Elements of Programming Languages
Assignment 3: Bidirectional Type Inference
Version 1.3.1 (November 19)
Due: November 21, 12pm

November 19, 2024

1 Introduction

This assignment is due November 21, at 12pm.

Please read over this handout carefully and look over the code before beginning work, as some of your questions may be answered later. Please let us know if there are any apparent errors or bugs. We will try to update this handout to fix any major problems and such updates will be announced to the course mailing list. The handout is versioned and the most recent version should always be available from the course web page.

2 Background

In this assignment you will implement the main components for a small, but realistic, language with records, variants, and a multiset collection type, *subtyping*, and *bidirectional typechecking*, called Frog. The syntax of Frog is similar to that of Giraffe from Assignment 2, with some differences in how functions are annotated with types. Instead, we will implement a form of typechecking for Frog called *bidirectional typing*. More information about bidirectional typing is available in the following paper:

> Jana Dunfield and Neel Krishnaswami. 2021. Bidirectional Typing. ACM Comput. Surv. 54, 5, Article 98 (June 2022), 38 pages. https://doi.org/10.1145/3450952

You do not need to read this paper to do this assignment, but sections 1–3 might be helpful as context explaining the rationale for the approach we will take.

Any valid Giraffe program from assignment 2 can be made into a valid Frog program if we adapt the type annotations. Type annotations can be added to ordinary or recursive functions by writing `sig f : ty` before the function definition, for example:

```
sig id : int -> int
let fun id(x) = x in ...
```

Recursive functions are also allowed with the same kind of type annotations:

```
sig fact : int -> int
let rec fact(n) = if (n == 0) then 1 else n*fact(n-1) in fact(4)
```

Frog includes records and record field access constructs along the lines discussed in class, and also allows a form of let-binding that pattern-matches a record against a record pattern. For example,

```
let <a=x,b=y> = <a=1,b=2> in x+y
```

typechecks and evaluates to $1 + 2 = 3$. Subtyping can be used to write functions that accept records of different types as follows:

```
sig getName : <name:string> -> string
let fun getName(x) = x.name in
let alice = <name = "Alice", age = 9> in
```

```
let bob = <name = "Bob", year = 1984> in
(getName alice, getName bob)
```

Frog also includes variant types and case analysis. For example, an option type can be defined as a variant type `[none:unit, some:int]`. Note that all cases of a variant have to have types, so we use the unit type when there is no meaningful data. The syntax `select c e` creates a variant expression with constructor `c` and contents `e`, and we can case analyze as follows:

```
sig f : [some: int, none: unit] -> int
let fun f(x) = case x of {some y -> y, none z -> 0} in
f(select some 42)
```

Subtyping can also be used with variants; the rules are similar to those for records but have important differences, discussed in more detail later.

Finally, Frog supports a collection type *bags* or *multisets*, that is, unordered collections where multiplicity matters (that is a collection `{|"abc"|}` with 1 copy of `"abc"` differs from one with two copies `{|"abc","abc"|}`). Bags have a form of comprehension syntax (see lecture 11), for example

```
{| concat(y,"z") | y <- {|"a", "aa", "aaa"|}, length(y) < 3|}
```

will evaluate to `{|"az","aaz"|}`.

Bags admit covariant subtyping so we can write functions that take bags containing different record types as follows:

```
sig f : {| <a: int, b: int> |} -> {| int |}
let fun f(x) = {|y.a + y.b | y <- x |} in
f {| <a=1,b=2,c="asdf" > |}
```

Implementing bidirectional typechecking is somewhat complex, but boils down to a simple idea: We decompose typechecking into two processes, *inference* and *checking*. When we are inferring a type, we write the typing judgment as $\Gamma \vdash e \Rightarrow \tau$ and we read this as "Assuming type information in Γ, we can infer type τ for expression e." We can use this judgment when we know Γ and e but not τ. In contrast, if we do know the expected type τ of e then we use the checking judgment $\Gamma \vdash e \Leftarrow \tau$, and we read this as "Assuming type information in Γ, we can check that expression e has type τ."

You will also implement substitution, some desugaring rules, and an evaluator for Frog. Evaluation is largely similar to evaluation for Giraffe, but in contrast to assignment 2, uses substitution instead of an environment. The main new features are to deal with records, variants, and multiset collections. For collections you will also implement a small interface in Scala that provides the collection operations, which will be used during evaluation.

3 Getting started

`Assn3.zip` contains a number of starting files; use the `unzip` command to extract them. We provide the following Scala files.

- `Syntax.scala` containing the abstract syntax of Frog and some additional useful code for reference cells, substitution.
- `Parser.scala` containing a parser using the Scala parser combinators library for parsing Frog code.
- `Bags.scala` containing an interface and skeleton implementation for bags (multisets) which you will need to fill in for Exercise 1.
- `Typer.scala` containing supporting code for exercises 2–3 where you will need to implement subtyping and bidirectional type checking.
- `Interpreter.scala` containing supporting code for exercises 4–6 where you will need to implement substitution, desugaring and evaluation, along with a main class/function for executing a read-eval-print loop or running Frog programs.

Although you only need to make changes to the `Bags.scala`, `Typer.scala` and `Interpreter.scala` files, it is recommended to read through the other files (especially `Syntax.scala`).

We include several example programs in a directory `examples`.

We also provide several scripts (which should work on a DICE, Linux or MacOS system):

- `compile.sh` compiles the code and pakcages it as a jar file (in a place where the remaining scripts will assume they can find it).
- `run.sh <file>` runs your standalone interpreter on a Frog file. If the filename is omitted then the read-eval-print interactive loop is started.
- `sample.sh <file>` works like `run.sh` but runs using a sample solution.
- `test.sh <file>` runs a test script on the current solution. The test scripts are in the `tests` subdirectory.

In addition a JAR file `Assn3Solution.jar` containing a working implementation is provided. The script `sample.sh` uses this.

If you are using a non-DICE system, such as Windows, these scripts may not work, but you can look at the contents to work out what you need to do instead.

3.1 Objectives

The rest of this handout defines exercises for you to complete, building on the partial implementation in the provided files. You may add your own function definitions or other code if necessary, but please use the existing definitions/types for the functions we ask you to write in the exercises, to simplify automated testing we may do. You should not need to change any existing code other than filling in definitions of functions as stated in the exercises below.

Your solutions may make use of Scala library operations, such as the list and list map operations that have been covered in previous assignments.

This assignment relies on material covered up to Lecture 14.

This assignment is graded on a scale of 100 points, and amounts to 20% of your final grade for this course. Your submissions will be marked and returned with feedback within 3 weeks if they are received by the due date.

Unlike the other two assignments, which were for feedback only, you must work on this assignment individually and not with others, in accordance with University policy on academic conduct. Please see the course web page for more information on this policy.

Submission instructions You should submit the three Scala files `Bags.scala`, `Typer.scala`, and `Interpreter.scala`. Please submit them using these names and including the suffix `.scala`, uploading all three directly to Gradescope and not in a zip file or directory. Any files in the submission other than these will be ignored so please be careful not to make changes to any of the other files in your solution. Please submit the files through Gradescope following the instructions on the Learn page for this assignment. The submission deadline is 12pm on November 21.

4 Frog Overview

The syntax of Frog is shown in Figure 1. This has three main differences from the Giraffe language from Assignment 2. First, type annotations on functions and recursive functions are given using signatures rather than in-line argument/result type annotations. Second, types and expression forms for *records* and *variants* are added, as discussed in Lecture 7. Third, a collection type for *bags* or *multisets* is provided, with a number of operations and a form of *comprehension syntax* (lecture 11).

Typechecking and type annotations As discussed later, Frog's type system is based on a technique called bidirectional typing. Functions are equipped with type signatures as follows:

```
sig f : ty1 -> ty2
let [fun|rec] f(x) = e in ...
```

Anonymous / recursive functions do not carry type annotations, but this means their use is limited to situations where the argument and (for recursive functions) result types can be identified from context.

Values	v	::=	$n \in \mathbb{N} \mid b \in \mathbb{B} \mid s \in \texttt{string}$	Constants
		|	$x \mid \backslash x.e \mid \mathsf{rec} f(x).e$	Functions
		|	(v_1, v_2)	Pairs
		|	unit	Unit value
		|	$\langle \ell_1 = v_1, \ldots, \ell_n = v_n \rangle$	Records
		|	$\mathsf{select}\ \ell\ v$	Variants
		|	$\{\mid v_1, \ldots, v_n \mid\}$	Multisets
Expressions	e	::=	$n \in \mathbb{N} \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2$	Numbers
		|	$b \in \mathbb{B} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \mid e_1 == e_2 \mid e_1 < e_2$	Booleans
		|	$s \in \texttt{string} \mid \mathsf{length}(e_1) \mid \mathsf{index}(e_1, e_2) \mid \mathsf{concat}(e_1, e_2)$	Strings
		|	$x \mid \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	Binding
		|	$\backslash x.e \mid \mathsf{rec} f(x).e \mid e_1\ e_2$	Functions
		|	$(e_1, e_2) \mid \mathsf{fst}(e) \mid \mathsf{snd}(e)$	Pairs
		|	unit	Unit
		|	$\langle \ell_1 = e_1, \ldots, \ell_n = e_n \rangle \mid e.\ell$	Records
		|	$\mathsf{select}\ \ell\ e \mid \mathsf{case}\ e\ \mathsf{of}\ \{\ell_1\ x_1 \to e_1, \ldots, \ell_n\ x_n \to e_n\}$	Variants
		|	$\{\mid e_1, \ldots, e_n \mid\} \mid \mathsf{when}(e_1, e_2) \mid \mathsf{count}(e_1, e_2)$	
		|	$\mathsf{sum}(e_1, e_2) \mid \mathsf{diff}(e_1, e_2) \mid \mathsf{flatMap}(e_1, e_2)$	Multisets
		|	$e : \tau$	Annotations
		|	$\mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2 \mid \mathsf{sig}\ f : \tau\ \mathsf{let}\ \mathsf{fun}\ f(x) = e_1\ \mathsf{in}\ e_2$	
		|	$\mathsf{sig}\ f : \tau\ \mathsf{let}\ \mathsf{rec}\ f(x) = e_1\ \mathsf{in}\ e_2$	
		|	$\mathsf{let}\ \langle \ell_1 = x_1, \ldots, \ell_n = x_n \rangle = e_1\ \mathsf{in}\ e_2$	
		|	$\{\mid e \mid p_1, \ldots, p_n \mid\} \mid \{\mid e \mid\}$	Syntactic Sugar
Comprehension Items	p	::=	$x \leftarrow e$	Binding
		|	e	Guard
		|	$\mathsf{let}\ x = e$	Let
Types	τ	::=	$\texttt{int} \mid \texttt{bool} \mid \texttt{string} \mid \texttt{unit} \mid \tau_1 \texttt{->} \tau_2 \mid \tau_1 * \tau_2$	
		|	$\langle \ell_1 : \tau_1, \ldots, \ell_n : \tau_n \rangle \mid [\ell_1 : \tau_1, \ldots, \ell_n : \tau_n] \mid \{\mid \tau \mid\}$	
Contexts	Γ	::=	$\cdot \mid \Gamma, x : \tau$	

Figure 1: Syntax of Frog

Bidirectional typing has limitations (particularly in the simple form we consider here), and there are times when a more sophisticated type system would be able to find a type that our system cannot. When this happens, it is helpful to be able to give the typechecker a hint as to the intended type of an expression. To do this, we can use the *type annotation operator* $e : \tau$, which essentially instructs the typechecker to check that e definitely does have type τ.

Records Record types are written `<l1=t1,...,ln=tn>` in concrete syntax, and likewise record expressions are written `<l1=e1,...,ln=en>`. Record field values can be extracted using field lookup `e.l`. Finally there is a variant of `let` that binds the field values of a record to variables:

```
let <l1=x1,...,ln=xn> = e1 in e2
```

This form is syntactic sugar which will be explained in greater detail later.

Record field order does not matter; that is, two records or record types are equal precisely when they have the same fields and each field is mapped to the same thing by both records/types. As a consequence, we require labels to be unique in a record.

Variants We write `select l e` to build an expression of variant type with constructor label `l` and contents `e`. By default, this will have type `[l:ty]` where `ty` is the type of `e`, but it can also have any other variant type that is a supertype of this type (see the Subtyping section below).

We do not require explicitly checking that the constructor labels in a `case` expression are unique, however since deep pattern matching is not supported, the effect of reusing such a label is that only the first occurrence

can ever match at run time.

Bags and Comprehensions The bag or multiset type, written `{|ty|}`, consists of values which are unordered collections of elements `{|v1,v2,...|}`, where the number of occurrences matters. Operations on bags include

- when(e_1, e_2) which takes a Boolean expression e_1 and a bag e_2 and returns either e_1 if e_2 is true, or the empty bag otherwise.
- sum(e_1, e_2) which combines the contents of two bags. For example sum$(\{|1,2,2,3|\}, \{|1,2|\})$ should evaluate to $\{|1,1,2,2,2,3|\}$.
- diff(e_1, e_2) which subtracts the contents of bag e_2 from bag e_1. For example diff$(\{|1,2,2,3|\}, \{|1,2|\})$ should evaluate to $\{|2,3|\}$.
- count(e_1, e_2) which returns the count of the number of elements in bag e_1 that are equal to e_2.
- flatMap(e_1, e_2) which takes a bag and a function, and performs a flatMap operation: combining all of the bags produced by evaluating the function on each element of e_1. For example flatMap$(\{|1,2,2,3|\}, \lambda x.\{|x,x|\})$ should evaluate to $\{|1,1,2,2,2,2,3,3|\}$.

In addition to these primitives, we permit a form of comprehension syntax for bags (see lecture 11), namely

```
{| e | p1, ..., pn |}
```

where `p1` through `pn` are comprehension items of the forms

- `x <- e` which binds `x` to each element of `e`,
- `e` which adds a Boolean expression `e` as a guard, or
- `let x = e` which binds `x` to the value of `e`

The scope of `x` in the expressions above is the rest of the list of comprehension items and the returned expression `e`. Thus, for example, in

```
{|x+y+z | x <- {|y,z|}, let y = 2*x, x == z|}
```

the variable `y` is free because it occurs in the first binding `x <- {|y|}`, the other two occurrences of `y` are bound or binding, all of the occurrences of `x` are bound or binding, and all of the occurrences of `z` are free.

Comprehensions can be translated down to expressions using the primitive operations by desugaring (see Desugaring, later). For example, the above expression can be written equivalently as

```
flatMap({|y,z|}, \x.
  let y = 2*x in
  when({|x+y+z|}, x == z)
)
```

Bag constructors $\{|e_1, \ldots, e_n|\}$ are required to have at least one element so that the type of the bag elements can be obtained by inferring a type for the first element. Bag comprehensions $\{|e \mid p_1, \ldots, p_n|\}$ are allowed to have empty lists of comprehension items p_i. We write $\{|e \mid |\}$ when the list of comprehension items is empty. This case is not immediately useful for programming since it behaves the same as a singleton bag constructor $\{|e|\}$, but it is useful internally for dealing with typing rules for bag constructors uniformly.

5 Implementing Multisets

We will use a Scala `trait` as an interface for multisets/bags, as follows (see `Bags.scala`):

```
trait Bag {
  type T[_]
  def toList[A](b: T[A]): List[A]
  def fromList[A](l: List[A]): T[A]
  def toString[A](b: T[A]) : String = toList(b).mkString("Bag(", ", ", ")")
  def sum[A](b1: T[A], b2: T[A]): T[A]
  def diff[A](b1: T[A], b2: T[A]): T[A]
  def flatMap[A,B](b: T[A], f: A => T[B]): T[B]
```

```
  def count[A](b: T[A], x: A): Int
}
```

Here, `type T[_]` declares that the `Bag` trait has a *abstract type constructor*, that is a type constructor `T` that takes another type as an argument. Implementing this feature will require filling in a particular type constructor such as `List[_]`. Note that this provides a customized `toString` function for printing bags, which relies on a `toList` method you will need to provide. You will need to implement this interface by defining a class (parameterized by `A`) called `BagImpl`:

```
object BagImpl extends Bag[A] {
  type T[_] = ...
  def toList[A](b: T[A]): List[A] = ...
  def fromList[A](l: List[A]): T[A] = ...
  def sum[A](b1: T[A], b2: T[A]): T[A] = ...
  def diff[A](b1: T[A], b2: T[A]): T[A] = ...
  def flatMap[A,B](b: T[A], f: A => T[B]): T[B] = ...
  def count[A](b: T[A], x: A): Int = ...
}
```

This can be implemented in several different ways. Two possibilities you may consider are to represent bags as lists of elements, in which case the `count` operation will need to count how many elements are in the list; or to represent bags as finite maps (e.g. `ListMap`s from elements to numbers, where the associated number is the number of copies of each element in the list). There are other possibilities as well.

Exercise 1. *Complete the implementation of the* `BagImpl` *object that implements all of the operations as outlined above.*

[10 marks]

6 Bidirectional Typechecking

6.1 Subtyping

Frog supports subtyping for record and variant types. Subtyping of record types allows both width and depth subtyping, and the order of fields is ignored; so for example the following subtyping relationship holds:

```
<a: int, b: <c: int, d: bool>> <: <b: <c: int>, a: int>
```

Variant types also admit subtyping, but a variant type τ_1 s a subtype of another τ_2 if all of the constructors in τ_1 are also present in τ_2 and the corresponding types are subtypes. Again, order is ignored, so for example

```
[b: <c: int, d: bool>, a: int] <: [a: int, b: <c: int>, e: string]
```

Note that as this example illustrates, record and variant types can be freely combined as well.

Subtyping is reflexive for base types, for example `int` is only a subtype or supertype of itself. Subtyping for pairing and function types is as defined in lectures, and for bag types is covariant.

The rules for subtyping are shown in Figure 2. In our bidirectional type system, the subsumption rule for subtyping will not be used freely, instead it will only be used in specific places, as discussed in the next section.

The notation $\tau \in \mathsf{BaseType}$ means that τ is one of the base types `int`, `bool`, `string`, or `unit`.

Exercise 2. *Define the function*

```
def subtype(ty1: Type, ty2: Type): Boolean
```

which tests whether a type is a subtype of another one using the rules in Figure 2.

[10 marks]

6.2 Typing Rules

The typing rules for inference and checking modes are shown in Figures 3 and 4, respectively. The inference rules are syntax-directed in that given Γ and e there is at most one rule we can use to derive $\Gamma \vdash e \Rightarrow \tau$ for some τ. The checking rules are not syntax-directed since there is one rule that is always applicable but should only be tried if no other rule applies. This rule is discussed further below.

$\boxed{\tau_1 <: \tau_2}$

$$\frac{\tau \in \mathsf{BaseType}}{\tau <: \tau} \qquad \frac{\tau_1 <: \tau_1' \qquad \tau_2 <: \tau_2'}{\tau_1 * \tau_2 <: \tau_1' * \tau_2'} \qquad \frac{\tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2'}{\tau_1 \text{ -> } \tau_2 <: \tau_1' \text{ -> } \tau_2'}$$

$$\frac{n \geq m \qquad \text{for all } \ell_i', \tau_i' \text{ there exists } \ell_j, \tau_j \text{ with } \ell_i' = \ell_j \text{ and with } \tau_i <: \tau_j'}{\langle \ell_1 = \tau_1, \ldots, \ell_n = \tau_n \rangle <: \langle \ell_1' = \tau_1', \ldots, \ell_m' = \tau_m' \rangle}$$

$$\frac{n \leq m \qquad \text{for all } \ell_i, \tau_i \text{ there exists } \ell_j', \tau_j' \text{ with } \ell_i = \ell_j' \text{ and with } \tau_i <: \tau_j'}{[\ell_1 = \tau_1, \ldots, \ell_n = \tau_n] <: [\ell_1' = \tau_1', \ldots, \ell_m' = \tau_m']} \qquad \frac{\tau <: \tau'}{\{|\tau|\} <: \{|\tau'|\}}$$

Figure 2: Subtyping rules

In Scala we will implement these judgements using two mutually recursive functions `tyCheck` and `tyInfer`. The expected behavior of tyCheck is either to return (necessarily) the unit value, indicating success, or to raise an exception if there is a type error. The expected behavior of tyInfer is to return the inferred type on success, or to raise an exception if there is a type error. A few cases of the implementation of each function are shown in the provided code to help give the idea.

One important feature to take note of in both sets of rules is that in there are no rules for dealing with comprehension items p directly, so these are not valid expressions on their own; instead, they can only appear within comprehension expressions $\{|e \mid p_1, \ldots, p_n|\}$. During typechecking of these expressions, we check the outermost (that is, the leftmost) comprehension item p_1 first, if any; depending on its form we may check or infer a type and/or add a variable to the context, and continue checking the rest of the comprehension $\{|e \mid p_2, \ldots, p_n|\}$. Once we have checked all comprehension items, we have a comprehension with just a variable and an empty list of such items left, which constructs a single-element multiset based on the expression.

Most of the other rules are similar to rules from Assignment 2 or from lectures, with the additional complication that some rules have both a checking and an inference form. Two additional rules are present that have no direct analogue, namely the rules for checking type annotations and for switching back from checking to inference mode.

$$\frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau} \qquad \frac{\Gamma \vdash e \Rightarrow \tau \qquad \tau <: \tau'}{\Gamma \vdash e \Leftarrow \tau'}$$

The first rule says that to infer a type for an expression that has a type annotation, we can use the type annotation itself, as long as we check that the expression does have that type. The second rule says that if we want to check that an expression has a type then it suffices to infer a type for the expression, and check that the inferred type is a subtype of the required type. Since this rule can always be applied, we only try it if no other more specific rules are applicable.

One final note is that in rules for constructs such as $e_1 == e_2$, $\mathsf{diff}(-,-)$, and $\mathsf{count}(-,-)$, the notation $\tau \in \mathsf{EqType}$ indicates that τ is a type whose values can be compared for equality. We only allow built-in equality to be used on base types or pairs or variants containing only equality types. A function `Typer.isEqType` is provided for checking this.

Exercise 3. *Implement the functions*

```
def tyInfer(ctx: Env, e: Expr): Type
def tyCheck(ctx: Env, e: Expr, ty: Type): Unit
```

which respectively correspond to the judgements $\Gamma \vdash e \Rightarrow \tau$ *and* $\Gamma \vdash e \Leftarrow \tau$.

[30 marks]

7 Evaluation

7.1 Substitution

Capture-avoiding substitution is needed for Frog both for desugaring and for evaluation, since the semantics you are to implement will use substitution to replace variables with their values. Luckily, we have defined the

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

$$\frac{n \in \mathbb{N}}{\Gamma \vdash n \Rightarrow \texttt{int}} \qquad \frac{\Gamma \vdash e_1 \Leftarrow \texttt{int} \qquad \Gamma \vdash e_2 \Leftarrow \texttt{int}}{\Gamma \vdash e_1 + e_2 \Rightarrow \texttt{int}} \qquad \frac{\Gamma \vdash e_1 \Leftarrow \texttt{int} \qquad \Gamma \vdash e_2 \Leftarrow \texttt{int}}{\Gamma \vdash e_1 - e_2 \Rightarrow \texttt{int}}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{int} \qquad \Gamma \vdash e_2 \Leftarrow \texttt{int}}{\Gamma \vdash e_1 * e_2 \Rightarrow \texttt{int}} \qquad \frac{b \in \mathbb{B}}{\Gamma \vdash b \Rightarrow \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau \qquad \Gamma \vdash e_2 \Leftarrow \tau \qquad \tau \in \mathsf{EqType}}{\Gamma \vdash e_1 == e_2 \Rightarrow \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{int} \qquad \Gamma \vdash e_2 \Leftarrow \texttt{int}}{\Gamma \vdash e_1 < e_2 \Rightarrow \texttt{bool}} \qquad \frac{\Gamma \vdash e \Leftarrow \texttt{bool} \qquad \Gamma \vdash e_1 \Rightarrow \tau \qquad \Gamma \vdash e_2 \Leftarrow \tau}{\Gamma \vdash \mathsf{if}\, e\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \Rightarrow \tau} \qquad \frac{s \in \texttt{string}}{\Gamma \vdash s \Rightarrow \texttt{string}}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{string}}{\Gamma \vdash \mathsf{length}(e_1) \Rightarrow \texttt{int}} \qquad \frac{\Gamma \vdash e_1 \Leftarrow \texttt{string} \qquad \Gamma \vdash e_2 \Leftarrow \texttt{int}}{\Gamma \vdash \mathsf{index}(e_1, e_2) \Rightarrow \texttt{string}} \qquad \frac{\Gamma \vdash e_1 \Leftarrow \texttt{string} \qquad \Gamma \vdash e_2 \Leftarrow \texttt{string}}{\Gamma \vdash \mathsf{concat}(e_1, e_2) \Rightarrow \texttt{string}}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x \Rightarrow \tau} \qquad \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash e : \tau \Rightarrow \tau} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 \Rightarrow \tau_2}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \Rightarrow \tau_2} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \texttt{ -> } \tau_2 \qquad \Gamma \vdash e_2 \Leftarrow \tau_1}{\Gamma \vdash e_1\, e_2 \Rightarrow \tau_2}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \Gamma \vdash e_2 \Rightarrow \tau_2}{\Gamma \vdash (e_1, e_2) \Rightarrow \tau_1 * \tau_2} \qquad \frac{\Gamma \vdash e \Rightarrow \tau_1 * \tau_2}{\Gamma \vdash \mathsf{fst}(e) \Rightarrow \tau_1} \qquad \frac{\Gamma \vdash e \Rightarrow \tau_1 * \tau_2}{\Gamma \vdash \mathsf{snd}(e) \Rightarrow \tau_2}$$

$$\frac{}{\Gamma \vdash \mathsf{unit} \Rightarrow \texttt{unit}} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n \Rightarrow \tau_n}{\Gamma \vdash \langle \ell_1 = e_1, \ldots, \ell_n = e_n \rangle \Rightarrow \langle \ell_1 = \tau_1, \ldots, \ell_n = \tau_n \rangle}$$

$$\frac{\Gamma \vdash e \Rightarrow \langle \ell_1 = \tau_1, \ldots, \ell_n = \tau_n \rangle \qquad 1 \leq j \leq n}{\Gamma \vdash e.\ell_j \Rightarrow \tau_j} \qquad \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash \mathsf{select}\, \ell\, e \Rightarrow [\ell : \tau]}$$

$$\frac{\Gamma \vdash e \Rightarrow [\ell_1 : \tau_1, \ldots, \ell_n : \tau_n] \qquad \Gamma, x_1 : \tau_1 \vdash e_1 \Rightarrow \tau \qquad \Gamma, x_2 : \tau_2 \vdash e_2 \Leftarrow \tau \qquad \ldots \qquad \Gamma, x_n : \tau_n \vdash e_n \Leftarrow \tau}{\Gamma \vdash \mathsf{case}\, e\, \mathsf{of}\, \{\ell_1\, x_1 \to e_1, \ldots, \ell_n\, x_n \to e_n\} \Rightarrow \tau}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau \qquad \Gamma \vdash e_2 \Leftarrow \tau \qquad \cdots \qquad \Gamma \vdash e_n \Leftarrow \tau}{\Gamma \vdash \{|e_1, \ldots, e_n|\} \Rightarrow \{|\tau|\}} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \{|\tau_1|\} \qquad \Gamma \vdash e_2 \Rightarrow \tau_1 \texttt{ -> } \{|\tau_2|\}}{\Gamma \vdash \mathsf{flatMap}(e_1, e_2) \Rightarrow \{|\tau_2|\}}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{bool} \qquad \Gamma \vdash e_2 \Rightarrow \{|\tau|\}}{\Gamma \vdash \mathsf{when}(e_1, e_2) \Rightarrow \{|\tau|\}} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \{|\tau|\} \qquad \Gamma \vdash e_2 \Leftarrow \{|\tau|\}}{\Gamma \vdash \mathsf{sum}(e_1, e_2) \Rightarrow \{|\tau|\}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \{|\tau|\} \qquad \Gamma \vdash e_2 \Leftarrow \{|\tau|\} \qquad \tau \in \mathsf{EqType}}{\Gamma \vdash \mathsf{diff}(e_1, e_2) \Rightarrow \{|\tau|\}} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \{|\tau|\} \qquad \Gamma \vdash e_2 \Leftarrow \tau \qquad \tau \in \mathsf{EqType}}{\Gamma \vdash \mathsf{count}(e_1, e_2) \Rightarrow \texttt{int}}$$

$$\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash \{|e \mid |\} \Rightarrow \{|\tau|\}} \qquad \frac{\Gamma \vdash e' \Leftarrow \texttt{bool} \qquad \Gamma \vdash \{|e \mid p_1 \ldots, p_n|\} \Rightarrow \{|\tau|\}}{\Gamma \vdash \{|e \mid e', p_1 \ldots, p_n|\} \Rightarrow \{|\tau|\}}$$

$$\frac{\Gamma \vdash e' \Rightarrow \{|\tau'|\} \qquad \Gamma, x : \tau' \vdash \{|e \mid p_1, \ldots, p_n|\} \Rightarrow \{|\tau|\}}{\Gamma \vdash \{|e \mid x \leftarrow e', p_1, \ldots, p_n|\} \Rightarrow \{|\tau|\}} \qquad \frac{\Gamma \vdash e' \Rightarrow \tau' \qquad \Gamma, x : \tau' \vdash \{|e \mid p_1 \ldots, p_n|\} \Rightarrow \{|\tau|\}}{\Gamma \vdash \{|e \mid \mathsf{let}\, x = e', p_1 \ldots, p_n|\} \Rightarrow \{|\tau|\}}$$

$$\frac{\begin{array}{c}\Gamma, x : \tau_1 \vdash e_1 \Leftarrow \tau_2 \\ \Gamma, f : \tau_1 \texttt{ -> } \tau_2 \vdash e_2 \Rightarrow \tau\end{array}}{\Gamma \vdash \mathsf{sig}\, f : \tau_1 \texttt{ -> } \tau_2\, \mathsf{let\ fun}\, f(x) = e_1\, \mathsf{in}\, e_2 \Rightarrow \tau} \qquad \frac{\begin{array}{c}\Gamma, f : \tau_1 \texttt{ -> } \tau_2, x : \tau_1 \vdash e_1 \Leftarrow \tau_2 \\ \Gamma, f : \tau_1 \texttt{ -> } \tau_2 \vdash e_2 \Rightarrow \tau\end{array}}{\Gamma \vdash \mathsf{sig}\, f : \tau_1 \texttt{ -> } \tau_2\, \mathsf{let\ rec}\, f(x) = e_1\, \mathsf{in}\, e_2 \Rightarrow \tau}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_1 * \tau_2 \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash e_2 \Rightarrow \tau}{\Gamma \vdash \mathsf{let}\, (x, y) = e_1\, \mathsf{in}\, e_2 \Rightarrow \tau}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \langle \ell_1 = \tau_1, \ldots, \ell_n = \tau_n \rangle \qquad \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_2 \Rightarrow \tau}{\Gamma \vdash \mathsf{let}\, \langle \ell_1 = x_1, \ldots, \ell_n = x_n \rangle = e_1\, \mathsf{in}\, e_2 \Rightarrow \tau}$$

Figure 3: Inference rules of Frog

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

$$\frac{\Gamma \vdash e \Leftarrow \texttt{bool} \qquad \Gamma \vdash e_1 \Leftarrow \tau \qquad \Gamma \vdash e_2 \Leftarrow \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Leftarrow \tau} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Leftarrow \tau_2} \qquad \frac{\Gamma, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \backslash x.e \Leftarrow \tau_1 \text{ -> } \tau_2}$$

$$\frac{\Gamma, f : \tau_1 \text{ -> } \tau_2, x : \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \mathsf{rec} f(x).e \Leftarrow \tau_1 \text{ -> } \tau_2} \qquad \frac{\Gamma \vdash e_1 \Leftarrow \tau_1 \qquad \Gamma \vdash e_2 \Leftarrow \tau_2}{\Gamma \vdash (e_1, e_2) \Leftarrow \tau_1 * \tau_2}$$

$$\frac{m \leq n \qquad \Gamma \vdash e_1 \Leftarrow \tau_1 \qquad \ldots \qquad \Gamma \vdash e_m \Leftarrow \tau_m \qquad \Gamma \vdash e_{m+1} \Rightarrow \tau_{m+1} \qquad \ldots \qquad \Gamma \vdash e_n \Rightarrow \tau_n}{\Gamma \vdash \langle \ell_1 = e_1, \ldots, \ell_n = e_n \rangle \Leftarrow \langle \ell_1 = \tau_1, \ldots, \ell_m = \tau_m \rangle}$$

$$\frac{\Gamma \vdash e \Leftarrow \langle \ell_j = \tau_j \rangle}{\Gamma \vdash e.\ell_j \Leftarrow \tau_j} \qquad \frac{\Gamma \vdash e \Leftarrow \tau_j \qquad 1 \leq j \leq n}{\Gamma \vdash \mathsf{select}\ \ell_j\ e \Leftarrow [\ell_1 : \tau_1, \ldots, \ell_n : \tau_n]}$$

$$\frac{\Gamma \vdash e \Rightarrow [\ell_1 : \tau_1, \ldots, \ell_n : \tau_n] \qquad \Gamma, x_1 : \tau_1 \vdash e_1 \Leftarrow \tau \qquad \ldots \qquad \Gamma, x_n : \tau_n \vdash e_n \Leftarrow \tau}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{\ell_1\, x_1 \to e_1, \ldots, \ell_n\, x_n \to e_n\} \Leftarrow \tau}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \tau \qquad \cdots \qquad \Gamma \vdash e_n \Leftarrow \tau}{\Gamma \vdash \{\!|e_1 \ldots, e_n|\!\} \Leftarrow \{\!|\tau|\!\}} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \{\!|\tau_1|\!\} \qquad \Gamma \vdash e_2 \Leftarrow \tau_1 \text{ -> } \{\!|\tau_2|\!\}}{\Gamma \vdash \mathsf{flatMap}(e_1, e_2) \Leftarrow \{\!|\tau_2|\!\}}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \texttt{bool} \qquad \Gamma \vdash e_2 \Leftarrow \{\!|\tau|\!\}}{\Gamma \vdash \mathsf{when}(e_1, e_2) \Leftarrow \{\!|\tau|\!\}} \qquad \frac{\Gamma \vdash e_1 \Leftarrow \{\!|\tau|\!\} \qquad \Gamma \vdash e_2 \Leftarrow \{\!|\tau|\!\}}{\Gamma \vdash \mathsf{sum}(e_1, e_2) \Leftarrow \{\!|\tau|\!\}}$$

$$\frac{\Gamma \vdash e_1 \Leftarrow \{\!|\tau|\!\} \qquad \Gamma \vdash e_2 \Leftarrow \{\!|\tau|\!\} \qquad \tau \in \mathsf{EqType}}{\Gamma \vdash \mathsf{diff}(e_1, e_2) \Leftarrow \{\!|\tau|\!\}} \qquad \frac{\Gamma \vdash e \Leftarrow \tau}{\Gamma \vdash \{\!|e \mid |\!\} \Leftarrow \{\!|\tau|\!\}}$$

$$\frac{\Gamma \vdash e' \Leftarrow \texttt{bool} \qquad \Gamma \vdash \{\!|e \mid p_1 \ldots, p_n|\!\} \Leftarrow \{\!|\tau|\!\}}{\Gamma \vdash \{\!|e \mid e', p_1 \ldots, p_n|\!\} \Leftarrow \{\!|\tau|\!\}} \qquad \frac{\Gamma \vdash e' \Rightarrow \{\!|\tau'|\!\} \qquad \Gamma, x : \tau' \vdash \{\!|e \mid p_1 \ldots, p_n|\!\} \Leftarrow \{\!|\tau|\!\}}{\Gamma \vdash \{\!|e \mid x \leftarrow e', p_1 \ldots, p_n|\!\} \Leftarrow \{\!|\tau|\!\}}$$

$$\frac{\Gamma \vdash e' \Rightarrow \tau' \qquad \Gamma, x : \tau' \vdash \{\!|e \mid p_1 \ldots, p_n|\!\} \Leftarrow \{\!|\tau|\!\}}{\Gamma \vdash \{\!|e \mid \mathsf{let}\ x = e', p_1 \ldots, p_n|\!\} \Leftarrow \{\!|\tau|\!\}} \qquad \frac{\Gamma, x : \tau_1 \vdash e_1 \Leftarrow \tau_2 \qquad \Gamma, f : \tau_1 \text{ -> } \tau_2 \vdash e_2 \Leftarrow \tau}{\Gamma \vdash \mathsf{sig}\ f : \tau_1 \text{ -> } \tau_2\ \mathsf{let\ fun}\ f(x) = e_1\ \mathsf{in}\ e_2 \Leftarrow \tau}$$

$$\frac{\begin{array}{c}\Gamma, f : \tau_1 \text{ -> } \tau_2, x : \tau_1 \vdash e_1 \Leftarrow \tau_2 \\ \Gamma, f : \tau_1 \text{ -> } \tau_2 \vdash e_2 \Leftarrow \tau\end{array}}{\Gamma \vdash \mathsf{sig}\ f : \tau_1 \text{ -> } \tau_2\ \mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2 \Leftarrow \tau} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_1 * \tau_2 \qquad \Gamma, x : \tau_1, y : \tau_2 \vdash e_2 \Leftarrow \tau}{\Gamma \vdash \mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2 \Leftarrow \tau}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \langle \ell_1 = \tau_1, \ldots, \ell_n = \tau_n \rangle \qquad \Gamma, x_1 : \tau_1, \ldots, x_n : \tau_n \vdash e_2 \Leftarrow \tau}{\Gamma \vdash \mathsf{let}\ \langle \ell_1 = x_1, \ldots, \ell_n = x_n \rangle = e_1\ \mathsf{in}\ e_2 \Leftarrow \tau} \qquad \frac{\Gamma \vdash e \Rightarrow \tau \qquad \tau <: \tau'}{\Gamma \vdash e \Leftarrow \tau'}$$

Figure 4: Checking rules of Frog

swapping operation for you, and most of the cases of substitution are similar to those for the Giraffe language of Assignment 2, so you can adapt them.

Function values may have expressions as subterms, but we expect that values are always *closed*, that is, have no free variables. Thus, for the case when the expression is actually a value `v: Value`, substitution does not need to do any work.

Similarly to assignment 2 you are to implement a substitution operation called `SubstExpr.subst` that substitutes an expression for a variable. Most of the cases are specified in the same way as in Assignment 2, and these cases can be reused verbatim in many cases. The cases for the lambda, rec, let fun and let rec constructs differ slightly in their different treatment of types using signatures, while most other cases are as shown in Figure 3 of Assignment 2, or are straightforward. The cases for the substitution operation behave on the distinctive constructs of Frog as follows:

$$
\begin{aligned}
v[e/x] &= v \\
\langle \ell_1 = e_1, \ldots, \ell_n = e_n \rangle [e/x] &= \langle \ell_1 = e_1[e/x], \ldots, \ell_n = e_n[e/x] \rangle \\
(e_0.\ell)[e/x] &= (e_0[e/x]).\ell \\
(\mathsf{let}\ \langle \ell_1 = x_1, \ldots, \ell_n = x_n \rangle = e_1\ \mathsf{in}\ e_2)[e/x] &= \\
\mathsf{let}\ \langle \ell_1 = y_1, \ldots, \ell_n = y_n \rangle = e_1[e/x]\ \mathsf{in}\ & (e_2(x_n \leftrightarrow y_n) \cdots (x_1 \leftrightarrow y_1))[e/x] \\
& (y_1, \ldots, y_n \text{ fresh}) \\
(\mathsf{select}\ \ell\ e_1)[e/x] &= \mathsf{select}\ \ell\ (e_1[e/x]) \\
(\mathsf{case}\ e_0\ \mathsf{of}\ \{\ell_1\, x_1 \to e_1, \ldots, \ell_n\, x_n \to e_n\})[e/x] &= \\
\mathsf{case}\ e_0[e/x]\ \mathsf{of}\ \{\ell_1\, y_1 \to e_1(x_1 \leftrightarrow y_1)[e/x], \ldots, \ell_n\, y_n & \to e_n(x_n \leftrightarrow y_n)[e/x]\} \\
& (y_1, \ldots, y_n \text{ fresh}) \\
\{\!|e_1, \ldots, e_n|\!\}[e/x] &= \{\!|e_1[e/x], \ldots, e_n[e/x]|\!\} \\
\mathsf{when}(e_1, e_2)[e/x] &= \mathsf{when}(e_1[e/x], e_2[e/x]) \\
\mathsf{sum}(e_1, e_2)[e/x] &= \mathsf{sum}(e_1[e/x], e_2[e/x]) \\
\mathsf{diff}(e_1, e_2)[e/x] &= \mathsf{diff}(e_1[e/x], e_2[e/x]) \\
\mathsf{count}(e_1, e_2)[e/x] &= \mathsf{count}(e_1[e/x], e_2[e/x]) \\
\mathsf{flatMap}(e_1, e_2)[e/x] &= \mathsf{flatMap}(e_1[e/x], e_2[e/x]) \\
\{\!|e_1 \mid p_1, \ldots, p_n|\!\}[e/x] &= \{\!|e_1[e/x] \mid (p_1, \ldots, p_n)[e/x]|\!\} \\
(x_1 \leftarrow e_1, p_1, \ldots, p_n)[e/x] &= (y \leftarrow e_1[e/x], (p_1, \ldots, p_n)(x_1 \leftrightarrow y)[e/x]) \\
& (y \text{ fresh}) \\
(\mathsf{let}\ x_1 = e_1, p_1, \ldots, p_n)[e/x] &= (\mathsf{let}\ y = e_1[e/x], (p_1, \ldots, p_n)(x_1 \leftrightarrow y)[e/x]) \\
& (y \text{ fresh}) \\
(e_1, p_1, \ldots, p_n)[e/x] &= (e_1[e/x], (p_1, \ldots, p_n)[e/x])
\end{aligned}
$$

Exercise 4. *Define the substitution function* `SubstExpr.subst` *for Frog expressions. This should take three arguments,* `e1`, `e2`, *and* `x`, *and return an expression* `subst(e1,e2,x)` *that is the result of capture avoiding substituton according to the above rules. This function should handle all cases of the* `Expr` *type—if it is not clear from the above instructions what the correct behavior for a case should be, please ask. (You may copy over the sample solution from Assignment 2 to use as a starting point.)*

[15 marks]

Note: The last four rules given above for substitution into bag constructs are not fully capture avoiding. Please implement them as ls, but you should instead ensure that substitution is always performed on desugared terms only.

7.2 Desugaring

Desugaring translates an expression in the full source language to a simpler sublanguage which is what is handled by the evaluator. In Frog, desugaring erases all type annotations, and replaces occurrences of constructs like let pair, let fun, and let rec more or less as in Assignment 2. Constructs such as let record and comprehensions are also desugared to simpler forms.

The following rules define the interesting cases of desugaring. Type annotations and type signatures are simply discarded wherever they appear. The cases involving comprehensions are the most interesting. Note in particular that the right-hand side of a rule involving comprehensions may still have further comprehension expressions to handle. This means that in the implementation of `desugar`, these cases should recursively call `desugar` to ensure that all occurrences of comprehension expressions are eventually eliminated. (This is not the only way to accomplish this, you are welcome to experiment with other ways.)

$$
\begin{array}{rcl}
e : \tau & \longrightarrow & e \\
\mathsf{sig}\ f : \tau\ \mathsf{let\ fun}\ f(x) = e_1\ \mathsf{in}\ e_2 & \longrightarrow & \mathsf{let}\ f = \backslash x.e_1\ \mathsf{in}\ e_2 \\
\mathsf{sig}\ f : \tau\ \mathsf{let\ rec}\ f(x) = e_1\ \mathsf{in}\ e_2 & \longrightarrow & \mathsf{let}\ f = \mathsf{rec} f(x).e_1\ \mathsf{in}\ e_2 \\
\mathsf{let}\ (x, y) = e_1\ \mathsf{in}\ e_2 & \longrightarrow & \mathsf{let}\ p = e_1\ \mathsf{in}\ e_2[\mathsf{fst}(p)/x, \mathsf{snd}(p)/y] \\
 & & \quad (p \notin FV(e_2)) \\
\mathsf{let}\ \langle \ell_1 = x_1, \ldots, \ell_n = x_n \rangle = e_1\ \mathsf{in}\ e_2 & \longrightarrow & \mathsf{let}\ r = e_1\ \mathsf{in}\ e_2[(r.\ell_1)/x_1, \ldots, (r.\ell_n)/x_n] \\
 & & \quad (r \notin FV(e_2)) \\
\{\!|e \mid |\!\} & \longrightarrow & \{\!|e|\!\} \\
\{\!|e \mid x \leftarrow e', p_1, \ldots, p_n|\!\} & \longrightarrow & \mathsf{flatMap}(e', \backslash x.\{\!|e \mid p_1, \ldots, p_n|\!\}) \\
\{\!|e \mid \mathsf{let}\ x = e', p_1, \ldots, p_n|\!\} & \longrightarrow & \mathsf{let}\ x = e'\ \mathsf{in}\ \{\!|e \mid p_1, \ldots, p_n|\!\} \\
\{\!|e \mid e', p_1, \ldots, p_n|\!\} & \longrightarrow & \mathsf{when}(e', \{\!|e \mid p_1, \ldots, p_n|\!\})
\end{array}
$$

Note: As mentioned above, the rules given for substitution of comprehension expressions are not fully capture avoiding. Substitution should therefore only be done after desugaring of subterms to eliminate comprehension operations.

Exercise 5. *Implement the function* `desugar` *that replaces the above constructs in* Frog *with their desugared forms. Desugaring should handle all possible cases of* `Expr` *and should rewrite all occurrences of the above desugaring rules. (As before, you may use the similar cases for Giraffe as a starting point.)*

[15 marks]

7.3 Implementing the Evaluator

The semantics of Frog is defined using large-step evaluation rules. The evaluation judgment is as follows:

$$e \Downarrow v$$

where e is the expression to be evaluated, and v is the value.

Most rules are similar to those given in lectures, specifically for arithmetic, booleans, variables, let-binding, functions, pairs, records, variants, and multisets as shown in Figure 5.

Notice that for flatMap, sum, and diff of multisets, we make use of the operations on multisets provided in exercise 1.

One subtlety to be aware of is that in the rule for $\mathsf{flatMap}(-,-)$, the second argument can be any kind of function: recursive or nonrecursive. When evaluating $\mathsf{flatMap}(-,-)$ using the `flatMap` function from exercise 1, we need to construct a Scala function which maps (element) values to (multiset) values, by evaluating this function v_2 on appropriate arguments y drawn from elements of the first argument, multiset v_1. This Scala function will need to call the evaluator function `eval` on an appropriate expression that passes argument y to function v_2 and evaluates to a result.

A second thing to be aware of is that in this specification, we have used the same syntax for expressions and values, but in the Scala code, there are separate case classes for expressions `Num`, `Bool`, `String`, `Lambda`, `Rec` and their corresponding values `NumV`, `BoolV`, `StringV`, `FunV`, and `RecV`. Thus, the first rule in Figure 5 corresponds to several cases the code needs to handle: one recognizing any `Value` evaluates to itself (which is provided in the starter code), and several additional rules for evaluating the various expression forms to their equivalent value forms, including for example

$\boxed{e \Downarrow v}$

$$\frac{v \text{ is a value}}{v \Downarrow v} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 + e_2 \Downarrow v_1 +_{\mathbb{N}} v_2} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 - e_2 \Downarrow v_1 -_{\mathbb{N}} v_2} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 * e_2 \Downarrow v_1 *_{\mathbb{N}} v_2} \qquad \frac{e \Downarrow \mathsf{true} \qquad e_1 \Downarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v}$$

$$\frac{e \Downarrow \mathsf{false} \qquad e_2 \Downarrow v}{\mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow v \qquad e_2 \Downarrow v}{e_1 == e_2 \Downarrow \mathsf{true}} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2 \qquad (v_1 \neq v_2)}{e_1 == e_2 \Downarrow \mathsf{false}} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 < e_2 \Downarrow n_1 <_{\mathbb{N}} n_2}$$

$$\frac{e \Downarrow v}{\mathsf{length}(e) \Downarrow \text{length of string } v} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\mathsf{index}(e_1, e_2) \Downarrow \text{the } v_2\text{-th character of } v_1}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\mathsf{concat}(e_1, e_2) \Downarrow \text{concatenation of } v_1 \text{ and } v_2} \qquad \frac{e_1 \Downarrow \backslash x.e \qquad e_2 \Downarrow v_1 \qquad e[v_1/x] \Downarrow v_2}{e_1\ e_2 \Downarrow v_2}$$

$$\frac{e_1 \Downarrow \mathsf{rec} f(x).e \qquad e_2 \Downarrow v_1 \qquad e[v_1/x, \mathsf{rec} f(x).e/f] \Downarrow v_2}{e_1\ e_2 \Downarrow v_2} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2[v_1/x] \Downarrow v}{\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 \Downarrow v} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)}$$

$$\frac{e \Downarrow (v_1, v_2)}{\mathsf{fst}(e) \Downarrow v_1} \qquad \frac{e \Downarrow (v_1, v_2)}{\mathsf{snd}(e) \Downarrow v_1} \qquad \frac{e_1 \Downarrow v_1 \qquad \cdots \qquad e_n \Downarrow v_n}{\langle \ell_1 = e_1, \ldots, \ell_n = e_n \rangle \Downarrow \langle \ell_1 = v_1, \ldots, \ell_n = v_n \rangle} \qquad \frac{e \Downarrow \langle \ell_1 = v_1, \ldots, \ell_n = v_n \rangle}{e.\ell_j \Downarrow v_j}$$

$$\frac{e \Downarrow v}{\mathsf{select}\ \ell\ e \Downarrow \mathsf{select}\ \ell\ v} \qquad \frac{e \Downarrow \mathsf{select}\ \ell_j\ v_j \qquad e_j[v_j/x_j] \Downarrow v}{\mathsf{case}\ e\ \mathsf{of}\ \{\ell_1\ x_1 \to e_1, \ldots, \ell_n\ x_n \to e_n\} \Downarrow v} \qquad \frac{e_1 \Downarrow v_1 \qquad \cdots \qquad e_n \Downarrow v_n}{\{\!|e_1, \ldots, e_n|\!\} \Downarrow \{\!|v_1, \ldots, v_n|\!\}}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\mathsf{flatMap}(e_1, e_2) \Downarrow \text{flatMap of the multiset } v_1 \text{ and function } v_2} \qquad \frac{e_1 \Downarrow \mathsf{true} \qquad e_2 \Downarrow v}{\mathsf{when}(e_1, e_2) \Downarrow v} \qquad \frac{e_1 \Downarrow \mathsf{false}}{\mathsf{when}(e_1, e_2) \Downarrow \{\!|\}\!\}}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\mathsf{sum}(e_1, e_2) \Downarrow \text{sum of two multisets } v_1, v_2} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\mathsf{diff}(e_1, e_2) \Downarrow \text{difference of two multisets } v_1, v_2}$$

$$\frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{\mathsf{count}(e_1, e_2) \Downarrow \text{count of } v_2 \text{ in the multiset } v_1}$$

Figure 5: Evaluation rules

```
    case Lambda(x, e) => FunV(x, e)
    case Rec(f, x, e) => RecV(f, x, e)
```

Exercise 6. *Implement the evaluator* `eval`*, whose type signature is as follows:*

```
def eval(expr: Expr): Value
```

[20 marks]

Change Log

- V1.1:
  - Changed occurrences of $\leq$ to $<:$ in some rules
  - Fixed misspelled package name in `Interpreter.scala`
  - Added clarifications about intended behavior of bag operations
- v1.1.1: (no changes to code)
  - Fixed error in variant subtyping example
  - Added sentences clarifying the expected behavior of `tyCheck` and `tyInfer`.
  - Added clarification that comprehension item lists can be empty, and explanation of the notation $\{|e \mid |\}$.
- v1.1.2: (no changes to code)
  - Fixed incorrect statement of the subsumption rule
  - Clarified treatment of evaluation of value forms
- v1.1.3 (no changes to code)
  - Corrected description of the substitution function and relationship to assignment 2
  - Corrected = to == in bag comprehension example
  - Clarified relationship to desugaring in assignment 2
  - Fixed typo in checking rule for rec
- v1.2
  - Reinstated tests/InterpreterTests.scala testing file which was inadvertently left out
  - Clarified that rules for evaluating values to themselves include rules for evaluating expression forms to equivalent value forms.
  - Added an example illustrating that flatMap needs to be able to handle functions that are not syntactic lambda-abstractions.
- v1.3
  - Fixed a typo and broken reference
  - Fixed `InterpreterTests.scala`
  - Clarified that the rules for substitution into comprehensions are not fully capture avoiding
  - Clarified that substitution needs to be performed on already-desugared terms to avoid this problem
- v1.3.1 (no changes to the code)
  - Fixed description of ‘when‘ in section 4 which was incon sistent with the rules (swapped roles of arguments)