

# V Syntax and Documentation

Arthur Giesel Vedana

October 31, 2017

# Contents

<b>Introduction</b>	<b>iii</b>
<b>1 Abstract Syntax and Semantics</b>	<b>1</b>
1.1 Abstract Syntax . . . . .	1
1.1.1 Expressions . . . . .	1
1.1.2 Types . . . . .	5
1.1.3 Traits . . . . .	6
1.2 Operational Semantics . . . . .	8
1.2.1 Paths . . . . .	9
1.2.2 Pattern Matching . . . . .	10
1.2.3 Big-Step Rules . . . . .	11
1.3 Type System . . . . .	20
1.3.1 Polymorphism . . . . .	20
1.3.2 Traits . . . . .	21
1.3.3 Type Inference System . . . . .	21
<b>2 Extended Language</b>	<b>27</b>
2.1 Abstract Syntax . . . . .	27
2.1.1 Additions . . . . .	28
2.2 Translation . . . . .	30
<b>3 Concrete Syntax</b>	<b>31</b>
3.1 Notation Conventions . . . . .	31
3.2 Basic Structure . . . . .	31
3.3 Identifiers and Operators . . . . .	32
3.4 Terms . . . . .	33
3.5 Functions and Declarations . . . . .	34
<b>4 Language Guide</b>	<b>35</b>
4.1 Basic Values . . . . .	35
4.2 Compound Values . . . . .	36
4.2.1 Lists . . . . .	36
4.2.2 Tuples . . . . .	38
4.2.3 Records . . . . .	38
4.3 Identifiers . . . . .	39
4.4 Patterns . . . . .	39
4.5 Constants . . . . .	40
4.6 Type Annotations . . . . .	40
4.7 Conditionals . . . . .	41
4.8 Match Expressions . . . . .	41
4.9 Operators . . . . .	42
4.9.1 Priority . . . . .	42
4.9.2 Associativity . . . . .	42

4.9.3	Table of Operators . . . . .	42
4.9.4	Operators as Functions . . . . .	43
4.9.5	Functions as Operators . . . . .	44
4.9.6	Defining new Operators . . . . .	44
4.10	Functions . . . . .	44
4.10.1	Named Functions . . . . .	45
4.10.2	Recursive Named Functions . . . . .	45
4.10.3	Lambdas . . . . .	46
4.10.4	Recursive Lambdas . . . . .	46
4.10.5	Function Type . . . . .	47
4.11	Partial Application and Currying . . . . .	47
4.12	Comments . . . . .	48
4.13	Libraries . . . . .	48
<b>5</b>	<b>Standard Library</b>	<b>50</b>
5.1	Operations on Basic Values . . . . .	50
5.1.1	Operations on All Values . . . . .	50
5.1.2	Operations on Numbers . . . . .	50
5.1.3	Operations on Booleans . . . . .	51
5.1.4	Operations on Functions . . . . .	51
5.1.5	Operations on Tuples . . . . .	52
5.1.6	Operations on Records . . . . .	53
5.2	Operations on Lists . . . . .	53
5.2.1	Basic Operations . . . . .	53
5.2.2	Generation Operations . . . . .	54
5.2.3	Transformation Operations . . . . .	55
5.2.4	Reduction Operations . . . . .	55
5.2.5	Sublist Operations . . . . .	56
5.2.6	Search Operations . . . . .	56
5.2.7	Indexing Operations . . . . .	57
5.2.8	Sorting Operations . . . . .	57
5.2.9	Zipping Operations . . . . .	57
5.3	String Conversion Operations . . . . .	58
<b>6</b>	<b>Changelog</b>	<b>58</b>

## 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.

$e$	$::=$	$\dots$
		$  \text{ func}$
		$  e_1 e_2$
		$  x$
$\text{func}$	$::=$	$\text{fn } x \Rightarrow e$
		$  \text{rec } f \ x \Rightarrow e$
		$  \text{rec } f : T \ x \Rightarrow e$
$x$	$::=$	$\{x_0, x_1, \dots\}$

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 ( $\text{fn } x \Rightarrow e$ ) 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 within 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:

$e$	::=	...
		<i>Builtin</i>
<i>Builtin</i>	::=	+
		-
		*
		÷
		-
		<
		≤
		>
		≥
		=
		≠
		∨
		∧
		get
		set
		stack
		distort

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

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

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

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

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

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

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

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

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

$e$	$::=$	$con$	
$con$	$::=$	$n$	(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.

$$\begin{aligned}
e &::= \dots \\
&| \{l_1 : e_1, \dots l_n : e_n\} \quad (n \geq 1) \\
&| \#l \\
&| \#(e_1, \dots e_n) \quad (n \geq 2) \\
l &::= \{l_1, l_2, \dots\}
\end{aligned}$$

The most basic type of path is a simple label  $\#l$ . An accessor can also be made by combining multiple accessors. The  $\#(e_1, \dots e_n)$  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.

Insert 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.

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

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  $l_1, \dots, l_n$ , one can use the pattern  $\{l_1 : p_1, \dots, l_n : p_n, \dots\}$ . This pattern will match any record whose set of labels is a superset of  $l_1, \dots, l_n$ .

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

$$\begin{aligned} e & ::= \dots \\ & \quad | \quad \text{match } e \text{ with } match_1, \dots, match_n \quad (n \geq 1) \\ \\ match & ::= p \rightarrow e \\ & \quad | \quad p \text{ when } e_1 \rightarrow e_2 \end{aligned}$$

**Exceptions** This expression always evaluates to a runtime error.

$$\begin{aligned} e & ::= \dots \\ & \quad | \quad \text{raise} \end{aligned}$$

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.

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

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.

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

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

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

$$\begin{array}{lcl}
T & ::= & \dots \\
& | & X^{\text{Traits}} \\
\\
X & ::= & X_1, X_2, \dots
\end{array}$$

### 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*).

$$\begin{array}{lcl}
\textit{Traits} & ::= & \emptyset \\
& | & \{\textit{Trait}\} \cup \textit{Traits} \\
\\
\textit{Trait} & ::= & \textit{Equatable} \\
& | & \textit{Orderable} \\
& | & \{l : \textit{Type}\} \quad (\text{Record Label})
\end{array}$$

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

By default, the following rules hold for conformance:

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

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

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

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

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

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

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

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

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

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

## 1.2 Operational Semantics

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

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

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

1. Arity of constructors  
If a constructor has arity  $n$ , it requires  $n$  arguments to become fully evaluated.
2. Associations between identifiers and values  
A new association is created every time a value is bound. This happens in `let` declarations, function application and `match` expressions.

Below are the definitions of both values and environments:

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

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

The values  $\ll \text{Builtin} . v_1, \dots v_n \gg$  and  $\ll \text{con} . v_1, \dots v_n \gg$  are partial applications of built-in functions and constructors, respectively.

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

### 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 function 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.

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

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

$$\begin{array}{c}
\text{traverse}(\text{path}_1, \{l_1 : v_1, \dots, l_n : v_n\}) = \text{rec}, r \\
\text{traverse}(\text{path}_2, \text{rec}, v) = v', \text{rec}' \\
\text{traverse}(\text{path}_1, \{l_1 : v_1, \dots, l_n : v_n\}, \text{rec}') = \text{rec}, r' \\
\hline
\text{traverse}(\text{path}_1 \cdot \text{path}_2, \{l_1 : v_1, \dots, l_n : v_n\}, v) = v', r'
\end{array}$$

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.

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

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

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

$$\begin{array}{c}
\{\} \vdash v_2 \Downarrow v' \quad \text{traverse}(\text{path}, \{l_1 : v_1, \dots, l_n : v_n\}, v') = v'', r \\
\{\} \vdash v_1 \Downarrow v'' \quad \text{traverse}(\text{path}[v_1, v_2], \{l_1 : v_1, \dots, l_n : v_n\}, v) = v''', r \\
\hline
\text{traverse}(\text{path}[v_1, v_2], \{l_1 : v_1, \dots, l_n : v_n\}, v) = v''', r
\end{array}$$

Initially, the provided update value is applied to the second component ( $v_2$ ) of the distorted accessor. This value is then provided as the new update value for a call to *traverse*, returning an old value and the updated record. This old value is then provided as argument to the first component ( $v_1$ ) of the distorted accessor, and the result of this is the output of the whole *traverse* (along with the updated record).

### 1.2.2 Pattern Matching

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

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

To aid in this matching, an 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:

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

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

$$\frac{\text{con}_1 = \text{con}_2 \quad \forall i \in [1, n] \quad \text{match}(p_i, v_i) = \text{vars}_i}{\text{match}(\text{con}_1 \ v_1, \dots v_n, \text{con}_2 \ p_1, \dots p_n) = \bigcup_{i=1}^n \text{vars}_i}$$

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

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

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

$$\neg \text{match}(p, v)$$

### 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.

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

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

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

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.

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

$$\frac{\text{env.aries}(\text{con}) = 0}{\text{env} \vdash \text{con} \Downarrow \text{con}} \quad (\text{BS-CON0})$$

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

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

When applying non recursive functions, a new association between the parameter identifier ( $x$ ) and the argument ( $v_2$ ) is added to the environment stored in the closure ( $\text{env}_1$ ). The body of the function ( $e$ ) is then evaluated using this new environment.

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

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

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

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

$$\frac{\text{env} \vdash e_1 \Downarrow \langle \text{rec } f : T \ x \Rightarrow e, \text{env}_1 \rangle \quad \text{env} \vdash e_2 \Downarrow v_2 \quad \{x \rightarrow v_2, f \rightarrow \langle \text{rec } f : T \ x \Rightarrow e, \text{env}_1 \rangle\} \cup \text{env}_1 \vdash e \Downarrow v}{\text{env} \vdash e_1 e_2 \Downarrow v} \quad (\text{BS-APPFNREC2})$$

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.

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \text{con} . v_1, \dots v_n \gg \quad n + 1 < \text{env.aries}(\text{con}) \quad \text{env} \vdash e_2 \Downarrow v}{\text{env} \vdash e_1 e_2 \Downarrow \ll \text{con} . v_1, \dots v_n, v \gg} \quad (\text{BS-APPCON})$$

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.



$$\frac{\text{env} \vdash e_1 \Downarrow \ll \text{con} . v_1, \dots v_n \gg \quad n + 1 = \text{env.arity}(\text{con}) \quad \text{env} \vdash e_2 \Downarrow v}{\text{env} \vdash e_1 e_2 \Downarrow \text{con } v_1, \dots v_n, v} \quad (\text{BS-APPCONTOTAL})$$

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

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

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.

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

$$\frac{\text{env} \vdash e_1 \Downarrow v \quad \text{env} \vdash e_2 \Downarrow \text{raise}}{\text{env} \vdash e_1 e_2 \Downarrow \text{raise}} \quad (\text{BS-APPRaise2})$$

**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  $\{l_1 : e_1, \dots l_n : e_n\}$  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.

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

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

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

**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.

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

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.

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

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.

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

**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.

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \text{get} . \#path \gg \quad \text{env} \vdash e_2 \Downarrow \{l_1 : v_1, \dots l_n : v_n\}}{\text{env} \vdash e_1 e_2 \Downarrow v'} \quad (\text{BS-GET})$$

$traverse(path, \{l_1 : v_1, \dots l_n : v_n\}) = v', r'$

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).

$$\frac{\text{env} \vdash e_1 \Downarrow \ll \text{set} . \#path, v \gg \quad \text{env} \vdash e_2 \Downarrow \{l_1 : v_1, \dots l_n : v_n\}}{\text{env} \vdash e_1 e_2 \Downarrow r'} \quad (\text{BS-SET})$$

$traverse(path, \{l_1 : v_1, \dots l_n : v_n\}, v) = v', r'$

**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.

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

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

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

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

$$\frac{\text{env} \vdash e_1 \Downarrow \llcorner \div . n_1 \gg \quad \text{env} \vdash e_2 \Downarrow 0}{\text{env} \vdash e_1 e_2 \Downarrow \text{raise}} \quad (\text{BS-}\div\text{ZERO})$$

$$\frac{\text{env} \vdash e_1 \Downarrow \llcorner \div . n_1 \gg \quad \text{env} \vdash e_2 \Downarrow n_2 \quad \|n_2\| \neq 0 \quad \|n\| = \|n_1\| \div \|n_2\|}{\text{env} \vdash e_1 e_2 \Downarrow n} \quad (\text{BS-}\div)$$

**Equality Operations** The equality operators ( $=$  and  $\neq$ ) test the equality of fully applied constructors and records.

$$\frac{\text{env} \vdash e_1 \Downarrow \llcorner = . (\text{con } v_1^1, \dots v_n^1) \gg \quad \text{env} \vdash e_2 \Downarrow (\text{con } v_1^2, \dots v_n^2) \quad \forall k \in [1, n] \quad \text{env} \vdash (= v_k^1) v_k^2 \Downarrow \text{true}}{\text{env} \vdash e_1 e_2 \Downarrow \text{true}} \quad (\text{BS-}=\text{CONTRUE})$$

$$\frac{\text{env} \vdash e_1 \Downarrow \llcorner = . (\text{con } v_1^1, \dots v_n^1) \gg \quad \text{env} \vdash e_2 \Downarrow (\text{con } v_1^2, \dots v_n^2) \quad \exists k \in [1, n] \quad \text{env} \vdash (= v_k^1) v_k^2 \Downarrow \text{false}}{\text{env} \vdash e_1 e_2 \Downarrow \text{false}} \quad (\text{BS-}=\text{CONFALSE})$$

$$\frac{\text{env} \vdash e_1 \Downarrow \llcorner = . (\text{con } v_1^1, \dots v_n^1) \gg \quad \text{env} \vdash e_2 \Downarrow (\text{con}' v_1^2, \dots v_n^2) \quad \text{con}' \neq \text{con}}{\text{env} \vdash e_1 e_2 \Downarrow \text{false}} \quad (\text{BS-}=\text{CONFALSE2})$$

$$\frac{\text{env} \vdash e_1 \Downarrow \llcorner = . \{l_1^1 : v_1^1, \dots l_n^1 : v_n^1\} \gg \quad \text{env} \vdash e_2 \Downarrow \{l_1^2 : v_1^2, \dots l_n^2 : v_n^2\} \quad \forall k \in [1, n] \quad l_k^1 = l_k^2 \wedge \text{env} \vdash (= v_k^1) v_k^2 \Downarrow \text{true}}{\text{env} \vdash e_1 e_2 \Downarrow \text{true}} \quad (\text{BS-}=\text{RECORDTRUE})$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \quad \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \quad \text{env} \vdash v_1 = v_3 \Downarrow \text{false} \quad \text{env} \vdash v_1 \text{ opIneq } v_3 \Downarrow b}{\text{env} \vdash e_1 \text{ opIneq } e_2 \Downarrow b} \quad (\text{BS-INEQLISTHEAD})$$

$$\frac{\text{env} \vdash e_1 \Downarrow v_1 :: v_2 \quad \text{env} \vdash e_2 \Downarrow v_3 :: v_4 \quad \text{env} \vdash v_1 = v_3 \Downarrow \text{true} \quad \text{env} \vdash v_2 \text{ opIneq } v_4 \Downarrow b}{\text{env} \vdash e_1 \text{ opIneq } e_2 \Downarrow b} \quad (\text{BS-INEQLISTTAIL})$$

**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.

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

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

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

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

**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.

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

$$\frac{\text{env} \vdash e_1 \Downarrow v \quad \neg \text{match}(p, v)}{\text{env} \vdash \text{let } p = e_1 \text{ in } e_2 \Downarrow \text{raise}} \quad (\text{BS-LET2})$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

#### 1.3.1 Polymorphism

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

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

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

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

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

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



### 1.3.2 Traits

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

### 1.3.3 Type Inference System

The type inference system is composed of two basic parts:

- Constraint Collection
- Constraint Unification

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

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

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

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

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

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

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

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

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

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

The following are the rules for the “match” function:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\Gamma \vdash n : \text{Int} \mid \{\} \quad (\text{T-Num})$$

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{c} X_1 \text{ is new} \quad X_2 \text{ is new} \quad \text{match}(p, X_1) = C, \text{env} \\ \{x \rightarrow X_2\} \cup \text{env} \cup \Gamma \vdash e : T_1 \mid C_1 \end{array}}{\Gamma \vdash \text{rec } x \ p \Rightarrow e : X_1 \rightarrow T_1 \mid C \cup C_1 \cup \{X_2 = X_1 \rightarrow T_1\}} \quad (\text{T-REC2})$$

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

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

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

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

$$\frac{X_1 \text{ is new}}{\Gamma \vdash raise : X_1 \mid \{\}} \quad (\text{T-RAISE})$$

## 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.

$e'$	$::=$	$func'$	
		$e'_1 e'_2$	
		$x$	
		<i>Builtin</i>	
		<i>con</i>	
		$\{l_1 : e'_1, \dots l_n : e'_n\}$	$(n \geq 1)$
		$\#l$	
		$\#(e'_1, \dots e'_n)$	$(n \geq 2)$
		<i>raise</i>	
		$match\ e'\ with\ match'_1, \dots match'_n$	$(n \geq 1)$
$func'$	$::=$	$\text{fn } x \Rightarrow e'$	
		$\text{rec } f\ x \Rightarrow e'$	
		$\text{rec } f : T\ x \Rightarrow e'$	
$match'$	$::=$	$p' \rightarrow e'$	
		$p' \text{ when } e'_1 \rightarrow e'_2$	

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

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

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

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

### 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.

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

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

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

**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.

$$\begin{array}{lcl}
func' & ::= & \dots \\
& | & \text{fn } p'_1, \dots p'_n \Rightarrow e' \quad (n \geq 1) \\
& | & \text{rec } f : T\ p'_1, \dots p'_n \Rightarrow e' \quad (n \geq 1) \\
& | & \text{rec } f\ p'_1, \dots p'_n \Rightarrow e' \quad (n \geq 1)
\end{array}$$



**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.

$$\begin{array}{lcl}
 e' & ::= & \dots \\
 & | & \text{let } decl' \text{ in } e \\
 \\ 
 decl' & ::= & p' = e' \\
 & | & f \ p'_1, \dots p'_n = e' \quad (n \geq 1) \\
 & | & \text{rec } x \ p'_1, \dots p'_n = e' \quad (n \geq 1) \\
 & | & x : T' \ p'_1, \dots p'_n = e' \quad (n \geq 1) \\
 & | & \text{rec } x : T' \ p'_1, \dots p'_n = e' \quad (n \geq 1) \\
 & | & \text{type alias } \tau = T'
 \end{array}$$

**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.

$$\begin{array}{lcl}
 e' & ::= & \dots \\
 & | & [e'_1, \dots e'_n] \quad (n \geq 0)
 \end{array}$$

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

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

**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.

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

**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.

$$e' ::= \dots \\ | [e'_1 \text{ for } p' \text{ in } e'_2]$$

**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.

$$e' ::= \dots \\ | (e'_1, \dots e'_n) \quad (n \geq 2)$$

## 2.2 Translation

## 3 Concrete Syntax

### 3.1 Notation Conventions

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

$[ \langle pat \rangle ]$	optional pattern
$\{ \langle pat \rangle \}$	zero or more occurrences of pattern
$( \langle pat \rangle )$	grouping of patterns
$\langle pat_1 \rangle   \langle pat_2 \rangle$	choice
$\langle pat \rangle \langle pat' \rangle$	all patterns accepted by $pat$ , except those also accepted by $pat'$
<b>text</b>	concrete characters (written in terminal font)

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

$$\langle sentence \rangle ::= \langle pat_1 \rangle | \langle pat_2 \rangle | \dots | \langle pat_n \rangle$$

Whitespace is always explicitly expressed in productions with the  $\sqcup$  character. It is used as a shorthand for the  $\langle whitespace \rangle$  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

$\langle program \rangle$	$::= \langle whitespace \rangle \langle expression \rangle \langle eof \rangle$
$\langle library \rangle$	$::= \langle whitespace \rangle \{ \langle declaration \rangle \} \langle eof \rangle$
$\langle whitespace \rangle$	$::= \{ \langle whitechars \rangle   \langle comment \rangle \}$
$\langle whitechars \rangle$	$::= \langle whitechar \rangle \{ \langle whitechar \rangle \}$
$\langle whitechar \rangle$	$::= \langle space \rangle   \langle tab \rangle   \langle newline \rangle$
$\langle newline \rangle$	$::= \langle return \rangle \langle linefeed \rangle   \langle return \rangle   \langle linefeed \rangle$
$\langle space \rangle$	$::= \text{space (' ')}$
$\langle tab \rangle$	$::= \text{horizontal tab ('\t')}$
$\langle return \rangle$	$::= \text{carriage return ('\r')}$
$\langle linefeed \rangle$	$::= \text{line feed ('\n')}$
$\langle comment \rangle$	$::= // \{ \langle any \rangle \langle newline \rangle \} \langle newline \rangle$
$\langle any \rangle$	$::= \text{any ASCII character}$

### 3.3 Identifiers and Operators

$\langle identifier \rangle$	$::= ( \langle idstart \rangle \{ \langle idcontinue \rangle \} ) \langle reservedid \rangle$
$\langle idstart \rangle$	$::= \langle small \rangle   \_$
$\langle idcontinue \rangle$	$::= \langle small \rangle   \langle large \rangle   \langle digit \rangle   '   \_   ?$
$\langle reservedid \rangle$	$::= \text{let}   \text{true}   \text{false}   \text{if}   \text{then}   \text{else}$ $  \text{rec}   \text{nil}   \text{raise}   \text{when}   \text{match}   \text{with}$ $  \text{try}   \text{except}   \text{for}   \text{in}   \text{import}   \text{infix}$ $  \text{infixl}   \text{infixr}   \text{type}   \text{alias}$
$\langle typeident \rangle$	$::= ( \langle typeidentstart \rangle \{ \langle idcontinue \rangle \} ) \langle reservedtype \rangle$
$\langle typeidentstart \rangle$	$::= \langle large \rangle   \_$
$\langle reservedtype \rangle$	$::= \text{Int}   \text{Bool}   \text{Char}$
$\langle small \rangle$	$::= \text{a}   \text{b}   \dots   \text{z}$
$\langle large \rangle$	$::= \text{A}   \text{B}   \dots   \text{Z}$
$\langle digit \rangle$	$::= \text{0}   \text{1}   \dots   \text{9}$
$\langle operator \rangle$	$::= \langle symbol \rangle \{ \langle symbol \rangle \}$
$\langle customop \rangle$	$::= \langle operator \rangle \langle reservedop \rangle$
$\langle reservedop \rangle$	$::= +   -   *   /   <=   <   =$ $  !=   >=   >   ::$
$\langle symbol \rangle$	$::= :   ?   !   \%   \$   \&   *   +   -$ $  .   /   <   =   >   @   ^       \sim$

### 3.4 Terms

$\langle term \rangle$	$::= \langle identifier \rangle$ $  \text{ true }   \text{ false } \quad (\text{booleans})$ $  \langle number \rangle$ $  \text{ nil } \quad (\text{empty list})$ $  \text{ raise } \quad (\text{exception})$ $  \langle char \rangle$ $  \langle string \rangle$ $  \langle parentheses \rangle$ $  \langle record \rangle$ $  \# \langle identifier \rangle \quad (\text{record access})$ $  \langle squarebrackets \rangle$ $  \text{ if } \neg \langle expression \rangle \text{ then } \neg \langle expression \rangle \text{ else } \neg \langle expression \rangle$ $  \langle match \rangle$ $  \langle lambda \rangle$ $  \langle reclambda \rangle$ $  \langle let \rangle$
$\langle number \rangle$	$::= \langle decimal \rangle   \langle binary \rangle   \langle octal \rangle   \langle hexadecimal \rangle$
$\langle decimal \rangle$	$::= \langle digit \rangle \{ \langle digit \rangle \}$
$\langle binary \rangle$	$::= 0 ( b   B ) \langle bindigit \rangle \{ \langle bindigit \rangle \}$
$\langle octal \rangle$	$::= 0 ( o   O ) \langle octdigit \rangle \{ \langle octdigit \rangle \}$
$\langle hexadecimal \rangle$	$::= 0 ( x   X ) \langle hexdigit \rangle \{ \langle hexdigit \rangle \}$
$\langle bindigit \rangle$	$::= 0   1$
$\langle octdigit \rangle$	$::= 0   1   \dots   7$
$\langle hexdigit \rangle$	$::= \langle digit \rangle   a   \dots   f   A   \dots   F$
$\langle char \rangle$	$::= ' ( \langle escape \rangle   \langle any \rangle \langle \backslash \rangle ) '$
$\langle string \rangle$	$::= " \{ \langle escape \rangle   \langle any \rangle \langle \backslash \rangle \} "$
$\langle escape \rangle$	$::= \backslash ( b   n   r   t   \backslash   "   ' )$
$\langle parentheses \rangle$	$::= ( \neg \langle expression \rangle ) \neg \quad (\text{parenthesised expression})$ $  ( \neg \langle expression \rangle_1 , \neg \dots , \neg \langle expression \rangle_n ) \neg \quad (\text{tuple, } n \geq 2)$ $  ( \neg \langle operator \rangle \neg ) \neg \quad (\text{prefix operator})$
$\langle record \rangle$	$::= \{ \neg \langle recordcomp \rangle_1 , \neg \dots , \neg \langle recordcomp \rangle_n \} \neg \quad (n \geq 1)$
$\langle recordcomp \rangle$	$::= \langle identifier \rangle \neg : \neg \langle expression \rangle$

$$\begin{aligned}
\langle match \rangle &::= \text{match } \sqcup \langle expression \rangle \text{ with } \sqcup \{ \langle matchcomp \rangle \} \\
\langle matchcomp \rangle &::= | \sqcup \langle pattern \rangle [ \text{when } \sqcup \langle expression \rangle ] \rightarrow \sqcup \langle expression \rangle \\
\langle expression \rangle &::= \langle operand \rangle \{ \langle operator \rangle \sqcup \langle operand \rangle \} \\
\langle operand \rangle &::= \langle application \rangle \\
&| \quad - \sqcup \langle application \rangle \quad \text{(unary negation)} \\
\langle application \rangle &::= \langle term \rangle \sqcup \{ \langle term \rangle \sqcup \}
\end{aligned}$$

$\langle \text{lambda} \rangle$	$::= \backslash \backslash \{ \langle \text{parameter} \rangle \} \rightarrow \backslash \langle \text{expression} \rangle$
$\langle \text{reclambda} \rangle$	$::= \text{rec } \backslash \langle \text{identifier} \rangle \backslash \{ \langle \text{parameter} \rangle \} [ : \backslash \langle \text{type} \rangle ] \rightarrow \backslash \langle \text{expression} \rangle$
$\langle \text{let} \rangle$	$::= \langle \text{declaration} \rangle ; \backslash \langle \text{expression} \rangle$
$\langle \text{declaration} \rangle$	$::= \text{let } \backslash \langle \text{pattern} \rangle = \backslash \langle \text{expression} \rangle$   $\text{let } \backslash [ \text{rec } \backslash \langle \text{funcname} \rangle \{ \langle \text{parameter} \rangle \} [ : \backslash \langle \text{type} \rangle ] = \backslash \langle \text{expression} \rangle$   $\text{import } \backslash \langle \text{string} \rangle \backslash$   $\text{type } \backslash \text{alias } \backslash \langle \text{typeident} \rangle \backslash = \backslash \langle \text{type} \rangle$
$\langle \text{funcname} \rangle$	$::= \langle \text{identifier} \rangle \backslash$   $[ \langle \text{fixity} \rangle \backslash \langle \text{digit} \rangle \backslash ] ( \backslash \langle \text{customop} \rangle \backslash ) \backslash$ (operator declaration)
$\langle \text{fixity} \rangle$	$::= \text{infixl} \mid \text{infixr} \mid \text{infix}$
$\langle \text{parameter} \rangle$	$::= \langle \text{pattuple} \rangle \backslash [ ( \backslash \langle \text{pattern} \rangle ) \backslash ] \langle \text{patvalue} \rangle \backslash$
$\langle \text{patvalue} \rangle$	$::= \langle \text{identifier} \rangle$   $\_$ (wildcard pattern)   $\text{true} \mid \text{false}$ (booleans)   $\langle \text{number} \rangle$   $\text{nil}$ (empty list)   $\langle \text{char} \rangle$   $\langle \text{string} \rangle$   $\langle \text{pattuple} \rangle$   $\langle \text{patrecord} \rangle$   $[ \backslash \langle \text{pattern} \rangle_1 , \dots , \backslash \langle \text{pattern} \rangle_n ] \backslash$ (list, $n \geq 0$ )

$$\begin{aligned}
\langle \text{pattuple} \rangle & ::= ( \neg \langle \text{pattern} \rangle_1 , \neg \dots , \neg \langle \text{pattern} \rangle_n ) \neg & (n \geq 2) \\
\langle \text{patrecord} \rangle & ::= \{ \neg \langle \text{patrecordcomp} \rangle_1 , \neg \dots , \neg \langle \text{patrecordcomp} \rangle_n \} \neg & (n \geq 1) \\
& \quad | \{ \neg \langle \text{patrecordcomp} \rangle_1 , \neg \dots , \neg \langle \text{patrecordcomp} \rangle_n , \dots \} \neg & \text{(partial record)} \\
\langle \text{patrecordcomp} \rangle & ::= \langle \text{identifier} \rangle \neg : \neg \langle \text{pattern} \rangle \\
\langle \text{pattern} \rangle & ::= \langle \text{patvalue} \rangle \neg : \neg \langle \text{type} \rangle \\
& \quad | \langle \text{patvalue} \rangle \neg \{ :: \neg \langle \text{patvalue} \rangle \neg \} \\
\langle \text{typevalue} \rangle & ::= \text{Int} \\
& \quad | \text{Bool} \\
& \quad | \text{Char} \\
& \quad | \langle \text{typeident} \rangle \\
& \quad | ( \neg \langle \text{type} \rangle_1 , \neg \dots , \neg \langle \text{type} \rangle_n ) \neg & \text{(tuple, } n \geq 2) \\
& \quad | [ \neg \langle \text{type} \rangle \neg ] \neg & \text{(list)} \\
& \quad | \langle \text{typerrecord} \rangle \\
\langle \text{typerrecord} \rangle & ::= \{ \neg \langle \text{typerrecordcomp} \rangle_1 , \neg \dots , \neg \langle \text{typerrecordcomp} \rangle_n \} \neg & (n \geq 1) \\
\langle \text{typerrecordcomp} \rangle & ::= \langle \text{identifier} \rangle \neg : \neg \langle \text{type} \rangle \\
\langle \text{type} \rangle & ::= \langle \text{typevalue} \rangle \neg \{ \neg \rightarrow \langle \text{typeValue} \rangle \neg \}
\end{aligned}$$

## 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 surrounded 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" → ' 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 `[]` (empty 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 functionality for performing computations with lists.

**Ranges** Ranges allow the easy creation of lists of integers in an 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 than the last

```

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

```

The second variant 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 explicitly 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 additional 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 available 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	<code>f x</code>	Function Application	Left
9	<code>f . g</code>	Function Composition	Right
	<code>x !! y</code>	List Indexing	Left
8	<code>x * y</code>	Multiplication	Left
	<code>x / y</code>	Division	Left
	<code>x % y</code>	Remainder	Left
7	<code>x + y</code>	Addition	Left
	<code>x - y</code>	Subtraction	Left
	<code>- x</code>	Unary Negation	None
6	<code>x :: y</code>	List Construction	Right
5	<code>x @ y</code>	List Concatenation	Right
4	<code>x == y</code>	Equals	None
	<code>x != y</code>	Not Equals	None
	<code>x &gt; y</code>	Greater Than	None
	<code>x &gt;= y</code>	Greater Than Or Equal	None
	<code>x &lt; y</code>	Less Than	None
	<code>x &lt;= y</code>	Less Than Or Equal	None
3	<code>x &amp;&amp; y</code>	Logical AND	Right
2	<code>x    y</code>	Logical OR	Right
1	<code>x \$ y</code>	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 remaining parameters. This greatly reduces the readability of the code, and is therefore not encouraged.

#### 4.9.6 Defining new Operators

It 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 ( $\rightarrow$ ), 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  $\rightarrow$  param2  $\rightarrow$  ...  $\rightarrow$  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  $\rightarrow$  Int`. The second parameter is an `Int`, and the return type is also `Int`.

```
( Int  $\rightarrow$  Int )  $\rightarrow$  Int  $\rightarrow$  Int
```

If the parenthesis were omitted, 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 available. 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 `.v`, 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.v1
let double x = x * 2;

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

double 4 // Invalid
```

## 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 → a
```

Identity function.

```
const :: a → b → 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 → Int → Int
```

Integer remainder, satisfying:

$$(x \div y) * y + (\text{remainder } x \ y) = x$$

```
(%) :: Int → Int → Int | left-associative, priority 8 |
```

Infix version of remainder

```
negate :: Int → Int
```

Unary negation, satisfying:

$$x + (\text{negate } x) = 0$$

```
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** ::  $(a \rightarrow b \rightarrow c) \rightarrow b \rightarrow a \rightarrow c$

flip f takes its first two arguments in reverse order of f.  
flip f x y = f y x

**apply** ::  $(a \rightarrow b) \rightarrow a \rightarrow b$

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 in higher order situations.

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

Infix version of apply. While it has the same functionality 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** ::  $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$

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)

**(.)** ::  $(b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow a \rightarrow c$  | 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** ::  $(a, b) \rightarrow a$

Returns the first component of a pair.



**snd** ::  $(a, b) \rightarrow b$

Returns the second component of a pair.

**swap** ::  $(a, b) \rightarrow (b, a)$

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** ::  $(a \rightarrow b \rightarrow (c, d)) \rightarrow b \rightarrow c$

Returns the value of a field in a record.

`get #label record`

**set** ::  $(a \rightarrow b \rightarrow (c, d)) \rightarrow a \rightarrow b \rightarrow d$

Returns the inputted record, modifying a single field value.

`set #label value record`

**modify** ::  $(a \rightarrow b \rightarrow (c, d)) \rightarrow (c \rightarrow a) \rightarrow b \rightarrow d$

Returns the inputted 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** ::  $[a] \rightarrow a$

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

**last** ::  $[a] \rightarrow a$

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, maintaining 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

$$\begin{aligned} \text{increment} > 0 &\implies n \leq \text{finish} \\ \text{increment} < 0 &\implies n \geq \text{finish} \end{aligned}$$

### 5.2.3 Transformation Operations

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

**reverse** ::  $[a] \rightarrow [a]$

Returns the elements of the input in reverse order.

**map** ::  $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$

`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** ::  $(b \rightarrow a \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$

`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** ::  $(a \rightarrow a \rightarrow a) \rightarrow [a] \rightarrow a$

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

**all** ::  $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool}$

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

**any** ::  $(a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Bool}$

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

**maximum** ::  $\text{Orderable } a \Rightarrow [a] \rightarrow a$

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*] → [*b*] → [(*a*, *b*)]

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

**zipWith** :: (*a* → *b* → *c*) → [*a*] → [*b*] → [*c*]

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

**unzip** :: [(*a*, *b*)] → ([*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.

## 6 Changelog

### v0.2

May 1, 2017

#### Additions

- **Match Expressions**   Abstract Syntax   Operational Semantics   Type System   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 omitted, the default values are left associative and priority 9).

- **Partial Record Pattern** Abstract Syntax Operational Semantics Type System Concrete Syntax

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 language is used for parsing, and then is translated into the core language.

- **Type Aliases** Concrete Syntax

Allows the programmer to specify type aliases, simplifying type declarations

## Changes

- **List Operations** Abstract Syntax Operational Semantics Type System

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

- **Non-Strict Semantics** Operational Semantics

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

- **Boolean Operations** Abstract Syntax Operational Semantics Type System

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

- **Record Projection (Accessor)** Abstract Syntax Operational Semantics Type System

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

- **Conditionals** Abstract Syntax Operational Semantics Type System

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

## Removals

- **Input and Output**   Abstract Syntax   Operational Semantics   Type System   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   Operational Semantics   Type System   Concrete Syntax

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

- **Try**   Abstract Syntax   Operational Semantics   Type System   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   Operational Semantics   Type System   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.