers, and so it maps identifiers from the input expression to new identifiers in the output expression.

$$
\begin{array}{lll}
\gamma & ::= & (aliases, ids) \\
\\
aliases & ::= & \{\} \mid \{\tau \to T\} \cup aliases \qquad (n \in \mathbb{N}) \\
ids & ::= & \{\} \mid \{x_1 \to x_2\} \cup ids
\end{array}
$$

Below will be sections describing the translation algorithms for the different types of expressions in the extended language. As to avoid clutter, only the rules that perform some sort of computation or modification on the expression will be displayed. This means that rules such as:

$$\gamma \vdash \text{Int} \Rightarrow \text{Int} \tag{Tr-T-Int}$$

will not be provided.

Similarly, composite expressions that simply call the translation algorithm recursively on their sub-expressions, without any modification to structure, will be omitted. This includes rules such as:

$$\frac{\gamma \vdash T_1' \Rightarrow T_1 \qquad \gamma \vdash T_2' \Rightarrow T_2}{\gamma \vdash (T_1' \to T_2') \Rightarrow (T_1 \to T_2)} \tag{Tr-T-Func}$$

2.2.1 Type Translation

Given the fact that trivial translations are not provided, there is only one translation rule that governs type translations.

$$\frac{\gamma.aliases(\tau) = T}{\gamma \vdash \tau \Rightarrow T} \tag{Tr-T-Alias}$$

2.2.2 Pattern Translation

The algorithm for translating patterns works on lists of patterns instead of single patterns. This is done to allow verification of repeated identifiers in a list of patterns.

Since functions allow multiple patterns as parameters, a verification is done to ensure that there are no repeated identifiers in any of the parameters. Furthermore, composite patterns, such as a list or tuple pattern, also cannot repeat identifiers in their sub-patterns, as this would cause ambiguous binding.

This verification is done from left to right, composing a set of identifiers already used in the pattern. Each verification uses the set of identifiers from the previous verification, and so the complete set of used identifiers is created.

The translation also returns a modified translation environment. This environment has new identifier mappings, one for each x pattern found.

$$\frac{\begin{array}{c}\text{id}_0 = \emptyset \qquad \gamma_0 = \gamma \\ \forall i \in [1,\ n]\ .\ \text{collectPatterns}(p_i', \text{id}_{i-1}, \gamma_{i-1}) = p_i, \text{id}_i, \gamma_i\end{array}}{\gamma \vdash [p_1',\ \ldots\ p_n'] \Rightarrow [p_1,\ \ldots\ p_n], \gamma_n} \tag{Tr-P}$$

To translate a single pattern, a list with only that pattern is created and then translated. This ensures that, even within a single pattern, no identifiers are allowed to repeat.

$$\frac{\gamma \vdash [p'] \Rightarrow [p], \gamma'}{\gamma \vdash p' \Rightarrow p, \gamma'} \quad \text{(Tr-P2)}$$

An auxiliary function, called "collectPatterns", is used in pattern translation. This function takes an extended pattern, a set of already used identifiers and a translation environment; and returns a core pattern and a new set of used identifiers.

This is the core of the pattern translation algorithm. If an identifier has already been used in a pattern (or list of patterns), the translation algorithm fails. If the identifier has not been used, a fresh identifier (guaranteed not to be in the environment) is generated. The original identifier is then associated to this new identifier and added to the environment.

$$\frac{x \notin \mathrm{id} \qquad x' \ \textit{is new}}{\mathrm{collectPatterns}(x, \mathrm{id}, \gamma) = x', \mathrm{id} \cup \{x\}, \gamma \cup \{x \to x'\}} \quad \text{(Tr-P-X)}$$

$$\frac{x \notin \mathrm{id} \qquad \gamma \vdash T' \Rightarrow T \qquad x' \ \textit{is new}}{\mathrm{collectPatterns}(x : T', \mathrm{id}, \gamma) = x' : T, \mathrm{id} \cup \{x\}, \gamma \cup \{x \to x'\}} \quad \text{(Tr-P-X2)}$$

Every other translation rule is a variation on iterating on sub-patterns to construct the list of used identifiers and final environment.

$$\mathrm{collectPatterns}(\_, \mathrm{id}, \gamma) = \_, \mathrm{id}, \gamma \quad \text{(Tr-P-Ignore)}$$

$$\frac{\gamma \vdash T' \Rightarrow T}{\mathrm{collectPatterns}(\_ : T', \mathrm{id}, \gamma) = \_ : T, \mathrm{id}, \gamma} \quad \text{(Tr-P-Ignore2)}$$

$$\frac{\begin{array}{c}\mathrm{id}_0 = \mathrm{id} \\ \forall i \in [1,\ n]\ .\ \mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i\end{array}}{\mathrm{collectPatterns}(con\ p'_1,\ \ldots\ p'_n, \mathrm{id}, \gamma) = con\ p_1,\ \ldots\ p_n, \mathrm{id}_n, \gamma_n} \quad \text{(Tr-P-Con)}$$

$$\frac{\begin{array}{c}\mathrm{id}_0 = \mathrm{id} \qquad \gamma \vdash T' \Rightarrow T \\ \forall i \in [1,\ n]\ .\ \mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i\end{array}}{\mathrm{collectPatterns}(con\ p'_1,\ \ldots\ p'_n : T', \mathrm{id}, \gamma) = con\ p_1,\ \ldots\ p_n : T, \mathrm{id}_n, \gamma_n} \quad \text{(Tr-P-Con2)}$$

$$\frac{\begin{array}{c}\mathrm{id}_0 = \mathrm{id} \\ \forall i \in [1,\ n]\ .\ \mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i\end{array}}{\mathrm{collectPatterns}(\{l_1 : p'_1,\ \ldots\ l_n : p'_n\}, \mathrm{id}, \gamma) = \{l_1 : p_1,\ \ldots\ l_n : p_n\}, \mathrm{id}_n, \gamma_n} \quad \text{(Tr-P-Record)}$$

$$\frac{\mathrm{id}_0 = \mathrm{id} \qquad \gamma \vdash T' \Rightarrow T \qquad \forall i \in [1,\ n]\ .\ \mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i}{\mathrm{collectPatterns}(\{l_1 : p'_1,\ \ldots\ l_n : p'_n\} : T', \mathrm{id}, \gamma) = \{l_1 : p_1,\ \ldots\ l_n : p_n\} : T, \mathrm{id}_n, \gamma_n} \text{(Tr-P-Record2)}$$

$$\frac{\mathrm{id}_0 = \mathrm{id} \qquad \forall i \in [1,\ n]\ .\ \mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i}{\mathrm{collectPatterns}(\{l_1 : p'_1,\ \ldots\ l_n : p'_n, \ldots\}, \mathrm{id}, \gamma) = \{l_1 : p_1,\ \ldots\ l_n : p_n, \ldots\}, \mathrm{id}_n \gamma_n} \text{(Tr-P-PartRec)}$$

$$\frac{\mathrm{id}_0 = \mathrm{id} \qquad \gamma \vdash T' \Rightarrow T \qquad \forall i \in [1,\ n]\ .\ \mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i}{\mathrm{collectPatterns}(\{l_1 : p'_1,\ \ldots\ l_n : p'_n, \ldots\} : T', \mathrm{id}, \gamma) = \{l_1 : p_1,\ \ldots\ l_n : p_n, \ldots\} : T, \mathrm{id}_n, \gamma_n} \text{(Tr-P-PartRec2)}$$

$$\frac{\mathrm{id}_0 = \mathrm{id} \qquad \forall i \in [1,\ n]\ .\ \mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i \qquad p = ::\ p_1\ (::\ p_2\ \ldots (::\ p_n\ nil)\ldots)}{\mathrm{collectPatterns}([p'_1,\ \ldots\ p'_n], \mathrm{id}, \gamma) = p, \mathrm{id}_n, \gamma_n} \text{(Tr-P-List)}$$

$$\frac{\mathrm{id}_0 = \mathrm{id} \qquad \gamma \vdash T' \Rightarrow T \qquad \forall i \in [1,\ n]\ .\ \mathrm{collectPatterns}(p'_i, \mathrm{id}_{i-1}, \gamma_{i-1}) = p_i, \mathrm{id}_i, \gamma_i \qquad p = ::\ p_1\ (::\ p_2\ \ldots (::\ p_n\ nil)\ldots)}{\mathrm{collectPatterns}([p'_1,\ \ldots\ p'_n] : T', \mathrm{id}, \gamma) = p : T, \mathrm{id}_n, \gamma_n} \text{(Tr-P-List2)}$$

Typing Patterns Some expressions require that untyped patterns be transformed into typed patterns. For this, fresh variable types are generated.

$$\mathrm{typeP}(patt' : T', \gamma) = patt' : T' \qquad \text{(Typ-Patt)}$$

$$\frac{X\ is\ new}{\mathrm{typeP}(patt', \gamma) = patt' : X} \text{(Typ-Patt2)}$$

2.2.3 Function Translation

Since functions have undergone massive changes from the core language, they need complex translation rules.

$$\frac{\gamma \vdash [p'_1,\ \ldots\ p'_n] \Rightarrow [p_1,\ \ldots\ p_n], \gamma' \qquad \forall i \in [1,\ n]\ .\ x_i\ is\ new \qquad \gamma' \vdash e' \Rightarrow e \qquad m = \texttt{match}\ (x_1,\ \ldots\ x_n)\ \texttt{with}\ (p_1,\ \ldots p_n) \rightarrow e \qquad f = \texttt{fn}\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow m}{\gamma \vdash (\texttt{fn}\ p'_1,\ \ldots\ p'_n \Rightarrow e') \Rightarrow f} \text{(Tr-F-Fn)}$$

$$\frac{\begin{array}{c}\gamma \vdash [p'_1,\ \ldots\ p'_n] \Rightarrow [p_1,\ \ldots\ p_n], \gamma' \\ f' \textit{ is new} \qquad \forall i \in [1,\ n]\ .\ x_i \textit{ is new} \\ \gamma' \cup \{f \rightarrow f'\} \vdash e' \Rightarrow e \\ m = \texttt{match}\ (x_1,\ \ldots\ x_n)\ \texttt{with}\ (p_1,\ \ldots p_n) \rightarrow e \\ fn = \texttt{rec}\ f'\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow m\end{array}}{\gamma \vdash (\texttt{rec}\ f\ \ p'_1,\ \ldots\ p'_n \Rightarrow e') \Rightarrow fn} \quad \text{(Tr-F-Rec)}$$

$$\frac{\begin{array}{c}\forall i \in [1, n]\ .\ \text{typeP}(p'_i, \gamma) = patt'_i : T'_i \\ T'' = T'_2 \rightarrow T'_3 \rightarrow \cdots \rightarrow T'_n \\ \gamma \vdash [patt'_1 : T'_1,\ \ldots\ p'_n : T'_n] \Rightarrow [p_1,\ \ldots\ p_n], \gamma' \\ f' \textit{ is new} \qquad \forall i \in [1,\ n]\ .\ x_i \textit{ is new} \\ \gamma' \cup \{f \rightarrow f'\} \vdash e' \Rightarrow e \qquad \gamma' \vdash T'' \Rightarrow T \\ m = \texttt{match}\ (x_1,\ \ldots\ x_n)\ \texttt{with}\ (p_1,\ \ldots p_n) \rightarrow e \\ fn = \texttt{rec}\ f : T\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow m\end{array}}{\gamma \vdash (\texttt{rec}\ f : T'\ \ p'_1,\ \ldots\ p'_n \Rightarrow e') \Rightarrow fn} \quad \text{(Tr-F-RecT)}$$

When functions only have one parameter, the `match` expression does not use a tuple, since tuples must have at least 2 components.

$$\frac{\begin{array}{c}\gamma \vdash p' \Rightarrow p, \gamma' \qquad x \textit{ is new} \\ \gamma' \vdash e' \Rightarrow e \\ m = \texttt{match}\ x\ \texttt{with}\ p \rightarrow e \\ f = \texttt{fn}\ x \Rightarrow m\end{array}}{\gamma \vdash (\texttt{fn}\ p' \Rightarrow e') \Rightarrow f} \quad \text{(Tr-F-Fn1)}$$

$$\frac{\begin{array}{c}\gamma \vdash p' \Rightarrow p, \gamma' \qquad x \textit{ is new} \\ \gamma' \vdash e' \Rightarrow e \\ m = \texttt{match}\ x\ \texttt{with}\ p \rightarrow e \\ f = \texttt{rec}\ x \Rightarrow m\end{array}}{\gamma \vdash (\texttt{rec}\ p' \Rightarrow e') \Rightarrow f} \quad \text{(Tr-F-Rec1)}$$

$$\frac{\begin{array}{c}\text{typeP}(p', \gamma) = p'' \\ \gamma \vdash p'' \Rightarrow p, \gamma' \qquad x \textit{ is new} \\ \gamma' \vdash e' \Rightarrow e \qquad \gamma' \vdash T' \Rightarrow T \\ m = \texttt{match}\ x\ \texttt{with}\ p \rightarrow e \\ f = \texttt{rec}\ x : T \Rightarrow m\end{array}}{\gamma \vdash (\texttt{rec}\ p' : T' \Rightarrow e') \Rightarrow f} \quad \text{(Tr-F-RecT1)}$$

As a further efficiency improvement, that `match` expression is only created when the parameters of the function are patterns. If every parameter of the function is a regular identifier (without type information), then no `match` expression is necessary. This is only done if the function does not specify a return type.

$$\frac{\gamma \vdash [x'_1,\ \ldots\ x'_n] \Rightarrow [x_1,\ \ldots\ x_n], \gamma' \quad \gamma' \vdash e' \Rightarrow e \quad f = \texttt{fn}\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow e}{\gamma \vdash (\texttt{fn}\ x'_1,\ \ldots\ x'_n \Rightarrow e') \Rightarrow f} \quad \text{(Tr-F-Fn2)}$$

$$\frac{\gamma \vdash [x'_1,\ \ldots\ x'_n] \Rightarrow [x_1,\ \ldots\ x_n], \gamma' \quad f' \textit{ is new} \quad \gamma' \cup \{f \to f'\} \vdash e' \Rightarrow e \quad fn = \texttt{rec}\ f'\ x_1 \Rightarrow \cdots \texttt{fn}\ x_n \Rightarrow e}{\gamma \vdash (\texttt{rec}\ f\ \ x'_1,\ \ldots\ x'_n \Rightarrow e') \Rightarrow fn} \quad \text{(Tr-F-Rec2)}$$

2.2.4 Declaration Translation

The translation of declarations works differently from other translations. The result of translating a declaration is a set of associations between patterns and expressions; along with an updated translation environment.

$$\frac{\gamma \vdash p' \Rightarrow p, \gamma' \quad \gamma \vdash e' \Rightarrow e}{\gamma \vdash (p' = e') \Rightarrow \{p \to e\}, \gamma'} \quad \text{(Tr-Decl)}$$

$$\frac{\gamma \vdash T' \Rightarrow T}{\gamma \vdash \texttt{type alias}\ \tau = T' \Rightarrow \{\}, \gamma \cup \{\tau \to T\}} \quad \text{(Tr-Decl-Alias)}$$

$$\frac{\gamma \vdash (\texttt{fn}\ p'_1,\ \ldots\ p'_n \Rightarrow e') \Rightarrow e \quad f' \textit{ is new}}{\gamma \vdash f\ p'_1,\ \ldots\ p'_n = e' \Rightarrow \{f' \to e\}, \gamma \cup \{f \to f'\}} \quad \text{(Tr-Decl-Func)}$$

$$\frac{\gamma \vdash (\texttt{fn}\ p'_1,\ \ldots\ p'_n \Rightarrow e') \Rightarrow e \quad f' \textit{ is new}}{\gamma \vdash f : T'\ p'_1,\ \ldots\ p'_n = e' \Rightarrow \{f' \to e\}, \gamma \cup \{f \to f'\}} \quad \text{(Tr-Decl-Func2)}$$

$$\frac{\gamma \vdash (\texttt{rec}\ f\ p'_1,\ \ldots\ p'_n \Rightarrow e') \Rightarrow e \quad f' \textit{ is new}}{\gamma \vdash \texttt{rec}\ f\ p'_1,\ \ldots\ p'_n = e' \Rightarrow \{f' \to e\}, \gamma \cup \{f \to f'\}} \quad \text{(Tr-Decl-Rec)}$$

$$\frac{\forall i \in [1, n]\ .\ \text{typeP}(p'_i, \gamma) = \mathit{patt}'_i : T'_i \quad T'' = T'_1 \to T'_2 \to \cdots \to T'_n \quad \gamma \vdash (\texttt{rec}\ f : T'\ \mathit{patt}'_1 : T'_1,\ \ldots\ \mathit{patt}'_n : T'_n \Rightarrow e') \Rightarrow e \quad f' \textit{ is new}}{\gamma \vdash \texttt{rec}\ f : T'\ p'_1,\ \ldots\ p'_n = e' \Rightarrow \{f' : T'' \to e\}, \gamma \cup \{f \to f'\}} \quad \text{(Tr-Decl-Rec2)}$$

2.2.5 Dot Translation

To properly translate the accessor and dot syntax expressions introduced in the extended language, it is necessary to define translations for both *acc* and *dot* expressions.

When translating an *acc*, the result is an expression e. The rules governing this translation are given below.

$$\gamma \vdash l \Rightarrow \#l \qquad \text{(Tr-Acc-Label)}$$

$$\gamma \vdash' x \Rightarrow x \qquad \text{(Tr-Acc-'x)}$$

$$\frac{\forall i \in [1, n] \,.\, \gamma \vdash dot i \Rightarrow e_i}{\gamma \vdash (dot_1,\ \ldots\ dot_n) \Rightarrow \#(e_1,\ \ldots\ e_n)} \qquad \text{(Tr-Acc-Joined)}$$

Similarly, translating a *dot* also yields an expression e. The two rules defining this translation are given below.

$$\frac{\gamma \vdash acc \Rightarrow e}{\gamma \vdash acc \Rightarrow e} \qquad \text{(Tr-Dot-Stacked)}$$

$$\frac{\gamma \vdash acc \Rightarrow e_1 \qquad \gamma \vdash dot \Rightarrow e_2}{\gamma \vdash acc \,.\, dot \Rightarrow \texttt{stack}\ e_1\ e_2} \qquad \text{(Tr-Dot-Stacked)}$$

Using the translations defined above, defining the translation of the accessor and dot access expressions is trivial:

$$\frac{\gamma \vdash dot \Rightarrow e}{\gamma \vdash \#dot \Rightarrow e} \qquad \text{(Tr-E-Acc)}$$

$$\frac{\gamma \vdash dot \Rightarrow e}{\gamma \vdash x \,.\, dot \Rightarrow \texttt{get}\ e\ x} \qquad \text{(Tr-E-Dot)}$$

2.2.6 Record Update Translation

$$\frac{x \text{ is new} \qquad f' = (\texttt{fn}\ x \Rightarrow x)}{\text{concat}(\varnothing, \gamma) = f', \gamma} \qquad \text{(Tr-Updates-Empty)}$$

$$\frac{x \text{ is new} \qquad f' = (\texttt{fn}\ x \Rightarrow \texttt{set}\ \#dot\ e'\ x) \qquad \text{concat}(updates, \gamma) = g', \gamma'}{\text{concat}(dot \leftarrow e' :: updates, \gamma) = g' \,.\, f', \gamma'} \qquad \text{(Tr-Updates-Set)}$$

$$\frac{x \text{ is new} \qquad f' = (\texttt{fn}\ x \Rightarrow \texttt{modify}\ \#dot\ e'\ x) \qquad \text{concat}(updates, \gamma) = g', \gamma'}{\text{concat}(dot \leftsquigarrow e' :: updates, \gamma) = g \,.\, f', \gamma'} \qquad \text{(Tr-Updates-Modify)}$$

$$\frac{\text{concat}(updates, \gamma) = g', \gamma'}{\text{concat}(decl' :: updates, \gamma) = \texttt{let}\ decl'\ \texttt{in}\ g', \gamma'} \text{ (Tr-Updates-Decl)}$$

$$\frac{\text{concat}(updates) = f', \gamma' \qquad \gamma' \vdash f' \Rightarrow f}{\gamma \vdash \texttt{update}\ updates \Rightarrow f} \text{ (Tr-E-Update)}$$

2.2.7 Expression Translation

The conditional expression translates into a `match` expression, testing whether the first sub-expression (e_1) is *true* or $false$ and evaluation e_2 or e_3, respectively.

$$\frac{\begin{array}{c}\gamma \vdash e'_1 \Rightarrow e_1 \qquad \gamma \vdash e'_2 \Rightarrow e_2 \qquad \gamma \vdash e'_3 \Rightarrow e_3 \\ e = \texttt{match}\ e_1\ \texttt{with}\ true \rightarrow e_2,\ false \rightarrow e_3\end{array}}{\gamma \vdash \texttt{if}\ e'_1\ \texttt{then}\ e'_2\ \texttt{else}\ e'_3 \Rightarrow e} \text{ (Tr-E-Cond)}$$

The list expression translates into nested applications of the :: constructor, ending with the *nil* (empty list) constructor.

$$\frac{\begin{array}{c}\forall i \in [1,\ n]\ .\ \gamma \vdash e'_i \Rightarrow e_i \\ e = ::\ e_1\ (::\ e_2\ \ldots (::\ e_n\ nil)\ldots)\end{array}}{\gamma \vdash [e'_1,\ \ldots\ e'_n] \Rightarrow e} \text{ (Tr-E-List)}$$

Similarly, the tuple expression translates into a complete application of a tuple (Tuple n) constructor, where n is the number of elements in the expression.

$$\frac{\begin{array}{c}\forall i \in [1,\ n]\ .\ \gamma \vdash e'_i \Rightarrow e_i \\ e = (\ \ldots\ (\text{Tuple}\ n\ e_1)\ e_2)\ \ldots e_n)\end{array}}{\gamma \vdash (e'_1,\ \ldots\ e'_n) \Rightarrow e} \text{ (Tr-E-Tuple)}$$

When translating `let` expressions, the declaration is translated into an ordered set of associations between patterns and expressions. Nested `let` expressions are then created with these associations.

$$\frac{\begin{array}{c}\gamma \vdash decl \Rightarrow \{p_1 \rightarrow e_1,\ \ldots\ p_n \rightarrow e_n\}, \gamma' \qquad \gamma' \vdash e' \Rightarrow e \\ ret = (\texttt{let}\ p_1 = e_1\ \texttt{in} \cdots \texttt{let}\ p_n = e_n\ \texttt{in}\ e)\end{array}}{\gamma \vdash \texttt{let}\ decl\ \texttt{in}\ e' \Rightarrow ret} \text{ (Tr-E-Let)}$$

There are two variations of ranges: one with an implicit step and one with an explicit step. Both of these rely on the existing of the function "range", which, when given a starting number, ending number and step, returns a list with the numbers.

The first variation uses a fixed step value of 1, while the second variation calculates its step by sutracting the second element of the range (e_2) from the first element (e_1).

$$\frac{\gamma \vdash e'_1 \Rightarrow e_1 \qquad \gamma \vdash e'_2 \Rightarrow e_2}{\gamma \vdash [e'_1 \,..\, e'_2] \Rightarrow \text{range } e_1\ e_2\ 1} \quad \text{(Tr-E-Range)}$$

$$\frac{\begin{array}{cc}\gamma \vdash e'_1 \Rightarrow e_1 & \gamma \vdash e'_2 \Rightarrow e_2 \\ \gamma \vdash e'_3 \Rightarrow e_3 & i = -\ e_2\ e_1\end{array}}{\gamma \vdash [e'_1,\ e'_2 \,..\, e'_3] \Rightarrow \text{range } e_1\ e_2\ i} \quad \text{(Tr-E-Range2)}$$

Comprehensions, similarly to ranges, rely on the function "map". A function is created with the pattern p' and the body e'_1. This function is then translated and passed as the first argument of the "map" function. The second argument of the function is the translation of the expression e'_2, which will eventually evaluate into a list.

$$\frac{\gamma \vdash (\texttt{fn}\ p' \Rightarrow e'_1) \Rightarrow f \qquad \gamma \vdash e'_2 \Rightarrow e_2}{\gamma \vdash [e'_1 \text{ for } p' \text{ in } e'_2] \Rightarrow \text{map } f\ e_2} \quad \text{(Tr-E-Comprehension)}$$

2.2.8 Do Notation Translation

$$\text{concat}(e', \gamma) = e', \gamma \quad \text{(Tr-Do-Base)}$$

$$\frac{\text{concat}(doTerms, \gamma) = f', \gamma'}{\text{concat}(p' \leftarrow e' :: doTerms, \gamma) = \text{bind } e'\ (\texttt{fn}\ p' \Rightarrow f'), \gamma'} \quad \text{(Tr-Do-Bind)}$$

$$\frac{\text{concat}(doTerms, \gamma) = f', \gamma'}{\text{concat}(e' :: doTerms, \gamma) = \text{bind } e'\ (\texttt{fn}\ \_ \Rightarrow f'), \gamma'} \quad \text{(Tr-Do-Term)}$$

$$\frac{\text{concat}(doTerms, \gamma) = f', \gamma'}{\text{concat}(decl' :: doTerms, \gamma) = \texttt{let}\ decl'\ \texttt{in}\ f', \gamma'} \quad \text{(Tr-Do-Decl)}$$

$$\frac{\text{concat}(doTerms) = f', \gamma' \qquad \gamma' \vdash f' \Rightarrow f}{\gamma \vdash \texttt{do}\ doTerms \Rightarrow f} \quad \text{(Tr-E-Do)}$$

3 Concrete Syntax

3.1 Notation Conventions

The following conventions are used for presenting the syntax of programs in *V*:

[⟨*pat*⟩]	optional pattern
{ ⟨*pat*⟩ }	zero or more occurrences of pattern
(⟨*pat*⟩)	grouping of patterns
⟨pat_1⟩ | ⟨pat_2⟩	choice
⟨*pat*⟩$_{\langle pat' \rangle}$	all patterns accepted by *pat*, except those also accepted by *pat′*
`text`	concrete characters (written in terminal font)

The syntax is described using a BNF grammar, with each production having the form:

⟨*sentence*⟩ ::= ⟨pat_1⟩ | ⟨pat_2⟩ | . . . | ⟨pat_n⟩

Whitespace is always explicitly expressed in productions with the ␣ character. It is used as a shorthand for the ⟨*whitespace*⟩ production. Literal characters will always be written in terminal font, so `|` and `[]` mean the literal characters, while | and [] are the choice and option pattern, respectively.

3.2 Basic Structure

⟨*program*⟩ ::= ⟨*whitespace*⟩ ⟨*expression*⟩ ⟨*eof*⟩

⟨*library*⟩ ::= ⟨*whitespace*⟩ { ⟨*declaration*⟩ } ⟨*eof*⟩

⟨*whitespace*⟩ ::= { ⟨*whitechars*⟩ | ⟨*comment*⟩ }

⟨*whitechars*⟩ ::= ⟨*whitechar*⟩ { ⟨*whitechar*⟩ }

⟨*whitechar*⟩ ::= ⟨*space*⟩ | ⟨*tab*⟩ | ⟨*newline*⟩

⟨*newline*⟩ ::= ⟨*return*⟩⟨*linefeed*⟩ | ⟨*return*⟩ | ⟨*linefeed*⟩

⟨*space*⟩ ::= space (' ')

⟨*tab*⟩ ::= horizontal tab ('\t')

⟨*return*⟩ ::= carriage return ('\r')

⟨*linefeed*⟩ ::= line feed ('\n')

⟨*comment*⟩ ::= `//` { ⟨*any*⟩$_{\langle newline \rangle}$ } ⟨*newline*⟩

⟨*any*⟩ ::= any ASCII character

3.3 Identifiers and Operators

⟨*identifier*⟩ ::= (⟨*idstart*⟩ { ⟨*idcontinue*⟩ })$_{\langle reservedid \rangle}$

⟨*idstart*⟩ ::= ⟨*small*⟩ | `_`

⟨*idcontinue*⟩ ::= ⟨*small*⟩ | ⟨*large*⟩ | ⟨*digit*⟩ | `'` | `_` | `?`

⟨*reservedid*⟩ ::= `let` | `true` | `false` | `if` | `then` | `else`
| `rec` | `nil` | `raise` | `when` | `match` | `with`
| `try` | `except` | `for` | `in` | `import` | `infix`
| `infixl` | `infixr` | `type` | `alias`

⟨*typeident*⟩ ::= (⟨*typeidentstart*⟩ { ⟨*idcontinue*⟩ })$_{\langle reservedtype \rangle}$

⟨*typeidentstart*⟩ ::= ⟨*large*⟩ | `_`

⟨*reservedtype*⟩ ::= `Int` | `Bool` | `Char`

⟨*small*⟩ ::= `a` | `b` | ... | `z`

⟨*large*⟩ ::= `A` | `B` | ... | `Z`

⟨*digit*⟩ ::= `0` | `1` | ... | `9`

⟨*operator*⟩ ::= ⟨*symbol*⟩ { ⟨*symbol*⟩ }

⟨*customop*⟩ ::= ⟨*operator*⟩$_{\langle reservedop \rangle}$

⟨*reservedop*⟩ ::= `+` | `-` | `*` | `/` | `<=` | `<` | `=`
| `!=` | `>=` | `>` | `::`

⟨*symbol*⟩ ::= `:` | `?` | `!` | `%` | `$` | `&` | `*` | `+` | `-`
| `.` | `/` | `<` | `=` | `>` | `@` | `^` | `|` | `~`

3.4 Terms

⟨*term*⟩ ::= ⟨*identifier*⟩
| `true` | `false` (booleans)
| ⟨*number*⟩
| `nil` (empty list)
| `raise` (exception)
| ⟨*char*⟩
| ⟨*string*⟩
| ⟨*parentheses*⟩
| ⟨*record*⟩
| `#` ⟨*identifier*⟩ (record access)
| ⟨*squarebrackets*⟩
| `if` ␣⟨*expression*⟩ `then` ␣⟨*expression*⟩ `else` ␣⟨*expression*⟩
| ⟨*match*⟩
| ⟨*lambda*⟩
| ⟨*reclambda*⟩
| ⟨*let*⟩

⟨*number*⟩ ::= ⟨*decimal*⟩ | ⟨*binary*⟩ | ⟨*octal*⟩ | ⟨*hexadecimal*⟩

⟨*decimal*⟩ ::= ⟨*digit*⟩ { ⟨*digit*⟩ }

⟨*binary*⟩ ::= `0` (`b` | `B`) ⟨*bindigit*⟩ { ⟨*bindigit*⟩ }

⟨*octal*⟩ ::= `0` (`o` | `O`) ⟨*octdigit*⟩ { ⟨*octdigit*⟩ }

⟨*hexadecimal*⟩ ::= `0` (`x` | `X`) ⟨*hexdigit*⟩ { ⟨*hexdigit*⟩ }

⟨*bindigit*⟩ ::= `0` | `1`

⟨*octdigit*⟩ ::= `0` | `1` | ... | `7`

⟨*hexdigit*⟩ ::= ⟨*digit*⟩ | `a` | ... | `f` | `A` | ... | `F`

⟨*char*⟩ ::= `'` (⟨*escape*⟩ | ⟨*any*⟩$_{\langle \text{'} | \backslash \rangle}$) `'`

⟨*string*⟩ ::= `"` { ⟨*escape*⟩ | ⟨*any*⟩$_{\langle \text{"} | \backslash \rangle}$ } `"`

⟨*escape*⟩ ::= `\` (`b` | `n` | `r` | `t` | `\` | `"` | `'`)

⟨*parentheses*⟩ ::= `(` ␣⟨*expression*⟩ `)`␣ (parenthesised expression)
| `(` ␣⟨*expression*⟩$_1$ `,` ␣... `,` ␣⟨*expression*⟩$_n$ `)` ␣ (tuple, $n \geq 2$)
| `(` ␣⟨*operator*⟩ ␣`)`␣ (prefix operator)

⟨*record*⟩ ::= `{` ␣⟨*recordcomp*⟩$_1$ `,` ␣... `,` ␣⟨*recordcomp*⟩$_n$ `}` ␣ ($n \geq 1$)

⟨*recordcomp*⟩ ::= ⟨*identifier*⟩ ␣`:` ␣⟨*expression*⟩

⟨*squarebrackets*⟩ ::= `[` ␣⟨*expression*⟩$_1$ `,` ␣`...` `,` ␣⟨*expression*⟩$_n$ `]` ␣ (list, $n \geq 0$)
| `[` ␣⟨*expression*⟩ `for` ␣⟨*pattern*⟩ `in` ⟨*expression*⟩ `]` ␣ (comprehension)
| `[` ␣⟨*expression*⟩ `[` `,` ␣⟨*expression*⟩ `]` `..` ␣⟨*expression*⟩ `]` ␣ (range)

⟨*match*⟩ ::= `match` ␣⟨*expression*⟩ `with` ␣{ ⟨*matchcomp*⟩ }

⟨*matchcomp*⟩ ::= `|` ␣⟨*pattern*⟩ [`when` ␣⟨*expression*⟩] `->` ␣⟨*expression*⟩

⟨*expression*⟩ ::= ⟨*operand*⟩ { ⟨*operator*⟩ ␣⟨*operand*⟩ }

⟨*operand*⟩ ::= ⟨*application*⟩
| `-` ␣⟨*application*⟩ (unary negation)

⟨*application*⟩ ::= ⟨*term*⟩ ␣{ ⟨*term*⟩ ␣}

3.5 Functions and Declarations

⟨*lambda*⟩ ::= `\` ␣{ ⟨*parameter*⟩ } `->` ␣⟨*expression*⟩

⟨*reclambda*⟩ ::= `rec` ␣⟨*identifier*⟩ ␣{ ⟨*parameter*⟩ } [`:`␣⟨*type*⟩] `->` ␣⟨*expression*⟩

⟨*let*⟩ ::= ⟨*declaration*⟩ `;` ␣⟨*expression*⟩

⟨*declaration*⟩ ::= `let` ␣⟨*pattern*⟩ `=` ␣⟨*expression*⟩
| `let` ␣[`rec` ␣] ⟨*funcname*⟩ { ⟨*parameter*⟩ } [`:` ␣⟨*type*⟩] `=` ␣⟨*expression*⟩
| `import` ␣⟨*string*⟩ ␣
| `type` ␣`alias` ␣⟨*typeident*⟩ ␣`=` ␣⟨*type*⟩

⟨*funcname*⟩ ::= ⟨*identifier*⟩ ␣
| [⟨*fixity*⟩ ␣⟨*digit*⟩ ␣] `(` ␣⟨*customop*⟩ ␣`)`␣ (operator declaration)

⟨*fixity*⟩ ::= `infixl` | `infixr` | `infix`

⟨*parameter*⟩ ::= ⟨*pattuple*⟩ ␣| `(` ␣⟨*pattern*⟩ `)` ␣| ⟨*patvalue*⟩ ␣

⟨*patvalue*⟩ ::= ⟨*identifier*⟩
| `_` (wildcard pattern)
| `true` | `false` (booleans)
| ⟨*number*⟩
| `nil` (empty list)
| ⟨*char*⟩
| ⟨*string*⟩
| ⟨*pattuple*⟩
| ⟨*patrecord*⟩
| `[` ␣⟨*pattern*⟩$_1$ `,` ␣`...` `,` ␣⟨*pattern*⟩$_n$ `]` ␣ (list, $n \geq 0$)

⟨*pattuple*⟩ ::= `(` ␣⟨*pattern*⟩$_1$ `,` ␣`...` `,` ␣⟨*pattern*⟩$_n$ `)` ␣ ($n \geq 2$)

⟨*patrecord*⟩ ::= `{` ␣⟨*patrecordcomp*⟩$_1$ `,` ␣`...` `,` ␣⟨*patrecordcomp*⟩$_n$ `}` ␣ ($n \geq 1$)
| `{` ␣⟨*patrecordcomp*⟩$_1$ `,` ␣`...` `,` ␣⟨*patrecordcomp*⟩$_n$ `,` `...` ␣`}` ␣
(partial record)

⟨*patrecordcomp*⟩ ::= ⟨*identifier*⟩ ␣`:` ␣⟨*pattern*⟩

⟨*pattern*⟩ ::= ⟨*patvalue*⟩ ␣`:` ␣⟨*type*⟩
| ⟨*patvalue*⟩ ␣`{` `::` ␣⟨*patvalue*⟩ ␣`}`

⟨*typevalue*⟩ ::= `Int`
| `Bool`
| `Char`
| ⟨*typeident*⟩
| `(` ␣⟨*type*⟩$_1$ `,` ␣`...` `,` ␣⟨*type*⟩$_n$ `)` ␣ (tuple, $n \geq 2$)
| `[` ␣⟨*type*⟩ ␣`]` ␣ (list)
| ⟨*typerecord*⟩

⟨*typerecord*⟩ ::= `{` ␣⟨*typerecordcomp*⟩$_1$ `,` ␣`...` `,` ␣⟨*typerecordcomp*⟩$_n$ `}` ␣ ($n \geq 1$)

⟨*typerecordcomp*⟩ ::= ⟨*identifier*⟩ ␣`:` ␣⟨*type*⟩

⟨*type*⟩ ::= ⟨*typevalue*⟩ ␣`{` `->` ⟨*typeValue*⟩ ␣`}`

4 Language Guide

4.1 Basic Values

There are 4 types of basic values available in the *V* language:

1. Integers
2. Booleans
3. Character
4. Strings

Integers Only positive integers (plus zero) are recognized. They are always specified in decimal format, using only the digits from 0 to 9.

Booleans Two values are available: `true` for true, and `false` for false.

Characters and Strings A character literal is a single Unicode character surround by single quotes (’). A string literal is a sequence of zero or more Unicode characters surrounded by double quotes ("). Technically, strings are not basic values, since they are just syntactic sugar for a list of characters.

”*abc*” $\rightarrow'$ *a*′ ::′ *b*′ ::′ *c*′ :: *nil*

Some characters must be escaped in order to insert them into either character or string literals. "Escaping" a character means preceding it by the backslash character (\). For character literals, the single quote must be escaped (\’), while string literals require the escaping of the double quote (\").

There is also support for ASCII escape codes to insert special characters in literals. These are the allowed escape codes and their resulting characters:

Escape code	Character
\b	backspace
\n	newline (line feed
\r	carriage return
\t	horizontal tab
\\	backslash
\’	single quote
\"	double quote

Any escape code can be used in either character or string literals. Furthermore, the special characters can be inserted directly into the literal. This means that multi line strings are supported by *V*. This also means that a single quote followed by a new line and a single quote is interpreted as a valid character literal (i.e. ’\n’).

4.2 Compound Values

4.2.1 Lists

Lists are ordered collections of values of the same type. There are no limits on the size of a list, even accepting lists with 0 values (the empty list).

Creating Lists An empty list can be created using either the `nil` keyword or the empty list literal, which is written as `[]` (empy square brackets).

To create a list with values, simply enclose the sequence of values, each separated by a comma, between square brackets.

```
[] // Empty list
[1, 2, 3] // List containing 3 values
```

Expanding Lists It is possible to add a value to the start of a list by using the list construction operator (`::`). The `append` function allowing the addition of a value to the end of a list. It is also possible to create a new list by using the concatenation operator (`@`), which adds two lists together.

```
let x = 0 :: [1, 2, 3];
// x is equal to [0, 1, 2, 3]

let y = append 4 [1, 2, 3];
// y is equal to [1, 2, 3, 4]

let z = [1, 2] @ [3, 4];
// z is equal to [1, 2, 3, 4]
```

Accessing Lists Any element of a list can be accessed by using the index (`!!`) operator. Lists are 0-indexed, which means the first value of a list is at index 0.

```
["a", "b", "c"] !! 0 // Returns "a"
```

An attempt to access an index outside the range of a list (that is, indexes equal to or greater than the size of the list) will result in a runtime error.

```
["a", "b", "c"] !! 5 // Runtime error
```

There are many other operations available for accessing elements of a list, including `head` (returns the first value of a list), `last`, `filter`, `maximum`, etc.

Complex Operations Although the *V* language does not directly support complex operations on lists, the standard library (see 5) provides a number of functions to manipulate lists. Among these are functions like `map`, `filter`, `sort`, `fold`, `sublist`, which provide basic funcionality for performing computations with lists.

Ranges Ranges allow the easy creation of lists of integers in a arithmetic progression.

There are two variants of ranges, one for simple integer counting and one for more complex progressions.

The first variant specifies the first and last value for the list. The list is then composed with every integer number between these values. Because of this, the first value must be smaller then the last

```
[1..5] // [1,2,3,4,5]
[3..7] // [3,4,5,6,7]
[5..3] // Invalid
```

The second variants specifies the first, second and last value for the list. The increment is the difference between the second and first value of the list, which can even be negative.

The increment is then added to each element until the largest possible value which is smaller than or equal to the last value. If the increment is negative, the list stops at the smallest possible value which is larger than or equal to the last value.

```
[1,3..10] // [1,3,5,7,9]
[5,4..1] // [5,4,3,2,1]
[5,3..0] // [5,3,1]
```

Comprehensions List comprehensions are a simple way to transform every value in a list, creating a new list.

In the example below, `ls` is a list containing every number from 1 to 10, inclusive. Using a list comprehension, `new` is a list containing every number from 2 to 11, since the code `x+1` is executed for every value in `ls`.

```
let ls = [1..10];
let new = [x+1 for x in ls];
```

4.2.2 Tuples

Tuples group multiple values, possibly of different types, into a single compound value. The minimum size of a tuple is 2, but there is no limit on its maximum size.

Tuples are specified inside parenthesis, with each of its values separated by commas.

```
(1, ``hello'')
(true, `c', 43)
```

Tuples are immutable, which means they cannot be changed once they are created. There is no way to add or remove elements from a tuple.

To access a specific field of a tuple, it must be deconstructed using a pattern. This can be done either by using a `let` expression or by pattern matching in a function definition.

```
let (x, y) = (1, ``hello'')
(\&&(x, y, z) -> x) (true, `c', 43)
```

Tuples are extremely useful as return values for functions that must convey more than one piece of information. Since every function can only return one value, tuples can be used to group the different values that the function must return.

4.2.3 Records

Records are, like tuples, groupings of multiple values of possibly different types. Unlike tuples, however, each value has its unique label. The smallest size for a record is 1, but there is no limit on its maximum size.

To construct a record, each value must be preceded by a label and a colon. Each label-value pair is separated by a comma, and the whole record is enclosed in curly brackets ({ }).

```
{name: ``Martha'', age: 32}
{day: 1, month: 1, year: 2000}
```

To get the value of a single field in a record, one can use the `get` function, passing the name of the field and the record. As can be seen below, the name of the field must be prefixed by the `#` character.

```
get #age {name: ``Martha'', age: 32} // Returns 32
get #month {day: 1, month: 1, year: 2000} // Returns 1
```

Unlike tuples, it is possible to alter a single field in a record. This is done by using the `set` function, which takes the name of the field (with the `#` character), the value to be set and the record.

It is also possible to modify a value of a field by using the `modify` function, which applies a function given as its parameter to the existing value.

It is important to note that these function do not change the original record, but instead returns a new, modified record.

```
set 3 #age {name: ``Martha'', age: 32}
// Returns {name: ``Martha'', age: 3}
set 8 #month {day: 1, month: 1, year: 2000}
// Returns {day: 1, month: 8, year: 2000}

modify #age (\x -> x * 2) {name: ``Martha'', age: 32}
// Returns {name: ``Martha'', age: 64}
```

4.3 Identifiers

Identifiers are used to name constants (in let declarations), functions and function arguments. When an identifier is expected, the identifier is defined as the longest possible sequence of valid characters. Any Unicode character is considered valid, with the exception of the following:

.	,	;	:	!	@	&
+	-	/	*	<	=	>
(	)	{	}	[	]	|
%	\	’	"	\n	\r	\t
_	‘					

Numerical digits are not allowed at the start of an identifier, but they can be used in any other position.

Furthermore, *V* has some reserved names that cannot be used by any identifier. They are the following:

let	true	false	if	then	else
rec	nil	raise	when	match	with
for	in	import	infix	infixl	infixr

4.4 Patterns

Patterns are rules for deconstructing values and binding their parts to identifiers. They can be used in constant declarations and function parameters, simplifying the extraction of data from compound values.

Pattern	**Examples**	**Comments**
Identifier	x, y, z	Matches any value and binds to the identifier
Number	1, 3	Matches and ignores the number
Boolean	true, false	Matches and ignores the boolean
Character	‘c’, ‘f’	Matches and ignores the character
Identifier	x, y, z	Matches any value and binds to the identifier
Wildcard	_	Matches and ignores any value
Tuple	(x, _, y)	Matches tuples with corresponding size
Record	{a: _, b: (x,y), c: x}	Matches records with corresponding labels
Partial Record	{a: _, b: (x,y), c: x, ...}	Matches records with at least the specified labels
Nil	nil, []	Matches the empty list
Cons	x :: y	Matches the head and tail of a non-empty list
List	[x, y, z]	Matches lists of corresponding size

Compound patterns, such as `Tuple`, `Record` and `List`, are composed of other patterns separated by commas. All patterns can have optional type annotations added to explicitly declare their types.

One thing to notice is that all patterns related to lists (`List`, `Cons` and `Nil`) and values (numbers, characters) can fail. If an attempt to match a pattern fails (e.g matching a non-empty list with `Nil`), the expression will raise an exception.

4.5 Constants

Constants are associations of identifiers to values. The value associated to a particular identifier cannot be changed after it is declared.

The keyword `let` is used to start a constant declaration, and a semicolon ends it. After the `let` keyword, any pattern can be used.

Below are examples of constant declarations:

```
let name = ``Steve'';
let age: Int = 32;
let (x: Int, y) = (4, true);
```

4.6 Type Annotations

Type annotations are used to explictly state the type of a constant, function argument or function return value. They are not necessary for most programs, since the interpreter can infer the type of any expression.

Sometimes, the programmer may want to create artificial constraints on a function argument, and type annotations allow this.

The table below shows every type that can be specified in type annotations. These types align with the types available in the *V* language, since every type can be used in a type annotation.

Type	**Example Values**	**Comments**
Int	1, 0, -3	
Bool	`true`, `false`	
Char	'c', ' '	
String	"abc", ""	This is syntactic sugar for `[Char]`
[*Type*]	[1, 2, 3], `nil`	List Type
(*Type*, ... *Type*)	(1, true, 'a')	Tuple Type
{id: *Type*, ... id: *Type*}	{a: 3, b: false}	Record Type
Type -> *Type*		Function Type (see 4.10.5)

4.7 Conditionals

V provides a conditional expression (`if ... then ... else`) to control the flow of a program. This expression tests a condition and, if its value is `true`, executes the first branch (known as the `then` clause). If the condition is `false`, the expression executes the second branch (the `else` clause).

```
if b then
  1+3 // Will execute if b is true
else
  2 // Will execute if b is false
```

The only accepted type for the condition of a conditional is Boolean. All types are accepted in the `then` and `else` branch, but they must be of the same type.

```
// This conditional is invalid code, since 4 and "hello" are of different type
if true then
  4
else
  "hello"
```

Unlike imperative languages, every conditional in *V* must specify both branches. This ensures that the conditional will always return a value.

It is possible to chain multiple conditionals together.

```
if grade > 10 then
  "The grade cannot be higher than 10"
else if grade < 0
  "The grade cannot be lower than 0"
else
  "The grade is valid"
```

4.8 Match Expressions

Match expressions are another way to control the flow of a program based on comparison with a list of patterns. Any number (greater or equal to 1) of patterns can be specified, and each one has a corresponding result expression.

```
match value with
| pattern1 -> result-expression1
| pattern2 when condition -> result-expression2
...
```

Each pattern is tested from top to bottom, stopping the comparison as soon as a valid match is found. When this happens, the corresponding result expression is evaluated and returned.

It is also possible to specify an aditional condition that must be satisfied for a pattern to be accepted. This condition can use any identifiers declared in its corresponding pattern, and it is not evaluated unless the pattern pattern returns a correct match.

4.9 Operators

V contains a number of infix binary operators to manipulate data.

Along with them, there is only one prefix unary operator, the negation operator. This operator is handled differently from a function application, both in its priority and its associativity.

4.9.1 Priority

Every operator is ordered within a priority system, in which operators at a higher priority level are evaluated first. The levels are ordered in numerical order (i.e. priority 9 is the highest level). For different operators at the same priority level, the evaluation is always done from left to right.

4.9.2 Associativity

Some operators can be composed several times in a row, such as addition or function application. For these operators, it is necessary to define how they are interpreted to return the desired value. There are 2 possible associativities that an operator can have:

- Left-associative
  `((a + b) + c) + d`
- Right-associative
  `a + (b + (c + d))`

4.9.3 Table of Operators

Below is a summary of every operator avaiable in the language, along with a small description and their associativities (if any). The table is ordered by decreasing priority level (the first operator has the highest priority).

Priority	Operator	Meaning	Associativity
10	`f x`	Function Application	Left
9	`f . g`	Function Composition	Right
	`x !! y`	List Indexing	Left
8	`x * y`	Multiplication	Left
	`x / y`	Division	Left
	`x % y`	Remainder	Left
7	`x + y`	Addition	Left
	`x - y`	Subtraction	Left
	`- x`	Unary Negation	None
6	`x :: y`	List Construction	Right
5	`x @ y`	List Concatenation	Right
4	`x == y`	Equals	None
	`x != y`	Not Equals	None
	`x > y`	Greater Than	None
	`x >= y`	Greater Than Or Equal	None
	`x < y`	Less Than	None
	`x <= y`	Less Than Or Equal	None
3	`x && y`	Logical AND	Right
2	`x || y`	Logical OR	Right
1	`x $ y`	Function Application	Right

4.9.4 Operators as Functions

It is possible to use any operator as a function by wrapping it in parenthesis. The left-hand operand becomes the first argument of the function, and the right-hand operand becomes the second argument.

This is useful mainly when passing operators as arguments to functions.

```
// Both expressions are equivalent
zipWith (\x y -> x + y) [1,2,3] [3,2,1]
zipWith (+) [1,2,3] [3,2,1]

// Adds 2 to every element in the list
map ((+) 2) [1,2,3]
```

As is shown in the last example, it is possible to provide the left-hand operand to obtain a partially applied function. If one wishes to provide the right-hand operand instead, it is possible to use the `flip` function, which changes the order of a function that takes two parameters.

```
// Divides 2 by every value in the list
map ((/) 2) [1,2,3]

// Divides every value in the list by 2
map (flip (/) 2) [1,2,3]
```

4.9.5 Functions as Operators

Wrapping a function name in backticks (`) will turn it into an infix binary operator. The first parameter of the function will become the left-hand operand, while the second parameter will become the right-hand operand.

```
4 `add` 5
add 4 5
```

It is possible to use this with functions that take more than 2 parameters, but then it becomes necessary to use parenthesis to pass the remaning parameters. This greatly reduces the readability of the code, and is therefore not encouraged.

4.9.6 Defining new Operators

Is is possible to define new operators to be used like regular operators. The syntax for this is the same as creating a new function, but the operator must be enclosed in parenthesis.

```
let (%+) x y = x % y + 1;

5 %+ 4 // 2
```

When declaring an operator, it is possible to also define its associativity and priority. This is done by using the keywords `infixl` (left associative), `infixr` (right associative) and `infix` (non-associative), followed by a number from 1 to 9.

```
let infixl 1 ($) f x = f x;
let f x = x + 2;

f $ 4 // 6
```

If the associativity and priority information is not provided, the operator will have priority 1 and be left associative.

The following are the list of characters allowed for operators.

?	!	%	&	*	+
-	.	/	<	=	>
@	^	|	~		

4.10 Functions

There are 4 types of functions that a programmer can declare:

1. Named functions
2. Recursive Named functions
3. Lambdas (unnamed functions)
4. Recursive Lambdas

4.10.1 Named Functions

These are functions that have a name by which they can be called after their definition. After the name, the programmer must specify one or more parameters, which can be any pattern. If an explicitly typed pattern is used, it must be enclosed in parenthesis. After every argument, the programmer can specify the return type of the function.

The body of a function can use any parameter declared in its definition to compute a return value. Since every expression in the language returns a value, any valid expression is accepted as the body of a function. The only constraint is that, if the definition specifies a return type, the value must be of that type.

Below are three examples of named functions:

```
let add x y =
  x + y
;

let duplicate (x: Int): Int =
  x * 2
;

let addTuple (x, y) =
  x + y
;
```

4.10.2 Recursive Named Functions

These functions differ from regular named functions by the fact that they can be called from within their own body. This means that the function can be called recursively, iterating over a certain value (or values). To indicate that a function is recursive, the keyword `rec` is added before its name. Below are two examples of recursive named functions:

```
let rec count ls =
  if empty? ls then
    0
  else
    1 + count (tail ls)
```

```
;

let rec factorial (x: Int): Int =
  if x == 0 then
    1
  else
    x * factorial (x - 1)
;
```

Here, both functions perform a test that determines whether the end condition is met. If the end condition is met, the function returns a simple value. If the end condition is not met, the function recursively calls itself with a modified value, continuing the iteration.

In the case of the `count` function, the recursion terminates when the input is an empty list. For the `factorial` function, an input equal to 0 terminates the recursion.

4.10.3 Lambdas

These are simple unnamed functions with a compact syntax that allows them to be written in a single line most of the time. This is useful mostly when passing lambdas as arguments to other functions, since they do not require creating a full named declaration.

The general syntax of a lambda is as follows:

```
\param1 param2 ... -> body
```

A backslash (\) indicates the start of a lambda, followed immediately by its parameters. Like in named functions, these can be any valid pattern. Unlike named functions, however, the return type of a lambda is never specified.

After the parameters, an arrow (->) indicates the start of the function body, which extends as far to the right as possible. Because of this, lambdas are usually enclosed in parenthesis to limit their scope.

Below are the same examples shown in the named functions section, but defined using lambda expressions. Notice that, without the use of parenthesis to enclose each lambda, the first function would try to include everything inside its body, resulting in a parsing error.

```
\\ add
(\x y -> x + y)

\\ duplicate
(\(x: Int) -> x * 2)

\\ AddTuple
(\(x, y) -> x + y)
```

4.10.4 Recursive Lambdas

Just like there is a recursive variant of named functions, there is a recursive variant of lambdas. These are compact expressions to define recursive functions. Like regular lambdas, they are used mostly to be passed as arguments to other functions, and are usually enclosed in parenthesis.

Unlike for regular lambdas, it is necessary to specify a name for a recursive lambda. Without a name, it would be impossible to call itself within its body. It is important to realize that, unlike with recursive named functions, this name is limited in scope to the inside of the lambda definition.

```
(rec fac x -> if x == 0 then 1 else x * fac (x - 1))

fac 4 // This is invalid code
```

With the example above, we see that the programmer tried to call a recursive lambda outside its definition. The offending code is outside the scope in which `fac` is available, resulting in invalid code.

4.10.5 Function Type

Every function has a type consisting of its parameter types and return type. Every parameter type is separated by an arrow (->), and the return type is also separated by a single arrow from the parameter types.

The syntax for a function type is as follows:

```
param1 -> param2 -> ... -> return
```

If one of the parameters is itself a function, it is possible to use parenthesis to indicate this.

The following function type defines a function that takes two parameters. The first is a function of type `Int -> Int`. The second parameter is an `Int`, and the return type is also `Int`.

```
(Int -> Int) -> Int -> Int
```

If the parenthesis were ommited, the type would describe a function that takes 3 parameters of type `Int`.

4.11 Partial Application and Currying

Technically, every function in *V* takes only one parameter. When a function is defined as having multiple parameters, it is actually a curried function.

As an example, take the following function, which returns the largest of two numbers:

```
let max x y =
  if x > y then
    x
  else
```

```
    y
;
```

This appears to be a function that takes two integers and returns an integer. In reality, `max` is a function that takes one integer and returns another function. This returned function takes one integer as a parameter and returns another integer.

This allows what is called *partial application*, which is when a function is called with too few arguments. This creates a function that "fixes" the applied arguments and returns a function that takes the remaining arguments.

Using the example above, we can write `max 5` to create a new function that takes only one argument. This function will then return the largest between its argument and the number 5.

It can then be bound to a name, just like any other function, and used elsewhere. This is also useful for quickly creating new functions with fixed data or to be passed as arguments.

```
let max5 = max 5;

max5 3 // Returns 5
max5 10 // Returns 10
```

String Conversions There are available functions to convert integers and booleans into and from strings. There are no included functions to convert compound types, but it is possible to create custom ones for each use case.

To convert strings to integers, the function is `parseInt`.

To convert integers to strings, the function is `printInt`.

To convert strings to booleans, the function is `parseBool`.

To convert booleans to strings, the function is `printBool`.

4.12 Comments

Comments are text that is ignored by the interpreter. They can be used to add notes or reminders for yourself or anyone that reads the source code.

Currently, only single line comments are avaiable. They begin with two forward-slashes (//) and continue until the end of the current line

```
// This is a comment on its own line.
3 + 4 // This line has code and a comment.
```

4.13 Libraries

Libraries are collections of constant and function declarations designed to be reused in multiple programs. These files can then be compiled to increase loading times when interpreting programs, or be loaded as parseable text files on their own.

To import a library in a program, the following syntax is used:

```
import "library"
```

The name of the library must be a string indicating the path of the library file. This path can either be relative to the program that is being executed or absolute. If a file extension is not provided for the file, it is assumed to be *vl*, which is the default extension used for compiled *V* libraries, or *v*, which is the extension for source code files in *V*.

Libraries can be imported anywhere in a program, and their functions will have their scope limited to wherever they were imported.

In the example below, we have a library with a single function `double`. This library is then imported inside a function in a program. Because the library was imported inside the scope of the function body, none of its functions can be called outside of it.

```
// math.vl
let double x = x * 2;

//----------------
// Program.v
let quadruple x =
  import "math"
  double (double x) // Valid
;

double 4 // Invalid
```

4.14 REPL

The REPL is a way to interactively test the language. Every expression written in the REPL is immediately evaluated and the result is printed in a new line.

```
> 4 + 5
9
```

Expressions can span multiple lines: if the parser cannot understand the current line as a legal expression (e.g because of a missing parentheses), you can continue to write code in a new line.

```
> if (4 > 5) then
  "Hello"
else
  "World"
"World"
```

If you write a binding, it is added to the current environment and can be used in subsequent expressions. The value and type of the binding are also printed to the terminal.

```
> let x = 4;
x: Int = 4
> if (x > 5) then "Hello" else "World"
"World"
```

There are also 4 commands available in the REPL that are not part of the language:

- <type>

  If you use this command, the REPL will print the type of the expression, instead of its value.

  ```
  > <type> fst "Hello"
  Char
  ```

- <list>

  This command lists any bindings that you have created in the environment, showing their identifier, type and value.

  ```
  > let x = 3;
  x: Int = 3
  > let y = (\x -> x);
  y: t -> t
  > <list>
  x: Int = 3
  y: t -> t
  ```

- <list-all>

  Similar to the <list> command, but this also lists all of the bindings that exist in the standard library.

  ```
  > <list-all>
  abs: Int -> Int
  all: (t -> Bool) -> [t] -> Bool
  ...
  ```

- <clear>

  Clears any bindings that you have created.

- <history>

  Prints all the expressions you have written to the console. This is useful if you want to copy experimental code to a file or share your code with others.

  ```
  > let x = 3;
  x: Int = 3
  > x + 4
  7
  > <history>
  let x = 3;
  x + 4
  ```

5 Standard Library

The Standard Library, called `stdlib`, is always imported in every *V* program. It provides basic functions for a number of use cases, ranging from numerical operations to function manipulation.

Some basic language features, such as list comprehensions and ranges, depend on the existence of the `stdlib`. This means that, while it is possible to create programs without importing the `stdlib`, doing so will most likely break any existing program.

5.1 Operations on Basic Values

5.1.1 Operations on All Values

The operations below are helper functions, designed to allow cleaner code.

id :: $a \to a$

Identity function.

const :: $a \to b \to a$

Always returns the first parameter it is passed.

5.1.2 Operations on Numbers

The 4 basic operations (addition, subtraction, multiplication and division) are built into the language. Other operations must be defined in terms of these.

One important thing to note is that the unary negation operator (`-`) is tightly coupled with the `negate` function defined in the `stdlib`. While the operator is defined inside the language, it depends on the presence of the `stdlib` to function.

remainder :: $Int \to Int \to Int$

Integer remainder, satisfying:

```
(x / y) * y + (remainder x y) = x
```

(%) :: $Int \to Int \to Int$ | left-associative, priority 8 |

Infix version of `remainder`

negate :: $Int \to Int$

Unary negation, satisfying:

```
x + (negate x) = 0
```

abs :: *Int* → *Int*

Absolute value

5.1.3 Operations on Booleans

Below are all the operations on booleans defined in the Standard Library.

and :: *Bool* → *Bool* → *Bool*

Boolean "AND"

(&&) :: *Bool* → *Bool* → *Bool* | right-associative, priority 3 |

Infix version of `and`

or :: *Bool* → *Bool* → *Bool*

Boolean "OR"

(||) :: *Bool* → *Bool* → *Bool* | right-associative, priority 2 |

Infix version of `or`

not :: *Bool* → *Bool*

Boolean "not"

```
not True = False
not False = True
```

xor :: *Bool* → *Bool* → *Bool*

Boolean "xor"

```
xor True True = False
xor True False = True
xor True False = True
xor False False = False
```

5.1.4 Operations on Functions

Basic manipulation of functions and application. Most of the usefulness of these functions come from their infix versions. They allow more compact and easier to read code

to be written, mainly reducing the need for parentheses.

flip :: $(a \to b \to c) \to b \to a \to c$

`flip f` takes its first two arguments in reverse order of `f`.

```
flip f x y = f y x
```

apply :: $(a \to b) \to a \to b$

This function simply applies its second argument to its first. While this seems redundant (after all, `apply f x` is the same as `f x`), it can be used higher order situations.

($) :: $(a \to b) \to a \to b$ | right-associative, priority 1 |

Infix version of `apply`. While it has the same funcionality as normal function application, it is right-associative with the lowest possible priority.

In some situations, this allows parentheses to be omitted.

```
f $ g $ h x = f (g (h x))
```

compose :: $(b \to c) \to (a \to b) \to a \to c$

Function composition. Applies the third argument to the second one, applying the resulting value to the first argument.

```
compose f g x = f (g x)
```

(.) :: $(b \to c) \to (a \to b) \to a \to c$ | right-associative, priority 9 |

Infix version of `compose`.
Can be used with `$` to reduce the number of parentheses needed.

```
f .  g .  h $ x = f (g (h x))
```

5.1.5 Operations on Tuples

The stdlib also provides basic functions for manipulating tuples with 2 components. For larger tuples, it is necessary to create custom functions.

fst :: $(a, b) \to a$

Returns the first component of a pair.

snd :: $(a, b) \to b$

Returns the second component of a pair.

swap :: $(a, b) \to (b, a)$

Swap the components of a pair.

5.1.6 Operations on Records

The functions below are used with record accessors (`#label`) to get, set and change individual fields in a record. While the functions themselves have a more generic type and can, therefore, be used in more circumstances, they were created with records in mind.

get :: $(a \to b \to (c, d)) \to b \to c$

Returns the value of a field in a record.

```
get #label record
```

set :: $(a \to b \to (c, d)) \to a \to b \to d$

Returns the inputed record, modifying a single field value.

```
set #label value record
```

modify :: $(a \to b \to (c, d)) \to (c \to a) \to b \to d$

Returns the inputed record, modifying a single field value by applying the old value to the specified function.

```
modify #label function record
```

5.2 Operations on Lists

5.2.1 Basic Operations

Basic functions to aid in using lists.

head :: $[a] \to a$

Returns the first element of a list, which must have at least one element.

last :: $[a] \to a$

Returns the last element of a list, which must have at least one element.

tail :: $[a] \rightarrow [a]$

Removes the first element of a list, which must have at least one element.

init :: $[a] \rightarrow [a]$

Removes the last element of a list, which must have at least one element.

tail :: $[a] \rightarrow [a]$

Removes the first element of a list, which must have at least one element.

empty? :: $[a] \rightarrow Bool$

Returns `True` if the list is empty, and `False` otherwise.

length :: $[a] \rightarrow Int$

Returns the number of elements in the list.

append :: $a \rightarrow [a] \rightarrow [a]$

Adds an element to the end of a list.

concat :: $[a] \rightarrow [a] \rightarrow [a]$

Appends two lists, maintaning order.

(@) **::** $[a] \rightarrow [a] \rightarrow [a]$ | right-associative, priority 5 |

Infix version of `concat`.

5.2.2 Generation Operations

These operations create lists based on input values.

range :: $Int \rightarrow Int \rightarrow Int \rightarrow [Int]$

`range start finish increment` generates a list of the form
[`start`, `start` + `increment`, `start` + 2 * `increment`, ..., `n`], where

$\texttt{increment} > 0 \implies \texttt{n} \leq \texttt{finish}$
$\texttt{increment} < 0 \implies \texttt{n} \geq \texttt{finish}$

5.2.3 Transformation Operations

These operations transform a list, altering its elements, their order, or both.

reverse :: $[a] \to [a]$

Returns the elements of the input in reverse order.

map :: $(a \to b) \to [a] \to [b]$

`map f ls` returns a list by applying the function `f` to each element of the list `ls`.

5.2.4 Reduction Operations

These operations take a list and reduce it to a simple value.

fold :: $(b \to a \to b) \to b \to [a] \to b$

`fold f acc ls` reduces the list using the function `f`, applying it to an accumulator (`acc`) and each element of the list, from left to right.

reduce :: $(a \to a \to a) \to [a] \to a$

The same as `fold`, but using the first element of the list as the `acc`

all :: $(a \to \mathit{Bool}) \to [a] \to \mathit{Bool}$

Checks whether all elements of a list satisfy a predicate. An empty list returns true.

any :: $(a \to \mathit{Bool}) \to [a] \to \mathit{Bool}$

Checks whether any elements of a list satisfy a predicate. An empty list returns false.

maximum :: $\mathit{Orderable}\ a \Rightarrow [a] \to a$

Returns the largest element of the list.

minimum :: *Orderable a => [a] → a*

Returns the smallest element of the list.

5.2.5 Sublist Operations

These operations return smaller segments of an existing list.

take :: *Int → [a] → [a]*

`take n ls` returns the first `n` elements of `ls`.

drop :: *Int → [a] → [a]*

`drop n ls` returns the list resulting from removing the first `n` elements of `ls`.

takeWhile :: *(a → Bool) → [a] → [a]*

`takeWhile p ls` returns the longest prefix of `ls` such that every element satisfies `p`.

dropWhile :: *(a → Bool) → [a] → [a]*

`dropWhile p ls` returns the suffix that remains after `takeWhile p ls`.

sublist :: *Int → Int → [a] → [a]*

`sublist start length ls` drops the first `start` elements of `ls`, and then takes the first `length` elements of the resulting list.

5.2.6 Search Operations

These operations search for specific elements in a list.

exists :: *Equatable a => a → [a] → Bool*

Tests whether the given element exists in the list.

filter :: *(a → Bool) → [a] → [a]*

`filter p ls` returns a sublist of `ls` such that every element satisfies `p`.

5.2.7 Indexing Operations

Manipulate a list through the index of its elements

indexOf :: *Equatable a => a → [a] → Int*

`indexOf t ls` returns the index of the first occurrence of `t` in `ls`. If the element does not occur, returns `-1`.

nth :: *Int → [a] → a*

`nth n ls` returns the element of `ls` at position `n`. If `n` is negative or larger than `length ls`, an exception is raised.

(!!) :: [a] → Int → a | left-associative, priority 9 |

The infix version of `nth`. It receives its operands in reverse order, allowing for expressions in the form `ls !! n`.

5.2.8 Sorting Operations

Sort lists.

sort :: *Orderable a => [a] → [a]*

Sorts a list in ascending order.

5.2.9 Zipping Operations

Operations that deal with tuples and lists.

zip :: $[a] \to [b] \to [(a, b)]$

Takes two lists and returns a list composed of corresponding pairs. It the lists have different lengths, elements of the larger one are discarded.

zipWith :: $(a \to b \to c) \to [a] \to [b] \to [c]$

Takes two lists and a function, returning a list composed of the result of applying the function to corresponding elements in each list. It the lists have different lengths, elements of the larger one are discarded.

unzip :: $[(a, b)] \to ([a], [b])$

Takes a list of pairs, returning a pair of lists, each containing the corresponding components of the original list.

5.3 String Conversion Operations

Converts values from and to strings.

parseInt :: *String* → *Int*

Converts a string into an integer. The only representation accepted is decimal (without a leading +), and the function raises an exception if parsing fails.

printInt :: *Int* → *String*

Converts an integer value into a string.

parseBool :: *String* → *Bool*

Converts a string into a boolean. Valid strings are `"true"` and `"false"`.

printBool :: *Bool* → *String*

Converts a boolean value into a string.

5.4 IO Operations

The standard library provides a few additional IO functions to complement the builtin write and read functions,

readLn :: *Void* → *IO String*

Reads a complete line from the standard input. A line ends with either a linefeed (\n) or a carriage return (\r).

writeLn :: *String* → *IO Void*

Prints the string to the standard output, followed by a linefeed (\n).

6 Changelog

v0.2

May 1, 2017

Additions

- **Match Expressions** Abstract Syntax [redacted] Concrete Syntax

  This is a structure to control the flow of a program by attempting to match a value against a list of patterns.

- **Prefix Notation for Operators** Concrete Syntax

  Wrapping any infix operator in parenthesis turns it into a function that takes two parameters.

- **Infix Notation for Functions** Concrete Syntax

  Wrapping any function name in backticks (‘) turns it into a binary operator.

- **Creating new Operators** Concrete Syntax

  When declaring a function, wrapping a name in parenthesis makes it a operator. Only a small number of characters are allowed for operator names, and it is possible to define the associativity and priority of the newly created operator (if this information is ommited, the default values are left associative and priority 9).

- **Partial Record Pattern** Abstract Syntax [redacted]

  This pattern does not create an equality constraint on records. Instead, it creates trait constraints, allowing for ad-hoc subtyping.

- **Extended Language** Abstract Syntax

  Created an extended language (syntax tree). This languaged is used for parsing, and then is translated into the core language.

- **Type Aliases** [redacted]

  Allows the programmer to specify type aliases, simplifying type declarations

Changes

- **List Operations** Abstract Syntax [redacted]

  The list operations `hd`, `tl` and `isempty` have been removed from the language defition. They have been added to the stdlib, and use pattern matching to recreate their funcionality.

- **Non-Strict Semantics** [redacted]

  Functions can now be non-strict, and compound values (lists, tuples, records) can contain undefined (`raise`) components.

- **Boolean Operations** Abstract Syntax [redacted]

  The boolean operators $\wedge$ (AND) and $\vee$ (OR) have been removed from the definition and added to the stdlib.

- **Record Projection (Accessor)** Abstract Syntax [redacted]

  Replaced record projection with record accessor, allowing for editing of individual fields in a record.

- **Conditionals** Abstract Syntax [redacted]

  Removed conditionals from core language, adding them to the extended language.

Removals

- **Input and Output** Abstract Syntax [redacted] Concrete Syntax

  Removed the input and output expressions. This was done because they are not compatible with the functional nature of V. IO will (probably) be added again in the future, but for now it must be removed.

- **Sequence Operator, Skip and Unit** Abstract Syntax [redacted] Concrete Syntax

  Since these are not used by any expressions, they have been removed.

- **Try** Abstract Syntax [redacted] Concrete Syntax

  Exceptions can no longer be handled. This was done because of the change to non-strict semantics.

- **Project Index (TuplePosition Trait)** Abstract Syntax [redacted] Concrete Syntax

  Removed the ability to project a specific component of a tuple, making patterns the only way to decompose tuples.

  As a result of this, the TuplePosition Trait has been removed.