Fabric: A Functional Language with Algebraic Data Types

1 INTRODUCTION

Fabric is a **f**unctional language with **a**lge**br**a**ic** data types. As the name implies, it features first-class functions and algebraic data types. More precisely, Fabric supports the following features:

- integers, booleans, strings, and the unit value
- 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
- expression sequencing
- conditional expressions (if and if-else expressions)
- immutable local variables
- mutually recursive immutable lazy local variables
- mutable local variables and assignments
- first-class functions and function application
- anonymous functions
- mutually recursive functions
- polymorphic first-order functions
- polymorphic algebraic data types and pattern matching
- static type checking

This document defines Fabric. First, it gives the syntax of Fabric: Section 2 describes the concrete syntax; Section 3 formalizes the desugaring rules; Section 4 shows the abstract syntax. Second, it describes type checking of Fabric in Section 5, type erasure of Fabric in Section 6, and the dynamic semantics of Fabric in Section 7. Third, it briefly introduces the standard library of Fabric in Section 8. Fourth, it defines the specification of a type checker and an interpreter of Fabric. In addition, Appendix A shows the typing rules of Fabric, and Appendix B shows the big-step operational semantics of Fabric.

2 CONCRETE SYNTAX

The concrete syntax of Fabric 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 Fabric, 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 Fabric:

```
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} ;
tnm  = id ;
tvar = "'" id ;

num  = ["-"] dgt {dgt} ;
str  = """ [ch] """ "<" expr "," expr ">" ;

expr = id ["[" type {"," type} "]"]
     | num | "true" | "false" | "(" ")" | str | "-" 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 | "{" expr {";" expr} "}"
     | "if" "(" expr ")" expr ["else" expr]
     | "val" id [":" type] "=" expr ";" expr
     | "var" id [":" type] "=" expr ";" expr
     | rdef {rdef} expr
     | "(" ")" "=>" expr | "(" id ":" type {"," id ":" type} ")" "=>" expr
     | id "=" expr
     | expr "(" ")" | expr "(" expr {"," expr} ")"
     | expr "match" "{" case {case} "}"
     | "(" expr ")" ;

vrnt = "case" id | "case" id "(" type {"," type} ")"
case = "case" id "=>" expr | "case" id "(" id {"," id} ")" "=>" expr
rdef = "lazy" "val" id ":" type "=" expr ";"
     | "def" id ["[" tvar {"," tvar} "]"] "(" ")" ":" type "=" expr ";"
     | "def" id ["[" tvar {"," tvar} "]"] "(" id ":"type {"," id ":" type} ")"
       ":" type "=" expr ";"
     | "type" tnm ["[" tvar {"," tvar} "]"] "{" vrnt {vrnt} "}" ;

type = tnm ["[" type {"," type} "]"] | tvar | "Int" | "Boolean" | "Unit"
     | "(" ")" "=>" type | type "=>" type | "("type {"," type} ")" "=>" type
     | "(" type ")" ;
```

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 Fabric 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.

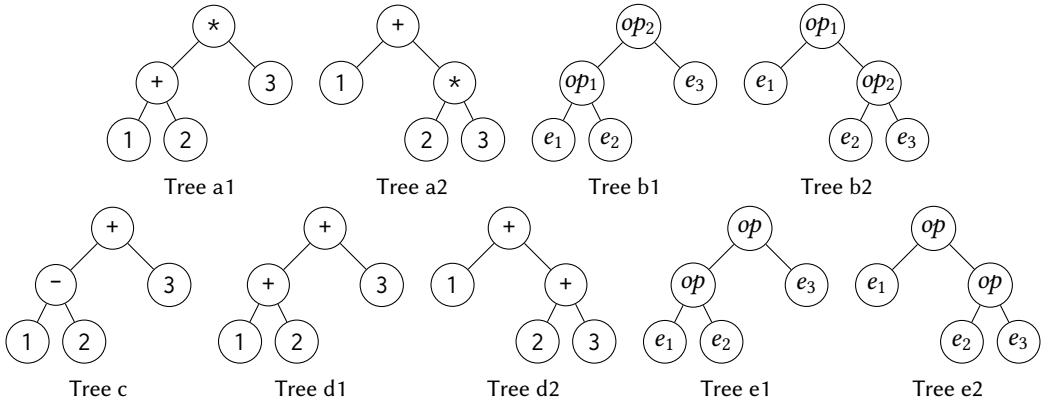

Fig. 1. Parse Trees

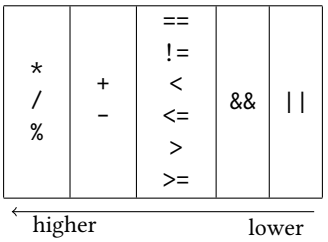

Fig. 2. Operator Precedence

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 Fabric, all the binary operators except `::` are left-associative. Only `::` is right-associative. Thus, `1 + 2 + 3` is parsed to only Tree d1.

We provide some example programs following the concrete syntax to help you understand the concrete syntax. At the same time, the examples give you a quick overview of Fabric.

Expressions separated by semicolons inside a pair of curly braces are sequenced. Note that expression sequencing thus always needs cruly braces, and semicolons are not an operator. It acts

like an operator only inside curly braces. Sequencing is left-associative, like all the other operators. The following is a typical usage of sequencing and results in `2` if both `f` and `g` terminate normally.

```
{
  f();
  g();
  2
}
```

Strings are not literals in Fabric. Instead, they are considered as finite foldable collections of integers. Thus, a string requires a value corresponding to the empty string and a function to folds the sequence. The following program results in `294`, which is the sum of the ascii codes of `'a'`, `'b'`, and `'c'`. It may seem weird at the first sight, but if it is used with user-defined lists, then it creates a perfect string, which is a list of integers. For more details, refer to Section 3.

```
"abc"<(x: Int, y: Int) => x + y, 0>
```

Fabric allows programmers to omit `else` branches in conditional expressions. When one `else` branch is absent, then it is equal to write `()`, which denotes the unit value, in the `else` branch. For example, the following programs are equivalent for some function `f` whose return type is `Unit`:

```
if (1 > 2)
  f()
else
  ()
```

```
if (1 > 2)
  f()
```

If an `else` branch is omitted in a nested conditional expressions, the existing `else` branch is combined with the innermost `then` branch. For instance, the following programs are equivalent for some functions `f` and `g` whose return types are `Unit`:

```
if (1 < 2)
  if (1 > 2)
    f()
else
  g()
```

```
if (1 < 2) {
  if (1 > 2)
    f()
  else
    g()
}
```

Both of the following programs pass type checking and have the same run-time behavior. The only difference is whether the type annotation of `x` exists or not. Note that type annotations for variables defined by `val` are optional.

```
val x: Int = 1;
x
```

```
val x = 1;
x
```

Variables defined by `var` are similar to those defined by `val` except that they are mutable. Their values can change by assignments. Type annotations for them also are optional.

```
var x: Int = 1;
x = 2
```

```
var x = 1;
x = 2
```

Unlike `val` and `var`, variables defined by `lazy val` always require type annotations. In addition, they can be mutually recursive if they are defined consecutively. Therefore, the following program passes type checking and results in `120` at run time:

```
lazy val fac: Int => Int = (x: Int) => if (x <= 1) 1 else x * fac(x - 1);
fac(5)
```

Just like `lazy val`, functions defined by `def` always require parameter type annotations and return type annotations. They can be mutually recursive as well. Thus, you may notice that `lazy`

`val` and `def` are quite the same except that `lazy val` leads to caching, while `def` does not. You will see their semantics in Section 7. The following program defines the factorial function like before but uses `def`:

```
def fac(x: Int): Int = if (x <= 1) 1 else x * fac(x - 1);
fac(5)
```

Since Fabric does not support type inference, every anonymous function requires type annotations for all the parameters of itself. Regardless of the number of parameters, parameters always must appear in a pair of parentheses. The following are example programs using anonymous functions and function applications:

```
((x: Int) => x + x)(2)
```

```
((x: Int, y: Int) => x * y)(3, 4)
```

The keyword `type` allows defining custom types. Every custom type in Fabric is an algebraic data type. Types defined together can be mutually recursive, like `lazy val` and `def`. The following is a typical program defining lists of integers:

```
type IntList {
  case IntNil
  case IntCons(Int, IntList)
}
IntCons(0, IntCons(1, IntCons(2, IntNil)))
```

Pattern matching handles values of custom types. The following defines a function computing the length of a given list.

```
def len(l: IntList): Int = l match {
  case IntNil => 0
  case IntCons(h, t) => 1 + len(t)
};
len(IntCons(0, IntCons(1, IntCons(2, IntNil))))
```

Different kinds of recursive definitions can be mutually recursive. In Fabric, three kinds of recursive definitions exist: lazy variables, functions, and custom types. For example, consecutively defined functions and lazy variables can be mutually recursive. The following defines functions checking the parity of a given natural number. It works even though `odd` is a lazy variable and `even` is a function.

```
lazy val odd: Int => Boolean = (x: Int) => if (x == 0) false else even(x - 1);
def even(x: Int): Boolean = if (x == 0) true else odd(x - 1);
even(5)
```

Parametric polymorphism is the most interesting and powerful feature in Fabric. Even though Fabric supports only a restrictive form of parametric polymorphism, it is still quite powerful. In Fabric, values cannot be polymorphic, while type definitions and non-anonymous function definitions can be polymorphic. For the ease of implementation, type variables are syntactically distinguished from the names of custom types: type variables always start with `'` (single quotation). This syntax is influenced by Standard ML. The following shows a polymorphic identity function:

```
def f['T](x: 'T): 'T = x
f[Int](1)
```

The following defines polymorphic lists in a typical way:

```
type List['T] {
  case Nil
```

$$
\begin{aligned}
[\![\texttt{-}e]\!] &= [\![e]\!] \texttt{ * -1} \\
[\![\texttt{!}e]\!] &= \texttt{if (}[\![e]\!]\texttt{) false else true} \\
[\![e_1 \texttt{ - } e_2]\!] &= [\![e_1]\!] \texttt{ + } [\![\texttt{-}e_2]\!] \\
[\![e_1 \texttt{ != } e_2]\!] &= [\![\texttt{!(}[\![e_1]\!] \texttt{ == } [\![e_2]\!]\texttt{)}]\!] \\
[\![e_1 \texttt{ <= } e_2]\!] &= \texttt{val } \underline{x_1} \texttt{ = } [\![e_1]\!]\texttt{;} \\
&\quad\ \texttt{val } \underline{x_2} \texttt{ = } [\![e_2]\!]\texttt{;} \\
&\quad\ [\![x_1 \texttt{ == } x_2 \texttt{ || } x_1 \texttt{ < } x_2]\!]
\end{aligned}
\qquad
\begin{aligned}
[\![e_1 \texttt{ > } e_2]\!] &= [\![\texttt{!(}[\![e_1]\!] \texttt{ <= } [\![e_2]\!]\texttt{)}]\!] \\
[\![e_1 \texttt{ >= } e_2]\!] &= [\![\texttt{!(}[\![e_1]\!] \texttt{ < } [\![e_2]\!]\texttt{)}]\!] \\
[\![e_1 \texttt{ \&\& } e_2]\!] &= \texttt{if (}[\![e_1]\!]\texttt{) } [\![e_2]\!] \texttt{ else false} \\
[\![e_1 \texttt{ || } e_2]\!] &= \texttt{if (}[\![e_1]\!]\texttt{) true else } [\![e_2]\!] \\
[\![\texttt{if (}e_1\texttt{) } e_2]\!] &= \texttt{if (}[\![e_1]\!]\texttt{) } [\![e_2]\!] \texttt{ else ()} \\
[\![\texttt{"}n_1 \cdots n_i\texttt{"<}e_1\texttt{, } e_2\texttt{>}]\!] &= \texttt{val } \underline{x} \texttt{ = } [\![e_1]\!]\texttt{;} \\
&\quad\ x\texttt{(}n_1 \cdots\texttt{, } x\texttt{(}n_i\texttt{, } [\![e_2]\!]\texttt{)} \cdots\texttt{)}
\end{aligned}
$$

Any other cases recursively desugar their subexpressions.

Fig. 3. Desugaring Rules

```
  case Cons('T, List['T])
}
Cons[Int](0, Cons[Int](1, Cons[Int](2, Nil[Int])))
```

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 Fabric consists of less sorts of expressions than the concrete syntax.

Figure 3 defines desugaring of Fabric expressions. Let e and x respectively denote an expression and an identifier. An expression e is desugared to $[\![e]\!]$.

4 ABSTRACT SYNTAX

We use an overline above mathematical material to indicate an appropriately punctuated sequence of zero or more repetitions of the material, with a subscript added to each metavariable in the material. For example, $t[\overline{\tau}]$ equals $t[\tau_1, \ldots, \tau_n]$ for some natural number n. Note that n can be zero, which allows $t[\overline{\tau}]$ to be the same as $t[\,]$.

We use a box around mathematical material to indicate an optional existence. For instance, $\mu\ x\boxed{:\tau}\texttt{=}e$ in e can be either $\mu\ x\texttt{=}e$ in e or $\mu\ x\texttt{:}\tau\texttt{=}e$ in e since the existence of $:\tau$ is optional.

Figure 4 describes the abstract syntax of Fabric. Metavariable x ranges over identifiers; n ranges over integers; b ranges over boolean literals, which are either true or false; t ranges over the names of custom types; α ranges over type variables, which appear when polymorphic functions and methods are defined; τ ranges over types; μ ranges over mutability tags for strict local variables, which are either var or val; w ranges over variant definitions; c ranges over cases in pattern matching; d ranges over recursive definitions, which are either lazy variables, functions, or custom types; e ranges over expressions.

The following briefly describes types:

- $t[\overline{\tau}]$ is a first-order type application. t is the name of a custom type; τ's are type arguments. If type arguments do not exist, then t is a monomorphic type. For example, `Fruit[ ]` can be a type where `Fruit` is the name of a monomorphic type. If there are one or more type arguments, then t is a polymorphic type. For instance, `List[Int]` can be a type where `List` is the name of a polymorphic type with one type parameter. `Int` is the only type argument in this case.
- α is a type variable.

Identifier	x	$\in$	Id	Expression	e ::=	$x[\overline{\tau}]$ (type application)
Number	n	$\in$	$\mathbb{Z}$		|	n (integer)
Boolean	b	::=	true		|	b (boolean)
		|	false		|	( ) (unit)
Type Name	t	$\in$	TN		|	$e + e$ (addition)
Type Variable	α	$\in$	TV		|	$e \times e$ (multiplication)
Type	τ	::=	$t[\overline{\tau}]$		|	$e \div e$ (division)
		|	α		|	e mod e (modulo)
		|	Int		|	$e = e$ (equal-to)
		|	Boolean		|	$e < e$ (less-than)
		|	Unit		|	e ; e (sequence)
		|	$\overline{\tau} \rightarrow \tau$		|	if e e e (conditional)
Mutability	μ	::=	var | val		|	μ x:τ=e in e (local variable)
Variant	w	::=	$x(\overline{\tau})$		|	$\overline{d}$ in e (recursive definitions)
Case	c	::=	$x(\overline{x}) \rightarrow e$		|	$\lambda \overline{x:\tau}.e$ (anonymous function)
Recursive Definition	d	::=	lazy x:τ=e		|	$x := e$ (assignment)
		|	def $x[\overline{\alpha}](\overline{x:\tau}):\tau=e$		|	$e(\overline{e})$ (function application)
		|	type $t[\overline{\alpha}]$ $\overline{w}$		|	e match $\overline{c}$ (pattern matching)

Fig. 4. Abstract Syntax

- Int is an integer type.
- Boolean is a boolean type.
- Unit is a unit type.
- $\overline{\tau_p} \rightarrow \tau_r$ is a function type. τ_p's are the types of the parameters; τ_r is the return type. For example, Int→Boolean is the type of a function from an integer to a boolean value.

The following briefly describes expressions:

- $x[\overline{\tau}]$ is an identifier instantiated with type arguments. x is the name of a variable, a parameter, a function, or a variant; τ's are type arguments. If type arguments do not exist, then x is a monomophic entity. For example, x[ ] can denote a variable whose name is x. If there are one or more type arguments, then x is a polymorphic entity. For instance, when there is a polymorphic function whose name is f, f[Int] denotes the function instantiated with Int. Note that all variables and parameters are monomorphic, while functions and variants can be either monomorphic or polymorphic.
- n is an integral literal; b is a boolean literal; ( ) is the unit value.
- $e_1 + e_2$, $e_1 \times e_2$, $e_1 \div e_2$, e_1 mod e_2, $e_1 = e_2$, and $e_1 < e_2$ are binary operations on integers.
- e_1 ; e_2 is a sequence of expressions. e_1 is evaluated before e_2 is evaluated.
- if e_1 e_2 e_3 is a conditional expression. e_1 is the condition; e_2 is the true branch; e_3 is the false branch.
- μ x=e_1 in e_2 is a local variable definition without a type annotation. It defines a local variable whose name is x and scope is e_2. e_1 determines the value of x. μ is either val or var. If μ is val, then x is an immutable variable. Otherwise, it is a mutable variable. μ x:τ=e in e is a local varaible definition with a type annotation for x. τ is the type of x.
- $\overline{d}$ in e defines recursive definitions. Each d is either a lazy variable definition, a function definition, or a type definition. Variables, functions, and types defined by d's can be used in

e. Identifiers introduced by d's must have distinct names; type identifiers introduced by d's must have distinct names.
  - lazy $x{:}\tau{=}e$ is a lazy variable definition. x is the name of the variable; τ is the type of the variable; e determines the value of the variable.
  - def $x[\overline{\alpha}](\overline{x_p{:}\tau_p}){:}\tau_r{=}e$ is a function definition. x is the name of the function; α's are the type parameters of the function; x_p's are the parameters of the function; τ_p's are type annotations for the parameters; τ_r is the return type of the function; e is the body of the function. α's must be distinct type variables; x_p's must have distinct names.
  - type $t[\overline{\alpha}]\ \overline{w}$ is a type definition. t is the name of the type; α's are the type parameters of the type; w's are the variants of the type. α's must be distinct type variables; all the variants must have distinct names. There must be one or more variants. Each variant definition w has the syntax of $x(\overline{\tau})$. x is the name of the variant; τ's are the types of the parameters of the variant.
- $\lambda\overline{x{:}\tau}.e$ is an anonymous function. x's are the parameters of the function; τ's are type annotations for the parameters; e is the body of the function. x's must have distinct names.
- $x := e$ updates the value of a variable denoted by x. e determines the new value of x.
- $e(\overline{e_a})$ is a function application. e is a function to be applied to arguments; e_a's are arguments for the function application.
- e match $\overline{c}$ is a pattern matching expression on e. c's are the cases of the pattern matching. All the cases must have distinct variant names. There must be one or more cases. Each case c has the syntax of $x(\overline{x_p}) \rightarrow e_c$. x is the name of a variant; values carried by a variant value can be bound to x_p's. e_c determines the result of the pattern matching. The scope of each x_p is e_c.

5 TYPE CHECKING

This section explains type checking of Fabric in a natural language. See Appendix A to find the formal static semantics of Fabric.

First, well-formedness of types has to be defined. We use the following metavariable and terminologies:

- Metavariable Γ ranges over type environments. A type environment contains two kinds of mappings:
  - A mapping from an identifier to a pair of a type scheme and a mutability tag. The following is the definition of a type scheme:

    $$\text{Type Scheme} \quad T \quad ::= \forall\overline{\alpha}.\tau$$

    A type scheme is a type followed by a list of zero or more type variables. The type of a type scheme is parametrized by the type variables of the type scheme. We use type schemes to represent both monomorphic and polymorhpic types. A type scheme representing a monomorphic type τ is $\forall.\tau$. A type scheme representing a polymorphic type parametrized by a single type variable is $\forall\alpha.\tau$. Note that type schemes exist only inside type environments. When a polymorphic type is used in an expression, the type has to be instantiated with a proper number of type arguments. Therefore, the type of every expression is a type, which is always monomorphic, not a type scheme.
  - A mapping from a type name to a type definition, whose syntax is type $t[\overline{\alpha}]\ \overline{w}$. A type definition type $t[\overline{\alpha}]\ \overline{w}$ creates a mapping from t to type $t[\overline{\alpha}]\ \overline{w}$.

  In addition, a type environment contains type variables.
- Substituting α with τ' in τ produces a type that is the same as τ except that every occurrence of α has been replaced with τ'. For example, substituting α with Boolean in α,$\alpha\rightarrow$Int results in Boolean,Boolean$\rightarrow$Int. If multiple substitutions happen simultaneously, each substitution

does not affect the others. For instance, consider $\alpha \to \beta$. If one substitutes α with β and β with γ simultaneously, the result is $\beta \to \gamma$, not $\gamma \to \gamma$.
- Every sentence explaining type checking assumes use of the current type environment, which is given to a case including the sentence at the beginning unless a particular type environment is specified.
- If we say "it is well-formed" while explaining well-formedness of τ, then τ is well-formed.
- We use the word "must" to represent requirements. If a requirement is violated while well-formedness of τ is explained, then τ is not well-formed.

The following explains how well-formedness of each type is checked.

Case $t[\tau_1, \ldots, \tau_n]$:
1. Let Γ be the current type environment.
2. For every $1 \le i \le n$,
   (a) τ_i must be well-formed.
3. t must be in the domain of Γ.
4. Let $\Gamma(t)$ be type $t[\alpha_1, \ldots, \alpha_m]\ \overline{w}$.
5. n must be the same as m. (The number of the type arguments must be the same as the number of the type parameters.)
6. It is well-formed.

Case α:
1. Let Γ be the current type environment.
2. α must be in Γ.
3. It is well-formed.

Case Int:
1. It is well-formed.

Case Boolean:
1. It is well-formed.

Case Unit:
1. It is well-formed.

Case $\tau_1, \ldots, \tau_n \to \tau$:
1. For every $1 \le i \le n$,
   (a) τ_i must be well-formed.
2. τ must be well-formed.
3. It is well-formed.

Now, type checking of expressions can be defined. We use the following terminologies:
- If we say "the type is τ" while explaining type checking of e, then the type of e is τ, and e is well-typed.
- We use the word "must" to represent requirements. If a requirement is violated while type checking of e is explained, then e is not well-typed.
- If we say "the type of e must be τ," then it implicitly adds the requirement that "e must be well-typed."
- If we say "let τ be the type of e," then it implicitly adds the requirement that "e must be well-typed."

The following explains how each expression is type-checked.

Case $x[\tau_1, \ldots, \tau_n]$:
1. Let Γ be the current type environment.
2. For every $1 \le i \le n$,
   (a) τ_i must be well-formed.
3. x must be in the domain of Γ.
4. Let $\Gamma(x)$ be $(\forall \alpha_1, \ldots, \alpha_m.\tau, \mu)$.
5. n must be the same as m. (The number of the type arguments must be the same as the number of the type parameters.)
6. For every $1 \le i \le n$,
   (a) Substitute every occurence of α_i in τ with τ_i. (Every substitution happens simultaneously.)
7. Let τ' be the result of substitution.
8. The type is τ'.

Case n:
1. The type is Int.

Case b:
1. The type is Boolean.

Case ( ):
1. The type is Unit.

Case $e_1 \oplus e_2$:

(∗) Suppose that $\oplus \in \{+, \times, \div, \mathsf{mod}\}$.
(1) The type of e_1 must be Int.
(2) The type of e_2 must be Int.
(3) The type is Int.

Case $e_1 \otimes e_2$:

(∗) Suppose that $\otimes \in \{=, <\}$.
(1) The type of e_1 must be Int.
(2) The type of e_2 must be Int.
(3) The type is Boolean.

Case e_1 ; e_2:

(1) e_1 must be well-typed.
(2) Let τ be the type of e_2.
(3) The type is τ.

Case if e_1 e_2 e_3:

(1) The type of e_1 must be Boolean.
(2) Let τ be the type of e_2.
(3) The type of e_3 must be τ.
(4) The type is τ.

Case $\mu\ x{:}\boxed{\tau_1'}{=}e_1$ in e_2:

(1) Let Γ be the current type environment.
(2) If $:\tau_1'$ exists, then
  (a) τ_1' must be well-formed.
(3) Let τ_1 be the type of e_1.
(4) If $:\tau_1'$ exists, then
  (a) τ_1' must be equal to τ_1.
(5) Add a mapping from x to $(\forall.\tau_1, \mu)$ to Γ.
(6) Let Γ' be the new type environment.
(7) Let τ be the type of e_2 under Γ'.
(8) The type is τ.

Case $d_1, \ldots, d_n$ in e:

(1) Let Γ_0 be the current type environment.
(2) For every $1 \le i \le n$,
  (a) If d_i is a type definition, then
    (i) Let d_i be type $t[\overline{\alpha}]\ \overline{w}$.
    (ii) t must not be in the domain of Γ_0.
  (b) Let d_i creates Γ_i'.
  (c) Add every mapping in Γ_i' to Γ_{i-1}.
  (d) Let Γ_i be the new type environment. .
(3) For every $1 \le i \le n$,
  (a) d_i must be well-formed under Γ_n.
(4) Let τ be the type of e under Γ_n.
(5) τ must be well-formed under Γ_0.
(6) The type is τ.

Case $\lambda x_1{:}\tau_1, \ldots, x_n{:}\tau_n.e$:

(1) Let Γ_0 be the current type environment.
(2) For every $1 \le i \le n$,
  (a) τ_i must be well-formed under Γ_0.
(3) For every $1 \le i \le n$,
  (a) Add a mapping from x_i to $(\forall.\tau_i, \mathsf{val})$ to Γ_{i-1}.
  (b) Let Γ_i be the new type environment.
(4) Let τ be the type of e under Γ_n.
(5) The type is $\tau_1, \ldots, \tau_n \rightarrow \tau$.

Case $x := e$:

(1) Let Γ be the current type environment.
(2) x must be in the domain of Γ.
(3) Let $\Gamma(x)$ be $(\forall\overline{\alpha}.\tau, \mu)$.
(4) The length of $\overline{\alpha}$ must be zero. (There must be no type parameter.)
(5) μ must be var.
(6) τ must be the type of e.
(7) The type is Unit.

Case $e(e_1, \ldots, e_n)$:

(1) Let τ_f be the type of e.
(2) τ_f must be a function type.
(3) Let τ_f be $\tau_1 \ldots \tau_m \rightarrow \tau$.
(4) n must be the same as m. (The number of the arguments must be the same as the number of the parameters.)
(5) For every $1 \le i \le n$,
  (a) τ_i must be the type of e_i.
(6) The type is τ.

Case e match $c_1, \ldots, c_n$:

(1) Let Γ be the current type environment.
(2) Let τ be the type of e.
(3) τ must be a type application.
(4) Let τ be $t[\tau_1, \ldots, \tau_m]$.
(5) t must be in the domain of Γ.
(6) Let $\Gamma(t)$ be type $t[\alpha_1, \ldots, \alpha_l]\ w_1, \ldots, w_k$.
(7) m must be the same as l. (The number of the type arguments must be the same as the number of the type parameters.)
(8) n must be the same as k. (The number of the cases must be the same as the number of the variants.)
(9) For every $1 \le i \le n$,
  (a) Let τ_i' be the type of c_i when $w_1, \ldots, w_k$, $\alpha_1, \ldots, \alpha_l$, and $\tau_1, \ldots \tau_m$ are given.
(10) For every $1 \le i \le n$,
  (a) τ_i' must be equal to τ_1'.
(11) The type is τ_1'.

In addition to the types of expressions, we need the types of cases. We use the following terminologies:

- If we say "the type is τ" while explaining type checking of c, then the type of c is τ, and c is well-typed.
- We use the word "must" to represent requirements. If a requirement is violated while type checking of c is explained, then c is not well-typed.
- If we say "let τ be the type of e," then it implicitly adds the requirement that "e must be well-typed."
- Type checking of c is possible only when a list of variant definitions, a list of type variables, and a list of types are given.

The following explains type checking of cases.

Case $x(x_1, \ldots, x_n) \rightarrow e$:

(1) Let $w_1, \ldots, w_k$, $\alpha_1, \ldots, \alpha_l$, and $\tau_1, \ldots \tau_m$ be given.
(2) l must be the same as m.
(3) Let Γ_0 be the current type environment.
(4) There must be w among $w_1, \ldots, w_k$ such that the name of w is x.
(5) Let w be $x(\tau'_1, \ldots, \tau'_h)$.
(6) n must be the same as h.
(7) For every $1 \leq i \leq h$,
  (a) For every $1 \leq j \leq l$,
    (i) Substitute every occurence of α_j in τ'_i with τ_j.
    (Every substitution happens simultaneously.)
  (b) Let τ''_i be the result of substitution.
  (c) Add a mapping from x_i to $(\forall.\tau''_i, \text{val})$ to Γ_{i-1}.
  (d) Let Γ_i be the new type environment.
(8) Let τ be the type of e under Γ_h.
(9) The type is τ.

We need to define well-formedness of recursive definitions. We use the following terminologies:

- If we say "it is well-formed" while explaining well-formedness of d, then d is well-formed.
- We use the word "must" to represent requirements. If a requirement is violated while well-formedness of d is explained, then d is not well-formed.

The following explains well-formedness of recursive definitions.

Case lazy $x{:}\tau{=}e$:

(1) τ must be well-formed.
(2) τ must be the type of e.
(3) It is well-formed.

Case def $x[\alpha_1, \ldots \alpha_n](x_1{:}\tau_1, \ldots, x_m{:}\tau_m){:}\tau{=}e$:

(1) Let Γ be the current type environment.
(2) For every $1 \leq i \leq n$,
  (a) α_i must not be in Γ.
(3) Add $\alpha_1, \ldots \alpha_n$ to Γ.
(4) Let Γ_0 be the new type environment.
(5) For every $1 \leq i \leq m$,
  (a) τ_i must be well-formed under Γ_0.
(6) τ must be well-formed under Γ_0.
(7) For every $1 \leq i \leq m$,
  (a) Add a mapping from x_i to $(\forall.\tau_i, \text{val})$ to Γ_{i-1}.
  (b) Let Γ_i be the new type environment.
(8) τ must be be the type of e under Γ_m.
(9) It is well-formed.

Case type $t[\alpha_1, \ldots, \alpha_n]\ w_1, \ldots, w_m$:

(1) Let Γ be the current type environment.
(2) For every $1 \leq i \leq n$,
  (a) α_i must not be in Γ.
(3) Add $\alpha_1, \ldots \alpha_n$ to Γ.
(4) Let Γ' be the new type environment.
(5) For every $1 \leq i \leq m$,
  (a) Let w_i be $x(\tau_1, \ldots, \tau_l)$.
  (b) For every $1 \leq j \leq l$,
    (i) τ_j must be well-formed under Γ'.
(6) It is well-formed.

Variant	W	$::=$	$x()$	$\mid E \div E$	(division)
		$\mid$	$x(\_)$	$\mid E \bmod E$	(modulo)
Case	C	$::=$	$x(\overline{x}) \rightarrow E$	$\mid E = E$	(equal-to)
Recursive Definition	D	$::=$	lazy $x{=}E$	$\mid E < E$	(less-than)
		$\mid$	def $x(\overline{x}){=}E$	$\mid E\ ;\ E$	(sequence)
		$\mid$	type $\overline{W}$	$\mid$ if $E\ E\ E$	(conditional)
Expression	E	$::=$	x (variable)	$\mid$ let $x{=}E$ in E	(local variable)
		$\mid$	n (integer)	$\mid \overline{D}$ in E	(recursive definition)
		$\mid$	b (boolean)	$\mid \lambda\overline{x}.E$	(anonymous function)
		$\mid$	$()$ (unit)	$\mid x := E$	(assignment)
		$\mid$	$E + E$ (addition)	$\mid E(\overline{E})$	(function application)
		$\mid$	$E \times E$ (multiplication)	$\mid E$ match $\overline{C}$	(pattern matching)

Fig. 5. Abstract Syntax after Type Erasure

We need to define type environments created by recursive definitions as well. We use the following terminology:

- If we say "the result is Γ" while explaining d, then d creates Γ.

The following explains type environments created by recursive definitions.

Case lazy $x{:}\tau{=}e$:

(1) Add a mapping from x to $(\forall.\tau, \mathsf{val})$ to the empty type environment.
(2) Let Γ be the new type environment.
(3) The result is Γ.

Case def $x[\overline{\alpha}](\overline{x{:}\tau}){:}\tau{=}e$:

(1) Add a mapping from x to $(\forall\overline{\alpha}.\overline{\tau}{\rightarrow}\tau, \mathsf{val})$ to the empty type environment.
(2) Let Γ be the new type environment.
(3) The result is Γ.

Case type $t[\overline{\alpha}]\ w_1, \ldots, w_n$:

(1) Add a mapping from t to type $t[\overline{\alpha}]\ w_1, \ldots, w_n$ to the empty type environment.
(2) Let Γ_0 be the new type environment.
(3) For every $1 \le i \le n$,
  (a) If w_i does not take any parameter, then
    (i) Let w_i be $x()$.
    (ii) Add a mapping from x to $(\forall\overline{\alpha}.t[\overline{\alpha}], \mathsf{val})$ to Γ_{i-1}.
    (iii) Let Γ_i be the new type environment.
  (b) Else if w_i takes one or more parameters, then
    (i) Let w_i be $x(\overline{\tau})$.
    (ii) Add a mapping from x to $(\forall\overline{\alpha}.\overline{\tau}{\rightarrow}t[\overline{\alpha}], \mathsf{val})$ to Γ_{i-1}.
    (iii) Let Γ_i be the new type environment.
(4) The result is Γ_n.

6 TYPE ERASURE

Type erasure removes type annotations from programs. Fabric uses type erasure since type annotations take no role at run time. They are used for only type checking. In this section, we define type erasure for Fabric.

First, we define an untyped version, which lacks type annotations, of Fabric. Figure 5 describes the abstract syntax of the untyped version. Metavariable W ranges over untyped variant definitions; C ranges over untyped cases in pattern matching; D ranges over untyped recursive definitions; E ranges over untyped expressions.

Most of the untyped version is similar to the typed version except that they lack type annotations. The following briefly describes notable changes:

$$
\boxed{\begin{array}{rcl} \mathit{erase}(e) & = & E \\ \mathit{erase}(d) & = & D \\ \mathit{erase}(w) & = & W \\ \mathit{erase}(c) & = & C \end{array}}
$$

$$
\begin{array}{rcl}
\mathit{erase}(x[\overline{\tau}]) & = & x \\
\mathit{erase}(n) & = & n \\
\mathit{erase}(b) & = & b \\
\mathit{erase}(()) & = & () \\
\mathit{erase}(e_1 + e_2) & = & \mathit{erase}(e_1) + \mathit{erase}(e_2) \\
\mathit{erase}(e_1 \times e_2) & = & \mathit{erase}(e_1) \times \mathit{erase}(e_2) \\
\mathit{erase}(e_1 \div e_2) & = & \mathit{erase}(e_1) \div \mathit{erase}(e_2) \\
\mathit{erase}(e_1 \ \mathsf{mod}\ e_2) & = & \mathit{erase}(e_1) \ \mathsf{mod}\ \mathit{erase}(e_2) \\
\mathit{erase}(e_1 = e_1) & = & \mathit{erase}(e_1) = \mathit{erase}(e_2) \\
\mathit{erase}(e_1 < e_1) & = & \mathit{erase}(e_1) < \mathit{erase}(e_2)
\end{array}
$$

$$
\begin{array}{rcl}
\mathit{erase}(e_1 \ ; \ e_2) & = & \mathit{erase}(e_1) \ ; \ \mathit{erase}(e_2) \\
\mathit{erase}(\mathsf{if}\ e_1\ e_2\ e_3) & = & \mathsf{if}\ \mathit{erase}(e_1)\ \mathit{erase}(e_2)\ \mathit{erase}(e_3) \\
\mathit{erase}(\mu\ x\boxed{:\tau}{=}e_1\ \mathsf{in}\ e_2) & = & \mathsf{let}\ x{=}\mathit{erase}(e_1)\ \mathsf{in}\ \mathit{erase}(e_2) \\
\mathit{erase}(\overline{d}\ \mathsf{in}\ e) & = & \overline{\mathit{erase}(d)}\ \mathsf{in}\ \mathit{erase}(e) \\
\mathit{erase}(\lambda\overline{x{:}\tau}.e) & = & \lambda\overline{x}.\mathit{erase}(e) \\
\mathit{erase}(x := e) & = & x := \mathit{erase}(e) \\
\mathit{erase}(e(\overline{e})) & = & \mathit{erase}(e)(\overline{\mathit{erase}(e)}) \\
\mathit{erase}(e\ \mathsf{match}\ \overline{c}) & = & \mathit{erase}(e)\ \mathsf{match}\ \overline{\mathit{erase}(c)} \\
\mathit{erase}(\mathsf{lazy}\ x{:}\tau{=}e) & = & \mathsf{lazy}\ x{=}e \\
\mathit{erase}(\mathsf{def}\ x[\overline{\alpha}](\overline{x{:}\tau}){:}\tau{=}e) & = & \mathsf{def}\ x(\overline{x}){=}e \\
\mathit{erase}(\mathsf{type}\ t[\overline{\alpha}]\ \overline{w}) & = & \mathsf{type}\ \overline{\mathit{erase}(w)} \\
\mathit{erase}(x(\,)) & = & x(\,) \\
\mathit{erase}(x(\overline{\tau})) & = & x(\_) \quad (\text{if } \#(\overline{\tau}) > 0) \\
\mathit{erase}(x(\overline{x}) \rightarrow e) & = & x(\overline{x}) \rightarrow \mathit{erase}(e)
\end{array}
$$

Fig. 6. Type Erasure

- A variant definition W is either $x(\,)$ or $x(\_)$. The former, $x(\,)$, represents a variant without any parameters; the latter, $x(\_)$, represents a variant with one or more parameters. It lacks not only type annotations but also the exact number of parameters. However, whether parameters exist or not is important at run time. If there is no parameter, the name of the variant itself denotes the only value of the variant. On the other hand, if there are one or more parameters, the name of the variant has to be applied to a proper number of arguments to produce a value of the variant. For example, when `type List[] Nil(), Cons(Int, List)` is given, both `Nil` and `Cons(1)` are well-typed expressions that result in values of the type `List`, while `Nil()` is not a well-typed expression.
- The syntax of a type definition is type $\overline{W}$, which lacks the name of the type. Since the names of types are unnecessary at run time, each type definition contains only a list of variants.
- The syntax of a variable definition is let $x{=}E$ in E. It lacks not only a type annotation for the variable but also the mutability tag. Whether a variable is mutable or immutable matters only during type checking. If a program passes type checking, it is guaranteed that the value of every immutable variable never changes. Therefore, mutability tags are not required at run time.

Figure 6 defines type erasure for FABRIC. $\mathit{erase}(e)$ denotes an untyped expression created by erasing types in e. Similarly, $\mathit{erase}(d)$, $\mathit{erase}(w)$, and $\mathit{erase}(c)$ respectively denotes applying type erasure to a recursive definition, a variant definition, and a case in pattern matching. Type erasure is defined in a straightforward recursive manner.

7 DYNAMIC SEMANTICS

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

Figure 7 defines values, addresses, environments, and stores. Metavariable v ranges over values. A value is one of the following:

- an integer n
- a boolean value b
- the unit value $(\,)$

$$
\begin{array}{rrcl}
\text{Value} & v & \in & \mathbb{V} \\
& v & ::= & n \mid b \mid () \mid \langle \lambda \overline{x}.E, \sigma \rangle \\
& & \mid & \langle\langle E, \sigma \rangle\rangle \mid \langle x \rangle \mid x(\overline{v})
\end{array}
\qquad
\begin{array}{rrcl}
\text{Address} & a & \in & \mathcal{A} \\
\text{Environment} & \sigma & \in & Id \hookrightarrow \mathcal{A} \\
\text{Store} & M & \in & \mathcal{A} \hookrightarrow \mathbb{V}
\end{array}
$$

Fig. 7. Definitions for Dynamic Semantics

- a closure $\langle \lambda \overline{x}.E, \sigma \rangle$
- a delayed computation $\langle\langle E, \sigma \rangle\rangle \in ExprV$
- a constructor $\langle x \rangle$
- a variant value $x(\overline{v})$

$\mathbb{V}$ is the set of every value. Metavariable a ranges over addresses, which are elements of an infinite set $\mathcal{A}$. Metavariable σ ranges over environments, which are finite maps from identifiers to addresses. Metavariable M ranges over stores, which are finite maps from addresses to values.

We use the following terminologies:

- Every sentence explaining the semantics assumes use of the current environment, which is given to a case including the sentence at the beginning unless a particular environment is specified.
- Every sentence assumes use of the most recently created store.
- 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 evaluation of expressions.

Case x:

(1) Let σ be the current environment.
(2) Let M be the current store.
(3) x must be in the domain of σ.
(4) Let a be $\sigma(x)$.
(5) a must be in the domain of M.
(6) Let v be $M(a)$.
(7) If v is not a delayed computation, then
  (a) The result is v.
(8) Else if v is a delayed computation, then
  (a) Let v be $\langle\langle E, \sigma' \rangle\rangle$.
  (b) Evaluate E under σ'.
  (c) Let v_1 be the result of E.
  (d) Let M_1 be the current store.
  (e) Add a mapping from a to v_1 to M_1.
  (f) The result is v_1.

Case n:

(1) The result is n.

Case b:

(1) The result is b.

Case ( ):

(1) The result is ( ).

Case $E_1 \otimes E_2$:

(*) Suppose that $\otimes \in \{+, \times, \div, \mathsf{mod}, =, <\}$.
(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 $\otimes$. 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 $v_1 \otimes v_2$.

Case E_1 ; E_2:

(1) Evaluate E_1.
(2) Evaluate E_2.
(3) Let v_2 be the result of E_2.
(4) The result is 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) Else 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 let x=E_1 in E_2:

(1) Let σ be the current environment.
(2) Evaluate E_1.
(3) Let v_1 be the result of E_1.
(4) Let M_1 be the current store.
(5) Let a not be in the domain of M_1.
(6) Add a mapping from x to a to σ.
(7) Let σ' be the new environment.
(8) Add a mapping from a to v_1 to M_1.
(9) Evaluate E_2 under σ'.
(10) Let v_2 be the result of E_2.
(11) The result is v_2.

Case $D_1, \ldots, D_n$ in E:

(1) Let σ be the current environment.
(2) Let σ_0 be the empty environment.
(3) Let M be the current store.
(4) For every $1 \le i \le n$,
(a) Let D_i creates σ_i'. (Every address in the range of σ_i' is not in the domain of M and the range of σ_{i-1}. In addition, all the addresses in the range of σ_i' are distinct from each other.)
(b) Add every mapping in σ_i' to σ_{i-1}.
(c) Let σ_i be the new environment.
(5) Add every mapping in σ_n to σ.
(6) Let σ' be the new environment.
(7) For every $1 \le i \le n$,
(a) Let D_i creates M_i' when σ' is given.
(b) Let M_i be the current store.
(c) Add every mapping in M_i' to M_i.
(8) Evaluate E under σ'.
(9) Let v be the result of E.
(10) The result is v.

Case $\lambda\overline{x}.E$:

(1) Let σ be the current environment.
(2) The result is $\langle \lambda\overline{x}.E, \sigma \rangle$.

Case $x := E$:

(1) Let σ be the current environment.
(2) x must be in the domain of σ.
(3) Evaluate E.
(4) Let v be the result of E.
(5) Let M_1 be the current store.
(6) Add a mapping from $\sigma(x)$ to v to M_1.
(7) The result is ( ).

Case $E(E_1, \ldots, E_n)$:

(1) Evaluate E.
(2) Let v be the result of E.
(3) v must be either a closure or a constructor.
(4) For every $1 \le i \le n$,
(a) Evaluate E_i. (If $i > 1$, it happens after the evaluation of E_{i-1}.)
(b) Let v_i be the result of E_i.
(5) If v is a closure, then
(a) Let v be $\langle \lambda x_1, \ldots, x_m.E_c, \sigma_0 \rangle$.
(b) n must be the same as m. (The number of the arguments must be the same as the number of the parameters.)
(c) Let M_n be the current store.
(d) For every $1 \le i \le n$,
(i) Let a_i be an address that is not in the domain of M_n and different from every a_j where $i \neq j$.
(e) For every $1 \le i \le n$,
(i) Add a mapping from x_i to a_i to σ_{i-1}.
(ii) Let σ_i be the new environment.
(f) For every $1 \le i \le n$,
(i) Let M_i' be the current store.
(ii) Add a mapping from a_i to v_i to M_i'.
(g) Evaluate E_c under σ_n.
(h) Let v_c be the result of E_c.
(i) The result is v_c.
(6) If v is a constructor, then
(a) Let v be $\langle x \rangle$.
(b) The result is $x(v_1, \ldots, v_n)$.

Case E match $C_1, \ldots, C_n$:

(1) Let σ_0 be the current environment.
(2) Evaluate E.
(3) Let v be the result of E.
(4) v must be a variant value.
(5) Let v be $x(v_1, \ldots, v_m)$.
(6) There must be C among $C_1, \ldots, C_n$ such that the name of C is x.
(7) Let C be $x(x_1, \ldots, x_k) \rightarrow E_c$.
(8) m must be the same as k.
(9) Let M_1 be the current store.
(10) For every $1 \le i \le m$,

(a) Let a_i be an address that is not in the domain of M_1 and different from every a_j where $i \neq j$.
(11) For every $1 \leq i \leq m$,
(a) Add a mapping from x_i to a_i to σ_{i-1}.
(b) Let σ_i be the new environment.
(12) For every $1 \leq i \leq m$,
(a) Let M'_i be the current store.
(b) Add a mapping from a_i to v_i to M'_i.
(13) Evaluate E_c under σ_m.
(14) Let v_c be the result of E_c.
(15) The result is v_c.

We need to define environments created by recursive definitions. We use the following terminology:

- If we say "the result is σ" while explaining D, then D creates σ.

The following explains environments created by recursive definitions.

Case lazy x=E:

(1) Add a mapping from x to a new address a to the empty environment.
(2) Let σ be the new environment.
(3) The result is σ.

Case def $x(\overline{x})$=E:

(1) Add a mapping from x to a new address a to the empty environment.
(2) Let σ be the new environment.
(3) The result is σ.

Case type $W_1, \ldots, W_n$:

(1) Let σ_0 be the empty environment.
(2) For every $1 \leq i \leq n$,
(a) If W_i is a variant definition without a parameter, then
(i) Let W_i be $x(\,)$.
(ii) Add a mapping from x to a new address a to σ_{i-1}.
(iii) Let σ_i be the new environment.
(b) Else if W_i is a variant definition with one or more parameters, then
(i) Let W_i be $x(\_)$.
(ii) Add a mapping from x to a new address a to σ_{i-1}.
(iii) Let σ_i be the new environment.
(3) The result is σ_n.

We need to define also stores created by recursive definitions. We use the following terminology:

- If we say "the result is M" while explaining D, then D creates M.
- D can create a store only when an environment is given.

The following explains stores created by recursive definitions.

Case lazy x=E:

(1) Let σ be a given environment.
(2) Add a mapping from $\sigma(x)$ to $\langle\langle E, \sigma\rangle\rangle$ to the empty store.
(3) Let M be the new store.
(4) The result is M.

Case def $x(\overline{x})$=E:

(1) Let σ be a given environment.
(2) Add a mapping from $\sigma(x)$ to $\langle\lambda\overline{x}.E, \sigma\rangle$ to the empty store.
(3) Let M be the new store.
(4) The result is M.

Case type $W_1, \ldots, W_n$:

(1) Let σ be a given environment.
(2) Let M_0 be the empty store.
(3) For every $1 \leq i \leq n$,
(a) If W_i is a variant definition without a parameter, then
(i) Let W_i be $x(\,)$.
(ii) Add a mapping from $\sigma(x)$ to $x(\,)$ to M_{i-1}.
(iii) Let M_i be the new store.
(b) Else if W_i is a variant definition with one or more parameters, then
(i) Let W_i be $x(\_)$.
(ii) Add a mapping from $\sigma(x)$ to $\langle x\rangle$ to M_{i-1}.
(iii) Let M_i be the new store.
(4) The result is M_n.

8 STANDARD LIBRARY

We provide the standard library of Fabric. The library consists of eight parts: utility functions for primitive types, which are integers, boolean values, and the unit value, pair types and utility functions, option types and utility functions, box types and utility functions, list types and utility functions, map types and utility functions, utility functions for strings, and utility functions to build parser combinators. This sections gives a brief overview of the Fabric standard library.

8.1 Primitive Types

```
def intEquals(l: Int, r: Int): Boolean
def intMax(l: Int, r: Int): Int
def intMin(l: Int, r: Int): Int
def booleanEquals(l: Boolean, r: Boolean): Boolean
def unitEquals(l: Unit, r: Unit): Boolean
```

8.2 Pairs

```
type Pair['T, 'S] {
  case Pair('T, 'S)
}
def pairFst['T, 'S](p: Pair['T, 'S]): 'T
def pairSnd['T, 'S](p: Pair['T, 'S]): 'S
def pairEquals['T, 'S](f: ('T, 'T) => Boolean, g: ('S, 'S) => Boolean):
  (Pair['T, 'S], Pair['T, 'S]) => Boolean
```

8.3 Options

```
type Option['T] {
  case None
  case Some('T)
}
def optionEquals['T](f: ('T, 'T) => Boolean): (Option['T], Option['T]) => Boolean
def optionFilter['T](o: Option['T], f: 'T => Boolean): Option['T]
def optionFilterNot['T](o: Option['T], f: 'T => Boolean): Option['T]
def optionFlatMap['T, 'S](o: Option['T], f: 'T => Option['S]): Option['S]
def optionFlatten['T](o: Option[Option['T]]): Option['T]
def optionForeach['T](o: Option['T], f: 'T => Unit): Unit
def optionGetOrElse['T](o: Option['T], t: 'T): 'T
def optionIsEmpty['T](o: Option['T]): Boolean
def optionMap['T, 'S](o: Option['T], f: 'T => 'S): Option['S]
def optionNonEmpty['T](o: Option['T]): Boolean
def optionToList['T](o: Option['T]): List['T]
```

8.4 Boxes

```
type Box['T] {
  case Box_(Option['T] => 'T)
}
def Box['T](t: 'T): Box['T]
def boxGet['T](b: Box['T]): 'T
def boxSet['T](b: Box['T], t: 'T): 'T
```

8.5 Lists

```
type List['T] {
  case Nil
  case Cons('T, List['T])
}
def List0['T](): List['T]
def List1['T](t1: 'T): List['T]
def List2['T](t1: 'T, t2: 'T): List['T]
def List3['T](t1: 'T, t2: 'T, t3: 'T): List['T]
def List4['T](t1: 'T, t2: 'T, t3: 'T, t4: 'T): List['T]
def List5['T](t1: 'T, t2: 'T, t3: 'T, t4: 'T, t5: 'T): List['T]
def listAppended['T](l: List['T], t: 'T): List['T]
def listConcat['T](l: List['T], r: List['T]): List['T]
def listCount['T](l: List['T], f: 'T => Boolean): Int
def listDrop['T](l: List['T], i: Int): List['T]
def listEquals['T](f: ('T, 'T) => Boolean): (List['T], List['T]) => Boolean
def listExists['T](l: List['T], f: 'T => Boolean): Boolean
def listFilter['T](l: List['T], f: 'T => Boolean): List['T]
def listFilterNot['T](l: List['T], f: 'T => Boolean): List['T]
def listFind['T](l: List['T], f: 'T => Boolean): Option['T]
def listFlatMap['T, 'S](l: List['T], f: 'T => List['S]): List['S]
def listFlatten['T](l: List[List['T]]): List['T]
def listFoldLeft['T, 'S](a: 'S, l: List['T], f: ('S, 'T) => 'S): 'S
def listFoldRight['T, 'S](l: List['T], a: 'S, f: ('T, 'S) => 'S): 'S
def listForall['T](l: List['T], f: 'T => Boolean): Boolean
def listForeach['T](l: List['T], f: 'T => Unit): Unit
def listGet['T](l: List['T], i: Int): Option['T]
def listIsEmpty['T](l: List['T]): Boolean
def listLength['T](l: List['T]): Int
def listMap['T, 'S](l: List['T], f: 'T => 'S): List['S]
def listNonEmpty['T](l: List['T]): Boolean
def listPrepended['T](l: List['T], t: 'T): List['T]
def listReverse['T](l: List['T]): List['T]
def listTake['T](l: List['T], i: Int): List['T]
def listUnzip['T, 'S](l: List[Pair['T, 'S]]): Pair[List['T], List['S]]
def listZip['T, 'S](l: List['T], r: List['S]): List[Pair['T, 'S]]
def listZipWithIndex['T](l: List['T]): List[Pair['T, Int]]
```

8.6 Maps

```
type Map['T, 'S] {
  case Map(List[Pair['T, 'S]], ('T, 'T) => Boolean)
}
def Map0['T, 'S](f: ('T, 'T) => Boolean): Map['T, 'S]
def Map1['T, 'S](f: ('T, 'T) => Boolean, t1: 'T, s1: 'S): Map['T, 'S]
def Map2['T, 'S](f: ('T, 'T) => Boolean, t1: 'T, s1: 'S, t2: 'T, s2: 'S): Map['T, 'S]
def Map3['T, 'S](
  f: ('T, 'T) => Boolean, t1: 'T, s1: 'S, t2: 'T, s2: 'S, t3: 'T, s3: 'S
): Map['T, 'S]
```

```
def Map4['T, 'S](
  f: ('T, 'T) => Boolean, t1: 'T, s1: 'S, t2: 'T, s2: 'S, t3: 'T, s3: 'S, t4: 'T, s4: 'S
): Map['T, 'S]
def Map5['T, 'S](
   f: ('T, 'T) => Boolean,
   t1: 'T, s1: 'S, t2: 'T, s2: 'S, t3: 'T, s3: 'S, t4: 'T, s4: 'S, t5: 'T, s5: 'S
): Map['T, 'S]
def mapFold['T, 'S, 'R](r: 'R, m: Map['T, 'S], f: ('R, 'T, 'S) => 'R): 'R
def mapGet['T, 'S](m: Map['T, 'S], t: 'T): Option['S]
def mapRemoved['T, 'S](m: Map['T, 'S], t: 'T): Map['T, 'S]
def mapToList['T, 'S](m: Map['T, 'S]): List[Pair['T, 'S]]
def mapUpdated['T, 'S](m: Map['T, 'S], t: 'T, s: 'S): Map['T, 'S]
```

8.7 Strings

```
val STRP = Cons[Int]
val EOS = Nil[Int]
val stringEquals = listEquals[Int](intEquals)
def substring(s: List[Int], i: Int, j: Int): List[Int]
```

8.8 Parser Combinators

```
type Parser['T] {
  case Parser(List[Int] => ParseResult['T])
}
type ParseResult['T] {
  case Success('T, List[Int])
  case Failure
}
def parse['T](parser: Parser['T], str: List[Int]): ParseResult['T]
def parseAll['T](parser: Parser['T], str: List[Int]): Option['T]
def parserConst(c: Int): Parser[Int]
def parserCond(p: Int => Boolean): Parser[Int]
def parserThen['T, 'S](l: () => Parser['T], r: () => Parser['S]): Parser[Pair['T, 'S]]
def parserOr['T](l: () => Parser['T], r: () => Parser['T]): Parser['T]
def parserMap['T, 'S](l: () => Parser['T], f: 'T => 'S): Parser['S]
```

9 TYPE CHECKER AND INTERPRETER SPECIFICATION

A type checker and an interpreter of FABRIC must satisfy the following conditions:

- If the type of e is τ, then `typeCheck(`e`)` must be equal to τ.
- If e is not well-typed, then `typeCheck(`e`)` must terminate by calling the `error` function. Error messages are implementation-dependent.
- If E results in v, then `interp(`E`)` must be equal to v.
- If execution of E terminates with a run-time error, then `interp(`E`)` must terminate by calling the `error` function. Error messages are implementation-dependent.
- If execution of E does not terminate, then `interp(`E`)` must not terminate or terminate due to stack overflow.

A reference interpreter of FABRIC is available at https://plrg.kaist.ac.kr/fabric.

A STATIC SEMANTICS

Figure 8 defines sets $\mathbb{M}$ and and $\mathbb{D}$, and metavariables δ and Δ. It formally defines type substitutions and type environments.

$$
\begin{array}{rrcl}
\text{Mutability} & \mu & \in & \mathbb{M} \\
\text{Type Definition} & \Delta & \in & \mathbb{D} \\
 & \Delta & ::= & \mathsf{type}\ t[\overline{\alpha}]\ \overline{w} \\
\text{Type Substitution} & \delta & ::= & [\overline{\alpha \leftarrow \tau}] \\
\text{Type Environment} & \Gamma & \in & (\mathit{Id} \hookrightarrow (\mathbb{T} \times \mathbb{M})) + (\mathit{TN} \hookrightarrow \mathbb{D}) + (\mathit{TV} \hookrightarrow \{\cdot\})
\end{array}
$$

Fig. 8. Definitions

Figure 9 defines some notations used in the appendices. In the figure, S denotes an arbitrary set. i, j, and k denote arbitary mathematical objects.

$$
\begin{array}{rcl@{\qquad}rcl}
S_1 + S_2 & \triangleq & \{i_1 \cup i_2 : i_1 \in S_1 \wedge i_2 \in S_2\} & \Gamma[\alpha] & \triangleq & \Gamma \cup \{(\alpha, \cdot)\} \\
S - i & \triangleq & \{(j,k) : (j,k) \in S \wedge j \neq i\} & \Gamma[x{:}\tau] & \triangleq & \Gamma[x{:}(\forall.\tau, \mathsf{val})] \\
S[i{:}j] & \triangleq & (S - i) \cup \{(i,j)\} & \Gamma[x{:}(\tau,\mu)] & \triangleq & \Gamma[x{:}(\forall.\tau,\mu)] \\
S[i{=}j] & \triangleq & (S - i) \cup \{(i,j)\} & \Gamma[x{:}T] & \triangleq & \Gamma[x{:}(T,\mu)] \text{ for some } \mu \\
S[i \mapsto j] & \triangleq & (S - i) \cup \{(i,j)\} & \Gamma(x) = T & \triangleq & \Gamma(x) = (T,\mu) \text{ for some } \mu \\
S; \{\overline{(i,j)}\} & \triangleq & (S - \overline{i}) \cup \{\overline{(i,j)}\} & \Gamma(x) = (\tau,\mu) & \triangleq & \Gamma(x) = (\forall.\tau,\mu) \\
\#(i_1,\ldots,i_n) & \triangleq & n & & &
\end{array}
$$

Fig. 9. Notations

From now on, we use underlines in addition to overlines. An underline below submaterial within overlined material prevents an overline from attaching subscripts to the metavariables in the submaterial; the effect is to require the underlined material to be the same in each copy. For example, $\overline{\tau\underline{\delta}}$ represents $\tau_1\delta, \ldots, \tau_n\delta$ for any $n \geq 0$.

Figure 10 formally defines the results of substituting types.

$\boxed{\tau\delta = \tau}$

$$
\begin{array}{rcl@{\qquad}rcl}
t[\overline{\tau}]\delta & = & t[\overline{\tau\underline{\delta}}] & \mathsf{Int}\delta & = & \mathsf{Int} \\
\alpha[\cdots\alpha{\leftarrow}\tau\cdots] & = & \tau & \mathsf{Boolean}\delta & = & \mathsf{Boolean} \\
\alpha[\overline{\alpha\leftarrow\tau}] & = & \alpha \ \ (\text{if} \bigwedge \overline{\alpha \neq \underline{\alpha}}) & \mathsf{Unit}\delta & = & \mathsf{Unit} \\
 & & & (\overline{\tau}{\rightarrow}\tau)\delta & = & \overline{\tau\underline{\delta}}{\rightarrow}\tau\delta
\end{array}
$$

Fig. 10. Type Substitutions

We use overlines and underlines for inference rules as well. If a rule contains an overline, it is, in fact, not a rule; it is a rule schema, which generates a set of infinitely many rules. For instance, the rule schema

$$\frac{\overline{\underline{\Gamma} \vdash \tau} \quad \Gamma \vdash \tau}{\Gamma \vdash \overline{\tau}{\rightarrow}\tau}$$

generates the following rules:

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash {\rightarrow}\tau} \qquad \frac{\Gamma \vdash \tau_1 \quad \Gamma \vdash \tau}{\Gamma \vdash \tau_1{\rightarrow}\tau} \qquad \cdots \qquad \frac{\Gamma \vdash \tau_1 \quad \cdots \quad \Gamma \vdash \tau_{10} \quad \Gamma \vdash \tau}{\Gamma \vdash \tau_1,\ldots,\tau_{10}{\rightarrow}\tau} \qquad \cdots$$

Figure 11 formally defines the well-formedness of types; Figure 12 formally defines the types of expressions and cases.

$$\boxed{\Gamma \vdash \tau}$$

$$\frac{\overline{\Gamma \vdash \tau} \quad t \in \mathit{Domain}(\Gamma) \quad \Gamma(t) = \text{type } t[\overline{\alpha}]\ \overline{w} \quad \#(\overline{\alpha}) = \#(\overline{\tau})}{\Gamma \vdash t[\overline{\tau}]}$$

$$\frac{\alpha \in \mathit{Domain}(\Gamma)}{\Gamma \vdash \alpha} \qquad \Gamma \vdash \mathsf{Int} \qquad \Gamma \vdash \mathsf{Boolean} \qquad \Gamma \vdash \mathsf{Unit} \qquad \frac{\overline{\Gamma \vdash \tau} \quad \Gamma \vdash \tau}{\Gamma \vdash \overline{\tau} \to \tau}$$

Fig. 11. Well-formedness of Types

$$\boxed{\Gamma \vdash e : \tau}$$

$$\frac{\overline{\Gamma \vdash \tau} \quad x \in \mathit{Domain}(\Gamma) \quad \Gamma(x) = \forall \overline{\alpha}.\tau}{\Gamma \vdash x[\overline{\tau}] : \tau[\overline{\alpha \leftarrow \tau}]}$$

$$\Gamma \vdash n : \mathsf{Int} \qquad \Gamma \vdash b : \mathsf{Boolean} \qquad \Gamma \vdash () : \mathsf{Unit}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \quad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 \oplus e_2 : \mathsf{Int}} \quad (\oplus \in \{+, \times, \div, \mathrm{mod}\}) \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \quad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 \otimes e_2 : \mathsf{Boolean}} \quad (\otimes \in \{=, <\})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \,;\, e_2 : \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Boolean} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\ e_1\ e_2\ e_3 : \tau} \qquad \frac{\boxed{\Gamma \vdash \tau_1} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma[x{:}(\tau_1, \mu)] \vdash e_2 : \tau_2}{\Gamma \vdash \mu\, x{:}\boxed{\tau_1} = e_1\ \mathsf{in}\ e_2 : \tau_2}$$

$$\frac{\overline{\Gamma \vdash d\ \mathsf{name\ ok}} \quad \Gamma' = \Gamma; \overline{\mathit{tbind}(d)} \quad \overline{\Gamma' \vdash d} \quad \Gamma' \vdash e : \tau \quad \Gamma \vdash \tau}{\Gamma \vdash \overline{d}\ \mathsf{in}\ e : \tau} \qquad \frac{\overline{\Gamma \vdash \tau} \quad \Gamma[\overline{x{:}\tau}] \vdash e : \tau}{\Gamma \vdash \lambda \overline{x{:}\tau}.e : \overline{\tau} \to \tau}$$

$$\frac{x \in \mathit{Domain}(\Gamma) \quad \Gamma(x) = (\tau, \mathrm{var}) \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \mathsf{Unit}} \qquad \frac{\Gamma \vdash e : \overline{\tau} \to \tau \quad \overline{\Gamma \vdash e : \tau}}{\Gamma \vdash e(\overline{e}) : \tau}$$

$$\frac{\Gamma \vdash e : t[\overline{\tau}] \quad t \in \mathit{Domain}(\Gamma) \quad \Gamma(t) = \text{type } t[\overline{\alpha}]\ \overline{w} \quad \#(\overline{w}) = \#(\overline{c}) \quad \delta = [\overline{\alpha \leftarrow \tau}] \quad \overline{\Gamma, \overline{w}, \delta \vdash c : \tau}}{\Gamma \vdash e\ \mathsf{match}\ \overline{c} : \tau}$$

$$\boxed{\Gamma, \overline{w}, \delta \vdash c : \tau}$$

$$\frac{w_i = x(\overline{\tau}) \quad \Gamma[\overline{x{:}\tau\delta}] \vdash e : \tau}{\Gamma, \overline{w}, \delta \vdash x(\overline{x}) \to e : \tau}$$

Fig. 12. Types of Expressions

Note that metavariables under the same overline have to be consistent in their numbers. For example, the rule schema

$$\frac{\overline{\Gamma \vdash \tau} \quad x \in \mathit{Domain}(\Gamma) \quad \Gamma(x) = \forall \overline{\alpha}.\tau}{\Gamma \vdash x[\overline{\tau}] : \tau[\overline{\alpha \leftarrow \tau}]}$$

generates

$$\frac{\Gamma \vdash \tau_1 \quad x \in \mathit{Domain}(\Gamma) \quad \Gamma(x) = \forall \alpha_1.\tau_1}{\Gamma \vdash x[\tau_1] : \tau[\alpha_1 \leftarrow \tau_1]}$$

but not

$$\frac{\Gamma \vdash \tau_1 \quad x \in \mathit{Domain}(\Gamma) \quad \Gamma(x) = \forall \alpha_1, \alpha_2.\tau_1}{\Gamma \vdash x[\tau_1] : \tau[\alpha_1 \leftarrow \tau_1, \alpha_2 \leftarrow \tau_2]}$$

Also, if a rule schema contains boxes, then the boxes have to be consistent, i.e. all of them exist simultaneously, or none of them exist. Therefore, the rule schema

$$\frac{\boxed{\Gamma \vdash \tau_1} \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma[x{:}(\tau_1, \mu)] \vdash e_2 : \tau_2}{\Gamma \vdash \mu\ x{:}\boxed{\tau_1}{=}e_1 \text{ in } e_2 : \tau_2}$$

never generates

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma[x{:}(\tau_1, \mu)] \vdash e_2 : \tau_2}{\Gamma \vdash \mu\ x{:}\tau_1{=}e_1 \text{ in } e_2 : \tau_2}$$

Figure 13, 14, and 15 define additional judgements for the static semantics.

$\boxed{\Gamma \vdash d \text{ name ok}}$

$$\Gamma \vdash \text{lazy } x{:}\tau{=}e \text{ name ok} \qquad \Gamma \vdash \text{def } x[\overline{\alpha}](\overline{x{:}\tau}){:}\tau{=}e \text{ name ok} \qquad \frac{t \notin \mathit{Domain}(\Gamma)}{\Gamma \vdash \text{type } t[\overline{\alpha}]\ \overline{w} \text{ name ok}}$$

Fig. 13. Well-formedness of Type Names

$\boxed{\begin{array}{l}\Gamma \vdash d \\ \Gamma \vdash w\end{array}}$

$$\frac{\Gamma \vdash \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash \text{lazy } x{:}\tau{=}e} \qquad \frac{\overline{\alpha \notin \mathit{Domain}(\underline{\Gamma})} \quad \Gamma' = \Gamma[\overline{\alpha}] \quad \overline{\underline{\Gamma'} \vdash \tau} \quad \Gamma' \vdash \tau \quad \Gamma'[\overline{x{:}\tau}] \vdash e : \tau}{\Gamma \vdash \text{def } x[\overline{\alpha}](\overline{x{:}\tau}){:}\tau{=}e}$$

$$\frac{\overline{\underline{\Gamma} \vdash \tau}}{\Gamma \vdash x(\overline{\tau})} \qquad \frac{\overline{\alpha \notin \mathit{Domain}(\underline{\Gamma})} \quad \Gamma' = \Gamma[\overline{\alpha}] \quad \overline{\underline{\Gamma'} \vdash w}}{\Gamma \vdash \text{type } t[\overline{\alpha}]\ \overline{w}}$$

Fig. 14. Well-formedness of Recursive Definitions

$\boxed{\begin{array}{r}\mathit{tbind}(d) = \Gamma \\ \mathit{tbind}(w, t[\overline{\alpha}]) = \Gamma\end{array}}$

$$\begin{array}{rcl}
\mathit{tbind}(\text{lazy } x{:}\tau{=}e) & = & [x{:}\tau] \\
\mathit{tbind}(\text{def } x[\overline{\alpha}](\overline{x{:}\tau}){:}\tau{=}e) & = & [x{:}\forall\overline{\alpha}.\overline{\tau} \rightarrow \tau] \\
\mathit{tbind}(\text{type } t[\overline{\alpha}]\ \overline{w}) & = & [t = \text{type } t[\overline{\alpha}]\ \overline{w}]; \overline{\mathit{tbind}(w, \underline{t[\overline{\alpha}]})} \\
\mathit{tbind}(x(\,), t[\overline{\alpha}]) & = & [x{:}\forall\overline{\alpha}.t[\overline{\alpha}]] \\
\mathit{tbind}(x(\overline{\tau}), t[\overline{\alpha}]) & = & [x{:}\forall\overline{\alpha}.\overline{\tau} \rightarrow t[\overline{\alpha}]] \quad (\text{if } \#(\overline{\tau}) > 0)
\end{array}$$

Fig. 15. Binding of Recursive Definitions

B BIG-STEP OPERATIONAL SEMANTICS

Figure 16 and 17 define the operational semantics of Fabric.

$$\boxed{\sigma, M \vdash e \Rightarrow v, M} \qquad \frac{x \in \mathit{Domain}(\sigma) \quad \sigma(x) \in \mathit{Domain}(M) \quad M(\sigma(x)) \notin \mathit{ExprV}}{\sigma, M \vdash x \Rightarrow M(\sigma(x)), M}$$

$$\frac{x \in \mathit{Domain}(\sigma) \quad \sigma(x) \in \mathit{Domain}(M) \quad M(\sigma(x)) = \langle\langle E, \sigma' \rangle\rangle \quad \sigma', M \vdash E \Rightarrow v, M_1}{\sigma, M \vdash x \Rightarrow v, M_1[\sigma(x) \mapsto v]}$$

$$\sigma, M \vdash n \Rightarrow n, M \qquad \sigma, M \vdash b \Rightarrow b, M \qquad \sigma, M \vdash ( ) \Rightarrow ( ), M$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow n_1, M_1 \quad \sigma, M_1 \vdash E_2 \Rightarrow n_2, M_2}{\sigma, M \vdash E_1 \odot E_2 \Rightarrow n_1 \odot n_2, M_2} \quad (\odot \in \{+, \times, =, <\})$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow n_1, M_1 \quad \sigma, M_1 \vdash E_2 \Rightarrow n_2, M_2 \quad n_2 \neq 0}{\sigma, M \vdash E_1 \otimes E_2 \Rightarrow n_1 \otimes n_2, M_2} \quad (\otimes \in \{\div, \mathsf{mod}\})$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow v_1, M_1 \quad \sigma, M_1 \vdash E_2 \Rightarrow v_2, M_2}{\sigma, M \vdash E_1 \; ; E_2 \Rightarrow v_2, M_2}$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow \mathsf{true}, M_1 \quad \sigma, M_1 \vdash E_2 \Rightarrow v_2, M_2}{\sigma, M \vdash \mathsf{if}\ E_1\ E_2\ E_3 \Rightarrow v_2, M_2} \qquad \frac{\sigma, M \vdash E_1 \Rightarrow \mathsf{false}, M_1 \quad \sigma, M_1 \vdash E_3 \Rightarrow v_2, M_2}{\sigma, M \vdash \mathsf{if}\ E_1\ E_2\ E_3 \Rightarrow v_2, M_2}$$

$$\frac{\sigma, M \vdash E_1 \Rightarrow v_1, M_1 \quad a \notin \mathit{Domain}(M_1) \quad \sigma[x \mapsto a], M_1[a \mapsto v_1] \vdash E_2 \Rightarrow v_2, M_2}{\sigma, M \vdash \mathsf{let}\ x{=}E_1\ \mathsf{in}\ E_2 \Rightarrow v_2, M_2}$$

$$\frac{\begin{array}{c}\overline{D\ \mathsf{creates}\ \sigma} \quad \sigma'' = \overline{\sigma} \quad \mathit{Domain}(\sigma'') = \{\overline{x}\} \quad M \cap \{\overline{\underline{\sigma''}(x)}\} = \emptyset \quad \mathit{distinct}(\overline{\underline{\sigma''}(x)}) \\ \sigma' = \sigma; \sigma'' \quad \overline{D, \underline{\sigma'}\ \mathsf{create}\ M'} \quad M_1 = M; \overline{M'} \quad \sigma', M_1 \vdash E \Rightarrow v, M_2\end{array}}{\sigma, M \vdash \overline{D}\ \mathsf{in}\ E \Rightarrow v, M_2}$$

$$\sigma, M \vdash \lambda\overline{x}.E \Rightarrow \langle \lambda\overline{x}.E, \sigma \rangle, M \qquad \frac{x \in \mathit{Domain}(\sigma) \quad \sigma, M \vdash E \Rightarrow v, M_1}{\sigma, M \vdash x := E \Rightarrow ( ), M_1[\sigma(x) \mapsto v]}$$

$$\frac{\begin{array}{c}\sigma, M \vdash E \Rightarrow \langle \lambda\overline{x}.E', \sigma' \rangle, M_0 \quad \overline{\underline{\sigma}, M_{i-1} \vdash E_i \Rightarrow v_i, M_i}^{1 \leq i \leq n} \\ \{\overline{a}\} \cap \mathit{Domain}(M_n) = \emptyset \quad \mathit{distinct}(\overline{a}) \quad \sigma'[\overline{x \mapsto a}], M_n[\overline{a \mapsto v}] \vdash E' \Rightarrow v, M'\end{array}}{\sigma, M \vdash E(\overline{E}) \Rightarrow v, M'}$$

Fig. 16. Evaluation of Expressions (1/2)

The overline notation may explicitly bind index variables by using a range expression. In such cases, the index variables may appear explicitly in the covered material, and subscripts are not automatically attached to metavariables. For example, $\overline{x}^{1 \leq i \leq n}$ represents $x_1, x_2, \ldots, x_n$ for any $n \geq 0$. In addition, the statement $\mathit{distinct}(k_1, \ldots, k_n)$ is true if and only if $k_i \neq k_j$ for every $1 \leq i < j \leq n$.

$$\frac{\sigma, M \vdash E \Rightarrow \langle x \rangle, M_0 \quad \overline{\underline{\sigma}, M_{i-1} \vdash E_i \Rightarrow v_i, M_i}^{1 \le i \le n}}{\sigma, M \vdash E(\overline{E}) \Rightarrow x(\overline{v}), M_n}$$

$$\frac{\begin{array}{c}\sigma, M \vdash E \Rightarrow x(\overline{v}), M_1 \quad C_i = x(\overline{x}) \rightarrow E' \\ \{\overline{a}\} \cap \mathit{Domain}(M_1) = \emptyset \quad \mathit{distinct}(\overline{a}) \quad \sigma[\overline{x \mapsto a}], M_1[\overline{a \mapsto v}] \vdash E' \Rightarrow v, M_2\end{array}}{\sigma, M \vdash E \text{ match } \overline{C} \Rightarrow v, M_2}$$

Fig. 17. Evaluation of Expressions (2/2)

Figure 18 formally defines environments and stores created by recursive definitions.

D creates σ	W creates σ	D,σ create M	W,σ create M

$$\text{lazy } x{=}E \text{ creates } [x \mapsto a] \qquad \text{lazy } x{=}E{,}\sigma \text{ create } [\sigma(x) \mapsto \langle\langle E, \sigma\rangle\rangle]$$

$$\text{def } x(\overline{x}){=}E \text{ creates } [x \mapsto a] \qquad \text{def } x(\overline{x}){=}E{,}\sigma \text{ create } [\sigma(x) \mapsto \langle \lambda\overline{x}.E, \sigma\rangle]$$

$$\frac{\overline{W \text{ creates } \sigma}}{\text{type } \overline{W} \text{ creates } \overline{\sigma}} \qquad \frac{\overline{W, \underline{\sigma} \text{ create } M}}{\text{type } \overline{W}, \sigma \text{ create } \overline{M}}$$

$$x(\,) \text{ creates } [x \mapsto a] \qquad x(\,), \sigma \text{ create } [\sigma(x) \mapsto x(\,)]$$

$$x(\_) \text{ creates } [x \mapsto a] \qquad x(\_), \sigma \text{ create } [\sigma(x) \mapsto \langle x \rangle]$$

Fig. 18. Binding of Recursive Definitions