Assignment 3:
PCF, **FPC**, and **PyCF**

15-312: Principles of Programming Languages (Fall 2023)

In this assignment you will explore concepts of self-reference in programming languages. In **PCF** we consider self-reference at the term level. Most often students encounter this form of self-reference when defining recursive functions, those that "call themselves" when applied to an argument. But self-reference is not inherently tied to functions, and may be usefully isolated using the type $\texttt{self}(\tau)$ of self-referential expressions of type τ. In **FPC** we consider self-reference at the type level, via recursive types $\texttt{rec}(t\,.\,\tau)$, where t may occur without restriction in τ. Recursive types may be used to define self-types, which shows that **FPC** subsumes **PCF**. Most students are familiar with recursive types in ML for defining data structures, such as natural numbers and lists, which are examples of inductive types. These definitions rely on the eager, by-value dynamics of **FPC**. When a lazy, by-name dynamics is used, the "same" recursive types, syntactically, define coinductive types, namely co-natural numbers and streams. In the first part of this assignment you will explore these relationships in more detail.

So-called "dynamically typed" languages are promoted as being more powerful and more flexible than their "statically typed" counterparts: there is no need to please a type checker to get your code to compile. A commonly-cited example is the ability to form heterogenous lists, those whose elements are of disparate types, say strings and numbers and booleans all intermixed, with the implication that this cannot be done in a static language. More generally, advocates of dynamic language stress the interactive nature of code development in which anything that parses does something, and one never has to deal with a type checker.

To explore these claims you will implement a dynamic version of **PCF**, called **PyCF**, in homage to a well-known dynamic language. The language is small, though easily extensible, but is faithful to the core tenets of dynamic typing. As you have learned in class, the irony is that dynamic typing is but a mode of use of static typing, one in which you confine your attention to a single recursive type. To substantiate this claim, you will formulate and implement a translation from **PyCF** into **FPC** in which a single recursive type classifies all values of the dynamic language. So-called dynamic types emerge as a combination of folding in a recursive type and tagging the components of a summand, leading to considerable run-time overhead. You will consider in detail the translation of a particular **PyCF** program to get a better feel for their needless inefficiency.

As Dana Scott said, untyped languages are really uni-typed. And here we ask, just why is that a good idea?

1 Self-Reference in Expressions and Types

Recall the type $\texttt{self}(\tau)$ of self-referential values of type τ, with introductory form $\texttt{self}[\tau](x.e)$ and eliminator form $\texttt{unroll}(e)$ subject to the dynamics

$$\texttt{unroll}(\texttt{self}[\tau](x.e)) \longmapsto \{\texttt{self}[\tau](x.e)/x\}e$$

with $\texttt{self}[\tau](x.e)$ val. Recall that the type $\texttt{self}(\tau)$ is definable in **FPC** as the recursive type $\texttt{rec}(t.t \to \tau)$, where $t \notin \tau$, with $\texttt{self}[\tau](x.e)$ being $\texttt{fold}(\lambda(x:\texttt{self}(\tau))e)$, and $\texttt{unroll}(e)$ being $\texttt{unfold}(e)(e)$.

Consider the extension of **FPC** defined in Appendix B.

Task 1.1 (10 pts).

Assume given a function $\texttt{halve} : \texttt{int} \to \texttt{int}$ that computes the floor of half of its argument. Thus, for example, $\texttt{halve}(3)$ evaluates to 1. You are to define:

1. $\texttt{lg}' : (\texttt{int} \to \texttt{int})\,\texttt{self}$, and
2. $\texttt{lg} : \texttt{int} \to \texttt{int}$.

The second, which computes the floor of the binary logarithm of its argument, is to be defined in terms of the first, an auxiliary function that, on input n computes $\lfloor \lg n \rfloor$.[1] Notice that $\texttt{lg}'$ is self-referential, and $\texttt{lg}$ is not!

Your solution must be given in terms of the self-types summarized above, within an eager, by-value interpretation of **FPC**.

For the next two tasks, define

$$T \triangleq \texttt{rec}\, t\, \texttt{is}\, \tau,$$

where

$$\tau \triangleq \texttt{E} \hookrightarrow \texttt{unit} + \texttt{N} \hookrightarrow t \times \texttt{bool} \times t.$$

The meaning of this type depends on whether **FPC** is understood eagerly/by-value or lazily/by-name.

Task 1.2 (20 pts).

For this task assume an **eager/by-value** dynamics for **FPC**.

Define the value $\texttt{Emp} \triangleq \texttt{fold}(\texttt{E} \cdot \langle\rangle)$ of type T, and $\texttt{Node}(t_1, b, t_2) \triangleq \texttt{fold}(\texttt{N} \cdot \langle t_1, b, t_2 \rangle)$, a value of type T when t_1 and t_2 are values of type T and b is a value of type $\texttt{bool}$.

In the eager setting this type is inductive in that it is possible to define a recursor for it with the following statics:

$$\frac{\Gamma \vdash e_1 : T \qquad \Gamma, x : \{\rho/t\}\tau \vdash e_2 : \rho}{\Gamma \vdash \texttt{Trec}[\rho](e_1\,;x.e_2) : \rho}$$

[1] $\lg n$ is the base-2 logarithm of n. For simplicity, your implementation may do anything when $n \leq 0$.

and the following dynamics:

$$\frac{}{\texttt{Trec}[\,\rho\,](\,\texttt{Emp}\,;\,x\,.\,e_2\,) \longmapsto \texttt{let}\,x\,\texttt{be}\,\texttt{E}\cdot\langle\rangle\,\texttt{in}\,e_2}$$

and

$$\frac{r_1 = \texttt{Trec}[\,\rho\,](\,t_1\,;\,x\,.\,e_2\,) \quad r_2 = \texttt{Trec}[\,\rho\,](\,t_2\,;\,x\,.\,e_2\,)}{\texttt{Trec}[\,\rho\,](\,\texttt{Node}(t_1,b,t_2)\,;\,x\,.\,e_2\,) \longmapsto \texttt{let}\,x\,\texttt{be}\,\texttt{N}\cdot\langle r_1,b,r_2\rangle\,\texttt{in}\,e_2}$$

Notice the similarity with the recursor for `nat`, the difference being that a node has two predecessors and a label.

Your tasks are as follows:

1. Describe, informally, the values of type T under this interpretation in terms of the above values.
2. Define, using the recursor, an in-order traversal function

   $$\texttt{inord} : T \to \texttt{bool}\,\texttt{list}.$$

   Feel free to use list operations, such as append and cons.
3. Define the recursor in **FPC** by giving a recursive function $R : (T \to \rho)\,\texttt{self}$, defined in terms of $x\,.\,e_2$, so that

   $$\texttt{Trec}[\,\rho\,](\,e_1\,;\,x\,.\,e_2\,) \triangleq \texttt{unroll}(\,R\,)(\,e_1\,).$$

   Your solution will follow very closely the dynamics, using `let` to bind the results of the recursive calls as in the premises of the dynamics rules.

Task 1.3 (20 pts). For this task assume a **lazy/by-name** dynamics for **FPC**.

If e is of type T, then the expression $\texttt{unfold}(\,e\,)$ is of type $\{T/t\}\tau$, the sum type given earlier.

In the lazy setting this type is coinductive in that it is possible to define a generator for it with the following statics:

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \{\sigma/t\}\tau}{\Gamma \vdash \texttt{Tgen}[\,\sigma\,](\,e_1\,;\,x\,.\,e_2\,) : T}$$

and dynamics:

$$\frac{t_1 = \texttt{Tgen}[\,\sigma\,](\,s_1\,;\,x\,.\,e_2\,) \quad t_2 = \texttt{Tgen}[\,\sigma\,](\,s_2\,;\,x\,.\,e_2\,)}{\texttt{unfold}(\,\texttt{Tgen}[\,\sigma\,](\,e_1\,;\,x\,.\,e_2\,)\,) \longmapsto \texttt{case}\,\{e_1/x\}e_2\,\{\texttt{E}\cdot\_ \hookrightarrow \texttt{E}\cdot\langle\rangle \mid \texttt{N}\cdot\langle s_1,b,s_2\rangle \hookrightarrow \texttt{N}\cdot\langle t_1,b,t_2\rangle\}}$$

Notice the similarity with the generator for the type `conat`, the difference being that a node has two predecessors, and a label.

1. Describe, informally, the values of this type in terms of its unfolding.

2. Define the function

$$\texttt{inord} : (\, T \to \texttt{bool}\,\texttt{stream}\,)\,\texttt{self}$$

that computes an inorder traversal of the given argument of type T. Feel free to use operations on streams such as append and cons of an element onto a stream. *Hint:* Your solution will be very similar to the one given for the inductive case. It will *not* require the use of the generator, because it does *not* compute a value of type T!

3. Define the generator in **FPC** by giving a recursive function $G : (\sigma \to T)\ \texttt{self}$, defined in terms of $x\,.\,e_2$, so that

$$\texttt{Tgen}[\,\sigma\,](\,e_1\,;x\,.\,e_2\,) = \texttt{unroll}(\,G\,)(\,e_1\,).$$

Your solution will rely on the lazy/by-name interpretation of **FPC**, and will follow very closely the dynamics given above.

2 The Zen of PyCF

In a dynamically typed language, such as Python, Ruby, JavaScript, or Scheme, an implementation must internally **tag** values with their class (often called "type") when they are created, and all operations must check the class of their operands (perform a "run-time type check").

In this section, we introduce **PyCF**, a dynamically-"typed" extension of **PCF**. For simplicity, we statically check that closed express do not have unbound variables, but there are no other static guarantees. The dynamics is more involved because it must impose, check, and remove classes during execution. Since these checks are baked into the dynamics, they are not visible in the syntax. Thus, it isn't possible to program without them!

2.1 Syntax

We summarize the syntax of this language here, writing ~~expressions~~ objects as d instead of e to distinguish from other languages. Rather than having (static) types, **PyCF** has (dynamic) classes which can be checked at run-time.

Sort		Abstract form	Concrete form	
Class	c ::=	`bool`	`bool`	
		`int`	`int`	
		`list`	`list`	
		`fun`	`fun`	
Object	d ::=	x	x	
		`bool`$[b]$	b	(`True`, `False`)
		`if`$(d; d_1; d_0)$	d_1 `if` d `else` d_0	
		`int`$[i]$	i	$(\ldots, -2, -1, 0, 1, 2, \ldots)$
		`plus`$(d_1; d_2)$	d_1 `+` d_2	(int addition and list append)
		`leq`$(d_1; d_2)$	d_1 `<=` d_2	
		`list`$(d_0; \ldots; d_{n-1})$	$[d_0, \ldots, d_{n-1}]$	
		`index`$(d; d_{\text{index}})$	d`[`d_{index}`]`	
		`len`(d)	`len(`d`)`	
		`fun`$(f\,.\,x\,.\,d)$		(see below)
		`ap`$(d; d_1)$	$d($ d_1 $)$	
		`let`$(d_1; x\,.\,d_2)$		(see below)
		`isinstance`$[c](d)$	`isinstance(`d`, `c`)`	

PyCF is an extension of **DPCF**, adding booleans, integers with primitive addition and less-than-or-equal-to operations, lists, let bindings, and run-time class checking.

In the concrete syntax, we parse `fun`$(f\,.\,x\,.\,d)$ and `let`$(d_1; x\,.\,d_2)$ as *declarations* to match familiar programming languages. Programs are lists of declarations, terminated by `if __name__ == "__main__": print(`d`)`, which gets desugared to a sequence of `lets`. For example, here is a program which sums the list `[1, 5, 3, 1, 2]`:

```
# Sum a list.
def sum(l):
    n = len(l)

    def sum_helper(i):
        return 0 if n <= i else l[i] + sum_helper(i + 1)

    return sum_helper(0)


test = [1, 5, 3, 1, 2]

if __name__ == "__main__":
    print(sum(test))
```

In abstract syntax, this program is is MAIN, where (up to α-equivalence):

$$\begin{aligned}
\textsc{SumHelper} &\triangleq \texttt{fun}(h\,.\,i\,.\,\texttt{if}(\texttt{leq}(n\,;i)\,;\texttt{int}[0]\,;\texttt{plus}(\texttt{index}(l\,;i)\,;\texttt{ap}(h\,;\texttt{plus}(i\,;\texttt{int}[1])))))\\
\textsc{Sum} &\triangleq \texttt{fun}(\mathit{sum}\,.\,l\,.\,\texttt{let}(\texttt{len}(l)\,;n\,.\,\texttt{let}(\textsc{SumHelper}\,;\mathit{helper}\,.\,\texttt{ap}(\mathit{helper}\,;\texttt{int}[0]))))\\
\textsc{Test} &\triangleq \texttt{list}(\texttt{int}[1],\texttt{int}[5],\texttt{int}[3],\texttt{int}[1],\texttt{int}[2])\\
\textsc{Main} &\triangleq \texttt{let}(\textsc{Sum}\,;\mathit{sum}\,.\,\texttt{let}(\textsc{Test}\,;\mathit{test}\,.\,\texttt{ap}(\mathit{sum}\,;\mathit{test})))
\end{aligned}$$

The abstract syntax of **PyCF** intentionally resembles that of **PCF**, but without any type information.[2] But appearances are deceiving! The dynamics of **PyCF** is radically different from that of **PCF**, for reasons that are much deeper than can be explained at the purely syntactic level.

2.2 Dynamics

Because the statics of **PyCF** is so permissive, the dynamics has to pick up the slack and detect errors at run-time. Consequently, we need the following set of judgments (defined in Appendix A):

d isnt c	d is not of class c
d val	d is a closed value
$d \longmapsto d'$	d steps to d'
d err	d incurs a run-time error

Task 2.1 (20 pts). Implement the `Index`, `Fun`, `Ap`, and `IsInstance` cases of the dynamics of **PyCF** in `lang-pycf/dynamics-pycf.sml`.

Testing As usual, you can test your code using `InterpreterPyCF`. For example:

[2] The concrete syntax may resemble[3] that of another familiar dynamically-"typed" language, as well... :)

[3] Don't take this *too* seriously, though - the semantics are sometimes different. For example, there is no mutability here (variables are true mathematical variables, not assignables), and `isinstance(True, int)` is false.

```
- InterpreterPyCF.repl ();
-> 1 + 2 + 3;
(Plus ((Plus ((Int 1), (Int 2))), (Int 3)))
Type: ok
Evaluating... val (Int 6)

-> [1, 2] + [3];
(Plus ((List [(Int 1), (Int 2)]), (List [(Int 3)])))
Type: ok
Evaluating... val (List [(Int 1), (Int 2), (Int 3)])

->
Interrupt
- InterpreterPyCF.evalFile "tests/triangle.pycf";
(Let ((Fun (triangle7 . (n8 . (If ((LEq (n8, (Int 0))), (Int 0),
   (Plus (n8, (Ap (triangle7, (Plus (n8, (Int ~1))))))))))),
   (triangle5 . (Ap (triangle5, (Int 10)))))))
Type: ok
Evaluating... val (Int 55)
val it = () : unit
```


2.3 Dynamic Programming (the other kind)

Now, let's experiment with programming in **PyCF**.

Task 2.2 (15 pts). Implement the following operations in **PyCF** using concrete syntax in `lang-pycf/tasks/`.

1. In `comprehension.pycf`, implement `comprehension` to satisfy the given specification.

   `comprehension([f, p, l])` should be analogous to `[f(x) for x in l if p(x)]` in other programming languages.

   Hint. You may find it useful to draw inspiration from `map.pycf` and `filter.pycf`.

2. In `sum_default.pycf`, implement `sum` to satisfy the given specification.

   In particular, `sum` should take in "multiple arguments" (really, a list of arguments). You can assume it will be provided only one or two arguments. The zeroth argument, always present, will be a list to sum. The first argument, which may or may not be present, will be a value to start the sum from, defaulting to `0`.

   This function is similar to `sum` in Python, which takes in an optional `start` argument.

3. In `matrix_sum.pycf`, implement `matrix_sum`, which sums the elements of a matrix represented by nested lists.

   For example:

   - `matrix_sum(5)` should be `5`, since its input is a zero-dimensional matrix (scalar).

- `matrix_sum([1, 4, 2])` should be `7`, summing the vector as a list.
- `matrix_sum([[5, 1], [6, 8]])` should be `20`, summing the two-dimensional matrix.

You may assume the input is built only of well-formed matrices; you need not worry about dimensions. Additionally, you are welcome to copy in functions from the given test files as you see fit.

To evaluate your file, you can use `InterpreterPyCF`. In each of the files, there are commented-out test cases you can uncomment and run. However, when submitting, make sure to comment them out again!

```
- InterpreterPyCF.evalFile "tasks/matrix_sum.pycf";
(Let ((Fun (matrix_sum8 . (m9 . (* your implementation... *)))),
   (matrix_sum6 . (Ap (matrix_sum6, (List [(List [(Int 5), (Int
   1)]), (List [(Int 6), (Int 8)])])))))))
Type: ok
Evaluating... val (Int 20)

val it = () : unit
(* remember: comment out test, uncomment `print(matrix_sum)' *)
```

3 Compiling PyCF to FPC

The run-time overhead of dynamic typing can be made explicit by translating **PyCF** into an eager version of **FPC** by regarding a **PyCF** object as an **FPC** expression with the type `dyn`, where `dyn` is defined as:

$$\begin{aligned}\texttt{dynview}[\tau] &\triangleq (\texttt{bool} \hookrightarrow \texttt{bool}) + (\texttt{int} \hookrightarrow \texttt{int}) + (\texttt{list} \hookrightarrow \texttt{list}(\,\tau\,)) + (\texttt{fun} \hookrightarrow \texttt{arr}(\,\tau\,;\,\tau\,))\\ \texttt{dyn} &\triangleq \texttt{rec}(\,d\,.\,\texttt{dynview}[d]\,)\\ &= \texttt{rec}(\,d\,.\,(\texttt{bool} \hookrightarrow \texttt{bool}) + (\texttt{int} \hookrightarrow \texttt{int}) + (\texttt{list} \hookrightarrow \texttt{list}(\,d\,)) + (\texttt{fun} \hookrightarrow \texttt{arr}(\,d\,;\,d\,))\,)\end{aligned}$$

When unfolded, `dyn` is a sum type whose summands correspond to the classes of **PyCF**:

$$\texttt{dynview}[\texttt{dyn}] = (\texttt{bool} \hookrightarrow \texttt{bool}) + (\texttt{int} \hookrightarrow \texttt{int}) + (\texttt{list} \hookrightarrow \texttt{list}(\,\texttt{dyn}\,)) + (\texttt{fun} \hookrightarrow \texttt{arr}(\,\texttt{dyn}\,;\,\texttt{dyn}\,))$$

For brevity, we write τ_c for the type of the summand with label c:

$$\begin{aligned}\tau_{\texttt{bool}} &\triangleq \texttt{bool}\\ \tau_{\texttt{int}} &\triangleq \texttt{int}\\ \tau_{\texttt{list}} &\triangleq \texttt{list}(\,\texttt{dyn}\,)\\ \tau_{\texttt{fun}} &\triangleq \texttt{arr}(\,\texttt{dyn}\,;\,\texttt{dyn}\,))\end{aligned}$$

where we define:

$$\texttt{bool} \triangleq (\texttt{true} \hookrightarrow \texttt{unit}) + (\texttt{false} \hookrightarrow \texttt{unit})$$

The run-time checks on **PyCF** amount to case analyses on the class of a value (since `dyn` is a sum): if the expected case is encountered, the value is returned, otherwise it results in an error.

3.1 Introduction

Suppose that $\Gamma \vdash d\ \mathsf{ok}$ is a well-formed **PyCF** expression in the **PyCF** context Γ. You are to define a translation $\overline{d}$ by induction on the structure of d that transforms d, a **PyCF** object, into an **FPC** expression that mimics d's behavior. We require static and dynamic correctness of the translation.

Theorem 3.1 (Static Correctness)**.** *If* $\Gamma \vdash d\ \mathsf{ok}$*, then* $\overline{\Gamma} \vdash \overline{e} : \texttt{dyn}$*, where* $\overline{\Gamma} = x_1 : \texttt{dyn}, \ldots, x_n : \texttt{dyn}$ *is the* **FPC** *context that corresponds to the* **PyCF** *context* $\Gamma = x_1\ \mathsf{ok}, \ldots, x_n\ \mathsf{ok}$*.*

Notice that we assume all variables have type `dyn`. We must be careful to maintain this invariant during translation when going under a binder!

The *dynamic correctness* is more difficult to specify precisely; the tools required are beyond the scope of this course. However, one may keep in mind the following guidelines for the translation.

1. A **PyCF** expression is erroneous (incurs a run-time error) iff its translation into **FPC** produces a run-time error.

2. A **PyCF** expression evaluates to a value of class c iff its translation into **FPC** evaluates to a value of type `dyn` of the form $\texttt{fold}(\, c \cdot v \,)$ for a value $v : \tau_c$.
3. A **PyCF** expression d should diverge iff its translation e diverges in **FPC**.
4. The order of evaluation of sub-expressions should be exactly the same under translation as it is in **PyCF**.

Now we define the helper expressions for dealing with dynamic class checking and recursive functions.

3.2 Tagging and Untagging

We wish to define two derived **FPC** constructs corresponding to tagging and untagging values according to their class:

$$\frac{\Gamma \vdash e : \tau_c}{\Gamma \vdash \texttt{new}[\, c \,](\, e \,) : \texttt{dyn}} \qquad \frac{\Gamma \vdash e : \texttt{dyn}}{\Gamma \vdash \texttt{cast}[\, c \,](\, e \,) : \tau_c}$$

In these rules, the type τ_c for class c is determined by the unrolling of the recursive type `dyn` as defined before. The purpose of `new` is to add a class tag to e and, dually, `cast` removes this class tag and returns the contained data if the class was correct and produces an error otherwise.

Task 3.1 (10 pts). Define the operation $\texttt{new}[\, c \,](\, e \,)$ with the above typing as a derived form in **FPC**, using the expression forms given in Appendix B. These operations may be thought of as the introduction forms for the recursive type `dyn` representing values of **PyCF**.

Task 3.2 (10 pts). Define the operation $\texttt{cast}[\, c \,](\, e \,)$ with the above typing in **FPC**, using the expression forms given in Appendix B. These are the elimination forms for the recursive type `dyn`.

Task 3.3 (10 pts). Implement $\texttt{new}[\, c \,](\, e \,)$ and $\texttt{cast}[\, c \,](\, e \,)$ in `translate/synext.sml`. You will find the functions in `translate/synext-helpers.sml` helpful.

Hint. You should *not* case on e, which could be an arbitrary expression of the appropriate type according to the rules above. In fact, e need not terminate, let alone be a value. Instead, you may wish to case on c (either explicitly or via auxiliary functions such as `classToSummand Dyn`).

3.3 Recursive Functions

To translate recursive functions from **PyCF**, we will need to recover recursive functions in **FPC**. In particular, we want a way to define recursive functions into **FPC** which admits the following typing rule:

$$\frac{\Gamma, f : \tau_1 \to \tau_2, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \texttt{fun}[\, \tau_1 \,;\, \tau_2 \,](\, f \,.\, x \,.\, e \,) : \tau_1 \to \tau_2}$$

Task 3.4 (10 pts). Implement $\texttt{fun}[\tau_1\,;\tau_2](\,f\,.\,x\,.\,e\,)$ in **FPC** in `translate/synext.sml`. This utility should be generic in **FPC**, and is not specifically tied up with **PyCF** and `dyn` type. You should make use of self types from `translate/synext-helpers.sml`, which are defined as follows:

$$\begin{aligned}\texttt{self}(\,\tau\,) &\triangleq \texttt{rec}(\,t\,.\,t \to \tau\,)\\ \texttt{self}[\,\tau\,](\,x\,.\,e\,) &\triangleq \texttt{fold}[\,t\,.\,t \to \tau\,](\,\lambda[\,\texttt{self}(\,\tau\,)\,](\,x\,.\,e\,)\,)\\ \texttt{unroll}(\,e\,) &\triangleq \texttt{unfold}(\,e\,)(\,e\,)\end{aligned}$$

Now, we may use recursive functions in our translation, given that they are definable in **FPC**.

We have to be careful translating recursive functions, since in the **FPC** definition $\texttt{fun}[\,\tau_1\,;\tau_2\,](\,f\,.\,x\,.\,e\,)$, variable f has type $\tau_1 \to \tau_2$ while x has type τ_1. However, in the translation, every variable must have type `dyn`.

3.4 Putting it All Together

We can now tackle the actual translation. Using these auxiliary notions, define the translation function $\overline{d}$ to satisfy the criteria specified above. To get you started, here are a few cases.

$$\begin{aligned}\overline{x} &= x\\ \overline{\texttt{bool}[\,b\,]} &= \texttt{new}[\,\texttt{bool}\,](\,b\,)\\ \overline{\texttt{if}(\,d\,;d_1\,;d_0\,)} &= \texttt{if}(\,\texttt{cast}[\,\texttt{bool}\,](\,\overline{d}\,)\,;\overline{d_1}\,;\overline{d_0}\,)\\ \overline{\texttt{list}(\,d_0\,;\ldots;d_{n-1}\,)} &= \texttt{new}[\,\texttt{list}\,](\,\texttt{list}(\,\overline{d_0}\,;\ldots;\overline{d_{n-1}}\,)\,)\\ \overline{\texttt{index}(\,d\,;d_{\text{index}}\,)} &= \texttt{index}(\,\texttt{cast}[\,\texttt{list}\,](\,\overline{d}\,)\,;\texttt{cast}[\,\texttt{int}\,](\,\overline{d_{\text{index}}}\,)\,)\end{aligned}$$

Task 3.5 (40 pts). Using the auxiliary constructs given above, define all the other cases for the translation $\overline{d}$.

Hint. Keep in mind that your translation must satisfy the properties listed above. In particular, according to Theorem 3.1, all variables in the context should be of type `dyn`, and all results should be of type `dyn`.

Task 3.6 (30 pts). Implement the translation in `translate/translate.fun`.

Note that you should use the syntax extensions in `translate/synext.sml`. You will not need to define your own labels for any purposes.

Testing Load `translate/sources.cm`. Then, you can test using `InterpreterTranslate`, which translates REPL and file inputs from **PyCF** to **FPC** before evaluation.

```
- InterpreterTranslate.repl ();
-> False if False else True;
(If ((Bool false), (Bool false), (Bool true)))
Type (source): ok
```

```
Compiling...
(* result of translating into FPC *)
Type (target): (Rec (d3064 . (Sum ["bool" ↪ (Sum ["false" ↪ (Prod
   []), "true" ↪ (Prod [])]), "fun" ↪ (Arrow (d3064, d3064)),
   "int" ↪ Int, "list" ↪ (List d3064)])))
Evaluating... val (Fold ((Inj ("bool", (Inj ("true", (Pair
   [])))))))
```

Notice that the type of the translated code and the result of evaluating the translated code are `dyn`; this should always be the case. Additionally, notice that the value is simply:

$$\texttt{fold}[\,d\,.\,\texttt{dynview}[d]\,](\,\texttt{bool}\cdot\texttt{true}\,)$$

Type annotations are *not* printed, since they are very long and obfuscate the output.

3.5 Reflection

Observe that the compiled code for seemingly-simple **PyCF** code becomes massive, due to the class tagging and checking. Consider the following function:

```
# Sum the numbers from 0 to n.
def triangle(n):
    return 0 if n <= 0 else n + triangle(n + -1)
```


One invariant of `triangle` is that it always returns an object of class `int`. However, in **PyCF**, we must perform many redundant tag checks at run-time: for example, we must check at each layer of recursion that both `n` and `triangle(n + -1)` are of class `int` before adding them together.

In **PyCF**, there's no way for us to get around these checks. However, if we work in **FPC**, we can wrap efficient code so it is compatible with compiled **PyCF** code. Continuing our example, consider an efficient (class-check free) version of the given function, e_{tri}:

$$e_{\text{tri}} \triangleq \texttt{fun}[\,\texttt{int}\,;\,\texttt{int}\,](\,f\,.\,n\,.\,\texttt{if}(\,\texttt{leq}(\,n\,;\,\texttt{int}[\,0\,]\,)\,;\,\texttt{int}[\,0\,]\,;\,\texttt{plus}(\,n\,;\,\texttt{ap}(\,f\,;\,\texttt{plus}(\,n\,;\,\texttt{int}[\,-1\,]\,)\,)\,)\,)\,)$$

This code isn't immediately compatible with compiled **PyCF** code, since it does not have type `dyn`. However, we can wrap e_{tri} such that it becomes compatible while still maintaining efficiency in the "inner loop" of computation.[4]

Task 3.7 (5 pts). Define efficient code $e_{\text{dyn-tri}} : \texttt{dyn}$ (i.e., compatible with "dynamically typed" code) to compute a triangle number using the efficient triangle number function $e_{\text{tri}} : \texttt{int} \to \texttt{int}$. Dynamically invoking $e_{\text{dyn-tri}}$ should perform a constant number of tags and casts, instead of performing tags and casts at each layer of recursion like the naive compiled code.

Hint. Your solution shouldn't depend on the implementation of e_{tri}; in fact, it should work for any expression of type $\texttt{int} \to \texttt{int}$.

[4] This is how some Python libraries, such as NumPy, manage to run fast: they use efficient C code under the hood which is wrapped in `PyObject` compatibility code.

A Definition of PyCF

A.1 Grammar

The grammar is given in Section 2.1.

A.2 Dynamics

A.2.1 Tag Refutation

$\boxed{d \text{ isnt } c}$

$$\frac{}{\texttt{bool}[\,b\,] \text{ isnt } \texttt{int}} \qquad \frac{}{\texttt{bool}[\,b\,] \text{ isnt } \texttt{list}} \qquad \frac{}{\texttt{bool}[\,b\,] \text{ isnt } \texttt{fun}}$$

$$\frac{}{\texttt{int}[\,i\,] \text{ isnt } \texttt{bool}} \qquad \frac{}{\texttt{int}[\,i\,] \text{ isnt } \texttt{list}} \qquad \frac{}{\texttt{int}[\,i\,] \text{ isnt } \texttt{fun}}$$

$$\frac{d_0 \text{ val} \quad \ldots \quad d_{n-1} \text{ val}}{\texttt{list}(\,d_0\,;\ldots;\,d_{n-1}\,) \text{ isnt } \texttt{bool}} \qquad \frac{d_0 \text{ val} \quad \ldots \quad d_{n-1} \text{ val}}{\texttt{list}(\,d_0\,;\ldots;\,d_{n-1}\,) \text{ isnt } \texttt{int}} \qquad \frac{d_0 \text{ val} \quad \ldots \quad d_{n-1} \text{ val}}{\texttt{list}(\,d_0\,;\ldots;\,d_{n-1}\,) \text{ isnt } \texttt{fun}}$$

$$\frac{}{\texttt{fun}(\,f\,.\,x\,.\,d\,) \text{ isnt } \texttt{bool}} \qquad \frac{}{\texttt{fun}(\,f\,.\,x\,.\,d\,) \text{ isnt } \texttt{int}} \qquad \frac{}{\texttt{fun}(\,f\,.\,x\,.\,d\,) \text{ isnt } \texttt{list}}$$

A.2.2 Stepping

$\boxed{d \text{ val}}$ $\boxed{d \text{ err}}$ $\boxed{d \longmapsto d'}$

$$\frac{}{\texttt{bool}[\,b\,] \text{ val}} \qquad \frac{d \longmapsto d'}{\texttt{if}(\,d\,;d_1\,;d_0\,) \longmapsto \texttt{if}(\,d'\,;d_1\,;d_0\,)} \qquad \frac{d \text{ err}}{\texttt{if}(\,d\,;d_1\,;d_0\,) \text{ err}}$$

$$\frac{}{\texttt{if}(\,\texttt{bool}[\,\texttt{true}\,]\,;d_1\,;d_0\,) \longmapsto d_1} \qquad \frac{}{\texttt{if}(\,\texttt{bool}[\,\texttt{false}\,]\,;d_1\,;d_0\,) \longmapsto d_0}$$

$$\frac{}{\texttt{int}[\,i\,]\ \mathsf{val}} \qquad \frac{d_1 \longmapsto d_1'}{\texttt{plus}(\,d_1\,;\,d_2\,) \longmapsto \texttt{plus}(\,d_1'\,;\,d_2\,)} \qquad \frac{d_1\ \mathsf{err}}{\texttt{plus}(\,d_1\,;\,d_2\,)\ \mathsf{err}}$$

$$\frac{d_1\ \mathsf{val} \qquad d_2 \longmapsto d_2'}{\texttt{plus}(\,d_1\,;\,d_2\,) \longmapsto \texttt{plus}(\,d_1\,;\,d_2'\,)} \qquad \frac{d_1\ \mathsf{val} \qquad d_2\ \mathsf{err}}{\texttt{plus}(\,d_1\,;\,d_2\,)\ \mathsf{err}} \qquad \frac{}{\texttt{plus}(\,\texttt{int}[\,i_1\,]\,;\,\texttt{int}[\,i_2\,]\,) \longmapsto \texttt{int}[\,i_1 + i_2\,]}$$

$$\frac{d_j\ \mathsf{val}\ (\text{for all } 0 \le j < m) \qquad d_j'\ \mathsf{val}\ (\text{for all } 0 \le j < n)}{\texttt{plus}(\,\texttt{list}(\,d_0\,;\,\ldots\,;\,d_{m-1}\,)\,;\,\texttt{list}(\,d'_0\,;\,\ldots\,;\,d'_{n-1}\,)\,) \longmapsto \texttt{list}(\,d_0\,;\,\ldots\,;\,d_{m-1}\,;\,d'_0\,;\,\ldots\,;\,d'_{n-1}\,)}$$

$$\frac{d\ \mathsf{val} \qquad d\ \mathsf{isnt}\ \texttt{int}}{\texttt{plus}(\,\texttt{int}[\,i\,]\,;\,d\,)\ \mathsf{err}} \qquad \frac{d_j\ \mathsf{val}\ (\text{for all } 0 \le j < m) \qquad d\ \mathsf{val} \qquad d\ \mathsf{isnt}\ \texttt{list}}{\texttt{plus}(\,\texttt{list}(\,d_0\,;\,\ldots\,;\,d_{m-1}\,)\,;\,d\,)\ \mathsf{err}}$$

$$\frac{d_1\ \mathsf{val} \qquad d_2\ \mathsf{val} \qquad d_1\ \mathsf{isnt}\ \texttt{int} \qquad d_1\ \mathsf{isnt}\ \texttt{list}}{\texttt{plus}(\,d_1\,;\,d_2\,)\ \mathsf{err}} \qquad \frac{d_1 \longmapsto d_1'}{\texttt{leq}(\,d_1\,;\,d_2\,) \longmapsto \texttt{leq}(\,d_1'\,;\,d_2\,)}$$

$$\frac{d_1\ \mathsf{err}}{\texttt{leq}(\,d_1\,;\,d_2\,)\ \mathsf{err}} \qquad \frac{d_1\ \mathsf{val} \qquad d_2 \longmapsto d_2'}{\texttt{leq}(\,d_1\,;\,d_2\,) \longmapsto \texttt{leq}(\,d_1\,;\,d_2'\,)} \qquad \frac{d_1\ \mathsf{val} \qquad d_2\ \mathsf{err}}{\texttt{leq}(\,d_1\,;\,d_2\,)\ \mathsf{err}}$$

$$\frac{i_1 \le i_2}{\texttt{leq}(\,\texttt{int}[\,i_1\,]\,;\,\texttt{int}[\,i_2\,]\,) \longmapsto \texttt{bool}[\,\texttt{true}\,]} \qquad \frac{i_1 \not\le i_2}{\texttt{leq}(\,\texttt{int}[\,i_1\,]\,;\,\texttt{int}[\,i_2\,]\,) \longmapsto \texttt{bool}[\,\texttt{false}\,]}$$

$$\frac{d_1\ \mathsf{isnt}\ \texttt{int}}{\texttt{leq}(\,d_1\,;\,d_2\,)\ \mathsf{err}} \qquad \frac{d_2\ \mathsf{isnt}\ \texttt{int}}{\texttt{leq}(\,\texttt{int}[\,i_1\,]\,;\,d_2\,)\ \mathsf{err}}$$

$$\frac{d_j \text{ val (for all } 0 \le j < k) \qquad d_k \longmapsto d'_k \qquad d'_m = d_m \text{ (for all } m \neq k)}{\texttt{list}(\, d_0 \,;\, \ldots \,;\, d_{n-1} \,) \longmapsto \texttt{list}(\, d'_0 \,;\, \ldots \,;\, d'_{n-1} \,)}$$

$$\frac{d_j \text{ val (for all } 0 \le j < k) \qquad d_k \text{ err}}{\texttt{list}(\, d_0 \,;\, \ldots \,;\, d_{n-1} \,) \text{ err}} \qquad \frac{d_j \text{ val (for all } 0 \le j < n)}{\texttt{list}(\, d_0 \,;\, \ldots \,;\, d_{n-1} \,) \text{ val}}$$

$$\frac{d \longmapsto d'}{\texttt{index}(\, d \,;\, d_{\text{index}} \,) \longmapsto \texttt{index}(\, d' \,;\, d_{\text{index}} \,)} \qquad \frac{d \text{ err}}{\texttt{index}(\, d \,;\, d_{\text{index}} \,) \text{ err}}$$

$$\frac{d \text{ val} \qquad d_{\text{index}} \longmapsto d'_{\text{index}}}{\texttt{index}(\, d \,;\, d_{\text{index}} \,) \longmapsto \texttt{index}(\, d \,;\, d'_{\text{index}} \,)} \qquad \frac{d \text{ val} \qquad d_{\text{index}} \text{ err}}{\texttt{index}(\, d \,;\, d_{\text{index}} \,) \text{ err}}$$

$$\frac{d_j \text{ val (for all } 0 \le j < n) \qquad 0 \le i < n}{\texttt{index}(\, \texttt{list}(\, d_0 \,;\, \ldots \,;\, d_{n-1} \,) \,;\, \texttt{int}[\, i \,] \,) \longmapsto d_i} \qquad \frac{d_j \text{ val (for all } 0 \le j < n) \qquad \neg(0 \le i < n)}{\texttt{index}(\, \texttt{list}(\, d_0 \,;\, \ldots \,;\, d_{n-1} \,) \,;\, \texttt{int}[\, i \,] \,) \text{ err}}$$

$$\frac{d \text{ val} \qquad d \text{ isnt } \texttt{list} \qquad d_{\text{index}} \text{ val}}{\texttt{index}(\, d \,;\, d_{\text{index}} \,) \text{ err}} \qquad \frac{d_j \text{ val (for all } 0 \le j < n) \qquad d_{\text{index}} \text{ val} \qquad d_{\text{index}} \text{ isnt } \texttt{int}}{\texttt{index}(\, \texttt{list}(\, d_0 \,;\, \ldots \,;\, d_{n-1} \,) \,;\, d_{\text{index}} \,) \text{ err}}$$

$$\frac{d \longmapsto d'}{\texttt{len}(\, d \,) \longmapsto \texttt{len}(\, d' \,)} \qquad \frac{d \text{ err}}{\texttt{len}(\, d \,) \text{ err}} \qquad \frac{d_j \text{ val (for all } 0 \le j < n)}{\texttt{len}(\, \texttt{list}(\, d_0 \,;\, \ldots \,;\, d_{n-1} \,) \,) \longmapsto \texttt{int}[\, n \,]}$$

$$\frac{d \text{ val} \qquad d \text{ isnt } \texttt{list}}{\texttt{len}(\, d \,) \text{ err}}$$

$$\frac{}{\texttt{fun}(\, f \,.\, x \,.\, d \,) \text{ val}} \qquad \frac{d \longmapsto d'}{\texttt{ap}(\, d \,;\, d_1 \,) \longmapsto \texttt{ap}(\, d' \,;\, d_1 \,)} \qquad \frac{d \text{ err}}{\texttt{ap}(\, d \,;\, d_1 \,) \text{ err}} \qquad \frac{d \text{ val} \qquad d_1 \longmapsto d'_1}{\texttt{ap}(\, d \,;\, d_1 \,) \longmapsto \texttt{ap}(\, d \,;\, d'_1 \,)}$$

$$\frac{d \text{ val} \qquad d_1 \text{ err}}{\texttt{ap}(\, d \,;\, d_1 \,) \text{ err}} \qquad \frac{d_1 \text{ val}}{\texttt{ap}(\, \texttt{fun}(\, f \,.\, x \,.\, d_2 \,) \,;\, d_1 \,) \longmapsto \{\texttt{fun}(\, f \,.\, x \,.\, d_2 \,), d_1/f, x\} d_2}$$

$$\frac{d \text{ val} \qquad d \text{ isnt } \texttt{fun} \qquad d_1 \text{ val}}{\texttt{ap}(\, d \,;\, d_1 \,) \text{ err}}$$

$$\frac{d_1 \longmapsto d'_1}{\texttt{let}(\, d_1 \,;\, x \,.\, d_2 \,) \longmapsto \texttt{let}(\, d'_1 \,;\, x \,.\, d_2 \,)} \qquad \frac{d_1 \text{ val}}{\texttt{let}(\, d_1 \,;\, x \,.\, d_2 \,) \longmapsto \{d_1/x\} d_2} \qquad \frac{d_1 \text{ err}}{\texttt{let}(\, d_1 \,;\, x \,.\, d_2 \,) \text{ err}}$$

$$\frac{d \longmapsto d'}{\texttt{isinstance}[\,c\,](\,d\,) \longmapsto \texttt{isinstance}[\,c\,](\,d'\,)} \qquad \frac{d \text{ err}}{\texttt{isinstance}[\,c\,](\,d\,) \text{ err}}$$

$$\frac{}{\texttt{isinstance}[\,\texttt{bool}\,](\,\texttt{bool}[\,b\,]\,) \longmapsto \texttt{bool}[\,\texttt{true}\,]} \qquad \frac{d \text{ isnt } \texttt{bool}}{\texttt{isinstance}[\,\texttt{bool}\,](\,d\,) \longmapsto \texttt{bool}[\,\texttt{false}\,]}$$

$$\frac{}{\texttt{isinstance}[\,\texttt{int}\,](\,\texttt{int}[\,i\,]\,) \longmapsto \texttt{bool}[\,\texttt{true}\,]} \qquad \frac{d \text{ isnt } \texttt{int}}{\texttt{isinstance}[\,\texttt{int}\,](\,d\,) \longmapsto \texttt{bool}[\,\texttt{false}\,]}$$

$$\frac{d_j \text{ val } (\text{for all } 0 \leq j < n)}{\texttt{isinstance}[\,\texttt{list}\,](\,\texttt{list}(\,d_0\,;\ldots;\,d_{n-1}\,)\,) \longmapsto \texttt{bool}[\,\texttt{true}\,]}$$

$$\frac{d \text{ isnt } \texttt{list}}{\texttt{isinstance}[\,\texttt{list}\,](\,d\,) \longmapsto \texttt{bool}[\,\texttt{false}\,]}$$

$$\frac{}{\texttt{isinstance}[\,\texttt{fun}\,](\,\texttt{fun}(\,f\,.\,x\,.\,d\,)\,) \longmapsto \texttt{bool}[\,\texttt{true}\,]} \qquad \frac{d \text{ isnt } \texttt{fun}}{\texttt{isinstance}[\,\texttt{fun}\,](\,d\,) \longmapsto \texttt{bool}[\,\texttt{false}\,]}$$

B Definition of FPC

In our formulation of **FPC**, we have:

- *labeled products and sums*, where there is a sort of labels which we assume are ordered
- a run-time error construct which causes the program to halt
- integer and list primitives

B.1 Grammar

Sort			Abstract form	Concrete form	
Label	l	::=	$\ldots$	$\ldots$	(labels)
Typ	τ	::=	$\mathtt{int}$	$\mathtt{int}$	(primitive)
			$\mathtt{list}(\tau)$	$\tau\,\mathtt{list}$	(primitive)
			$\mathtt{prod}(l \hookrightarrow \tau_l \mid l \in L)$	$\langle \tau_l \rangle_{l \in L}$	
			$\mathtt{sum}(l \hookrightarrow \tau_l \mid l \in L)$	$[\tau_l]_{l \in L}$	
			$\mathtt{arr}(\tau_1 ; \tau_2)$	$\tau_1 \rightarrow \tau_2$	
			$\mathtt{rec}(t \,.\, \tau)$	$\mathtt{rec}\, t\, \mathtt{is}\, \tau$	
Exp	e	::=	x	x	
			$\mathtt{error}[\tau]$	$\mathtt{error}[\tau]$	
			$\mathtt{int}[i]$	i	$(\ldots, -2, -1, 0, 1, 2, \ldots)$
			$\mathtt{plus}(e_1 ; e_2)$	$e_1 + e_2$	(integer addition)
			$\mathtt{leq}(d_1 ; d_2)$	$d_1 <= d_2$	
			$\mathtt{list}(e_0 ; \ldots ; e_{n-1})$	$[e_0 ; \ldots ; e_{n-1}]$	
			$\mathtt{append}(e_1 ; e_2)$	$e_1 \mathbin{@} e_2$	
			$\mathtt{index}(e ; e_{\text{index}})$	$e[e_{\text{index}}]$	
			$\mathtt{len}(e)$	$\mathtt{len}(e)$	
			$\mathtt{tpl}(l \hookrightarrow e_l \mid l \in L)$	$\langle l \hookrightarrow e_l \mid l \in L \rangle$	(labeled tuple)
			$\mathtt{pr}[l](e)$	$e \cdot l$	(projection from a labeled prod.)
			$\mathtt{in}[l][(l \hookrightarrow \tau_l)_{l \in L}](e)$	$l \cdot e$	(injection into a labeled sum)
			$\mathtt{case}(e ; l \hookrightarrow x_l \,.\, e_l \mid l \in L)$	$\mathtt{case}\, e\, \{ l \cdot x_l \hookrightarrow e_l \mid l \in L \}$	(labeled case)
			$\mathtt{fold}[t \,.\, \tau](e)$	$\mathtt{fold}(e)$	
			$\mathtt{unfold}(e)$	$\mathtt{unfold}(e)$	
			$\lambda[\tau](x \,.\, e)$	$\lambda(x : \tau)\, e$	
			$\mathtt{ap}(e ; e_1)$	$e(e_1)$	

B.2 Statics

B.2.1 Type Well-formedness

$\boxed{\Delta \vdash \tau\ \mathsf{type}}$

Informally: "τ only uses type variables present in Δ", where $\Delta ::= t_1\ \mathsf{type}, \ldots, t_n\ \mathsf{type}$.

$$\frac{}{\Delta, t\ \mathsf{type} \vdash t\ \mathsf{type}} \qquad \frac{}{\Delta \vdash \mathtt{int}\ \mathsf{type}} \qquad \frac{\Delta \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{list}(\tau)\ \mathsf{type}} \qquad \frac{\Delta \vdash \tau_l\ \mathsf{type}\ (\text{for all } l \in L)}{\Delta \vdash \mathtt{prod}(l \hookrightarrow \tau_l \mid l \in L)\ \mathsf{type}}$$

$$\frac{\Delta \vdash \tau_l\ \mathsf{type}\ (\text{for all } l \in L)}{\Delta \vdash \mathtt{sum}(l \hookrightarrow \tau_l \mid l \in L)\ \mathsf{type}} \qquad \frac{\Delta \vdash \tau_1\ \mathsf{type} \qquad \Delta \vdash \tau_2\ \mathsf{type}}{\Delta \vdash \mathtt{arr}(\tau_1 ; \tau_2)\ \mathsf{type}} \qquad \frac{\Delta, t\ \mathsf{type} \vdash \tau\ \mathsf{type}}{\Delta \vdash \mathtt{rec}(t \,.\, \tau)\ \mathsf{type}}$$

B.2.2 Expression Typing

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

$$\frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \frac{\vdash \tau\ \mathsf{type}}{\Gamma \vdash \mathtt{error}[\tau] : \tau}$$

$$\frac{}{\Gamma \vdash \mathtt{int}[i] : \mathtt{int}} \qquad \frac{\Gamma \vdash e_1 : \mathtt{int} \qquad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash \mathtt{plus}(e_1 ; e_2) : \mathtt{int}} \qquad \frac{\Gamma \vdash e_1 : \mathtt{int} \qquad \Gamma \vdash e_2 : \mathtt{int}}{\Gamma \vdash \mathtt{leq}(e_1 ; e_2) : \mathtt{bool}}$$

Note, we are use the following abbreviation:

$\mathtt{unit} \triangleq \mathtt{prod}(l \hookrightarrow - \mid l \in \varnothing)$ (empty labeled product)

$\mathtt{bool} \triangleq \mathtt{sum}(l \hookrightarrow \mathtt{unit} \mid l \in \mathtt{true}, \mathtt{false})$

$$\frac{\vdash \tau\ \mathsf{type} \qquad \Gamma \vdash e_j : \tau\ (\text{for all } 0 \le j < n)}{\Gamma \vdash \mathtt{list}(e_0 ; \ldots ; e_{n-1}) : \mathtt{list}(\tau)} \qquad \frac{\Gamma \vdash e_1 : \mathtt{list}(\tau) \qquad \Gamma \vdash e_2 : \mathtt{list}(\tau)}{\Gamma \vdash \mathtt{append}(e_1 ; e_2) : \mathtt{list}(\tau)}$$

$$\frac{\Gamma \vdash e : \mathtt{list}(\tau) \qquad \Gamma \vdash e_{\text{index}} : \mathtt{int}}{\Gamma \vdash \mathtt{index}(e ; e_{\text{index}}) : \tau} \qquad \frac{\Gamma \vdash e : \mathtt{list}(\tau)}{\Gamma \vdash \mathtt{len}(e) : \mathtt{int}}$$

$$\frac{\vdash \tau_1\ \mathsf{type} \qquad \Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda[\tau_1](x \,.\, e) : \mathtt{arr}(\tau_1 ; \tau_2)} \qquad \frac{\Gamma \vdash e : \mathtt{arr}(\tau_1 ; \tau_2) \qquad \Gamma \vdash e_1 : \tau_1}{\Gamma \vdash \mathtt{ap}(e ; e_1) : \tau_2}$$

$$\frac{\Gamma \vdash e_l : \tau_l\ (\text{for all } l \in L)}{\Gamma \vdash \mathtt{tpl}(l \hookrightarrow e_l \mid l \in L) : \mathtt{prod}(l \hookrightarrow \tau_l \mid l \in L)} \qquad \frac{\Gamma \vdash e : \mathtt{prod}(l \hookrightarrow \tau_l \mid l \in L)}{\Gamma \vdash \mathtt{pr}[l](e) : \tau_l}$$

$$\frac{\vdash \tau_l\ \mathsf{type}\ (\text{for all } l \in L) \qquad \Gamma \vdash e : \tau_l}{\Gamma \vdash \mathtt{in}[l][(l \hookrightarrow \tau_l)_{l \in L}](e) : \mathtt{sum}(l \hookrightarrow \tau_l \mid l \in L)}$$

$$\frac{\vdash \tau\ \mathsf{type} \qquad \Gamma \vdash e : \mathtt{sum}(l \hookrightarrow \tau_l \mid l \in L) \qquad \Gamma, x_l : \tau_l \vdash e_l : \tau\ (\text{for all } l \in L)}{\Gamma \vdash \mathtt{case}(e ; l \hookrightarrow x_l \,.\, e_l \mid l \in L) : \tau}$$

$$\frac{t\ \mathsf{type} \vdash \tau\ \mathsf{type} \qquad \Gamma \vdash e : \{\mathtt{rec}(t\,.\,\tau)/t\}\tau}{\Gamma \vdash \mathtt{fold}[t\,.\,\tau](e) : \mathtt{rec}(t\,.\,\tau)} \qquad \frac{\Gamma \vdash e : \mathtt{rec}(t\,.\,\tau)}{\Gamma \vdash \mathtt{unfold}(e) : \{\mathtt{rec}(t\,.\,\tau)/t\}\tau}$$

B.3 Dynamics

$$\frac{}{\mathtt{error}[\tau]\ \mathsf{err}}$$

$$\frac{}{\mathtt{int}[i]\ \mathsf{val}}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{plus}(e_1\,;e_2) \longmapsto \mathtt{plus}(e_1'\,;e_2)} \qquad \frac{e_1\ \mathsf{val} \qquad e_2 \longmapsto e_2'}{\mathtt{plus}(e_1\,;e_2) \longmapsto \mathtt{plus}(e_1\,;e_2')}$$

$$\frac{}{\mathtt{plus}(\mathtt{int}[i_1]\,;\mathtt{int}[i_2]) \longmapsto \mathtt{int}[i_1 + i_2]}$$

$$\frac{e_1 \longmapsto e_1'}{\mathtt{leq}(e_1\,;e_2) \longmapsto \mathtt{leq}(e_1'\,;e_2)} \qquad \frac{e_1\ \mathsf{val} \qquad e_2 \longmapsto e_2'}{\mathtt{leq}(e_1\,;e_2) \longmapsto \mathtt{leq}(e_1\,;e_2')}$$

$$\frac{i_1 \leq i_2}{\mathtt{leq}(\mathtt{int}[i_1]\,;\mathtt{int}[i_2]) \longmapsto \mathtt{true}} \qquad \frac{i_1 \not\leq i_2}{\mathtt{leq}(\mathtt{int}[i_1]\,;\mathtt{int}[i_2]) \longmapsto \mathtt{false}}$$

Note, we are use the following abbreviation:

$$\begin{aligned}
\mathtt{triv} &\triangleq \mathtt{tpl}(l \hookrightarrow - \mid l \in \varnothing) && \text{(empty labeled tuple)}\\
\mathtt{true} &\triangleq \mathtt{in}[\mathtt{true}][\mathtt{true},\mathtt{false}](\mathtt{triv})\\
\mathtt{false} &\triangleq \mathtt{in}[\mathtt{false}][\mathtt{true},\mathtt{false}](\mathtt{triv})
\end{aligned}$$

$$\frac{e_j \ \mathsf{val} \ (\text{for all } 0 \le j < k) \qquad e_k \longmapsto e'_k \qquad e'_m = e_m \ (\text{for all } m \ne k)}{\texttt{list}(\, e_0 \,;\, \ldots \,;\, e_{n-1} \,) \longmapsto \texttt{list}(\, e'_0 \,;\, \ldots \,;\, e'_{n-1} \,)}$$

$$\frac{e_j \ \mathsf{val} \ (\text{for all } 0 \le j < k) \qquad e_k \ \mathsf{err}}{\texttt{list}(\, e_0 \,;\, \ldots \,;\, e_{n-1} \,) \ \mathsf{err}} \qquad \frac{e_j \ \mathsf{val} \ (\text{for all } 0 \le j < n)}{\texttt{list}(\, e_0 \,;\, \ldots \,;\, e_{n-1} \,) \ \mathsf{val}}$$

$$\frac{e_1 \longmapsto e'_1}{\texttt{append}(\, e_1 \,;\, e_2 \,) \longmapsto \texttt{append}(\, e'_1 \,;\, e_2 \,)} \qquad \frac{e_1 \ \mathsf{val} \qquad e_2 \longmapsto e'_2}{\texttt{append}(\, e_1 \,;\, e_2 \,) \longmapsto \texttt{append}(\, e_1 \,;\, e'_2 \,)}$$

$$\frac{e_j \ \mathsf{val} \ (\text{for all } 0 \le j < m) \qquad e'_j \ \mathsf{val} \ (\text{for all } 0 \le j < n)}{\texttt{append}(\, \texttt{list}(\, e_0 \,;\, \ldots \,;\, e_{m-1} \,) \,;\, \texttt{list}(\, e'_0 \,;\, \ldots \,;\, e'_{n-1} \,) \,) \longmapsto \texttt{list}(\, e_0 \,;\, \ldots \,;\, e_{m-1} \,;\, e'_0 \,;\, \ldots \,;\, e'_{n-1} \,)}$$

$$\frac{e \longmapsto e'}{\texttt{index}(\, e \,;\, e_{\text{index}} \,) \longmapsto \texttt{index}(\, e' \,;\, e_{\text{index}} \,)} \qquad \frac{e \ \mathsf{val} \qquad e_{\text{index}} \longmapsto e'_{\text{index}}}{\texttt{index}(\, e \,;\, e_{\text{index}} \,) \longmapsto \texttt{index}(\, e \,;\, e'_{\text{index}} \,)}$$

$$\frac{e_j \ \mathsf{val} \ (\text{for all } 0 \le j < n) \qquad 0 \le i < n}{\texttt{index}(\, \texttt{list}(\, e_0 \,;\, \ldots \,;\, e_{n-1} \,) \,;\, \texttt{int}[\, i \,] \,) \longmapsto e_i} \qquad \frac{e_j \ \mathsf{val} \ (\text{for all } 0 \le j < n) \qquad \neg(0 \le i < n)}{\texttt{index}(\, \texttt{list}(\, e_0 \,;\, \ldots \,;\, e_{n-1} \,) \,;\, \texttt{int}[\, i \,] \,) \ \mathsf{err}}$$

$$\frac{e \longmapsto e'}{\texttt{len}(\, e \,) \longmapsto \texttt{len}(\, e' \,)} \qquad \frac{e \ \mathsf{err}}{\texttt{len}(\, e \,) \ \mathsf{err}} \qquad \frac{e_j \ \mathsf{val} \ (\text{for all } 0 \le j < n)}{\texttt{len}(\, \texttt{list}(\, e_0 \,;\, \ldots \,;\, e_{n-1} \,) \,) \longmapsto \texttt{int}[\, n \,]}$$

$$\frac{e_l \ \mathsf{val} \ (\text{for all } l < k) \qquad e_k \longmapsto e'_k \qquad e'_m = e_m \ (\text{for all } m \ne k)}{\texttt{tpl}(\, l \hookrightarrow e_l \mid l \in L \,) \longmapsto \texttt{tpl}(\, l \hookrightarrow e'_l \mid l \in L \,)} \qquad \frac{e_l \ \mathsf{val} \ (\text{for all } l < k) \qquad e_k \ \mathsf{err}}{\texttt{tpl}(\, l \hookrightarrow e_l \mid l \in L \,) \ \mathsf{err}}$$

$$\frac{e_l \ \mathsf{val} \ (\text{for all } l)}{\texttt{tpl}(\, l \hookrightarrow e_l \mid l \in L \,) \ \mathsf{val}} \qquad \frac{e \longmapsto e'}{\texttt{pr}[\, l \,](\, e \,) \longmapsto \texttt{pr}[\, l \,](\, e' \,)} \qquad \frac{e_l \ \mathsf{val} \ (\text{for all } l)}{\texttt{pr}[\, l \,](\, \texttt{tpl}(\, l \hookrightarrow e_l \mid l \in L \,) \,) \longmapsto e_l}$$