Sums and Recursion

In this assignment, we will develop Recursive ALFA, an extension of ALF_p from A4 with sum types, recursive expressions, and recursive types.

Please submit answers to the 4 written questions on Gradescope under the "A5 Written" assignment. The 2 coding questions will be via the Learn OCaml platform, which stores your code on our server (see below). Your use of a late day will be determined by your A5 Written submission time on Gradescope. The version of your code stored on the server at the regular deadline will be used for grading unless you also submit the written portion late.

1 Syntax

We can define the structural syntax and the on-paper concrete syntax for Recursive ALFA expressions as shown in Fig. 1. Here, n ranges over integers and x and y range over expression variables, and t over type variables. The notation $x.e$ is used to indicate that the variable x is bound for use in e, and similarly $x.y.e$ for two variables, and $t.\tau$ for type variables. We write $\tau?$ for optional type annotations. We will use the concrete syntax in the remainder of the handout.

2 Sums

Booleans are the simplest sum type, but ALFA also supports more general binary sum types. Sum types are all about making choices, one of the most fundamental concepts in computation. For example, consider the OCaml sum (a.k.a. variant) type `bread` from A1:

```
type bread =
| White
| Wheat
| Multigrain of int
```

Here, the programmer has three choices when constructing a value of `bread` type: they can use the constants `White` or `Wheat`, or they can apply `Multigrain` to an integer. That integer is *conditional data*, i.e. it is only available in one of the three cases.

If we pass the choice we make to a function, that function can *case analyze* on its input using OCaml's **`match`** expression:

```
let numgrains(b : bread) =
  match b with
  | White -> 0
  | Wheat -> 1
  | Multigrain n -> n
```

Many things in the world involve making one of several choices, with additional data provided only for some of these choices, and then passing that choice on to someone else who will make different decisions depending on which choice you made! For example, you've encountered many web forms in your life that ask you a question like *"How did you find*

			Structural Form	**Concrete Form**
Typ	τ	::=	Num	Num
		|	Bool	Bool
		|	Arrow$(\tau_{\mathsf{in}}, \tau_{\mathsf{out}})$	$\tau_{\mathsf{in}} \to \tau_{\mathsf{out}}$ (right assoc., precedence 1)
		|	Prod(τ_1, τ_2)	$\tau_1 \times \tau_2$ (right assoc., precedence 3)
		|	Unit	Unit
				1 (on paper only)
		|	Sum(τ_L, τ_R)	$\tau_L + \tau_R$ (right assoc., precedence 2)
		|	TVar$[t]$	t
		|	Rec$(t.\tau)$	rec t is τ (prefix, precedence 0)
Expr	e, v	::=	NumLit$[n]$	$\underline{n}$
		|	UnOp$[u](e)$	ue
		|	BinOp$[b](e_1, e_2)$	$e_1\, b\, e_2$
		|	True	True
		|	False	False
		|	If(e_1, e_2, e_3)	if e_1 then e_2 else e_3 (prefix, precedence 0)
		|	Var$[x]$	x
		|	LetAnn$(\tau?, e_1, x.e_2)$	let $x : \tau$ be e_1 in e_2 (prefix, precedence 0)
				let x be e_1 in e_2
		|	Fix$(\tau?, x.e)$	fix $(x : \tau) \to e$ (prefix, precedence 0)
				fix $x \to e$
		|	Fun$(\tau_{\mathsf{in}}?, x.e)$	fun $(x : \tau_{\mathsf{in}}) \to e$ (prefix, precedence 0)
				fun $x \to e$
		|	Ap(e_1, e_2)	$e_1\ e_2$ (left assoc., precedence 6)
		|	Pair(e_1, e_2)	(e_1, e_2)
		|	Triv	()
		|	PrjL(e)	e.fst (postfix, precedence 8)
		|	PrjR(e)	e.snd (postfix, precedence 8)
		|	LetPair$(e_1, x.y.e_2)$	let (x, y) be e_1 in e_2 (prefix, precedence 0)
		|	InjL(e)	L e (prefix, precedence 5)
		|	InjR(e)	R e (prefix, precedence 5)
		|	Case$(e_1, x.e_2, y.e_3)$	case e_1 of L$(x) \to e_2$ else R$(y) \to e_3$ (prefix, precedence 0)
		|	Roll(e)	roll(e)
		|	Unroll(e)	unroll(e)
UnOp	u	::=	OpNeg	$-$ (precedence 7)
BinOp	b	::=	OpLt	$<$ (left assoc., precedence 2)
		|	OpGt	$>$ (left assoc., precedence 2)
		|	OpEq	$=?$ (left assoc., precedence 2)
		|	OpPlus	$+$ (left assoc., precedence 3)
		|	OpMinus	$-$ (left assoc., precedence 3)
		|	OpTimes	$*$ (left assoc., precedence 4)

Figure 1: Syntax of Recursive ALFA

us?". The answer can be expressed as a sum type, where only some of the choices have conditional data:

```
type referral =
  | PersonalRecommendation of person
  | Search
  | Email
  | Other of string
```

In ALFA, we will include a minimal version of this feature. In particular, we will support only *binary sums*.

To understand binary sums, it is helpful to imagine a world where OCaml did not allow you to define your own sum types. Instead, you had only the following definition with exactly two constructors, named `L` and `R` respectively, each of which must take an argument given by the two type parameters, `'left` and `'right`. In OCaml type definitions, parameters are given before the type name, using parentheses and commas when there is more than one. (There is no connection with product types here, which use the `*` symbol!) It might help to think about `BinarySum<Left, Right>` from C++, which takes in two type parameters separated by a comma.

```
type ('left, 'right) binarysum =
| L of 'left
| R of 'right
```

So for example, the type `(int, float) binarysum` has values `L n` for `n : int` and `R r` for `r : float`, because we substituted the type `int` for the type variable `'left` and `float` for `'right` in the type definition above.

We can use `binarysum` to define a type `bread2` that is isomorphic to `bread` as follows:

```
type bread2 = (unit, (unit, int) binarysum) binarysum
```

Notice that we use two binary sums to encode a three-way choice. The outer binary sum distinguishes between `White` and non-`White`, and the inner between `Wheat` and `Multigrain`. We use the `unit` type to handle the situation where there is no other data of interest to supply to the corresponding constructor. To see the correspondence more clearly, we can define the following emulated "constructors" for `bread2`, corresponding to the actual constructors of the original `bread` type:

```
let white : bread2 = L ()
let wheat : bread2 = R (L ())
let multigrain : int -> bread2 = fun n -> R (R n)
```

Two types are isomorphic, written $\tau_1 \cong \tau_2$, if we can define functions *into* and *out* (called the "witnesses" to the isomorphism) that compose in either direction to the identity function.

$$\begin{array}{rcll} \mathit{into} & : & \tau_1 \to \tau_2 & \\ \mathit{out} & : & \tau_2 \to \tau_1 & \\ \mathit{out}\ (\mathit{into}\ x) & \equiv & x & \text{for all } x : \tau_1 \\ \mathit{into}\ (\mathit{out}\ y) & \equiv & y & \text{for all } y : \tau_2 \end{array}$$

We can show that `bread` $\cong$ `bread2` with the following witnesses.

```
let into : bread -> bread2 = fun b ->
  match b with
  | White -> white
  | Wheat -> wheat
  | Multigrain n -> multigrain n

let out : bread2 -> bread = fun b2 ->
  match b2 with
  | L () -> White
  | R (L ()) -> Wheat
  | R (R n) -> Multigrain n
```

You can prove the necessary equivalences above for yourself as an exercise.

This exercise motivates our definitions for binary sums in ALFA. First, to give the programmer a choice when they introduce a value of a binary sum type, we define two introductory forms, called *left injection*, written $\mathsf{L}\ e$, and *right injection*, written $\mathsf{R}\ e$. Both of these can be given the sum type, $\tau_L + \tau_R$, and whether it is a left or right injection is what determines whether e must be of type τ_L or τ_R, respectively. So you can think of $\tau_L + \tau_R$ in ALFA as corresponding to `('left, 'right) binarysum` in the OCaml exercise above.

$$\frac{\Gamma \vdash e \Leftarrow \tau_L}{\Gamma \vdash (\mathsf{L}\ e) \Leftarrow (\tau_L + \tau_R)}\ \text{(A-InjL)} \qquad \frac{\Gamma \vdash e \Leftarrow \tau_R}{\Gamma \vdash (\mathsf{R}\ e) \Leftarrow (\tau_L + \tau_R)}\ \text{(A-InjR)}$$

We cannot define synthesis rules for injections because that would require guessing the type for the injection that was not chosen.

Injections are values once their arguments are values, i.e. they are the introduction forms for sum types:

$$\frac{e \Downarrow v}{\mathsf{L}\ e \Downarrow \mathsf{L}\ v}\ \text{(E-InjL)} \qquad \frac{e \Downarrow v}{\mathsf{R}\ e \Downarrow \mathsf{R}\ v}\ \text{(E-InjR)}$$

Note that you can't have injections that have no conditional data, like the constants `White` and `Wheat` in OCaml. If there is no data of interest, we can just use the trivial value of **`Unit`** type as we did above! So in ALFA, we can express the `bread` type from OCaml using a binary sum type, which we will abbreviate bread in the remainder of the document (ALFA doesn't itself have support for type abbreviations internal to the language):

$$\mathsf{bread} = \mathsf{Unit} + (\mathsf{Unit} + \mathsf{Num})$$

For convenience, we can define the "constructors" as follows:

$$\begin{array}{l}\mathsf{let}\ \mathit{White} : \mathsf{bread}\ \mathsf{be}\ \mathsf{L}\ ()\ \mathsf{in} \\ \mathsf{let}\ \mathit{Wheat} : \mathsf{bread}\ \mathsf{be}\ \mathsf{R}\ (\mathsf{L}\ ())\ \mathsf{in} \\ \mathsf{let}\ \mathit{Multigrain} : \mathsf{Num} \to \mathsf{bread}\ \mathsf{be}\ \mathsf{fun}\ n \to \mathsf{R}\ (\mathsf{R}\ n)\ \mathsf{in}\ \ldots\end{array}$$

The idea is that binary sum types do not bother with names for the constructors – instead, there are always two constructors, L and R. If you need more than two options, you can

chain together the sums as we did above. So `White` corresponds to L, `Wheat` corresponds to R L, and `Multigrain` corresponds to R R). It's sort of a binary encoding! (In fact, that is often how it is implemented in a compiler.) In addition, each constructor *must* take an argument. In OCaml, `White` and `Wheat` do not take arguments, so in ALFA, we use the trivial value, (), of Unit type.

Given an expression that synthesizes sum type, we can case analyze on it using case. (Think **match** expressions in OCaml, but again in their most primitive form.) In analytic position, both branches of the case expression must analyze against the given type. In synthetic position, both branches must synthesize the same type, just like if expressions. Each branch has access to the argument of the corresponding injection via a specified variable (x and y in the rules below).

$$\frac{\Gamma \vdash e_{\text{scrut}} \Rightarrow \tau_\ell + \tau_r \qquad \Gamma, x : \tau_\ell \vdash e_\ell \Leftarrow \tau \qquad \Gamma, y : \tau_r \vdash e_r \Leftarrow \tau}{\Gamma \vdash (\mathsf{case}\ e_{\text{scrut}}\ \mathsf{of}\ \mathsf{L}(x) \to e_\ell\ \mathsf{else}\ \mathsf{R}(y) \to e_r) \Leftarrow \tau} \quad \text{(A-Case)}$$

$$\frac{\Gamma \vdash e_{\text{scrut}} \Rightarrow \tau_\ell + \tau_r \qquad \Gamma, x : \tau_\ell \vdash e_\ell \Rightarrow \tau \qquad \Gamma, y : \tau_r \vdash e_r \Rightarrow \tau}{\Gamma \vdash (\mathsf{case}\ e_{\text{scrut}}\ \mathsf{of}\ \mathsf{L}(x) \to e_\ell\ \mathsf{else}\ \mathsf{R}(y) \to e_r) \Rightarrow \tau} \quad \text{(S-Case)}$$

Evaluation proceeds by evaluating the scrutinee to an injection, then taking the appropriate branch and substituting in the conditional data.

$$\frac{e_{\text{scrut}} \Downarrow \mathsf{L}\ v_{\text{data}} \qquad [v_{\text{data}}/x]e_\ell \Downarrow v}{(\mathsf{case}\ e_{\text{scrut}}\ \mathsf{of}\ \mathsf{L}(x) \to e_\ell\ \mathsf{else}\ \mathsf{R}(y) \to e_r) \Downarrow v} \quad \text{(E-Case-L)}$$

$$\frac{e_{\text{scrut}} \Downarrow \mathsf{R}\ v_{\text{data}} \qquad [v_{\text{data}}/y]e_r \Downarrow v}{(\mathsf{case}\ e_{\text{scrut}}\ \mathsf{of}\ \mathsf{L}(x) \to e_\ell\ \mathsf{else}\ \mathsf{R}(y) \to e_r) \Downarrow v} \quad \text{(E-Case-R)}$$

So consider again the OCaml function `numgrains`, repeated below:

```
let numgrains (b : bread) =
  match b with
  | White -> 0
  | Wheat -> 1
  | Multigrain n -> n
```

We can write this using case analysis in ALFA as follows (using the textual syntax here for convenience):

```
let numgrains be fun (b : bread) ->
  case b of
  L(_) -> _0_
  else R(y) ->
    case y of
    L(_) -> _1_
    else R(n) -> n
in ...
```

Notice that we had to nest case analyses, because ALFA has only binary sums. We used _ for variables that stand for the trivial value.

Given the syntactic awkwardness, no real language will have just binary sums in this form, just like no real language will support only binary products, i.e. pairs. As with binary products, the reason we are studying binary sums is because they capture the *essence* of a wide variety of other language features. OCaml exposes product types via tuples and records and sum types via its variant types and pattern matching system as just discussed. Object-oriented languages, if you squint enough, support products and sums using objects, enumerations, and class-based inheritance together with virtual methods (If you're curious, look up the *visitor pattern* in your favorite OO language. It allows only a pale imitation of proper variant types and pattern matching. The omission or obfuscation of sum types in mainstream languages is the source of significant problems. These languages make it difficult to make simple choices!)

Question 1: Burger King (Written, 15%) Convert the following OCaml definitions to ALFA:

```
type burger =
| Plain of int           (* denotes number of patties *)
| Royale of bool         (* denotes w/ or w/o cheese *)
| BigKahuna
| BLT of int * bool      (* denotes num bacons and if lettuce
                          * (always tomato) *)

let calories (burg : burger) =
  match burg with
  | Plain numPatties -> 230 + numPatties * 110
  | Royale hasCheese -> 910 + (if hasCheese then 90 else 0)
  | BigKahuna -> 1540
  | BLT (numBacon, hasLettuce) ->
      550 + (if numBacon > 0
        then (if hasLettuce then 0 else 330)
        else 0)
```

First, define the burger type as an ALFA sum type:

burger=

Then define the four constructors:

let $Plain$: Num → burger be ?? in
let $Royale$: Bool → burger be ?? in
let $BigKahuna$: burger be ?? in
let BLT : Num × Bool → burger be ?? in ...

Finally, define the `calories` function.

let $calories$: burger → Num be ?? in ...

HINT: Make sure you are using ALFA's syntax only, not OCaml's syntax! (You can check using the `parse` function in Learn OCaml).

HINT: Make sure the expressions that appear in the holes (??) above are well-typed when they appear in the given let bindings, according to the rules above! Check carefully, perhaps using your solution to the coding problem below!

3 Recursive Expressions (i.e. Fixpoints)

ALFA did not support recursion. Recursive ALFA adds support for recursion using a very general construct: *fixpoint expressions*, a.k.a. *self-referential expressions*, $\mathsf{fix}\ (x : \tau) \to e_{\text{body}}$. The variable x stands for the fixpoint expression itself in the fixpoint's body:

$$\frac{[(\mathsf{fix}\ (x : \tau) \to e_{\text{body}})/x]e_{\text{body}} = e_{\text{unrolled}} \qquad e_{\text{unrolled}} \Downarrow v}{(\mathsf{fix}\ (x : \tau) \to e_{\text{body}}) \Downarrow v}$$

or written more concisely:

$$\frac{[(\mathsf{fix}\ (x : \tau) \to e_{\text{body}})/x]e_{\text{body}} \Downarrow v}{(\mathsf{fix}\ (x : \tau) \to e_{\text{body}}) \Downarrow v}\ \text{(E-Fix)}$$

How can this not cause an infinite recursion (non-terminating evaluation)? Well, if the body of the fixpoint is a function, then the substitution step will move the fixpoint inside the function. This is called "unrolling" the fixpoint (see e_{unrolled} in the first version of the rule above). The result is a function, which is a value, and so evaluation stops!

For example, we can define the recursive function `fib` in ALFA as follows:

```
let fib : Num -> Num be
  fix fib ->
    fun n -> if n < 2 then 1
             else fib(n - 1) + fib(n - 2)
in ... (* the let-bound fib is in scope only here *)
```

Notice how the self-reference is brought into scope within the definition using the fixpoint. Otherwise, it would not be in scope because **`let`** only brings the variable in scope in the body of the let expression, i.e. after the **`in`**. We need not have used the variable `fib` as the self-reference here, i.e. the following is an equivalent definition using `self` instead of `fib` for the self-reference:

```
let fib : Num -> Num be
  fix self ->
    fun n -> if n < 2 then 1
             else self(n - 1) + self(n - 2)
in ... (* fib is in scope only here *)
```

This combination of **`let`** and **`fix`** is how recursive function definitions work under the hood – OCaml / Hazel just inserts the fixpoint for you automatically.

We can break down this definition as follows, using some abbreviations that will be helpful for us:

$$e_{\text{fib}}{=}\mathsf{fix}\ fib \to e_{\text{body}}$$
$$e_{\text{body}}{=}\mathsf{fun}\ n \to \mathsf{if}\ n < \underline{2}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ fib(n-\underline{1}) + fib(n-\underline{2})$$

The value of e_{fib} according to the rule above is its unrolling, i.e.

$$v_{\text{fib}}{=}[e_{\text{fib}}/fib]e_{\text{body}} = \mathsf{fun}\ n \to \mathsf{if}\ n < \underline{2}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ e_{\text{fib}}(n-\underline{1}) + e_{\text{fib}}(n-\underline{2})$$

Notice that the variables fib have been replaced with the definition of fib itself, e_{fib}, and the outermost form is now fun rather than fix. The fixpoints have been moved inside, where it is not necessary for them to be further unrolled until the function is later applied and the recursive branch is taken. For example:

$$\begin{aligned}
& & & e_{\mathsf{fib}}\ \underline{2} \\
\text{(Unroll)} & \equiv & & v_{\mathsf{fib}}\ \underline{2} \\
\text{(Apply)} & \equiv & & \mathsf{if}\ \underline{2} < \underline{2}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ e_{\text{fib}}\ (\underline{2}-\underline{1}) + e_{\text{fib}}\ (\underline{2}-\underline{2}) \\
\text{(Guard Eval)} & \equiv & & \mathsf{if\ False\ then}\ \underline{1}\ \mathsf{else}\ e_{\text{fib}}\ (\underline{2}-\underline{1}) + e_{\text{fib}}\ (\underline{2}-\underline{2}) \\
\text{(Branch)} & \equiv & & e_{\text{fib}}\ (\underline{2}-\underline{1}) + e_{\text{fib}}\ (\underline{2}-\underline{2}) \\
\text{(Unroll)} & \equiv & & v_{\text{fib}}\ (\underline{2}-\underline{1}) + e_{\text{fib}}\ (\underline{2}-\underline{2}) \\
\text{(Arg Eval)} & \equiv & & v_{\text{fib}}\ \underline{1} + e_{\text{fib}}\ (\underline{2}-\underline{2}) \\
\text{(Apply)} & \equiv & & (\mathsf{if}\ \underline{1} < \underline{2}\ \mathsf{then}\ \underline{1}\ \mathsf{else}\ ...) + e_{\text{fib}}\ (\underline{2}-\underline{2}) \\
\text{(Guard Eval)} & \equiv & & (\mathsf{if\ True\ then}\ \underline{1}\ \mathsf{else}\ ...) + e_{\text{fib}}\ (\underline{2}-\underline{2}) \\
\text{(Branch)} & \equiv & & \underline{1} + e_{\text{fib}}\ (\underline{2}-\underline{2}) \\
& & ... &
\end{aligned}$$

Question 2: Remainder (Written, 15%) Consider the gcd function in OCaml, which subtracts the smaller value from the larger value recursively until both values are equal, returning the GCD of the two original inputs:

```
let rec gcd a b =
  if a =? b then a
  else if a > b then gcd (a - b) b
  else gcd a (b - a)
```

1. Fill in the hole in the following abbreviations to define the gcd function, which we abbreviate e_{gcd}, in ALFA, again using correct syntax and types:

$$e_{\mathsf{gcd}}{=}\mathsf{fix}\ (gcd : \mathsf{Num} \to \mathsf{Num} \to \mathsf{Num}) \to e_{\mathsf{fun}}$$
$$e_{\mathsf{fun}}{=}\mathsf{fun}\ a \to \mathsf{fun}\ b \to\ ??$$

2. What is v_{gcd}, the value of e_{gcd}, according to the rules of ALFA above? You may use the abbreviations above in your answer if you would like, but perform any necessary substitutions.

3. Show through a series of equivalences that $e_{\mathsf{gcd}}\ \underline{15}\ \underline{6} \equiv \underline{3}$. You may use the abbreviations above in your answer. You may also use the abbreviation v_{gcd} to stand for your answer to part 2. Each step should be justified by one of the following reasons:

   - (Unroll): Unrolling of a fixpoint
   - (Arg Eval): Evaluation of an argument of a function application
   - (Apply): Substitution of argument value into a function body.
   - (Projection): Projection of a tuple using let pair or projection operations.
   - (Guard Eval): Evaluation of the guard of an **`if`** expression to a value.
   - (Branch): Choice of if branch

4 Recursive Types

Recursive types, $\mathsf{rec}\ t\ \mathsf{is}\ \tau_{\mathsf{body}}$, are similar to recursive expressions: they are types that contain self-references. The type variable t is the self reference, i.e. it stands for the recursive type itself in the recursive type's body, τ_{body}.

Let us consider how the OCaml `intlist` type:

```
type intlist =
  | Nil
  | Cons of int * intlist
```

corresponds to the following recursive type, which we will abbreviate NumList below:

$$\mathsf{NumList}{=}\mathsf{rec}\ numlist\ \mathsf{is}\ (1 + \mathsf{Num} \times numlist)$$

The `Nil` constructor corresponds to the left arm of the sum type in the body of NumList. Because there is no data associated with the `Nil` constructor, we use the type 1, i.e. Unit, of trivial data, as discussed in A4. The `Cons` constructor corresponds to the right arm of the sum type. Its argument type, **`int * intlist`**, corresponds to the ALFA product type $\mathsf{Num} \times numlist$. The type variable $numlist$ is the self-reference.

How do we create expressions of types like NumList? We cannot simply use left or right injection, because those are the introduction forms for sum types, i.e. types of the form $\tau_1 + \tau_2$, whereas we want a value of a recursive type, which just happens to have a sum type in its body. We need to unroll the recursion, just like we did with fixpoints above. So we need an introductory form for recursive types that takes mediates the unrolling. This introductory form for recursive types is $\mathsf{roll}(e)$.

Similarly, we need elimination operations on values of recursive type, and we cannot use existing operations like case analysis directly. Instead, we need an elimination operation specifically for recursive types. The elimination form for recursive types is $\mathsf{unroll}(e)$, because it acts on rolled up values.

Their typing rules are given below.

$$\frac{\Gamma \vdash e_{\mathsf{body}} : [(\mathsf{rec}\ a\ \mathsf{is}\ \tau_{\mathsf{body}})/a]\tau_{\mathsf{body}}}{\Gamma \vdash \mathsf{roll}(e_{\mathsf{body}}) : \mathsf{rec}\ a\ \mathsf{is}\ \tau_{\mathsf{body}}} \quad \text{(T-Roll)}$$

$$\frac{\Gamma \vdash e : \mathsf{rec}\ a\ \mathsf{is}\ \tau_{\mathsf{body}}}{\Gamma \vdash \mathsf{unroll}(e) : [(\mathsf{rec}\ a\ \mathsf{is}\ \tau_{\mathsf{body}})/a]\tau_{\mathsf{body}}} \quad \text{(T-Unroll)}$$

Notice how the rules instantiate the self-reference in the type's body by substitution. This is called a one-step unrolling of the type. The introductory form, $\mathsf{roll}(e_{\mathsf{body}})$, takes an expression e_{body} of the unrolled type as its argument and turns it into an expression of the "rolled up" recursive type. The elimination form goes in the other direction, i.e. it takes as input an expression of recursive type and unrolls it, so its type is the unrolling of the recursive type.

During evaluation, unrolls simply eliminate rolls – all of the action is in the type system:

$$\frac{e \Downarrow v}{\mathsf{roll}(e) \Downarrow \mathsf{roll}(v)} \quad \text{(E-Roll)} \qquad \frac{e \Downarrow \mathsf{roll}(v)}{\mathsf{unroll}(e) \Downarrow v} \quad \text{(E-Unroll)}$$

For example, the empty list (corresponding to `Nil`) is $\mathsf{roll}(\mathsf{L}\ ())$ because the unrolling of the recursive type Numlist is the following sum type:

$$[\mathsf{NumList}/\mathit{numlist}](1 + \mathsf{Num} \times \mathit{numlist}) = 1 + \mathsf{Num} \times \mathsf{NumList}$$

Notice how the unrolling of the recursive type NumList is a sum type, rather than a recursive type. The recursive type, which we are just abbreviating as NumList (but could write out fully), has moved inside. Notice how this is similar to how fix works at the expression level!

Cons can similarly be expressed as an uncurried function of type $\mathsf{Num} \times \mathsf{NumList} \to \mathsf{NumList}$:

$$\mathsf{fun}\ \mathit{hdtl} \to \mathsf{roll}(\mathsf{R}\ \mathit{hdtl})$$

If you are unclear about how this works, try carefully deriving that the expression above has the function type $\mathsf{Num} \times \mathsf{NumList} \to \mathsf{NumList}$.

To avoid having to apply **`roll`** manually, we can define the constructors as values when we write ALFA programs:

```
let Nil : NumList be roll(L ()) in
let Cons : Num -> NumList -> NumList be
  fun hd -> fun tl -> roll(R (hd, tl))
```

Pattern matching on a list in OCaml corresponds to case analysis on the *unrolling* in ALFA. For example, we can define the length function as the value of the fixpoint below:

```
let length : NumList -> Num be
  fix length -> fun xs ->
    case unroll(xs) of
    L(x) -> 0
    else R(y) -> 1 + length y.1
```

We needed to apply **unroll** to go from xs, which has the recursive type abbreviated NumList, to its unrolling, the sum type given above, which is suitable for case analysis.

We can engage with the same exercise for *any* recursive type from OCaml, not just lists. For example, recall the definition of binary trees from lecture, here specialized for integers:

```
type btree =
| Leaf of int
| Node of btree * btree
```

This corresponds to the Recursive ALFA type BTree defined below, again by simply translating each constructor to an arm of the sum.

$$\mathsf{BTree}{=}\mathsf{rec}\ \mathit{btree}\ \mathsf{is}\ (\mathsf{Num} + \mathit{btree} \times \mathit{btree})$$

Again, we can define the two constructors as functions.

```
let Leaf : Num -> BTree be fun n -> roll(L n)
let Node : (BTree * BTree) -> BTree be
  fun children -> roll(R children)
```

It turns out that the primitive features of essentially all languages can similarly be understood as some (often more *ad hoc*) combination of sums, products, and recursive types. For example, any finite collection of subclasses of an abstract base class form a sum type, virtual method dispatch allows for case analysis, the fields form a product type, and this/self references are a manifestation of the fact that classes are recursive types.

Question 3: Self-Hosting (Written, 20%) It turns out that with recursive types, Recursive ALFA is powerful enough to be *self-hosting*, i.e. it is possible to implement a Recursive ALFA typechecker and evaluator using Recursive ALFA itself (just like C can be and is implemented using C and OCaml can and is implemented using OCaml itself!) This is because tree types, including binary trees like the example above but also richer syntax tree types, are recursive sum types.

We used this fact in the coding exercises in the previous assignment – the syntax of ALF expressions and types was expressed in Hazel using its support for recursive sum types. So you can encode the syntax of AL, ALF, ALFA, and now even Recursive ALFA types and expressions as Recursive ALFA expressions of a corresponding recursive type!

We won't ask you to do a full self-hosted implementation of Recursive ALFA, but as an exercise, start by defining a Recursive ALFA type abbreviated Typ whose values encode the structural syntax of Recursive ALFA types themselves (see Fig. 1). Then fill in the holes, ?, to define each of the constructors. Finally, define a function that determines whether the given Typ is a compound type (i.e. an arrow, product, sum, or recursive type).

Don't panic! Recall that the structural syntax for types defined in the syntax table is simply a recursive datatype, just like NumList and BTree above, so this is less tricky than it might initially sound.

It might help to refer to the OCaml type definition corresponding to the structural syntax for types given in the Learn-OCaml assignment. Then, translate that to a Recursive ALFA type and corresponding constructors, just like we did for NumList and BTree above.

You may assume a type Id that classifies encodings of Recursive ALFA identifiers into Recursive ALFA[1]

Id=(assume some suitable definition, omitted)
Typ= ?

```
let Num : Typ be ? in
let Bool : Typ be ? in
let Arrow : Typ * Typ -> Typ be ? in
let Prod : Typ * Typ -> Typ be ? in
let Unit : Typ be ? in
let Sum : Typ * Typ -> Typ be ? in
let TVar : Id -> Typ be ? in
let Rec : Id * Typ -> Typ be ? in
let isCompound : Typ -> Bool be
  fun (ty: Typ) -> ? in
```

Question 4: Recursive ALFA Evaluator (LearnOCaml, 10%) Implement an evaluator for Recursive ALFA. We have already included the cases for constructs from prior assignments, so you only need to implement fixpoints, roll, and unroll according to the rules.

The first part of your job is to implement substitution for the forms new to ALFA ML.

```
subst : Exp.t -> Identifier.t -> Exp.t -> Exp.t
```

Next, you should use substitution to implement a function `eval : Exp.t -> Value.t` that correctly implements the evaluation semantics assuming it is given a closed, well-typed expression. In other words, the following correctness theorem should be true:

Theorem 1 (Evaluator Correctness)**.** `eval` $\lfloor e \rfloor \equiv \lfloor v \rfloor$ *iff* $e \Downarrow v$ *and* $\vdash e : \tau$.

Notice the assumption that the input is a closed, well-typed expression. Your implementation should `raise IllTyped` when the OCaml exhaustiveness checker forces you to consider a case that is ruled out by the typing assumption.

Careful: your evaluator can fail to terminate if the given Recursive ALFA expression has unfounded recursion. Don't write any tests that cause this to happen.

4.1 Type Validity

We can now move on to semantics. Because we have type variables, t, in Recursive ALFA, we need to make sure the programmer has not referred to type variables that have not yet been bound. We do so by defining a *type validity judgement*, $\Delta \vdash \tau$ type, pronounced "τ is a valid type, assuming Δ". Here, Δ is the *type validation context*, and it operates much like the typing context Γ that we learned about in A4 in that it tracks assumptions about variables. Here, the assumption we want to track is of the form t type, i.e. that the type

[1]For example, τ_{id} might be a list of characters, and characters might be encoded using Num.

variable t does actually stand for a valid type. Type validation contexts Δ are finite sets of these assumptions, i.e. they take the following form:

$$t_1 \text{ type}, \cdots, t_n \text{ type} \quad (n \geq 0)$$

We say that a type is closed and valid if it is valid under the empty type validation context. For example, rec a is $(b+a)$ is not a closed, valid type because we need the assumption that the free variable b stands for a type. We can, however, derive that $b \text{ type} \vdash (\text{rec } a \text{ is } (b+a)) \text{ type}$ using the following rules. Remember, all we are doing is checking that type variables are bound before they are used here.

Base types like Num, Bool, and Unit are valid under any type validation context:

$$\frac{}{\Delta \vdash \text{Num type}} \text{ (TV-Num)} \qquad \frac{}{\Delta \vdash \text{Bool type}} \text{ (TV-Bool)} \qquad \frac{}{\Delta \vdash 1 \text{ type}} \text{ (TV-Unit)}$$

The rules for arrow types, binary product types, and binary sum types thread through the type validation context inductively:

$$\frac{\Delta \vdash \tau_{\text{in}} \text{ type} \qquad \Delta \vdash \tau_{\text{out}} \text{ type}}{\Delta \vdash \tau_{\text{in}} \to \tau_{\text{out}} \text{ type}} \text{ (TV-Arrow)} \qquad \frac{\Delta \vdash \tau_\ell \text{ type} \qquad \Delta \vdash \tau_r \text{ type}}{\Delta \vdash \tau_\ell \times \tau_r \text{ type}} \text{ (TV-Prod)}$$

$$\frac{\Delta \vdash \tau_\ell \text{ type} \qquad \Delta \vdash \tau_r \text{ type}}{\Delta \vdash \tau_\ell + \tau_r \text{ type}} \text{ (TV-Sum)}$$

The type validation context is extended when checking the validity of recursive types. Extending Δ makes t available for use in the type body, τ.

$$\frac{\Delta, t \text{ type} \vdash \tau \text{ type}}{\Delta \vdash (\text{rec } t \text{ is } \tau) \text{ type}} \text{ (TV-Rec)}$$

We look in the type validation context for a corresponding assumption when we encounter a type variable. In other words, it has to have been bound further down the derivation (i.e. further "up"/"out" in the program). This rule should remind you of the variable typing rule from A4, here at the type level.

$$\frac{t \text{ type} \in \Delta}{\Delta \vdash t \text{ type}} \text{ (TV-TVar)}$$

We use this judgement whenever we encounter a type annotation and need to make sure it is a closed and valid type. In Recursive ALFA, we can have optional type annotations on functions, fixpoints, and let expressions:

$$\frac{\vdash \tau_{\text{in}} \text{ type} \qquad \Gamma, x : \tau_{\text{in}} \vdash e : \tau_{\text{out}}}{\Gamma \vdash (\text{fun } (x : \tau_{\text{in}}) \to e) : \tau_{\text{in}} \to \tau_{\text{out}}} \text{ (T-FunAnn)} \qquad \frac{\vdash \tau \text{ type} \qquad \Gamma, x : \tau \vdash e : \tau}{\Gamma \vdash (\text{fix } (x : \tau) \to e) : \tau} \text{ (T-FixAnn)}$$

$$\frac{\vdash \tau_1 \text{ type} \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash (\text{let } x \ : \ \tau_1 \text{ be } e_1 \text{ in } e_2) : \tau_2} \text{ (T-LetAnn)}$$

Question 5: Bottomless Recursion (Written, 10%) Derive the following judgement using the rules above, supplemented with rules from the previous assignment as needed. This is an example showing that a fixpoint without a base case can be given any type at all, here $\mathsf{NumList} \to \mathsf{BTree}$ as an example. (If we attempt to apply it to an argument, that function application will fail to terminate, but that doesn't stop us from giving it a type!)

$$\vdash (\mathsf{fix}\ (f : \mathsf{NumList} \to \mathsf{BTree}) \to \mathsf{fun}\ (x : \mathsf{NumList}) \to f(x)) : \mathsf{NumList} \to \mathsf{BTree}$$

You can expand abbreviations like NumList and BTree which were defined earlier in the handout whenever it is convenient to do so. Since both are abbreviations of recursive types, your derivations will need to apply the (TV-Rec) rule. You can define derivation abbreviations as needed as in prior derivations. You can reuse a derivation abbreviation in multiple places if it is helpful.

4.2 Type Equivalence

When are two types equivalent? This turns out to be a critically important question in any non-trivial programming language, because typechecking requires checking for type equivalence in many situations (e.g. checking that the "then" and "else" branches of a conditional have the same type).

For ALF and ALFA, the answer was straightforward: two types are equivalent if and only if they are syntactically identical.

In Recursive ALFA, the question of type equivalence is a little more subtle. Why? Because two types that differ only in the choice of bound variable identifiers are semantically equivalent. We call this equivalence "alpha-equivalence" for historic reasons, and write it $\tau \equiv_\alpha \tau'$. For example, consider the types NumList from above and $\mathsf{NumList}'$ defined as follows:

$$\mathsf{NumList} = \mathsf{rec}\ numlist\ \mathsf{is}\ 1 + \mathsf{Num} \times numlist$$
$$\mathsf{NumList}' = \mathsf{rec}\ self\ \mathsf{is}\ 1 + \mathsf{Num} \times self$$

These two types are alpha-equivalent because all that differs is that we chose the identifier $self$ rather than $numlist$ for the self-reference in the second. Indeed, we could keep defining such types, because there is an infinite equivalence class of types alpha equivalent to these.

Now that type equivalence is a bit more sophisticated, we need to use explicit equivalence in rules, like those for if expressions, that involve checking that two types are equivalent:

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : \tau_2 \qquad \Gamma \vdash e_3 : \tau_3 \qquad \tau_2 \equiv_\alpha \tau_3}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \tau_3}\ \text{(T-If)}$$

Since differences like these only matter to human readers and do not have any formal significance, it turns out to be far more practical to work not with individual types but with the entire alpha-equivalence classes of types at once. How might we do that? The trick is to eliminate identifiers entirely from our operational representation and instead represent only the *binding tree* of the type. For example, we might draw the binding tree for the type NumList as shown in Fig. 2, omitting the identifier and instead drawing an arrow directly to

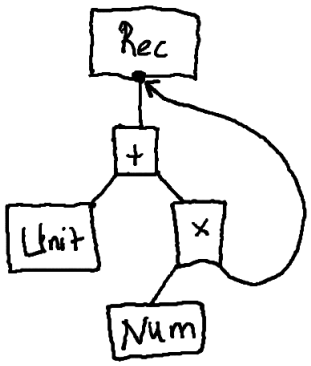

Figure 2: The binding tree for all types alpha-equivalent to NumList.

the node in the tree that introduces the binding being referred to. All types alpha-equivalent to NumList have exactly the same binding tree, so the binding tree is actually a representation of the entire equivalence class.

4.2.1 De Bruijn Indices

We can represent binding trees not just graphically as in Fig. 2 but also symbolically. The most common approach is to use a *de Bruijn*[2] *indexed representation* of binding trees. In this representation, instead of type variables, a, we use a natural number called a de Bruijn index, written $\overleftarrow{n}$, that specifies how many parent binding sites in the tree to "skip" to get to the intended binding site. For example, consider the following type (don't worry about what it means, just consider it syntactically):

$$\mathsf{rec}\ a\ \mathsf{is}\ (1 + (\mathsf{rec}\ b\ \mathsf{is}\ (a \to \mathsf{Num} \times b)) \times a)$$

We can represent this and every other type in its alpha-equivalence class with the same de Bruijn indexed representation as follows:

$$\mathsf{rec}\ \cdot\ \mathsf{is}\ (1 + (\mathsf{rec}\ \cdot\ \mathsf{is}\ (\overleftarrow{1} \to \mathsf{Num} \times \overleftarrow{0})) \times \overleftarrow{0})$$

In this example, the first use of a corresponds to de Bruijn index $\overleftarrow{1}$ because you have to skip one type variable binding (i.e. the binding of b by the recursive type) to get to the binding of a. The first (and only) use of b has de Bruijn index $\overleftarrow{0}$ because it is bound by the closest parent binder, i.e. the recursive type. The second use of a is not under the recursive type body, so it does not need to skip a binding and the de Bruijn index in that case is also $\overleftarrow{0}$. Notice that we no longer need to mention the identifiers at all – where they would be bound, we write $\cdot$ to indicate simply that a binding exists.

A de Bruijn index that is too high given the number of bindings it takes to reach it in the given term is called a *free index* because it corresponds to a free variable. Free indices

[2]pronounced approximately "de brown"

will arise when you "peel off" binders in order to substitute into the body of a type. For example, the body of the recursive type above is $1 + (\mathsf{rec}\ \cdot\ \mathsf{is}\ (\overleftarrow{1} \to \mathsf{Num} \times \overleftarrow{0})) \times \overleftarrow{0}$. Here, only the left-most instance of $\overleftarrow{0}$, referring to the variable b bound by the recursive type, is a bound index. The other two have become free because we peeled off the recursive type on the outside.

Question 6: De Bruijn Index Conversion (Written, 10%)

a) Convert the following type with type variables into de Bruijn index representation

   1) $(\mathsf{rec}\ y\ \mathsf{is}\ (\mathsf{rec}\ x\ \mathsf{is}\ ((\mathsf{rec}\ z\ \mathsf{is}\ (z \times y)) \times x + y) \times x)) \times \mathsf{Num}$
   2) $\mathsf{rec}\ m\ \mathsf{is}\ ((\mathsf{rec}\ n\ \mathsf{is}\ (m \times n)) + (\mathsf{rec}\ n\ \mathsf{is}\ (m + (\mathsf{rec}\ n\ \mathsf{is}\ (n \times m)))))$

b) Convert the following de Bruijn index representation to have type variables (of your choice)

   1) $(\mathsf{rec}\ \cdot\ \mathsf{is}\ (\overleftarrow{0} \to 1)) \to (\mathsf{rec}\ \cdot\ \mathsf{is}\ (\mathsf{rec}\ \cdot\ \mathsf{is}\ (1 \times (\overleftarrow{0} \to \overleftarrow{1}))))$
   2) $\mathsf{rec}\ \cdot\ \mathsf{is}\ ((\mathsf{rec}\ \cdot\ \mathsf{is}\ ((\overleftarrow{0} + \overleftarrow{1}) \times (\mathsf{rec}\ \cdot\ \mathsf{is}\ (\overleftarrow{2} + \overleftarrow{0})))) \to \overleftarrow{0})$

Question 7: De Bruijn Indices (LearnOCaml, 20%) Complete the OCaml code shown in Fig. 3. Of the six modules, the first four will be available in the prelude. Your task is to implement the last two: `DeltaUtil` and `TypUtil`. (The code is organized this way for testing purposes.) You do not need to prove the theorems below, but they should guide your thinking. The functions here are the key to implementing a typechecker for Recursive ALFA similar to that in previous assignments, which we will provide for reference (after the extended late deadline for A4 – see Piazza).

Type Validation Contexts The `Delta.t` type implements type validation contexts, Δ, as lists of identifiers. This is similar to the Contexts exercise from Assignment 3, differing mainly in that the `DeltaUtil.lookup` function return the de Bruijn index for the given identifier when it exists. Hint: This is straightforward if you implement `DeltaUtil.extend` so that the identifier with de Bruijn index n is always at position n of the list, where position 0 is the head.

The only tricky question to consider here is whether the list should contain duplicates or not after you extend the context with an identifier that has already appeared, i.e. when an identifier has been shadowed. For example, you'll want to think about the following type, where b is shadowed:

$$\mathsf{rec}\ a\ \mathsf{is}\ (1 + \mathsf{rec}\ b\ \mathsf{is}\ (\mathsf{rec}\ b\ \mathsf{is}\ (a + \mathsf{Num} \times b)) \times a)$$

Type Validity and Equivalence The `Typ.t` type represents types and the `Typ.d` type represents alpha-equivalence classes of types using the method of de Bruijn indices just described. Because the two definitions differ only in how variables and binders are represented, we parameterize over these choices via the parameterized type `('var, 'binder) Typ.p`.

```
module Identifier = struct type t = string end

module DBIdx = struct type t = int end

module Delta = struct
  type t = Identifier.t list
  let empty : t = []
end

module Typ = struct
  (* parameterized representation of types *)
  type ('var, 'binding) p =
  | TVar of 'var
  | TRec of 'binding * ('var, 'binding) p
  | TForall of 'binding * ('var, 'binding) p
  | TNum | TBool | TUnit
  | TProd of ('var, 'binding) p * ('var, 'binding) p
  | TSum of ('var, 'binding) p * ('var, 'binding) p
  | TArrow of ('var, 'binding) p * ('var, 'binding) p

  (* types that use particular identifiers *)
  type t = (Identifier.t, Identifier.t) p

  (* de Bruijn indexed representations of
   * alpha-equivalence classes *)
  type d = (DBIdx.t, unit) p

  (* type equivalence once you have a de Bruijn indexed
   * representation is simply structural equality! *)
  let dequiv (d1 : d) (d2 : d) : bool = (d1 = d2)
end

module DeltaUtil = struct
  let extend : t -> Identifier.t -> t =
    raise NotImplemented
  let lookup : t -> Identifier.t -> DBIdx.t option =
    raise NotImplemented
end

module TypUtil = struct
  let rec valid (ctx : Delta.t) (ty : t) : d option =
    raise NotImplemented
  let valid_and_closed = valid Delta.empty

  (* substitute d1 for free de Bruijn index 0 in d2 *)
  let subst0 (d1 : d) (d2 : d) : d = raise NotImplemented
end
```


Figure 3: De Bruijn Indexed Types

That is, `Typ.p` is a parameterized type (like `'a list`), except it now takes two type parameters, `'var` and `'binder`, rather than just one. (We saw a similar parameterized type with two parameters in the handout for A4.) The two parameters are written with parentheses and a comma, but careful: this has nothing to do with product types (which use `*`). It is just a type with two type parameters. The constructors of both types have the same names, and you work with them using pattern matching against those constructors.

We can implement individual types by instantiating both of these parameters with `Identifier.t`. We call this type `Typ.t`.

Alternatively, we can represent a de Bruijn indexed type equivalence class, which we call `Typ.d`, by instantiating `'var` with a de Bruijn index (an integer, though we define it as the type `DBIdx.t` for clarity) and `'binder` with **`unit`**, because we no longer need to explicitly represent bindings (we refer to them only by index).

Both `Typ.d` and `Typ.t` use the same constructors. It is only their argument types that are determined by the choice of parameters. For example, we can write the encoding of the NumList type, and its corresponding de Bruijn indexed representation, as follows:

```
let numlist_t : Typ.t =
  Typ.TRec("numlist", Typ.TSum (
    Typ.TUnit,
    Typ.TProd (Typ.TNum, Typ.TVar("numlist"))))
let numlist_d : Typ.d =
  Typ.TRec((), Typ.TSum (
    Typ.TUnit,
    Typ.TProd (Typ.TNum, Typ.TVar(0))))
```

Implement the function `TypUtil.valid`, which checks that the input type is valid according to the rules given in Sec. 4.1. If so, it returns the corresponding de Bruijn indexed representation.

The following correctness theorems should hold for your implementation.

Theorem 2 (Type Validation Correctness)**.** `TypUtil.valid` $\lfloor\Delta\rfloor$ $\lfloor\tau\rfloor \equiv$ `Some` $\lfloor\overleftarrow{\tau}\rfloor$ *where* $\overleftarrow{\tau}$ *is the de Bruijn indexed representation of* τ *iff* $\Delta \vdash \tau$ `type`*.*

This function allows us to implement the function `Typ.dequiv` using simple structural equality. We would use this function when we need to check for type equivalence in a type checker for Recursive ALFA, such as in the T-If rule described above.

Type Variable Substitution Variables, including type variables, are ultimately given meaning by substitution. Implement a substitution function, `TypUtil.subst0 d1 d2`, that substitutes `d1` for the free variable that would be given index `0` at the root of the type `d2` (we'll use this to implement unrolling in the typechecker). You can assume there are no free indices in `d1`, i.e. it is closed. You may also assume that there are no free indices greater than `0` relative to the root in `d2`. Hint: think about how the index you are looking for changes each time you cross a binder. You will need a helper function.

This function is critical in implementing recursive type unrolling as found in the typing rules T-Roll and T-Unroll.