FIBER: A Language with Functions, Integers, Booleans, Eagerness, and Recursion

1 INTRODUCTION

FIBER is a toy language for the KAIST CS320 course. FIBER stands for a language with **f**unctions, **i**ntegers, **b**ooleans, **e**agerness[1], and **r**ecursion. As the name implies, it features integers, booleans, first-class functions, and recursive functions. In addition, it is an eager language. More precisely, FIBER supports the following features:

- integers and booleans
- basic arithmetic operators, including negation, addition, subtraction, multiplication, division, and modulo
- basic relational operators, including equal-to, not-equal-to, less-than, less-than-or-equal-to, greater-then, and greater-then-or-equal-to.
- basic boolean operators, including negation, conjunction, and disjunction
- conditional expressions (if-else expressions)
- tuples of arbitrary lengths greater than one
- projections for tuples
- lists, which are cons or nil
- primitives for lists: isEmpty, nonEmpty, head, and tail
- immutable local variables
- immutable local variable binding via pattern matching on tuples
- first-class functions and function application
- anonymous functions
- mutually recursive functions
- dynamic type tests

This document is the specification of FIBER. First, it gives the syntax of FIBER: Section 2 describes the concrete syntax; Secion 3 formalizes the desugaring rules; Section 4 shows the abstract syntax. Second, it defines the semantics of FIBER in a natural langauge in Section 5. The formal big-step operational semantics of FIBER can be found in A.

2 CONCRETE SYNTAX

The concrete syntax of FIBER is written in the extended Backus–Naur form. To improve the readability, we use different colors for different kinds of objects. Syntactic elements of the extended Backus–Naur form, rather than FIBER, are written in purple. For example, we use `=`, `|`, and `;`. Note that `{ }` denotes a repetition of zero or more times, and `[ ]` denotes an optional existence. Nonterminals are written in blue. For example, `expr` is a nonterminal denoting expressions. Any other objects written in black are terminals. For instance, `"true"` and `"false"` are terminals representing boolean literals.

The following is the concrete syntax of FIBER:

```
ltr  = "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L"
     | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X"
     | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j"
```

[1] *Eagerness* denotes the most usual function application semantics, that the arguments of a function application are evaluated before the function body is evaluated.

```
     | "k"  | "l"  | "m"  | "n"  | "o"  | "p"  | "q"  | "r"  | "s"  | "t"  | "u"  | "v"
     | "w"  | "x"  | "y"  | "z" ;
pdgt = "1"  | "2"  | "3"  | "4"  | "5"  | "6"  | "7"  | "8"  | "9" ;
dgt  = "0"  | pdgt ;

sch  = ltr | "_" ;
ch   = sch | dgt ;
id   = sch {ch} ;

idx  = pdgt {dgt} ;
num  = ["-"] dgt {dgt} ;

expr = id | num | "true" | "false" | "-" expr | "!" expr
     | expr "+"  expr | expr "-"  expr | expr "*"  expr | expr "/"  expr
     | expr "%"  expr | expr "==" expr | expr "!=" expr | expr "<"  expr
     | expr "<=" expr | expr ">"  expr | expr ">=" expr | expr "&&" expr
     | expr "||" expr | "if" "(" expr ")" expr "else" expr
     | "(" expr "," expr {"," expr} ")" | expr "." "_" idx
     | "Nil" | expr "::" expr | expr "." "isEmpty"
     | expr "." "nonEmpty" | expr "." "head" | expr "." "tail"
     | "val" id "=" expr ";" expr
     | "val" "(" id "," id {"," id} ")" "=" expr ";" expr
     | "(" ")" "=>" expr | id "=>" expr | "(" id {"," id} ")" "=>" expr
     | fdef {fdef} expr
     | expr "(" ")" | expr "(" expr {"," expr} ")"
     | expr "." "isInstanceOf" "[" type "]"
     | "(" expr ")"| "{" expr "}" ;

fdef = "def" id "(" ")" "=" expr ";"
     | "def" id "(" id {"," id} ")" "=" expr ";" ;

type = "Int" | "Boolean" | "Tuple" | "List" | "Function" ;
```

Note that whitespaces, such as `' '`, `'\t'`, and `'\n'`, are omitted from the above specification. You can insert any kinds of whitespaces between any two terminals to make a valid nonterminal, except `id` and `num`. For example, since we have `expr = num | "-" expr`, if one parses `-1` and `- 1`, then both will succeed, and the results will be the same. On the other hand, because you cannot insert whitespaces at the middle of terminals, `tr ue` cannot be parsed while `true` can be parsed correctly.

The concrete syntax of Fiber is *ambiguous*. It means that a single string can be parsed in multiple ways. For example, `1 + 2 * 3` can result in both Tree a1 and Tree a2 in Figure 1.

To resolve the ambiguity of the concrete syntax, we define *precedence* between binary operators. If op_1 precedes op_2, then e_1 op_1 e_2 op_2 e_3 can result in only Tree b1. On the other hand, if op_2 precedes op_1, Tree b2 is the only possible result.

Figure 2 shows precedence. One appearing earlier in the table precedes one appearing later. For example, since `*` precedes `+`, `1 + 2 * 3` is parsed to only Tree a2. Operators in the same box of the table have the same precedence. If they appear in a single expression, then one appearing first in

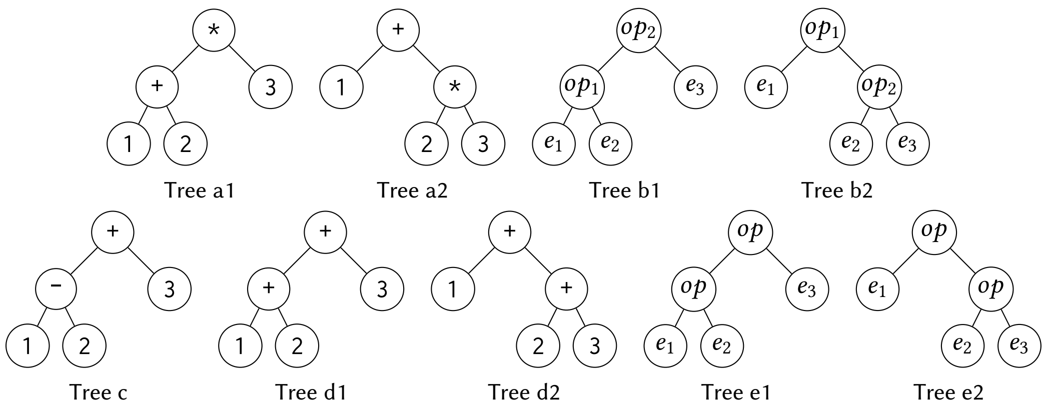

Fig. 1. Parse Trees

`*` `/` `%`	`+` `-`	`==` `!=` `<` `<=` `>` `>=`	`&&`	`||`	`::`

← higher lower

Fig. 2. Operator Precedence

the expression has the higher precedence in the expression. For instance, `1 - 2 + 3` results in Tree c because `-` and `+` have the same precedence, but `-` appears first in the expression.

Alas, precedence is not enough to resolve the ambiguity. We have problems when an operator appears more than once in an expression. For example, `1 + 2 + 3` can result in both Tree d1 and Tree d2.

We introduce *associativity* of binary operators to solve the problem. A binary operator can be either left-associative or right-associative. If op_1 is left-associative, then e_1 op e_2 op e_3 can result in only Tree e1. On the other hand, if op is right-associative, Tree e2 is the only possible result. In Fiber, all the binary operators except `::` are left-associative. Only `::` is right-associative. Thus, `1 + 2 + 3` is parsed to only Tree d1.

3 DESUGARING

To simplify the implementation of the interpreting phase, the parsing phase of the interpreter desugars a given expression. Desugaring rewrites some subexpressions with other expressions. Due to desugaring, the abstract syntax of Fiber consists of less sorts of expressions than the concrete syntax.

Figure 3 defines desugaring of Fiber expressions. Let e and x respectively denote an expression and an identifier. An expression e is desugared to $[\![e]\!]$. For example, after desugaring, `-(1 + 2)` becomes `(1 + 2) * -1`. Note that lines under identifiers imply that the identifiers must be fresh, i.e. have different names from existing ones. For instance, if the entire program is `1 <= 2`, then below is a valid desugaring result.

$$
\begin{aligned}
\llbracket \mathtt{-}e \rrbracket &= \llbracket e \rrbracket \ \mathtt{*\ -1} \\
\llbracket \mathtt{!}e \rrbracket &= \mathtt{if}\ \mathtt{(}\llbracket e \rrbracket\mathtt{)}\ \mathtt{false\ else\ true} \\
\llbracket e_1\ \mathtt{-}\ e_2 \rrbracket &= \llbracket e_1 \rrbracket\ \mathtt{+}\ \llbracket \mathtt{-}e_2 \rrbracket \\
\llbracket e_1\ \mathtt{!=}\ e_2 \rrbracket &= \llbracket \mathtt{!(}\llbracket e_1 \rrbracket\ \mathtt{==}\ \llbracket e_2 \rrbracket\mathtt{)} \rrbracket \\
\llbracket e_1\ \mathtt{<=}\ e_2 \rrbracket &= \mathtt{val}\ \underline{x_1}\ \mathtt{=}\ \llbracket e_1 \rrbracket\mathtt{;} \\
&\ \mathtt{val}\ \underline{x_2}\ \mathtt{=}\ \llbracket e_2 \rrbracket\mathtt{;} \\
&\ \llbracket x_1\ \mathtt{==}\ x_2\ \mathtt{||}\ x_1\ \mathtt{<}\ x_2 \rrbracket \\
\llbracket e_1\ \mathtt{>}\ e_2 \rrbracket &= \llbracket \mathtt{!(}\llbracket e_1 \rrbracket\ \mathtt{<=}\ \llbracket e_2 \rrbracket\mathtt{)} \rrbracket \\
\llbracket e_1\ \mathtt{>=}\ e_2 \rrbracket &= \llbracket \mathtt{!(}\llbracket e_1 \rrbracket\ \mathtt{<}\ \llbracket e_2 \rrbracket\mathtt{)} \rrbracket
\end{aligned}
$$

$$
\begin{aligned}
\llbracket e_1\ \mathtt{\&\&}\ e_2 \rrbracket &= \mathtt{if}\ \mathtt{(}\llbracket e_1 \rrbracket\mathtt{)}\ \llbracket e_2 \rrbracket\ \mathtt{else\ false} \\
\llbracket e_1\ \mathtt{||}\ e_2 \rrbracket &= \mathtt{if}\ \mathtt{(}\llbracket e_1 \rrbracket\mathtt{)}\ \mathtt{true\ else}\ \llbracket e_2 \rrbracket \\
\llbracket e\mathtt{.nonEmpty} \rrbracket &= \llbracket \mathtt{!(}\llbracket e \rrbracket\mathtt{.isEmpty)} \rrbracket \\
\llbracket \mathtt{val}\ \mathtt{(}x_1\mathtt{,}\cdots\mathtt{,}x_i\mathtt{)}\ \mathtt{=}\ e_1\mathtt{;}\ e_2 \rrbracket &= \mathtt{val}\ \underline{x}\ \mathtt{=}\ \llbracket e_1 \rrbracket\mathtt{;} \\
&\ \mathtt{val}\ x_1\ \mathtt{=}\ x\mathtt{.\_1;} \\
&\ \cdots \\
&\ \mathtt{val}\ x_i\ \mathtt{=}\ x\mathtt{.\_}i\mathtt{;} \\
&\ \llbracket e_2 \rrbracket \\
\llbracket \mathtt{(}e\mathtt{)} \rrbracket &= \llbracket e \rrbracket \\
\llbracket \mathtt{\{}e\mathtt{\}} \rrbracket &= \llbracket e \rrbracket
\end{aligned}
$$

Any other cases recursively desugar their subexpressions.

Fig. 3. Desugaring Rules

```
val x = 1;
val y = 2;
x == y || x < y
```

Since the desugaring rules are defined recursively, they can handle complex programs correctly. For example, `-(1 + 2)` becomes `(1 + 2) * -1 * -1` instead of `-(1 + 2) * -1` after desugaring.

4 ABSTRACT SYNTAX

Figure 4 describes the abstract syntax of Fiber. Metavariable x ranges over identifiers; i ranges over indices of tuples, which are positive integers; n ranges over integers; b ranges over boolean literals, which are either true or false; e ranges over expressions; d ranges over recursive function definitions; τ ranges over types, which are either Int, Boolean, Tuple, List, or Function.

The following briefly describes expressions:

- $e_1 + e_2$, $e_1 \times e_2$, $e_1 \div e_2$, e_1 mod e_2, $e_1 = e_2$, and $e_1 < e_2$ are binary operations on integers.
- if e_1 e_2 e_3 is a conditional expression.
- $(e_1, \cdots, e_i)$ creates a tuple of length i. Length i must be greater than one.
- $e.i$ is a projection from a tuple. The beginning index is one.
- Nil creates the empty list.
- $e_1 :: e_2$ creates a nonempty list.
- e.isEmpty, e.head, and e.tail are unary operations on a list.
- val x=e_1 in e_2 defines a local variable whose name is x and scope is e_2.
- $\lambda x_1 \cdots x_i.e$ defines an anonymous function whose parameters are $x_1, \cdots, x_i$ and body is e. The names of the parameters must be distinct from each other.
- def $x(x_1, \cdots, x_i)$ $=e$ defines a (possibly recursive) function whose name is x, parameters are $x_1, \cdots, x_i$, and body is e. The names of the parameters must be distinct from each other.
- $d_1 \cdots d_i$ e defines functions from d_1 to d_i. The names of the functions must be distinct from each other. They can be mutually recursive and used in e.
- $e(e_1, \cdots, e_i)$ is a function application. e is a function; $e_1, \cdots, e_i$ are arguments.
- e is τ tests the type of a given value.

$$
\begin{array}{rcll}
\text{Identifier} & x & \in & \mathit{Id} \\
\text{Index} & i & \in & \mathbb{Z}^{+} \\
\text{Number} & n & \in & \mathbb{Z} \\
\text{Boolean} & b & ::= & \mathsf{true} \\
 & & \mid & \mathsf{false} \\
\text{Function} & d & ::= & \mathsf{def}\ x(x,\cdots,x) = e \\
\text{Types} & \tau & ::= & \mathsf{Int} \\
 & & \mid & \mathsf{Boolean} \\
 & & \mid & \mathsf{Tuple} \\
 & & \mid & \mathsf{List} \\
 & & \mid & \mathsf{Function}
\end{array}
$$

$$
\begin{array}{rclll}
\text{Expression} & e & ::= & x & \text{(variable)} \\
 & & \mid & n & \text{(integer)} \\
 & & \mid & b & \text{(boolean)} \\
 & & \mid & e + e & \text{(addition)} \\
 & & \mid & e \times e & \text{(multiplication)} \\
 & & \mid & e \div e & \text{(division)} \\
 & & \mid & e\ \mathsf{mod}\ e & \text{(modulo)} \\
 & & \mid & e = e & \text{(equal-to)} \\
 & & \mid & e < e & \text{(less-than)} \\
 & & \mid & \mathsf{if}\ e\ e\ e & \text{(conditional)} \\
 & & \mid & (e,\cdots,e) & \text{(tuple; length > 1)} \\
 & & \mid & e.i & \text{(projection)} \\
 & & \mid & \mathsf{Nil} & \text{(nil)} \\
 & & \mid & e :: e & \text{(cons)} \\
 & & \mid & e.\mathsf{isEmpty} & \text{(is-empty)} \\
 & & \mid & e.\mathsf{head} & \text{(head)} \\
 & & \mid & e.\mathsf{tail} & \text{(tail)} \\
 & & \mid & \mathsf{val}\ x{=}e\ \mathsf{in}\ e & \text{(local variable)} \\
 & & \mid & \lambda x \cdots x.e & \text{(anonymous function)} \\
 & & \mid & d \cdots d\ e & \text{(recursive function)} \\
 & & \mid & e(e,\cdots,e) & \text{(function application)} \\
 & & \mid & e\ \mathsf{is}\ \tau & \text{(type test)}
\end{array}
$$

Fig. 4. Abstract Syntax

5 SEMANTICS

This section explains the semantics of FIBER in a natural language. See Appendix A to find the formal big-step operational semantics.

To explain the semantics, we need the definition of a value. A value is one of the following:

- an integer
- a boolean
- a tuple whose length is greater than one and elements are values
- the empty list
- a nonempty list, which consists of a value and a (empty or nonempty) list
- a closure, which is a function with an environment

In this section, we use the following metavariables and terminologies:

- Metavariable v ranges over values.
- Metavariable σ ranges over environments, which are maps from identifiers to values.
- If we say "the result is v" while explaining evaluation of e, then v is the result of e.
- We use the word "must" to represent requirements. If a requirement is violated, then a run-time error occurs. Any occurrence of a run-time error immediately terminates the execution.

The following explains how each expression is evaluated.

Case x:

(1) Let σ be the current environment.
(2) x must be in the domain of σ.
(3) The result is $\sigma(x)$.

Case n:

(1) The result is n.

Case b:

(1) The result is b.

Case $e_1 \oplus e_2$:

(*) Suppose that $\oplus \in \{+, \times, \div, \mathsf{mod}, =, <\}$.
(1) Evaluate e_1.
(2) Let v_1 be the result of e_1.
(3) Evaluate e_2.
(4) Let v_2 be the result of e_2.
(5) (v_1, v_2) must be in the domain of $\oplus$. Note that $+, \times \in (\mathbb{Z}, \mathbb{Z}) \to \mathbb{Z}$, $\div, \mathsf{mod} \in (\mathbb{Z}, \mathbb{Z} \setminus \{0\}) \to \mathbb{Z}$, and $=, < \in (\mathbb{Z}, \mathbb{Z}) \to \{\mathsf{true}, \mathsf{false}\}$.
(6) The result is $v_1 \oplus v_2$.

Case $\mathsf{if}\ e_1\ e_2\ e_3$:

(1) Evaluate e_1.
(2) Let v_1 be the result of e_1.
(3) v_1 must be a boolean.
(4) If v_1 is true, then
  (a) Evaluate e_2.
  (b) Let v_2 be the result of e_2.
  (c) The result is v_2.
(5) If v_1 is false, then
  (a) Evaluate e_3.
  (b) Let v_3 be the result of e_3.
  (c) The result is v_3.

Case $(e_1, \cdots, e_i)$:

(1) Evaluate e_1.
(2) Let v_1 be the result of e_1.
(3) Evaluate e_{k+1} in the same manner after evaluating e_k.
(4) Repeat (3) until e_i is evaluated.
(5) The result is a tuple consisting of the values from v_1 to v_i.

Case $e.i$:

(1) Evaluate e.
(2) Let v be the result of e.
(3) v must be a tuple whose length is greater than or equal to i.
(4) The result is the ith element of v. Note that the beginning index is one.

Case Nil:

(1) The result is the empty list.

Case $e_1 :: e_2$:

(1) Evaluate e_1.
(2) Let v_1 be the result of e_1.
(3) Evaluate e_2.
(4) Let v_2 be the result of e_2.
(5) v_2 must be either the empty list or a nonempty list.
(6) The result is a nonempty list whose head is v_1 and tail is v_2.

Case $e.\mathsf{isEmpty}$:

(1) Evaluate e.
(2) Let v be the result of e.
(3) v must be either the empty list or a nonempty list.
(4) If v is the empty list, then
  (a) The result is true.
(5) Else if v is a nonempty list, then
  (a) The result is false.

Case $e.\mathsf{head}$:

(1) Evaluate e.
(2) Let v be the result of e.
(3) v must be a nonempty list.
(4) The result is the head of v.

Case $e.\mathsf{tail}$:

(1) Evaluate e.
(2) Let v be the result of e.
(3) v must be a nonempty list.
(4) The result is the tail of v.

Case $\mathsf{val}\ x{=}e_1\ \mathsf{in}\ e_2$:

(1) Evaluate e_1.
(2) Let v_1 be the result of e_1.
(3) Add a mapping from x to v_1 to the current environment.
(4) Let σ_{new} be the new environment.
(5) Evaluate e_2 under σ_{new}.
(6) Let v_2 be the result of e_2.
(7) The result is v_2.

Case $\lambda x_1 \cdots x_i.e$:

(1) Let σ be the current environment.
(2) The result is a closure whose parameters are from x_1 to x_i, body is e, and environment is σ.

Case $d_1 \cdots d_i\ e$:

(1) Let $x_1, \cdots, x_i$ be the names of $d_1, \cdots, d_i$.
(2) Let $v_1, \cdots, v_i$ be the closures of $d_1, \cdots, d_i$. If d_j equals $\mathsf{def}\ x_j(x_{j1}, \cdots, x_{jk}) {=} e_j$, then v_j consists of the parameters $x_{j1}, \cdots, x_{jk}$ and the body e_j.

(3) Add a mapping from x's to v's to the current environment.
(4) Let σ_{new} be the new environment.
(5) The environment of every v_k needs to be σ_{new}.
(6) Evaluate e under σ_{new}.
(7) The result is v.

Case $e(e_1, \cdots, e_i)$:

(1) Evaluate e.
(2) Let v be the result of e.
(3) Evaluate e_1.
(4) Let v_1 be the result of e_1.
(5) Evaluate e_{k+1} in the same manner after evaluating e_k.
(6) Repeat (6) until e_i is evaluated.
(7) v must be a closure.
(8) The number of parameters must equal the number of arguments.
(9) Let $x_1, \cdots, x_i$ be the names of the parameters of v.
(10) Let e_c be the body of v.
(11) Let σ_c be the environment of v.
(12) Add a mapping from x's to v's to σ_c.
(13) Let σ_{new} be the new environment.
(14) Evaluate e_c under σ_{new}.
(15) Let v_c be the result of e_c.
(16) The result is v_c.

Case e is τ:

(∗) The type of a value is as the following:
- The type of an integer is Int.
- The type of a boolean is Boolean.
- The type of a tuple is Tuple.
- The type of the empty list is List.
- The type of a nonempty list is List.
- The type of a closure is Function.

(1) Evaluate e.
(2) Let v be the result of e.
(3) If the type of v is τ, then
  (a) The result is true.
(4) If the type of v is not τ, then
  (a) The result is false.

The division and modulo operations are defined as the following, which is the same as the semantics of many real-world languages:

- If $n_1 \geq 0$ and $n_2 > 0$, then $n_1 \div n_2$ is the quotient when n_1 is divided by n_2.
- If $n_1 \geq 0$ and $n_2 < 0$, then the $n_1 \div n_2$ is the negation of the quotient when n_1 is divided by $-n_2$.
- If $n_1 < 0$ and $n_2 > 0$, then the $n_1 \div n_2$ is the negation of the quotient when $-n_1$ is divided by n_2.
- If $n_1 < 0$ and $n_2 < 0$, then the $n_1 \div n_2$ is the quotient when $-n_1$ is divided by $-n_2$.
- If $n_1 \geq 0$, then the $n_1 \bmod n_2$ is the remainder when n_1 is divided by $|n_2|$.
- If $n_1 < 0$, then the $n_1 \bmod n_2$ is the negation of the remainder when $-n_1$ is divided by $|n_2|$.

5.1 Interpreter Specification

An interpreter of FIBER must satisfy the following conditions:

- It provides a function named `interp` that takes a FIBER expression as an argument and returns a FIBER value.
- If v is the result of e under the empty environment, then `interp(`e`)` equals v.
- If the evaluation of e terminates due to a run-time error, then `interp(`e`)` terminates by calling the `error` function. Each error message can be any string.
- If the evaluation of e runs forever under the empty environment, then `interp(`e`)` runs forever or terminates due to stack overflow.

A reference interpreter of FIBER is available at https://plrg.kaist.ac.kr/fiber.

A BIG-STEP OPERATIONAL SEMANTICS

$$
\begin{array}{rrcl}
\text{Value} & v & \in & \mathbb{V} \\
 & v & ::= & n \mid b \mid (v,\cdots,v) \mid \mathsf{Nil} \mid v :: v \mid \langle \lambda x \cdots x.e, \sigma \rangle \\
\text{Environment} & \sigma & \in & Id \xrightarrow{\text{fin}} \mathbb{V}
\end{array}
$$

Fig. 5. Values and Environments

$$
\boxed{type(v) = \tau}
\qquad
\begin{array}{rcl}
type(n) & = & \mathsf{Int} \\
type((v_1,\cdots,v_i)) & = & \mathsf{Tuple} \\
type(v_1 :: v_2) & = & \mathsf{List}
\end{array}
\qquad
\begin{array}{rcl}
type(b) & = & \mathsf{Boolean} \\
type(\mathsf{Nil}) & = & \mathsf{List} \\
type(\langle \lambda x_1 \cdots x_i.e, \sigma \rangle) & = & \mathsf{Function}
\end{array}
$$

Fig. 6. Types of Values

$$
\boxed{\sigma \vdash e \Rightarrow v}
\qquad
\frac{x \in Domain(\sigma)}{\sigma \vdash x \Rightarrow \sigma(x)}
\qquad
\sigma \vdash n \Rightarrow n
\qquad
\sigma \vdash b \Rightarrow b
$$

$$
\frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2}
\qquad
\frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 \times e_2 \Rightarrow n_1 \times n_2}
$$

$$
\frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2 \quad n_2 \neq 0}{\sigma \vdash e_1 \div e_2 \Rightarrow n_1 \div n_2}
\qquad
\frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2 \quad n_2 \neq 0}{\sigma \vdash e_1 \ \mathsf{mod}\ e_2 \Rightarrow n_1 \ \mathsf{mod}\ n_2}
$$

$$
\frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 = e_2 \Rightarrow n_1 = n_2}
\qquad
\frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 < e_2 \Rightarrow n_1 < n_2}
$$

$$
\frac{\sigma \vdash e_1 \Rightarrow \mathsf{true} \quad \sigma \vdash e_2 \Rightarrow v}{\sigma \vdash \mathsf{if}\ e_1\ e_2\ e_3 \Rightarrow v}
\qquad
\frac{\sigma \vdash e_1 \Rightarrow \mathsf{false} \quad \sigma \vdash e_3 \Rightarrow v}{\sigma \vdash \mathsf{if}\ e_1\ e_2\ e_3 \Rightarrow v}
$$

$$
\frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \cdots \quad \sigma \vdash e_i \Rightarrow v_i}{\sigma \vdash (e_1,\cdots,e_i) \Rightarrow (v_1,\cdots,v_i)}
\qquad
\frac{\sigma \vdash e \Rightarrow (v_1,\cdots,v_i,\cdots,v_{i'})}{\sigma \vdash e.i \Rightarrow v_i}
$$

$$
\sigma \vdash \mathsf{Nil} \Rightarrow \mathsf{Nil}
\qquad
\frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma \vdash e_2 \Rightarrow v_2 \quad type(v_2) = \mathsf{List}}{\sigma \vdash e_1 :: e_2 \Rightarrow v_1 :: v_2}
$$

Fig. 7. Evaluation of Expressions (1/2)

$$\frac{\sigma \vdash e \Rightarrow \mathsf{Nil}}{\sigma \vdash e.\mathsf{isEmpty} \Rightarrow \mathsf{true}} \qquad \frac{\sigma \vdash e \Rightarrow v_1 :: v_2}{\sigma \vdash e.\mathsf{isEmpty} \Rightarrow \mathsf{false}}$$

$$\frac{\sigma \vdash e \Rightarrow v_1 :: v_2}{\sigma \vdash e.\mathsf{head} \Rightarrow v_1} \qquad \frac{\sigma \vdash e \Rightarrow v_1 :: v_2}{\sigma \vdash e.\mathsf{tail} \Rightarrow v_2}$$

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \quad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \mathsf{val}\ x{=}e_1\ \mathsf{in}\ e_2 \Rightarrow v_2} \qquad \sigma \vdash \lambda x_1 \cdots x_i.e \Rightarrow \langle \lambda x_1 \cdots x_i.e, \sigma \rangle$$

$$\frac{\begin{array}{c} d_1 = \mathsf{def}\ x_1(x_{11}, \cdots, x_{1i_1}) = e_1 \quad \cdots \quad d_i = \mathsf{def}\ x_i(x_{i1}, \cdots, x_{ii_i}) = e_i \\ v_1 = \langle \lambda x_{11} \cdots x_{1i_1}.e_1, \sigma' \rangle \quad \cdots \quad v_i = \langle \lambda x_{i1} \cdots x_{ii_i}.e_i, \sigma' \rangle \\ \sigma' = \sigma[x_1 \mapsto v_1, \cdots, x_i \mapsto v_i] \quad \sigma' \vdash e \Rightarrow v \end{array}}{\sigma \vdash d_1 \cdots d_i\ e \Rightarrow v}$$

$$\frac{\begin{array}{c} \sigma \vdash e \Rightarrow \langle \lambda x_1 \cdots x_i.e', \sigma' \rangle \quad \sigma \vdash e_1 \Rightarrow v_1 \quad \cdots \quad \sigma \vdash e_i \Rightarrow v_i \\ \sigma'[x_1 \mapsto v_1, \cdots, x_i \mapsto v_i] \vdash e' \Rightarrow v \end{array}}{\sigma \vdash e(e_1, \cdots, e_i) \Rightarrow v}$$

$$\frac{\sigma \vdash e \Rightarrow v}{\sigma \vdash e\ \mathsf{is}\ \tau \Rightarrow \mathit{type}(v) = \tau}$$

Fig. 8. Evaluation of Expressions (2/2)