

# X-FIBER: A Language with Exceptions, Functions, Integers, Booleans, Eagerness, and Recursion

## 1 INTRODUCTION

X-FIBER is a toy language for the CS320 course. X-FIBER stands for a language with **e**xceptions, **f**unctions, **i**ntegers, **b**ooleans, **e**agerness, and **r**ecursion. As the name implies, it extends FIBER. Like FIBER, it is an eager language and features integers, booleans, first-class functions, and recursive functions. In addition, it provides first-class continuations, exceptions, and exception handlers. More precisely, X-FIBER supports the following features (the bold parts are the features not in FIBER):

- 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-than, and greater-than-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
- **first-class continuations**
- **return expressions**
- **exceptions and exception handlers**

This document defines X-FIBER and provides a guide to the project. First, it gives the syntax of X-FIBER: Section 2 describes the concrete syntax; Section 3 formalizes the desugaring rules; Section 4 shows the abstract syntax. Second, it describes the semantics of X-FIBER in Section 5. Appendix A shows the small-step semantics of X-FIBER.

## 2 CONCRETE SYNTAX

The concrete syntax of X-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 X-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 X-FIBER (the parts in boxes are the cases not in 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"
      | "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
      | "vcc" id ";" expr
      | "(" ")" "=>" expr | id "=>" expr | "(" id {"," id} ")" "=>" expr
      | fdef {fdef} expr
      | expr "(" ")" | expr "(" expr {"," expr} ")"
      | expr "." "isInstanceOf" "[" type "]"
      | "return" expr
      | "throw" expr
      | "try" expr "catch" expr
      | "(" 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 program. 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 X-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.

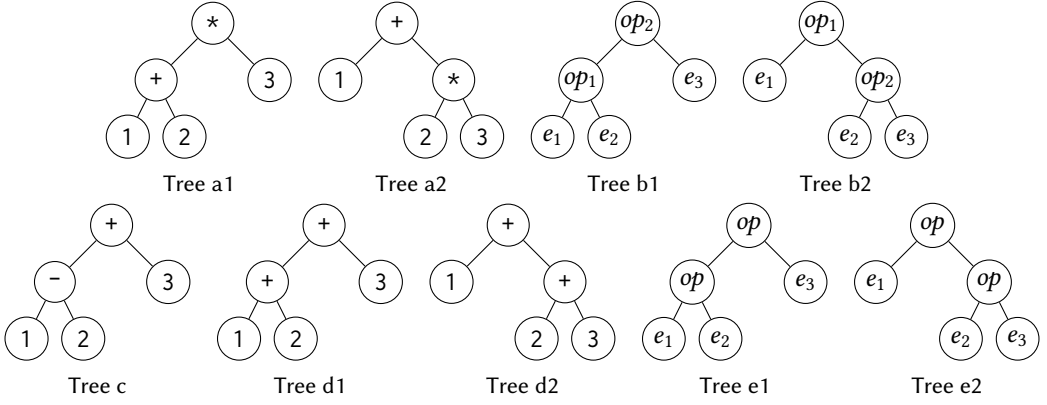


Fig. 1. Parse Trees

		==			
		!=			
*	+	<	&&		::
/	-	<=			
%		>			
		>=			
← higher			lower		

Fig. 2. Operator Precedence

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 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 X-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.

$\llbracket -e \rrbracket = \llbracket e \rrbracket * -1$ $\llbracket !e \rrbracket = \text{if } (\llbracket e \rrbracket) \text{ false else true}$ $\llbracket e_1 - e_2 \rrbracket = \llbracket e_1 \rrbracket + \llbracket -e_2 \rrbracket$ $\llbracket e_1 != e_2 \rrbracket = \llbracket !(\llbracket e_1 \rrbracket == \llbracket e_2 \rrbracket) \rrbracket$ $\llbracket e_1 <= e_2 \rrbracket = \text{val } \underline{x_1} = \llbracket e_1 \rrbracket;$ $\text{val } \underline{x_2} = \llbracket e_2 \rrbracket;$ $\llbracket [x_1 == x_2 \mid \mid x_1 < x_2] \rrbracket$ $\llbracket e_1 > e_2 \rrbracket = \llbracket !(\llbracket e_1 \rrbracket <= \llbracket e_2 \rrbracket) \rrbracket$ $\llbracket e_1 >= e_2 \rrbracket = \llbracket !(\llbracket e_1 \rrbracket < \llbracket e_2 \rrbracket) \rrbracket$ $\llbracket e_1 \&\& e_2 \rrbracket = \text{if } (\llbracket e_1 \rrbracket) \llbracket e_2 \rrbracket \text{ else false}$ $\llbracket e_1 \mid\mid e_2 \rrbracket = \text{if } (\llbracket e_1 \rrbracket) \text{ true else } \llbracket e_2 \rrbracket$ $\llbracket e.\text{nonEmpty} \rrbracket = \llbracket !(\llbracket e \rrbracket.\text{isEmpty}) \rrbracket$ <div style="border: 1px solid black; padding: 5px; display: inline-block;"> <math display="block">\llbracket \text{return } e \rrbracket = \text{return}(\llbracket e \rrbracket)</math> </div> $\llbracket (e) \rrbracket = \llbracket e \rrbracket$ $\llbracket \{e\} \rrbracket = \llbracket e \rrbracket$	<div style="border: 1px solid black; padding: 5px; display: inline-block;"> <math display="block">\llbracket (x_1, \dots, x_i) =&gt; e \rrbracket =</math> <math display="block">(x_1, \dots, x_i) =&gt;</math> <math display="block">\text{vcc return; } \llbracket e \rrbracket</math> </div> <div style="border: 1px solid black; padding: 5px; display: inline-block; margin-top: 10px;"> <math display="block">\llbracket \text{def } x(x_1, \dots, x_i) = e; \rrbracket =</math> <math display="block">\text{def } x(x_1, \dots, x_i) =</math> <math display="block">\text{vcc return; } \llbracket e \rrbracket;</math> </div> $\llbracket \text{val } (x_1, \dots, x_i) = e_1; e_2 \rrbracket =$ $\text{val } \underline{x} = \llbracket e_1 \rrbracket;$ $\text{val } x_1 = x._1;$ $\dots$ $\text{val } x_i = x._i;$ $\llbracket e_2 \rrbracket$
---	---

Any other cases recursively desugar their subexpressions.

Fig. 3. Desugaring Rules

Due to desugaring, the abstract syntax of X-FIBER consists of less sorts of expressions than the concrete syntax.

Figure 3 defines desugaring of X-FIBER expressions (the parts in boxes are the rules not in FIBER). Let  $e$  and  $x$  respectively denote an expression and an identifier. An expression  $e$  is desugared to  $\llbracket e \rrbracket$ .

#### 4 ABSTRACT SYNTAX

Figure 4 describes the abstract syntax of X-FIBER (the parts in boxes are the cases not in 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;  $\tau$  ranges over types, which are either Int, Boolean, Tuple, List, or Function.

The following briefly describes expressions:

- $e_1 + e_2$ ,  $e_1 \times e_2$ ,  $e_1 \div e_2$ ,  $e_1 \bmod e_2$ ,  $e_1 = e_2$ , and  $e_1 < e_2$  are binary operations on integers.
- if  $e_1 \ e_2 \ e_3$  is a conditional expression.
- $(e_1, \dots, e_i)$  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.
- Cons  $e_1 \ e_2$  creates a nonempty list.
- $e.\text{isEmpty}$ ,  $e.\text{head}$ , and  $e.\text{tail}$  are unary operations on a list.
- $\text{val } x = e_1 \text{ in } e_2$  defines a local variable whose name is  $x$  and scope is  $e_2$ .
- $\text{vcc } x \text{ in } e$  defines a local variable whose name is  $x$  and scope is  $e$ . The current continuation is bound to  $x$ .
- $\lambda x_1 \dots x_i. e$  defines an anonymous function whose parameters are  $x_1, \dots, x_i$  and body is  $e$ . The names of the parameters must be distinct from each other.
- $\text{def } x(x_1, \dots, x_i) = e$  defines a (possibly recursive) function whose name is  $x$ , parameters are  $x_1, \dots, x_i$ , and body is  $e$ . The names of the parameters must be distinct from each other.

Identifier $x \in Id$ Index $i \in \mathbb{Z}^+$ Number $n \in \mathbb{Z}$ Boolean $b ::= \text{true}$   $\text{false}$ Function $d ::= \text{def } x(x, \dots, x) = e$ Types $\tau ::= \text{Int}$   $\text{Boolean}$   $\text{Tuple}$   $\text{List}$   $\text{Function}$	Expression $e ::=$	$x$	(variable)
		$  n$	(integer)
		$  b$	(boolean)
		$  e + e$	(addition)
		$  e \times e$	(multiplication)
		$  e \div e$	(division)
		$  e \bmod e$	(modulo)
		$  e = e$	(equal-to)
		$  e < e$	(less-than)
		$  \text{if } e \ e \ e$	(conditional)
		$  (e, \dots, e)$	(tuple; length > 1)
		$  e.i$	(projection)
		$  \text{Nil}$	(nil)
		$  \text{Cons } e \ e$	(cons)
		$  e.\text{isEmpty}$	(is-empty)
		$  e.\text{head}$	(head)
		$  e.\text{tail}$	(tail)
		$  \text{val } x=e \text{ in } e$	(local variable)
		$  \text{vcc } x \text{ in } e$	(continuation)
		$  \lambda x \dots x. e$	(anonymous function)
		$  d \dots d \ e$	(recursive function)
		$  e(e, \dots, e)$	(function application)
		$  e \text{ is } \tau$	(type test)
		$  \text{throw } e$	(exception)
		$  \text{try } e \text{ catch } e$	(exception handler)

Fig. 4. Abstract Syntax

- $d_1 \dots d_i \ e$  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  $e$ .
- $e(e_1, \dots, e_i)$  is a function application.  $e$  is a function;  $e_1, \dots, e_i$  are arguments.
- $e \text{ is } \tau$  tests the type of a given value.
- $\text{throw } e$  throws an exception.
- $\text{try } e_1 \text{ catch } e_2$  registers an exception handler.

## 5 SEMANTICS

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

- a continuation, which denotes the remaining computation at some point of execution

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

- Metavariable  $v$  ranges over values.
- Metavariable  $\sigma$  ranges over environments, which are maps from identifiers to values.
- Metavariable  $\kappa$  ranges over continuations. As continuations are values, we can treat  $\kappa$  as a value.
- Metavariable  $H$  ranges over possibly-registered exception handlers. If there is no handler,  $H$  does not have any information, and we write  $H = \cdot$ . If there is a handler, then  $H$  is a tuple of an expression, an environment, a continuation, and an exception handler. Then, we write  $H = \langle e, \sigma, \kappa, H' \rangle$ .
- If we say “the result is  $v$ ” while explaining evaluation of  $e$ , then  $e$  results in  $v$ .
- We use the word “must” to represent requirements. If a requirement is violated, then a run-time error occurs. A run-time error differs from an exception. Any occurrence of a run-time error immediately terminates the execution.
- The evaluation begins with the empty environment, and without an exception handler.

The following explains how each expression is evaluated.

**Case  $x$ :**

- (1) Let  $\sigma$  be the current environment.
- (2)  $x$  must be in the domain of  $\sigma$ .
- (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  $\oplus \in \{+, \times, \div, \text{mod}, =, <\}$ .
- (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  $+, \times \in (\mathbb{Z}, \mathbb{Z}) \rightarrow \mathbb{Z}$ ,  $\div, \text{mod} \in (\mathbb{Z}, \mathbb{Z} \setminus \{0\}) \rightarrow \mathbb{Z}$ , and  $=, < \in (\mathbb{Z}, \mathbb{Z}) \rightarrow \{\text{true}, \text{false}\}$ .
- (6) The result is  $v_1 \oplus v_2$ .

**Case 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  $(e_1, \dots, e_i)$ :**

- (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  $i$ th element of  $v$ . Note that the beginning index is one.

**Case Nil:**

- (1) The result is the empty list.

**Case Cons  $e_1$   $e_2$ :**

- (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  $e.\text{isEmpty}$ :**

- (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  $e.\text{head}$ :**

- (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  $e.\text{tail}$ :**

- (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  $\text{val } x = e_1 \text{ in } e_2$ :**

- (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  $\sigma_{\text{new}}$  be the new environment.
- (5) Evaluate  $e_2$  under  $\sigma_{\text{new}}$ .
- (6) Let  $v_2$  be the result of  $e_2$ .
- (7) The result is  $v_2$ .

**Case  $\text{vcc } x \text{ in } e$ :**

- (1) Let  $\kappa$  be the current continuation.
- (2) Add a mapping from  $x$  to  $\kappa$  to the current environment.
- (3) Let  $\sigma_{\text{new}}$  be the new environment.
- (4) Evaluate  $e$  under  $\sigma_{\text{new}}$ .
- (5) Let  $v$  be the result of  $e$ .
- (6) The result is  $v$ .

**Case  $\lambda x_1 \dots x_i.e$ :**

- (1) Let  $\sigma$  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  $\sigma$ .

**Case  $d_1 \dots d_i e$ :**

- (1) Let  $x_1, \dots, x_i$  be the names of  $d_1, \dots, d_i$ .
- (2) Let  $v_1, \dots, v_i$  be the closures of  $d_1, \dots, d_i$ .  
If  $d_j$  equals  $\text{def } x_j(x_{j1}, \dots, x_{jk}) = e_j$ , then

$v_j$  consists of the parameters  $x_{j1}, \dots, x_{jk}$  and the body  $e_j$ .

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

**Case  $e(e_1, \dots, e_i)$ :**

- (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 either a closure or a continuation.
- (8) If  $v$  is a closure, then
  - (a) The number of parameters must equal the number of arguments.
  - (b) Let  $x_1, \dots, x_i$  be the names of the parameters of  $v$ .
  - (c) Let  $e_c$  be the body of  $v$ .
  - (d) Let  $\sigma_c$  be the environment of  $v$ .
  - (e) Add a mapping from  $x$ 's to  $v$ 's to  $\sigma_c$ .
  - (f) Let  $\sigma_{\text{new}}$  be the new environment.
  - (g) Evaluate  $e_c$  under  $\sigma_{\text{new}}$ .
  - (h) Let  $v_c$  be the result of  $e_c$ .
  - (i) The result is  $v_c$ .
- (9) Else if  $v$  is a continuation, then
  - (a) There must be exactly one argument.
  - (b) Use  $v$  as the current continuation.
  - (c) The result is  $v_1$ .

**Case  $e \text{ is } \tau$ :**

- (\*) 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`.
  - The type of a continuation is `Function`.
- (1) Evaluate  $e$ .
- (2) Let  $v$  be the result of  $e$ .
- (3) If the type of  $v$  is  $\tau$ , then

- (a) The result is true.
- (4) If the type of  $v$  is not  $\tau$ , then
  - (a) The result is false.

**Case** throw  $e$ :

- (1) Evaluate  $e$ .
- (2) Let  $v$  be the result of  $e$ .
- (3) There must be an exception handler.
- (4) Let  $\langle e_h, \sigma_h, \kappa_h, H_h \rangle$  be the exception handler.
- (5) Use  $\kappa_h$  as the current continuation.
- (6) Use  $H_h$  as the current exception handler.
- (7) Evaluate  $e_h$  under  $\sigma_h$ .
- (8) Let  $v_h$  be the result of  $e_h$ .
- (9)  $v_h$  must be either a closure or a continuation.
- (10) If  $v_h$  is a closure, then
  - (a)  $v_h$  must have exactly one parameter.
  - (b) Let  $x$  be the name of the parameter of  $v_h$ .

- (c) Let  $e_c$  be the body of  $v_h$ .
- (d) Let  $\sigma_c$  be the environment of  $v_h$ .
- (e) Add a mapping from  $x$  to  $v$  to  $\sigma_c$ .
- (f) Let  $\sigma_{new}$  be the new environment.
- (g) Evaluate  $e_c$  under  $\sigma_{new}$ .
- (h) Let  $v_c$  be the result of  $e_c$ .
- (i) The result is  $v_c$ .
- (11) Else if  $v_h$  is a continuation, then
  - (a) Use  $v_h$  as the current continuation.
  - (b) The result is  $v$ .

**Case** try  $e_1$  catch  $e_2$ :

- (1) Let  $\sigma$  be the current environment.
- (2) Let  $\kappa$  be the current continuation.
- (3) Let  $H$  be the current exception handler.
- (4) Let  $H_{new}$  be  $\langle e_2, \sigma, \kappa, H \rangle$ .
- (5) Use  $H_{new}$  as the current exception handler.
- (6) Evaluate  $e_1$ .
- (7) Let  $v$  be the result of  $e_1$ .
- (8) The result is  $v$ .

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 X-FIBER must satisfy the following conditions:

- It provides a function named `interp` that takes an X-FIBER expression as an argument and returns an X-FIBER value.
- If  $v$  is the result of  $e$ , then `interp( $e$ )` equals  $v$ .
- If the evaluation of  $e$  runs forever, then `interp( $e$ )` runs forever or terminates due to stack overflow.
- If the evaluation of  $e$  terminates due to a run-time error, then `interp( $e$ )` terminates by calling the error function. Error messages are implementation-dependent.

A reference interpreter of X-FIBER is available at <https://plrg.kaist.ac.kr/x-fiber>.



## A SMALL-STEP SEMANTICS

Value	$v \in \mathbb{V}$
	$v ::= n \mid b \mid (v, \dots, v) \mid \text{Nil} \mid \text{Cons } v \ v \mid \langle \lambda x \dots x.e, \sigma \rangle \mid \langle k \parallel s \rangle$
Environment	$\sigma \in \text{Id} \xrightarrow{\text{fin}} \mathbb{V}$
Handler	$H ::= \cdot \mid \langle e, \sigma, k \parallel s, H \rangle$
Continuation	$k ::= \square \mid \sigma, H \vdash e :: k \mid (+) :: k \mid (\times) :: k \mid (\div) :: k \mid (\text{mod}) :: k$ $\mid (=) :: k \mid (<) :: k \mid (\sigma, H, e, e) :: k \mid ((i)) :: k \mid (.i) :: k$ $\mid (\text{Cons}) :: k \mid (\text{isEmpty}) :: k \mid (\text{head}) :: k \mid (\text{tail}) :: k$ $\mid (x, \sigma, H, e) :: k \mid (@i) :: k \mid ([\tau]) :: k \mid (H) :: k \mid (\leftrightarrow) :: k$
Stack	$s ::= \blacksquare \mid v :: s$

Fig. 5. Definitions for Semantics

$$\boxed{\text{type}(v) = \tau}$$

$$\begin{array}{ll}
 \text{type}(n) = \text{Int} & \text{type}(\text{Nil}) = \text{List} \\
 \text{type}(b) = \text{Boolean} & \text{type}(\text{Cons } v_1 \ v_2) = \text{List} \\
 \text{type}((v_1, \dots, v_i)) = \text{Tuple} & \text{type}(\langle \lambda x_1 \dots x_i.e, \sigma \rangle) = \text{Function} \\
 & \text{type}(\langle k \parallel s \rangle) = \text{Function}
 \end{array}$$

Fig. 6. Types of Values

$$\begin{array}{c}
 \boxed{k \parallel s \rightarrow k \parallel s} \\
 \hline
 x \in \text{Domain}(\sigma) \\
 \hline
 \sigma, H \vdash x :: k \parallel s \rightarrow k \parallel \sigma(x) :: s \\
 \sigma, H \vdash n :: k \parallel s \rightarrow k \parallel n :: s \\
 \sigma, H \vdash b :: k \parallel s \rightarrow k \parallel b :: s \\
 \sigma, H \vdash e_1 + e_2 :: k \parallel s \rightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (+) :: k \parallel s \\
 (+) :: k \parallel n_2 :: n_1 :: s \rightarrow k \parallel n_1 + n_2 :: s \\
 \sigma, H \vdash e_1 \times e_2 :: k \parallel s \rightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (\times) :: k \parallel s \\
 (\times) :: k \parallel n_2 :: n_1 :: s \rightarrow k \parallel n_1 \times n_2 :: s \\
 \sigma, H \vdash e_1 \div e_2 :: k \parallel s \rightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (\div) :: k \parallel s \\
 (\div) :: k \parallel n_2 :: n_1 :: s \rightarrow k \parallel n_1 \div n_2 :: s \\
 \hline
 n_2 \neq 0 \\
 \hline
 (\div) :: k \parallel n_2 :: n_1 :: s \rightarrow k \parallel n_1 \div n_2 :: s
 \end{array}$$

Fig. 7. Evaluation of Expressions (1/3)

$$\begin{array}{l}
\sigma, H \vdash e_1 \text{ mod } e_2 :: k \parallel s \rightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (\text{mod}) :: k \parallel s \\
\\
\frac{n_2 \neq 0}{(\text{mod}) :: k \parallel n_2 :: n_1 :: s \rightarrow k \parallel n_1 \text{ mod } n_2 :: s} \\
\sigma, H \vdash e_1 = e_2 :: k \parallel s \rightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (=) :: k \parallel s \\
(=) :: k \parallel n_2 :: n_1 :: s \rightarrow k \parallel n_1 = n_2 :: s \\
\sigma, H \vdash e_1 < e_2 :: k \parallel s \rightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (<) :: k \parallel s \\
(<) :: k \parallel n_2 :: n_1 :: s \rightarrow k \parallel n_1 < n_2 :: s \\
\sigma, H \vdash \text{if } e_1 \ e_2 \ e_3 :: k \parallel s \rightarrow \sigma, H \vdash e_1 :: (\sigma, H, e_2, e_3) :: k \parallel s \\
(\sigma, H, e_1, e_2) :: k \parallel \text{true} :: s \rightarrow \sigma, H \vdash e_1 :: k \parallel s \\
(\sigma, H, e_1, e_2) :: k \parallel \text{false} :: s \rightarrow \sigma, H \vdash e_2 :: k \parallel s \\
\sigma, H \vdash (e_1, \dots, e_i) :: k \parallel s \rightarrow \sigma, H \vdash e_1 :: \dots :: \sigma, H \vdash e_i :: ((i)) :: k \parallel s \\
((i)) :: k \parallel v_i :: \dots :: v_1 :: s \rightarrow k \parallel (v_1, \dots, v_i) :: s \\
\sigma, H \vdash e.i :: k \parallel s \rightarrow \sigma, H \vdash e :: (.i) :: k \parallel s \\
(.i) :: k \parallel (v_1, \dots, v_i, \dots, v_j) :: s \rightarrow k \parallel v_i :: s \\
\sigma, H \vdash \text{Nil} :: k \parallel s \rightarrow k \parallel \text{Nil} :: s \\
\sigma, H \vdash \text{Cons } e_1 \ e_2 :: k \parallel s \rightarrow \sigma, H \vdash e_1 :: \sigma, H \vdash e_2 :: (\text{Cons}) :: k \parallel s \\
\\
\frac{\text{type}(v_2) = \text{List}}{(\text{Cons}) :: k \parallel v_2 :: v_1 :: s \rightarrow k \parallel \text{Cons } v_1 \ v_2 :: s} \\
\sigma, H \vdash e.\text{isEmpty} :: k \parallel s \rightarrow \sigma, H \vdash e :: (\text{isEmpty}) :: k \parallel s \\
(\text{isEmpty}) :: k \parallel \text{Nil} :: s \rightarrow k \parallel \text{true} :: s \\
(\text{isEmpty}) :: k \parallel \text{Cons } v_1 \ v_2 :: s \rightarrow k \parallel \text{false} :: s \\
\sigma, H \vdash e.\text{head} :: k \parallel s \rightarrow \sigma, H \vdash e :: (\text{head}) :: k \parallel s \\
(\text{head}) :: k \parallel \text{Cons } v_1 \ v_2 :: s \rightarrow k \parallel v_1 :: s \\
\sigma, H \vdash e.\text{tail} :: k \parallel s \rightarrow \sigma, H \vdash e :: (\text{tail}) :: k \parallel s \\
(\text{tail}) :: k \parallel \text{Cons } v_1 \ v_2 :: s \rightarrow k \parallel v_2 :: s
\end{array}$$

Fig. 8. Evaluation of Expressions (2/3)

$$\begin{array}{c}
\sigma, H \vdash \text{val } x=e_1 \text{ in } e_2 :: k \parallel s \rightarrow \sigma, H \vdash e_1 :: (x, \sigma, H, e_2) :: k \parallel s \\
(x, \sigma, H, e) :: k \parallel v :: s \rightarrow \sigma[x \mapsto v], H \vdash e :: k \parallel s \\
\sigma, H \vdash \text{vcc } x \text{ in } e :: k \parallel s \rightarrow \sigma[x \mapsto \langle k \parallel s \rangle], H \vdash e :: k \parallel s \\
\sigma, H \vdash \lambda x_1 \cdots x_i. e :: k \parallel s \rightarrow k \parallel \langle \lambda x_1 \cdots x_i. e, \sigma \rangle :: s \\
\frac{d_1 = \text{def } x_1(x_{11}, \dots, x_{1j_1}) = e_1 \quad \cdots \quad d_i = \text{def } x_i(x_{i1}, \dots, x_{ij_i}) = e_i \quad v_1 = \langle \lambda x_{11} \cdots x_{1j_1}. e_1, \sigma' \rangle \quad \cdots \quad v_i = \langle \lambda x_{i1} \cdots x_{ij_i}. e_i, \sigma' \rangle \quad \sigma' = \sigma[x_1 \mapsto v_1, \dots, x_i \mapsto v_i]}{\sigma, H \vdash d_1 \cdots d_i e :: k \parallel s \rightarrow \sigma', H \vdash e :: k \parallel s} \\
\sigma, H \vdash e(e_1, \dots, e_i) :: k \parallel s \rightarrow \sigma, H \vdash e :: \sigma, H \vdash e_1 :: \cdots :: \sigma, H \vdash e_i :: (@i) :: k \parallel s \\
(@i) :: k \parallel v_i :: \cdots :: v_1 :: \langle \lambda x_1 \cdots x_i. e, \sigma \rangle :: s \rightarrow \sigma[x_1 \mapsto v_1, \dots, x_i \mapsto v_i], H \vdash e :: k \parallel s \\
(@1) :: k \parallel v :: \langle k' \parallel s' \rangle :: s \rightarrow k' \parallel v :: s' \\
\sigma, H \vdash e \text{ is } \tau :: k \parallel s \rightarrow \sigma, H \vdash e :: ([\tau]) :: k \parallel s \\
([\tau]) :: k \parallel v :: s \rightarrow k \parallel \text{type}(v) = \tau :: s \\
\sigma, H \vdash \text{throw } e :: k \parallel s \rightarrow \sigma, H \vdash e :: (H) :: k \parallel s \\
\langle \langle e, \sigma, k' \parallel s', H \rangle \rangle :: k \parallel v :: s \rightarrow \sigma, H \vdash e :: (\leftrightarrow) :: (@1) :: k' \parallel v :: s' \\
\sigma, H \vdash \text{try } e_1 \text{ catch } e_2 :: k \parallel s \rightarrow \sigma, \langle e_2, \sigma, k \parallel s, H \rangle \vdash e_1 :: k \parallel s \\
(\leftrightarrow) :: k \parallel v_2 :: v_1 :: s \rightarrow k \parallel v_1 :: v_2 :: s
\end{array}$$

Fig. 9. Evaluation of Expressions (3/3)

$$\frac{k \parallel s \rightarrow^* k \parallel s \quad k_1 \parallel s_1 \rightarrow^* k_2 \parallel s_2 \quad k_2 \parallel s_2 \rightarrow k_3 \parallel s_3}{k_1 \parallel s_1 \rightarrow^* k_3 \parallel s_3}$$

Fig. 10. Reflexive Transitive Closure

### A.1 Interpreter Specification

An interpreter of X-FIBER must satisfy the following conditions:

- (1) It provides a function named `interp` that takes an X-FIBER expression as an argument and returns an X-FIBER value.
- (2) If  $\emptyset, \cdot \vdash e :: \square \parallel \blacksquare \rightarrow^* \square \parallel v :: \blacksquare$ , then `interp(e)` equals to  $v$ .
- (3) If  $\forall (k, s) \in \{(k, s) : \emptyset, \cdot \vdash e :: \square \parallel \blacksquare \rightarrow^* k \parallel s\}. \exists k'. \exists s'. k \parallel s \rightarrow k' \parallel s'$ , then `interp(e)` runs forever or terminates due to stack overflow.
- (4) If  $\nexists v. \emptyset, \cdot \vdash e :: \square \parallel \blacksquare \rightarrow^* \square \parallel v :: \blacksquare$  and (3) is not the case, then `interp(e)` terminates by calling the error function. Error messages are implementation-dependent.