Last updated: January 26, 2023

1. Introduction

This manual describes the λ^+ programming language, which is a simple, functional language small enough that a complete interpreter for λ^+ can be implemented in a quarter-long course. The programming language is similar in nature to the untyped λ-calculus, but extends the λ-calculus with constructs such as let bindings and named functions to make it more convenient to program in λ^+. The language is also very similar to real-world functional programming languages, such as Lisp and OCaml.

This manual gives an informal overview of the language, describes its syntax, and gives precise semantics to the language. At the beginning of the semester, students should only focus on the overview discussion of λ^+ in Section 2. Additional sections will be added to this manual as we progress through the course.

2. Overview

A λ^+ program is simply an expression, and executing the program is equivalent to evaluating the expression. For example, the simple expression

```
8
```

is a valid λ^+ program, and the value of this program is the integer 8.

The most basic expressions in λ^+ are integer constants and binary arithmetic operators, such as `+`, `*`, `-`. For example,

```
(3 + 6 - 1) * 2
```

is a valid λ^+ expression with value 16.

2.1. Let Bindings.

Let bindings in λ^+ allow us to name and reuse expressions. Specifically, an expression of the form

```
let x = e1 in e2
```

binds the value of e1 to identifier x and evaluates e2 under this binding. The expression e2 is referred to as the body of the let expression, and e1 is called the *initializer*. The value of the let expression is the result of evaluating e2. For example,

```
let x = 3+5 in x-2
```

evaluates to 6, while the expression

```
let x = 3+5 in x+y
```

yields the run-time error:

```
Unbound variable y
```

since the identifier y is not bound (i.e. has not been "defined") in the body of the let expression.

Let expressions in λ^+ can be arbitrarily nested. For example, consider the nested let expressions:

```
let x = 3+5 in
let y = 2*x in
y+x
```

This is a valid λ^+ expression and evaluates to 24. Observe that the body of the first let expression is `let y = 2*x in y+x` while the body of the second (nested) let expression is y+x. As another example, consider the nested let expression:

```
let x = let x = 3 in
x+1 in x
```

evaluates to 4. The initializer for the first (outer-level) let expression is `let x=3 in x+1`, which evaluates to 4. Thus, the value of x in the body of the outer let expression is 4. As a final example of let expressions, consider:

```
let x = 2 in
let x = 3 in
x
```

evaluates to 3, since each identifier refers to the most recently bound value.

2.2. **"Boolean" operations.** For simplicity, λ^+ does not have the typical boolean values of true and false. Instead, λ^+ treats 0 as false and non-zero values as true. The supported boolean operations are $\&\&, ||$ as well as the comparison operators $=, >, <$. For example, the following expression evaluates to 1:

```
((1 = 1) || 3 = 4) && 1
```

λ^+ also supports *if-then-else expressions*, which evaluate to the then-expression when the condition expression is non-zero or the else-expression otherwise.

```
if 1 then 2 + 3 else 3 * 4
```

This will evaluate to 5.

Note that it is possible to simulate "else if" clauses by nesting multiple if-then-else. For example,

```
let x = 1 in
if x = 0
  then 3
  else if x = 1 then 5
  else 7
```

is understood as

```
let x = 1 in
if x = 0
  then 3
  else (if x = 1 then 5 else 7)
```

and evaluates to 5.

2.3. **Lambda Expressions and Applications.** As in λ-calculus, λ^+ also provides lambda expressions of the form:

```
lambda x1, ..., xn. e
```

For example, the λ^+ expression

```
lambda x, y. x+y
```

corresponds to an unnamed (anonymous) function that takes two arguments x and y and evaluates their sum. The above λ^+ expression is equivalent to the λ^+ expression:

```
lambda x. lambda y. x+y
```

The transformation from the first lambda expression `lambda x, y. x+y` to the second expression `lambda x. (lambda y. x+y)` is known as *currying*. We could remove multi-argument lambdas from the language without reducing its expressive power; in fact, implementations of λ^+ only need to implement single-argument lambdas, with multi-argument lambdas treated as "syntactic sugar" in λ^+. That is, multi-argument lambdas merely provide a more convenient way to write expressions that can already be expressed using other constructs in the language (namely single-argument lambdas, here).

Of course, for lambda expressions to be useful, we also need to be able to apply arguments to lambda abstractions. Application in λ^+ is the same as application in λ-calculus, with the form `e1 e2 ... en`. As in λ-calculus, application is left-associative, so that `e1 e2 ... en` is read as `(((e1 e2) e3) ... ) en)`.

The expression `(lambda x. e1) e2` evaluates e2 with e1 bound to x. For example, the application expression

```
((lambda x. (lambda y. x + y)) 6) 7
```

evaluates to 13. Note that this can be conveniently written in λ^+ as

```
(lambda x, y. x + y) 6 7
```

Keep in mind that lambdas are right-associative while application is left-associative, i.e. the following are equivalent:

```
lambda x. lambda y. x + y     <==>    lambda x. (lambda y. x + y)
f e1 e2                       <==>    (f e1) e2
```

As a more interesting example, consider the application expression

```
(lambda x, y. x + y) 6
```

which evaluates to the lambda expression

```
lambda y. 6 + y
```

This example illustrates an interesting feature of λ^+: Expressions in λ^+ do not have to evaluate to constants; they can be *partially evaluated* functions, such as `lambda y. (6 + y)` in this example.

Here, we highlight two possible mistakes one can make using application expressions in λ^+. First, someone may write

```
lambda x. x 4
```

to try to apply a function `lambda x. x` to 4, but what this will really do is create a function that accepts an argument `x` and then apply `x` to 4. The correct way of writing this expression is

```
(lambda x. x) 4
```

As a second caveat, the application expression

```
((let x = 2 in x) 3)
```

is a syntactically valid λ^+ expression but will yield the run-time error:

```
Run-time error in expression (let x = 2 in x 3)
Only lambda expressions can be applied to other expressions
```

The problem here is that the first expression e1 in the application (e1 e2) must evaluate to a lambda expression. Only functions can be applied to arguments. On the other hand, the following expression

```
let x = lambda y. y in
(x 3)
```

is both syntactically and semantically valid and evaluates to 3.

2.4. **Named Function Definitions.** In addition to `lambda` expressions, which correspond to anonymous function definitions, the λ^+ language also makes it possible to define named functions using the syntax:

```
fun f with x1, ..., xn = e1 in e2
```

Here f is the name of the function being defined, x1, ...xn are the arguments of function f, and e1 is the body of function f. The value of the fun expression is the result of evaluating e2 under this definition of f.

Using named function definitions, we can now define recursive functions. The following program defines a recursive function for computing factorial, and the expression f 4 in the body evaluates to 4!, i.e., 24.

```
fun f with n =
  if n = 0
  then 1
  else n * (f (n-1))
in f 4
```

Like multi-argument lambdas, named function definitions are also "syntactic sugar" in λ^+. Specifically, the function definition

```
fun f with x = e1 in e2
```

is equivalent to the following let expression:

```
let f = fix (lambda f. lambda x. e1)
in e2
```

where the fix operator models *fixed-point combinators* in untyped lambda calculus.

Here is another example that illustrates the use of named functions in λ^+:

```
fun even with x =
  if x = 0 then 1
  else if x = 1 then 0
  else even (x - 2)
in
fun odd with x = even (x + 1)
in
odd 7
```

This λ^+ program evaluates to 1, as expected. To see why, observe that the sequence of function calls is:

```
  odd 7
= even 8
= even 6
= even 4
= even 2
= even 0
= 1
```

2.5. **Lists.** In addition to integers, the λ^+ language also supports linked lists. A list in λ^+ is a general data structure consisting of either:

- The empty list constant Nil
- The *cons cell* $e_1 \mathbin{@} e_2$, where e_1 is the *head* of the list and e_2 is the *tail* of the list.

By convention, we take @ to be right-associative, so that 1@2@3@ Nil is read as 1@(2@(3@ Nil)). λ^+ supports the following list operations:

- isnil e: The expression (isnil e) evaluates to 1 if e is Nil, or 0 otherwise.
- $e_1 \mathbin{@} e_2$: The expression $e_1 \mathbin{@} e_2$ evaluates to a new list with head corresponding to the value of e_1 and tail corresponding to the value of e_2.
- $!\,e$: The expression $!\,e$ yields the head of the list if e evaluates to a cons cell, or causes a runtime error otherwise. For example, !(2@3@ Nil) evaluates to 2.
- $\#\,e$: The expression $\#\,e$ yields the tail of the list if e evaluates to a cons cell, or causes a runtime error otherwise. For example, #(2@3@ Nil) evaluates to 3@ Nil, and #(1@2@3@ Nil) evaluates to 2@3@ Nil and #(2@ Nil) evaluates to Nil.

Consider the following example of a function on lists:

```
fun length with list  =
  if isnil list
    then 0
    else (length #list) + 1
in
length (1 @ 2 @ 2 @ 1 @ Nil)
```

Here, we define a function length to compute the number of elements in a list and use this function on the list (1@2@2@1@ Nil). The above expression evaluates to 4 (exercise: try expanding this expression out by hand).

As another example, here is a program that adds n to each element in a given list:

```
fun add with l, n =
  if isnil l
  then Nil
  else (!l + n) @ (add #l n)
in add (1 @ 2 @ 3 @ Nil) 2
```

This program evaluates to the list 3 @ 4 @ 5 @ Nil.

3. Syntax

In any programming language, there are two types of "syntax" that the language designer is concerned with. The first type is *concrete syntax*, the syntax used for the source code that the programmer will type in. The concrete syntax will specify how the linear sequence of *tokens* making up the source code should be converted into a data structure called a *parse tree*. The parse tree will then be converted into an *abstract syntax tree*, which is a data structure consisting of the core parts of the language with all irrelevant notational constructs (e.g. parentheses, braces, etc.) removed. Interpreters and compilers will operate on abstract syntax trees.

3.1. **Concrete Syntax.**

The complete *concrete syntax* of λ^+ is specified by the context-free grammar presented in Figure 1.

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{Expr} \\
\mathit{Expr} & ::= & \mathit{let}\ \mathit{ID}\ =\ \mathit{Expr}\ \mathit{in}\ \mathit{Expr} \\
& | & \mathit{fun}\ \mathit{ID}\ \mathit{with}\ \mathit{Idlist}\ =\ \mathit{Expr}\ \mathit{in}\ \mathit{Expr} \\
& | & \mathit{lambda}\ \mathit{Idlist}.\ \mathit{Expr} \\
& | & \mathit{fix}\ \mathit{Expr} \\
& | & \mathit{if}\ \mathit{Expr}\ \mathit{then}\ \mathit{Expr}1\ \mathit{else}\ \mathit{Expr}2 \\
& | & \mathit{Expr}1 \oplus \mathit{Expr}2 \qquad (\oplus \in \{+, -, *, =, >, <, \&\&, ||\}) \\
& | & !\,\mathit{Expr} \\
& | & \#\,\mathit{Expr} \\
& | & \mathit{Expr}1\ \mathit{Expr}2 \\
& | & \mathsf{Nil} \\
& | & \mathit{Expr}1\,@\,\mathit{Expr}2 \\
& | & \mathsf{isnil}\ \mathit{Expr} \\
& | & \mathsf{INT\_CONST} \\
& | & \mathit{ID}
\end{array}
$$

Figure 1. Concrete syntax of the λ^+ language.

Observe that this grammar in Figure 1 is ambiguous, and we discuss the intended meaning of the ambiguous constructs. The first source of ambiguity in the grammar is binary operators. For example, the λ^+ expression 2*3+4 can be parsed in two ways: either as (2*3)+4 or as 2*(3+4). To disambiguate the grammar, we therefore need to declare the precedence and associativity of operators. Figure 2 shows the precedence of operators, where operators higher up in the figure have higher precedence than those lower down in the figure. Operators shown on the same line have the same precedence.

To illustrate how precedence declarations allow us to resolve ambiguities, consider again the expression 2*3+4. Since * has higher precedence than +, this means the expression should be

!, #
@
*
+, −
<, >, =
&&, ||

Figure 2. Operator precedence

parsed as (2*3)+4, instead of 2*(3+4). Similarly, since ! has precedence than @, this means the expression !x@y should be understood as (!x)@y rather than !(x@y).

Observe that precedence declarations are not sufficient to resolve all ambiguities concerning these operators; we also need associativity declarations. For example, precedence declarations alone are not sufficient to decide whether the expression 1+2+3 should be parsed as (1+2)+3 or as 1+(2+3). To resolve this issue, we also need to specify the associativity of the binary operators.

In the λ^+ language, the binary operators +, -, *, /, =, <, and > are all left-associative; the only right-associative operator is @. This indicates that the expression 1+2+3 should be parsed as (1+2)+3, while the expression 1@2@3 should be parsed as 1@(2@3).

3.2. **Abstract Syntax.** The complete *abstract syntax* of λ^+ is specified by the context-free grammar presented in Figure 3.

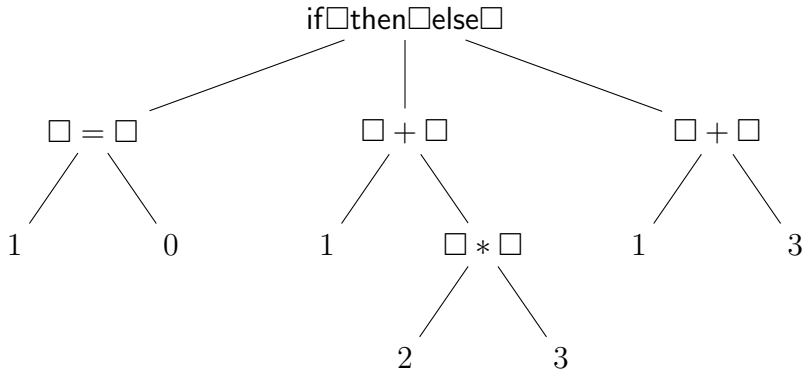
Syntactic construct			Description
$\oplus \in \mathit{ArithOp}$	::=	$+ \mid - \mid *$	
$\odot \in \mathit{Pred}$	::=	$= \mid < \mid > \mid$ && $\mid \mid\mid$	
$\diamond \in \mathit{BinOp}$	::=	$\oplus \mid \odot$	
$e \in \mathit{Expr}$	::=	i	integer
	$\mid$	x	variable
	$\mid$	$e_1 \diamond e_2$	binary op
	$\mid$	$\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$	if-then-else
	$\mid$	$\mathsf{lambda}\ x.\ e$	lambda abstraction
	$\mid$	$\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$	let binding
	$\mid$	fix	fixed-point operator
	$\mid$	$e_1\ e_2$	function application
	$\mid$	Nil	empty list
	$\mid$	$e_1 \mathbin{@} e_2$	cons cell
	$\mid$	$!\, e$	list head
	$\mid$	$\#\, e$	list tail
	$\mid$	$\mathsf{isnil}\ e$	is nil predicate

FIGURE 3. Abstract syntax of the λ^+ language.

The abstract syntax is a mathematical description of how to inductively construct an abstract syntax tree (AST). For example, the expression

$$\mathsf{if}(1 = 0)\,\mathsf{then}(1 + (2 * 3))\,\mathsf{else}(1 + 3)$$

corresponds to the following AST:



Note that there is inherently no ambiguity in an AST, because it is a data structure that encodes the order of operations.

The remaining sections in the manual will only use the abstract syntax, so that we can precisely reason about the expressions in our program.

4. Operational Semantics

Recall that a *semantics* of λ^+ defines the meaning of every expression in the λ^+. In *operational semantics*, we mathematically model a machine that can execute a program using some *state*. Some different methods to operational semantics are:

- A type of operational semantics called *small-step operational semantics* is based on defining *transitions* between states. To evaluate an expression, one constructs an initial state from the program and then takes transitions until a *final state* is reached. This is similar to the automata that you learned about in CS 138.
- Another type is *big-step operational semantics*, which is based on defining (often recursively or inductively) how a program can be directly executed from its expression down to a final *value*.

Although small-step is much more popular among programming language theorists, big-step is usually more intuitive to most people and how languages are typically implemented in practice. Thus, we will be using (and you will be implementing) a big-step operational semantics for λ^+ in this course.

4.1. The Machine Model.

Before we begin, we must understand what our machine model will require. We will need some notion of when our machine is "done" executing a program; let us use the term *value* to refer to the final expression obtained by executing a program.

We formally define a value using the following inductive definition:

- Any integer i is a value.
- Any lambda expression $\mathsf{lambda}\, x.\ e$ is a value.
- Nil is a value.
- If v_1, v_2 are values, then $v_1 \mathbin{@} v_2$ are values.
- No other expression is a value.

For example, the following expressions are all values:

$$10$$
$$\mathsf{lambda}\, x.\ 1 + 2$$
$$\mathsf{Nil}$$
$$10 \mathbin{@} \mathsf{lambda}\, y.\ y$$

The following expressions are **not** values:

$$1 + 2$$
$$(\mathsf{lambda}\, x.\ 1 + 2)10$$
$$\mathsf{if\ Nil\ then}\ 10\ \mathsf{else}\ 20$$
$$(1 + 2) \mathbin{@} \mathsf{Nil}$$

We denote values by variations of the symbol v.

4.1.1. *The Evaluation Relation.*

We can now define how program states are related to final values:

Definition 4.1 (Big-step evaluation relation)**.** The big-step evaluation relation has the form

$$e \Downarrow v$$

that asserts that e will evaluate to v.

The idea is that whenever this relation holds, then e will evaluate to the final value v. As the reader shall see shortly, this will allow us to both 1) precisely define the meaning of "evaluation"; and 2) provide us an algorithm to perform evaluation.

4.2. **Arithmetic and Boolean Operations.** Now, we finally have all of the tools required for us to be able to specify the semantics of λ^+ expressions. Let us start by defining the *evaluation rule* for the simplest of λ^+ expressions, namely integer constants:

$$\frac{}{i \Downarrow i}\ \textsc{Int}$$

The evaluation rule Int says that any integer constant i will evaluate to itself.

This rule is not very useful by itself, so let's include an evaluation rule for addition:

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2}{e_1 + e_2 \Downarrow i_1 + i_2}\ \textsc{Add}$$

which says that if e_1 and e_2 both evaluate to integers, then $e_1 + e_2$ evaluates to the sum of those integers. Note that we abuse notation here: the $+$ on the left-hand side of the conclusion is merely a symbol in the AST, while we take the $+$ on the right-hand side of the conclusion to mean integer addition.

Equipped with these two rules, we can now show how they can be used. Consider the following *proof tree* or *derivation* of how $(1 + 2) + 4$ evaluates to 7:

$$\dfrac{\textsc{Add}\ \dfrac{\textsc{Int}\ \dfrac{}{1 \Downarrow 1} \qquad \textsc{Int}\ \dfrac{}{2 \Downarrow 2}}{1 + 2 \Downarrow 3} \qquad \textsc{Int}\ \dfrac{}{4 \Downarrow 4}}{(1 + 2) + 4 \Downarrow 7}\ \textsc{Add}$$

This says that in order to evaluate $(1+2)+4$, first we must match against some evaluation rule. The only rule that matches is the Add rule, so then we recursively proceed to evaluate $1 + 2$. But again only the Add rule matches, so we evaluate 1 to 1 by the Int rule (and similarly for 2). Overall, $1 + 2$ evaluates to 3 under Add, so then we continue through with the use of the Add rule on $(1 + 2) + 4$ to evaluate to 7. Note that this can be easily turned into an algorithm which can evaluate the expression.

The addition rule can be generalized to all of the arithmetic operators in λ^+:

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2}{e_1 \oplus e_2 \Downarrow i_1 \oplus i_2}\ \textsc{Arith}$$

where $\oplus \in \{+, -, *, \}$. Note that we are abusing the notation slightly here: the $\oplus$ in $i_1 \oplus i_2$ is meant as the actual operation on the integers, whereas the $\oplus$ in $e_1 \oplus e_2$ is used as part of a tree of symbols.

The predicate operators $\odot \in \{=, <, >\}$ evaluate to 0 if the predicate does not hold and to 1 otherwise, as stated in the following operational semantic rules:

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad i_1 \odot i_2}{e_1 \odot e_2 \Downarrow 1}\ \textsc{PredTrue}$$

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad \neg(i_1 \odot i_2)}{e_1 \odot e_2 \Downarrow 0}\ \textsc{PredFalse}$$

Again, we are abusing notation here, where the $i_1 \odot i_2$ are taken to mean the actual predicate on integers, not on symbols in λ^+.

Lastly, we present the semantics for if-then-else expressions. The operational semantics for this expression consists of two inference rules, one in which the conditional evaluates to a non-zero value, and another in which it evaluates to zero:

$$\frac{e_1 \Downarrow i \qquad i \neq 0 \qquad e_2 \Downarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v}\ \textsc{IfTrue}$$

$$\frac{e_1 \Downarrow 0 \qquad e_3 \Downarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v}\ \textsc{IfFalse}$$

In the first rule IfTrue, if the expression e_1 evaluates to a non-zero integer i, then the if-then-else expression overall evaluates to the expression e_1 in the then branch, which yields the value v. Observe that in the case where e_1 evaluates to a non-zero value, the expression e_3 is never evaluated. The second rule IfFalse is similar to first one, except that when e_1 evaluates to 0, then evaluation proceeds by evaluating e_3.

4.3. **Functions and Application.** As in λ-calculus, the lambda abstractions and applications follow the substitution model of evaluation. However, we present them here for completeness.

First, consider lambda abstractions. Since they are essentially function definitions, we cannot really evaluate them until they are "called", i.e., they are applied to some value. Thus, lambda abstractions just evaluate to themselves:

$$\frac{}{\mathsf{lambda}\ x.\ e \Downarrow \mathsf{lambda}\ x.\ e}\ \textsc{Lambda}$$

Now consider the application rule.

$$\frac{e_1 \Downarrow \mathsf{lambda}\ x.\ e_1' \qquad e_2 \Downarrow v \qquad [x \mapsto v]e_1' \Downarrow v'}{(e_1\ e_2) \Downarrow v'}\ \textsc{App}$$

To evaluate the application $(e_1\ e_2)$, we first evaluate the expression e_1. Note that application is semantically nonsensical if the expression e_1 is not a lambda abstraction; thus, the operational semantics "get stuck" if e_1 is not a lambda abstraction of the form $\mathsf{lambda}\ x.\ e_1'$. This notion of "getting stuck" in the operational semantics corresponds to having a runtime error. Assuming the expression e_1 evaluates to a lambda expression $\mathsf{lambda}\ x.\ e_1'$, next we evaluate the argument e_2 to v. Lastly, we substitute x with v in the function body $e1'\ x$ as in β-reduction in lambda calculus.

Note that we evaluate e_2 first before binding it to x; thus, λ^+ uses a *call-by-value* evaluation order. An alternate semantics may choose not to evaluate e_2, instead directly substituting it into the function body; this alternate evaluation order is called *call-by-name*.

4.4. **Let Bindings and Variables.** The meaning of let bindings in λ^+ can be defined in terms of lambda abstractions and applications[1]: a let expression $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$ is equivalent to $(\mathsf{lambda}\, x.\ e_2)e_1$, so the rule for let expressions is defined as as follows:

$$\frac{e_1 \Downarrow v_1 \qquad [x \mapsto v_1]e_2 \Downarrow v_2}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \Downarrow v_2}\ \textsc{Let}$$

To evaluate a let expression $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$, we first evaluate the initial expression e_1, which yields value v_1. Then, to evaluate the body e_2, we substitute occurrences of identifier x in e_2 with value v_1, and evaluate the substituted expression, which yields value v_2, the result of evaluating the entire let expression.

4.5. **The Fixed-point Operator.** The fix operator behaves like various fixed-point operators in untyped lambda calculus: it approximates a recursive function by unrolling the generator of the recursive function on demand:

$$\frac{e \Downarrow \mathsf{lambda}\, f.\ e' \qquad [f \mapsto \mathsf{fix}\ (\mathsf{lambda}\, f.\ e')]e' \Downarrow v}{\mathsf{fix}\ \ e \Downarrow v}\ \textsc{Fix}$$

That is, to evaluate a fixed-point expression $\mathsf{fix}\ \ e$, we first evaluate e to a lambda expression $\mathsf{lambda}\, f.\ e'$, which is the generator of a recursive function that refers to itself as f. We then apply the lambda expression to a copy of the fixed-point expression (i.e. substituting $\mathsf{fix}\,(\mathsf{lambda}\, f.\ e')$ for f in e'), essentially unrolling the body of the recursive function once.

4.6. **List Operators.** Recall that a list is either the empty list Nil, or it is a *cons cell* $(e_1 \mathbin{@} e_2)$ where e_1 is the head of the list and e_2 is the tail of the list. The evaluation rules for constructing lists are shown below:

$$\frac{}{\mathsf{Nil} \Downarrow \mathsf{Nil}}\ \textsc{Nil} \qquad\qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \mathbin{@} e_2 \Downarrow v_1 \mathbin{@} v_2}\ \textsc{Cons}$$

Now that we have defined the meaning of Nil, we can now state the semantics of the isnil operator. Since any list value can either be Nil or a cons cell, we have two cases:

$$\frac{e \Downarrow \mathsf{Nil}}{\mathsf{isnil}\, e \Downarrow 1}\ \textsc{IsNilTrue} \qquad\qquad \frac{e \Downarrow v_1 \mathbin{@} v_2}{\mathsf{isnil}\, e \Downarrow 0}\ \textsc{IsNilFalse}$$

Here, which rule matches triggered will depend on whether e evaluates to Nil or not. If e is not a list, then the evaluation will get stuck.

Next, let us discuss the semantics of the ! and # operators. These operators can clearly only be applied to cons cells, so the evaluation rules are simply:

$$\frac{e \Downarrow v_1 \mathbin{@} v_2}{!\, e \Downarrow v_1}\ \textsc{Head} \qquad\qquad \frac{e \Downarrow v_1 \mathbin{@} v_2}{\#\, e \Downarrow v_2}\ \textsc{Tail}$$

4.7. **Summary.** The evaluation rules in the big-step operational semantics for λ^+ are summarized in Figure 4.

[1]The reason we retain let bindings in the abstract syntax, instead of treating it as syntactic sugars, is because the static behavior (i.e. the typing rule) of a let binding is different from the static behavior of its de-sugared form, and hence requires special treatment. For more details, please refer to the next section.

$$\frac{}{i \Downarrow i}\ \textsc{Int} \qquad \frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2}{e_1 \oplus e_2 \Downarrow i_1 \oplus i_2}\ \textsc{Arith}$$

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad i_1 \odot i_2}{e_1 \odot e_2 \Downarrow 1}\ \textsc{PredTrue}$$

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad \neg(i_1 \odot i_2)}{e_1 \odot e_2 \Downarrow 0}\ \textsc{PredFalse}$$

$$\frac{e_1 \Downarrow i \qquad i \neq 0 \qquad e_2 \Downarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v}\ \textsc{IfTrue} \qquad \frac{e_1 \Downarrow 0 \qquad e_3 \Downarrow v}{\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \Downarrow v}\ \textsc{IfFalse}$$

$$\frac{}{\mathsf{lambda}\, x.\ e \Downarrow \mathsf{lambda}\, x.\ e}\ \textsc{Lambda}$$

$$\frac{e_1 \Downarrow \mathsf{lambda}\, x.\ e_1' \qquad e_2 \Downarrow v \qquad [x \mapsto v]e_1' \Downarrow v'}{(e_1\ e_2) \Downarrow v'}\ \textsc{App}$$

$$\frac{e_1 \Downarrow v_1 \qquad [x \mapsto v_1]e_2 \Downarrow v_2}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \Downarrow v_2}\ \textsc{Let}$$

$$\frac{e \Downarrow \mathsf{lambda}\, f.\ e' \qquad [f \mapsto \mathsf{fix}\ (\mathsf{lambda}\, f.\ e')]e' \Downarrow v}{\mathsf{fix}\ \ e \Downarrow v}\ \textsc{Fix}$$

$$\frac{}{\mathsf{Nil} \Downarrow \mathsf{Nil}}\ \textsc{Nil} \qquad \frac{e_1 \Downarrow v_1 \qquad e_2 \Downarrow v_2}{e_1 \mathbin{@} e_2 \Downarrow v_1 \mathbin{@} v_2}\ \textsc{Cons}$$

$$\frac{e \Downarrow \mathsf{Nil}}{\mathsf{isnil}\, e \Downarrow 1}\ \textsc{IsNilTrue} \qquad \frac{e \Downarrow v_1 \mathbin{@} v_2}{\mathsf{isnil}\, e \Downarrow 0}\ \textsc{IsNilFalse}$$

$$\frac{e \Downarrow v_1 \mathbin{@} v_2}{!\, e \Downarrow v_1}\ \textsc{Head} \qquad \frac{e \Downarrow v_1 \mathbin{@} v_2}{\#\, e \Downarrow v_2}\ \textsc{Tail}$$

FIGURE 4. Evaluation rules in the big-step operational semantics of the λ^+ language.

5. Types

5.1. **Basic definitions.** The syntax of *types* in λ^+ is shown in Figure 5.

$$\begin{array}{rcl} \mathsf{T} & ::= & \mathsf{T}_1 \to \mathsf{T}_2 \\ & | & \mathsf{Int} \\ & | & \mathsf{List}[\mathsf{T}] \end{array}$$

Figure 5. Types of the λ^+ language.

Definition 5.1 (Typing Environment)**.** A typing environment (or typing context) Γ is a mapping from variables to types

$$\Gamma = x_1 : \mathsf{T}_1, x_2 : \mathsf{T}_2, \ldots, x_n : \mathsf{T}_n$$

As a shorthand, we will use $\Gamma, x : \mathsf{T}$ to mean the typing environment obtained by extending Γ with $x : \mathsf{T}$. Note that order within a typing environment does not matter if the variables are unique: $x : \mathsf{T}_1, y : \mathsf{T}_2$ is the same as $y : \mathsf{T}_2, x : \mathsf{T}_1$. However, order *does* matter if the same variable is repeated: for example, the the type of x in the context $\Gamma = x : \mathsf{T}_1, x : \mathsf{T}_2$ is T_2. In this case, we can simply write $\Gamma = x : \mathsf{T}_2$ since we can safely remove $x : \mathsf{T}_1$.

Definition 5.2 (Typing Relation)**.** The typing relation

$$\Gamma \vdash e : \mathsf{T}$$

asserts that e has type T under the typing environment Γ.

For example,

$$\vdash 1 : \mathsf{Int}$$

asserts that 1 has type Int under the empty typing environment.

5.2. **Explanation of the typing rules.** TODO (please see the relevant lecture slides for the explanation in the meantime).

As with evaluation rules, we have a set of *typing rules* that allow us to construct a derivation tree (i.e. a proof) that some expression has some type.

5.3. **Summary.** The typing rules for λ^+ are summarized in Figure 6.

$$\frac{}{\Gamma \vdash i : \mathsf{Int}}\ \text{T-Int} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int} \qquad \square \in \{+, -, *\}}{\Gamma \vdash e_1 \square e_2 : \mathsf{Int}}\ \text{T-Arith}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int} \qquad \square \in \{\&\&, ||\}}{\Gamma \vdash e_1 \square e_2 : \mathsf{Int}}\ \text{T-Logic}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int} \qquad \square \in \{<, >\}}{\Gamma \vdash e_1 \square e_2 : \mathsf{Int}}\ \text{T-Ineq} \qquad \frac{\Gamma \vdash e_1 : \mathsf{T} \qquad \Gamma \vdash e_2 : \mathsf{T}}{\Gamma \vdash e_1 = e_2 : \mathsf{Int}}\ \text{T-Eq}$$

$$\frac{\Gamma, x : \mathsf{T}_1 \vdash e : \mathsf{T}_2}{\Gamma \vdash \mathsf{lambda}\, x : \mathsf{T}_1.\ e : \mathsf{T}_1 \to \mathsf{T}_2}\ \text{T-Lambda}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{T}_1 \to \mathsf{T}_2 \qquad \Gamma \vdash e_2 : \mathsf{T}_1}{\Gamma \vdash (e_1\ e_2) : \mathsf{T}_2}\ \text{T-App}$$

$$\frac{\Gamma \vdash e : \mathsf{T} \to \mathsf{T}}{\Gamma \vdash \mathsf{fix}\, e : \mathsf{T}}\ \text{T-Fix}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{T} \qquad \Gamma \vdash e_3 : \mathsf{T}}{\Gamma \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : \mathsf{T}}\ \text{T-If} \qquad \frac{\Gamma \vdash e : \mathsf{List}[\mathsf{T}]}{\Gamma \vdash \mathsf{isnil}\, e : \mathsf{Int}}\ \text{T-IsNil}$$

$$\frac{x : \mathsf{T} \in \Gamma}{\Gamma \vdash x : \mathsf{T}}\ \text{T-Var} \qquad \frac{\Gamma \vdash e_1 : \mathsf{T}_1 \qquad \Gamma, x : \mathsf{T}_1 \vdash e_2 : \mathsf{T}_2}{\Gamma \vdash \mathsf{let}\, x : \mathsf{T}_1 = e_1\, \mathsf{in}\, e_2 : \mathsf{T}_2}\ \text{T-Let}$$

$$\frac{}{\Gamma \vdash \mathsf{Nil}[\mathsf{T}] : \mathsf{List}[\mathsf{T}]}\ \text{T-Nil} \qquad \frac{\Gamma \vdash e_1 : \mathsf{T} \qquad \Gamma \vdash e_2 : \mathsf{List}[\mathsf{T}]}{\Gamma \vdash e_1 \mathbin{@} e_2 : \mathsf{List}[\mathsf{T}]}\ \text{T-Cons}$$

$$\frac{\Gamma \vdash e : \mathsf{List}[\mathsf{T}]}{\Gamma \vdash\, !\, e : \mathsf{T}}\ \text{T-Head} \qquad \frac{\Gamma \vdash e : \mathsf{List}[\mathsf{T}]}{\Gamma \vdash \#\, e : \mathsf{List}[\mathsf{T}]}\ \text{T-Tail}$$

FIGURE 6. Typing rules in the λ^+ language.

6. Type Inference

In order to turn the typing rules into a type checking algorithm, we had to add type annotations to the variable bindings. In this section, we will explore how to write an algorithm to *infer* the types of variable bindings so that the user will be able to write well-typed λ^+ programs without type annotations.

One popular paradigm for type inference is *constraint-based* type inference. Such an algorithm will first use the typing rules to generate a set of *constraints* in the form of equations relating known types and unknown *type variables*. Then, the algorithm will *solve* the set of constraints to obtain a *substitution* for the type variables, which can then be used to replace all of the type variables with concrete types.

6.1. **A simple example.**

Before we discuss the algorithm to infer types in general, let us first consider how we might *manually* infer the types in the following λ^+ program:

```
(lambda x. !x @ x) (5 @ Nil)
```

Consider: what does this program do? Can you determine the type of this expression just by reading the code? Keep your answers in mind as you read the following discussion.

6.1.1. *Generating constraints.*

If the program is well-typed, the only rule that would allow it to be well-typed is T-App. We'll try to infer the type of the lambda first, and then we'll infer the type of the argument afterward.

In our standard type checking algorithm, we would not be able to determine the type of `lambda x. !x @ x`. The T-Lambda rule requires that we already know what the type of `x` is before we recursively check the function body. Instead, we can instead use an unknown type variable X_0 as a placeholder for the type of `x`, and we'll ensure that everything works out later when we have enough information to figure out the actual type.

Using this placeholder type for `x`, let's now descend into the function body. The only situation where `!x @ x` is well-typed is when the T-Cons rule is used, so we can deduce the following information:

- If the head `!x` is well-typed, then it must be so by the T-Head rule, and `x` must be a list over some unknown type. If we generate another type variable X_1 to represent this unknown type, then we need to ensure that the following equation holds:

  $$\mathsf{X}_0 = \mathsf{List}[\mathsf{X}_1]$$

  Assuming that this equation holds, the type of `!x` must therefore be X_1.
- If the tail `x` is well-typed, then it must be so by the T-Var rule. But then the type of `x` is just our type variable X_0.
- The T-Cons rule syntactically requires the head to be a known type T and the tail to be a known type $\mathsf{List}[\mathsf{T}]$. We instead deduced that the head is of some unknown type X_1 and that the type of the tail is some unknown type X_0, which would not match the rule. In other words, we currently do not have enough information to type check this cons operation. However, if we can guarantee at some later point that

  $$\mathsf{X}_0 = \mathsf{List}[\mathsf{X}_1]$$

  then we will be able to apply the rule at that point. Assuming that the equation holds, the return type of the lambda body is $\mathsf{List}[\mathsf{X}_1]$ (or X_0).

Finally, we can deduce from T-Lambda that the type of the lambda itself should be $\mathsf{X}_0 \to \mathsf{List}[\mathsf{X}_1]$.

Let's determine the type of the argument expression `5 @ Nil`. The only typing rule that would apply is T-Cons. Clearly 5 is of type Int, but the type of Nil is not so obvious. By the T-Nil rule, Nil is going to be a list type of some unknown type. Again, we'll generate a placeholder type X_2 to represent the type of the elements, so that the type of Nil is $\mathsf{List}[\mathsf{X}_2]$. In the T-Cons rule again, we must then ensure that the type of the head is equal to the type of the elements of the tail:

$$\mathsf{Int} = \mathsf{X}_2$$

If we assume that this equation holds, then the type of `5 @ Nil` is $\mathsf{List}[\mathsf{X}_2]$.

Lastly, we can now check the conditions of the T-App rule. The left expression is required to be a function type; indeed, its type is $\mathsf{X}_0 \to \mathsf{List}[\mathsf{X}_1]$. The right expression is then required to be of type X_0, but we have deduced that the right expression is of type $\mathsf{List}[\mathsf{X}_2]$. We can impose the following restriction to ensure that they are the same:

$$\mathsf{X}_0 = \mathsf{List}[\mathsf{X}_2]$$

If we assume that this equation holds, then the T-App rule tells us that the type of the whole application is simply the result type of the function, or $\mathsf{List}[\mathsf{X}_1]$.

Collecting all of the equations that we assumed would hold, we end up with the system of equations

$$\begin{aligned}\mathsf{X}_0 &= \mathsf{List}[\mathsf{X}_1]\\ \mathsf{Int} &= \mathsf{X}_2\\ \mathsf{X}_0 &= \mathsf{List}[\mathsf{X}_2]\end{aligned}$$

Note: the example shown here is slightly simplified for illustrative purposes. In the general procedure, we would also need to generate additional type variables and equations for the T-App rule, as we would not know what the argument and result types of the function would be.

6.1.2. *Solving constraints.* The above equations constitute a *set of constraints* because they constrain the set of possible types that we can assign to the type variables. If we can solve these constraints, then we will be able to find a *substitution* for our placeholders that would allow us to show that the program is well-typed according to the typing rules. Conversely, if we cannot solve these constraints, then our program is ill-typed because at least one of the assumptions that we made above would lead to a contradiction (i.e. our type system cannot prove the safety of the program).

As humans, we can immediately find a solution to the constraints above:

$$\begin{aligned}\mathsf{X}_0 &= \mathsf{List}[\mathsf{Int}]\\ \mathsf{X}_1 &= \mathsf{Int}\\ \mathsf{X}_2 &= \mathsf{Int}\end{aligned}$$

If we were to annotate our program with these placeholder variables and then apply this *substitution*, we would end up with a program that we can typecheck successfully in our traditional type checking algorithm.

However, a computer is not (yet) capable of the same reasoning power as a human, so we must devise a methodical way to solve these constraints. For now, let's consider how we would solve this manually; later we will show a more generalized method to solve these types of constraints.

Consider again our constraints

$$\begin{aligned} \mathsf{X}_0 &= \mathsf{List}[\mathsf{X}_1] \\ \mathsf{Int} &= \mathsf{X}_2 \\ \mathsf{X}_0 &= \mathsf{List}[\mathsf{X}_2] \end{aligned}$$

We would like to find a substitution σ for X_0, X_1, X_2 such that no placeholder variables appear in their substituted type. Initially, σ will be empty because we have not done anything yet. We can solve the constraints by eliminating variables until all the variables are on the left hand side, as we would in high school algebra.

We will process our constraints from top to bottom. The first constraint already has a type variable on its left hand side, so we can directly add it to our substitution: $\sigma = [\mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1]]$. Next, we need to perform the substitution in the remaining constraints. This gives us

$$\begin{aligned} \sigma &= [\mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1]] \\ \mathsf{Int} &= \mathsf{X}_2 \\ \mathsf{List}[\mathsf{X}_1] &= \mathsf{List}[\mathsf{X}_2] \end{aligned}$$

The next constraint gives us the substitution $\mathsf{X}_2 = \mathsf{Int}$, so now we have

$$\begin{aligned} \sigma &= [\mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1], \mathsf{X}_2 \mapsto \mathsf{Int}] \\ \mathsf{List}[\mathsf{X}_1] &= \mathsf{List}[\mathsf{Int}] \end{aligned}$$

Our final constraint is $\mathsf{List}[\mathsf{X}_1] = \mathsf{List}[\mathsf{Int}]$. Since types are simply a tree data structure, these two types are equal if and only if we have $\mathsf{X}_1 = \mathsf{Int}$. So we generate a *new* constraint:

$$\begin{aligned} \sigma &= [\mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1], \mathsf{X}_2 \mapsto \mathsf{Int}] \\ \mathsf{X}_1 &= \mathsf{Int} \end{aligned}$$

which we can immediately add to our substitution:

$$\sigma = [\mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1], \mathsf{X}_2 \mapsto \mathsf{Int}, \mathsf{X}_1 \mapsto \mathsf{Int}]$$

Applying our entire substitution to each right-hand side type inside the same substitution then yields our desired solution.

In the subsequent sections, we will consider a more generalized form of this inference procedure that can handle arbitrary λ^+ programs.

6.2. **Updated Type Syntax.** Because we need to account for type variables, we must extend our syntax of types as shown in Figure 7.

$$\begin{array}{rcl} \mathsf{T} & ::= & \mathsf{T}_1 \to \mathsf{T}_2 \\ & | & \mathsf{Int} \\ & | & \mathsf{List}[\mathsf{T}] \\ & | & \mathsf{X} \end{array}$$

FIGURE 7. Types in constraint-based typing.

We can formally define a constraint as follows:

Definition 6.1. An *equality constraint* is a pair of types $\mathsf{T}_1, \mathsf{T}_2$, written as $\mathsf{T}_1 = \mathsf{T}_2$.

For example, the following are constraints:

$$\begin{aligned} \mathsf{X} &= \mathsf{List}[\mathsf{Int}] \\ \mathsf{X}_1 &= \mathsf{X}_2 \\ \mathsf{Int} &= \mathsf{Int} \to \mathsf{List}[\mathsf{Int}] \\ \mathsf{List}[\mathsf{X}_1] &= \mathsf{List}[\mathsf{Int}] \end{aligned}$$

6.3. **Constraint Typing.** The constraint generation process is specified using the *constraint typing rules* in Figure 8, which are modified versions of our regular typing rules. The reader should go through the following textual description first before examining the rules in depth.

While the constraint typing rules are similar to the regular typing rules, note that there are two key differences. First, note that expressions may be typed with concrete types or type variables or combinations of each. Thus, we must formulate our type checking as additional constraints on these "arbitrary" types, such as in the CT-Arith rule.

Second, these rules are not entirely formal; they implicitly assume that there is some state tracking the set of constraints as well as the set of generated type variables. For example, the $\mathsf{X}_1 = \mathsf{Int}$ and $\mathsf{X}_2 = \mathsf{Int}$ in the CT-Arith rule will implicitly "add" the two constraints to the output of our constraint generation procedure. The "X fresh" means that we will generate a new type variable X that has not been used yet. We could change the form of the typing relation to include the state, but it would complicate the presentation.

$$\frac{}{\Gamma \vdash i : \mathsf{Int}}\ \text{CT-Int} \qquad \frac{\Gamma \vdash e_1 : \mathsf{T}_1 \qquad \Gamma \vdash e_2 : \mathsf{T}_2 \qquad \square \in \{+, -, *\} \qquad \mathsf{T}_1 = \mathsf{Int} \qquad \mathsf{T}_2 = \mathsf{Int}}{\Gamma \vdash e_1 \square e_2 : \mathsf{Int}}\ \text{CT-Arith}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{T}_1 \qquad \Gamma \vdash e_2 : \mathsf{T}_2 \qquad \square \in \{\&\&, ||\} \qquad \mathsf{T}_1 = \mathsf{Int} \qquad \mathsf{T}_2 = \mathsf{Int}}{\Gamma \vdash e_1 \square e_2 : \mathsf{Int}}\ \text{CT-Logic}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{T}_1 \qquad \Gamma \vdash e_2 : \mathsf{T}_2 \qquad \square \in \{<, >\} \qquad \mathsf{T}_1 = \mathsf{Int} \qquad \mathsf{T}_2 = \mathsf{Int}}{\Gamma \vdash e_1 \square e_2 : \mathsf{Int}}\ \text{CT-Ineq} \qquad \frac{\Gamma \vdash e_1 : \mathsf{T}_1 \qquad \Gamma \vdash e_2 : \mathsf{T}_2 \qquad \mathsf{T}_1 = \mathsf{T}_2}{\Gamma \vdash e_1 = e_2 : \mathsf{Int}}\ \text{CT-Eq}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{T}_1 \qquad \Gamma \vdash e_2 : \mathsf{T}_2 \qquad \Gamma \vdash e_3 : \mathsf{T}_3 \qquad \mathsf{T}_1 = \mathsf{Int} \qquad \mathsf{T}_2 = \mathsf{T}_3}{\Gamma \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : \mathsf{T}_2}\ \text{CT-If}$$

$$\frac{\mathsf{X}\ \mathsf{fresh} \qquad \Gamma, x : \mathsf{X} \vdash e : \mathsf{T}}{\Gamma \vdash \mathsf{lambda}\, x.\ e : \mathsf{X} \to \mathsf{T}}\ \text{CT-Lambda}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{T}_1 \qquad \Gamma \vdash e_2 : \mathsf{T}_2 \qquad \mathsf{X}_1, \mathsf{X}_2\ \mathsf{fresh} \qquad \mathsf{T}_1 = \mathsf{X}_1 \to \mathsf{X}_2 \qquad \mathsf{T}_2 = \mathsf{X}_1}{\Gamma \vdash (e_1\ e_2) : \mathsf{X}_2}\ \text{CT-App}$$

$$\frac{\mathsf{X}\ \mathsf{fresh} \qquad \Gamma \vdash e : \mathsf{T} \qquad \mathsf{T} = \mathsf{X} \to \mathsf{X}}{\Gamma \vdash \mathsf{fix}\, e : \mathsf{X}}\ \text{CT-Fix}$$

$$\frac{x : \mathsf{T} \in \Gamma}{\Gamma \vdash x : \mathsf{T}}\ \text{CT-Var} \qquad \frac{\mathsf{X}\ \mathsf{fresh} \qquad \Gamma \vdash e_1 : \mathsf{T}_1 \qquad \Gamma, x : \mathsf{X} \vdash e_2 : \mathsf{T}_2 \qquad \mathsf{X} = \mathsf{T}_1}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \mathsf{T}_2}\ \text{CT-Let}$$

$$\frac{\mathsf{X}\ \mathsf{fresh}}{\Gamma \vdash \mathsf{Nil} : \mathsf{List}[\mathsf{X}]}\ \text{CT-Nil} \qquad \frac{\Gamma \vdash e_1 : \mathsf{T}_1 \qquad \Gamma \vdash e_2 : \mathsf{T}_2 \qquad \mathsf{T}_2 = \mathsf{List}[\mathsf{T}_1]}{\Gamma \vdash e_1 \mathbin{@} e_2 : \mathsf{List}[\mathsf{T}_1]}\ \text{CT-Cons}$$

$$\frac{\Gamma \vdash e : \mathsf{T} \qquad \mathsf{X}\ \mathsf{fresh} \qquad \mathsf{T} = \mathsf{List}[\mathsf{X}]}{\Gamma \vdash \mathsf{isnil}\, e : \mathsf{Int}}\ \text{CT-IsNil}$$

$$\frac{\Gamma \vdash e : \mathsf{T} \qquad \mathsf{X}\ \mathsf{fresh} \qquad \mathsf{T} = \mathsf{List}[\mathsf{X}]}{\Gamma \vdash\, !\, e : \mathsf{X}}\ \text{CT-Head} \qquad \frac{\Gamma \vdash e : \mathsf{T} \qquad \mathsf{X}\ \mathsf{fresh} \qquad \mathsf{T} = \mathsf{List}[\mathsf{X}]}{\Gamma \vdash \#\, e : \mathsf{List}[\mathsf{X}]}\ \text{CT-Tail}$$

FIGURE 8. Constraint typing rules in the λ^+ language.

Algorithm 1 Unification Algorithm

```
procedure UNIFY(C)                                  ▷ Returns a substitution that solves the constraints
    if C = ∅ then
        []                                          ▷ return the empty substitution
    else
        let {T1 = T2} ∪ C' = C                      ▷ split C into T1 = T2 and C'
        if T1 = T2 then
            unify(C')
        else if T1 = X ∧ X ∉ FV(T2) then
            unify(C'[X ↦ T2]) ∘[X ↦ T2]
        else if T2 = X ∧ X ∉ FV(T1) then
            unify(C'[X ↦ T1]) ∘[X ↦ T1]
        else if (T1 = T11 → T12) ∧ (T2 = T21 → T22) then
            unify(C' ∪ {T11 = T21, T12 = T22})
        else
            fail
```

6.4. **Unification.** A famous algorithm for solving equality constraints is the *unification algorithm*. Some key ideas behind the algorithm are: 1) constraints relating a type variable to a type can be adapted to become part of the solution; and 2) two types are equal if all of their nodes are equal (recall that types are tree data structures). For example,

$$\mathsf{T}_{11} \to \mathsf{T}_{12} = \mathsf{T}_{21} \to \mathsf{T}_{22} \text{ if and only if } \mathsf{T}_{11} = \mathsf{T}_{21} \text{ and } \mathsf{T}_{12} = \mathsf{T}_{22}$$

Using these facts, we can derive the unification algorithm. Algorithm 1 is a slightly modified version of the unification algorithm presented in Chapter 22 of *Types and Programming Languages* [2]. On your homework assignment, you will need to extend the algorithm to handle list types as well.

Before we begin, first let us formally define what a substitution is.

Definition 6.2. A *substitution* is a mapping σ from type variable to type. We use $[\mathsf{X}_0 \mapsto \mathsf{T}_0, \ldots, \mathsf{X}_n \mapsto \mathsf{T}_n]$ to denote a substitution with X_i bound to T_i for $i = 0 \ldots n$. For example, [] is the empty substitution.

Definition 6.3. A substitution σ may be *applied* to a type T, written $\sigma(\mathsf{T})$, to replace all type variables occurring in T with their bindings in σ (if they exist). For example, $[\mathsf{X}_0 \mapsto \mathsf{Int}](\mathsf{X}_0 \to \mathsf{X}_1) = \mathsf{Int} \to \mathsf{X}_1$.

Definition 6.4. A substitution σ_1 may be *composed* with another substitution σ_2, denoted by $\sigma_1 \circ \sigma_2$, to form a new substitution that contains 1) all bindings $\mathsf{X} \mapsto \mathsf{T}$ in σ_1 such that X is not bound in σ_2 and 2) $\mathsf{X} \mapsto \sigma_1(\mathsf{T})$ if $\mathsf{X} \mapsto \mathsf{T}$ in σ_2. Informally, looking up a binding X in $(\sigma_1 \circ \sigma_2)$ can be thought of as: if X is bound to T in σ_2, then return $\sigma_1(\mathsf{T})$, otherwise look up X in σ_1. For example:

$$\begin{aligned} \sigma_1 &= [\mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1], \mathsf{X}_2 \mapsto \mathsf{Int}] \\ \sigma_2 &= [\mathsf{X}_1 \mapsto \mathsf{X}_2] \\ \sigma_1 \circ \sigma_2 &= [\mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1], \mathsf{X}_1 \mapsto \mathsf{Int}, \mathsf{X}_2 \mapsto \mathsf{Int}] \end{aligned}$$

Note that if the types in the bindings of σ_1 contain no type variables, then the only type variables in $\sigma_1 \circ \sigma_2$ must come from type variables in σ_2 that are not bound in σ_1.

[2] Benjamin C. Pierce. 2002. Types and Programming Languages (1st. ed.). The MIT Press.

Now we are ready to describe the algorithm. If no constraints are provided to UNIFY, then there is a trivial solution: the empty substitution []. Otherwise, we select one constraint $\mathsf{T}_1 = \mathsf{T}_2$ from the given set of constraints C, and handle it as follows:

- If T_1 and T_2 are equal (in the sense of the trees being equal), then we recursively unify the remaining constraints C'. In other words, if T_1 and T_2 are equal, then the constraint $\mathsf{T}_1 = \mathsf{T}_2$ is trivially satisfied and we do not need to consider it.
- Now suppose T_1 is actually a type variable X. We proceed by first substituting all occurrences of X in our remaining constraints with T_2, and then recursively unifying them. The idea is that if this recursive call is successful, then it is indeed the case that $\mathsf{X} = \mathsf{T}_2$ is valid, so we can add $\mathsf{X} \mapsto \mathsf{T}_2$ to our final substitution.

  The side condition $\mathsf{X} \notin FV(\mathsf{T}_2)$ is known as the *occurs check*. This is necessary to prevent the algorithm from generating a solution involving a cyclic substitution like $[\mathsf{X} \mapsto \mathsf{X} \to \mathsf{X}]$, which would be nonsensical since our types are finite trees. Note that attempting to eliminate all type variables with such a cyclic substitution would result in an infinite loop!
- The case when T_2 is a type variable X is similar to the above case.
- If T_1 and T_2 are both functions, then we note that equality on two function types is defined as:

  $$\mathsf{T}_{11} \to \mathsf{T}_{12} = \mathsf{T}_{21} \to \mathsf{T}_{22} \text{ if and only if } \mathsf{T}_{11} = \mathsf{T}_{21} \text{ and } \mathsf{T}_{12} = \mathsf{T}_{22}$$

  Thus, we can "break down" the equality on the functions into the two "smaller" constraints on the RHS of this definition, and then proceed as usual.
- If no case matches, then the constraint is an equality of two unequal types (e.g. $\mathsf{Int} = \mathsf{Int} \to \mathsf{Int}$), or the occurs check has failed, etc. Thus, there is no solution and the algorithm fails.

Note: in an actual implementation, one of several different strategies can be used for ensuring that all type variables are "fully" substituted. Here, we use composition of substitutions (such as in Definition 6.4) to incrementally substitute type variables during unification. Another strategy is to simply add bindings to the substitutions during unification, and then substitute them fully afterwards. A third strategy, which is actually used in high-performance implementations of unification, is to avoid substitution altogether and use a union-find data structure instead.