Last updated: March 4, 2024 at 7:09pm

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 integer and boolean arithmetic, 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 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. **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.

2.3. **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. Named functions are also syntactic sugars, and the λ^+ implementation translates it into a let-binding and an anonymous function:

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

2.4. **Boolean expressions.** λ^+ has two introduction forms for constructing boolean values – true and false – and one elimination form for consuming boolean values: if e_1 then e_2 else e_3. It also provides integer comparison operators $=, >, <$.

For example,

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

will evaluate to 12, while

```
if 1 = 1 then 3 < 4 else true
```

will evaluate to false.

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.

2.5. **Recursive Functions.** By default, named functions are non-recursive. To define recursive functions, you can add keyword `rec` after `fun`. 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 rec f with n =
  if n = 0
  then 1
  else n * (f (n-1))
in f 4
```

Recursive function definitions are also "syntactic sugar" in λ^+. Specifically, the function definition

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

is equivalent to the following let expression:

```
let f = fix f is lambda x. e1
in e2
```

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

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

```
fun rec 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
```

2.6. **Lists.** In addition to integers, the λ^+ language also supports linked lists. A list in λ^+ is a general data structure that can be constructed from:

- the empty list constant Nil, or
- the *cons* constructor $e_1 :: e_2$, 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 $1 :: 2 :: 3 :: \mathsf{Nil}$ is read as $1 :: (2 :: (3 :: \mathsf{Nil}))$.

To use/consume a given list, λ^+ supports pattern-matching on lists:

```
match e1 with
| Nil -> e2
| x::y -> e3
end
```

The meaning of the above expression is:

(1) Evaluate e1 to some valuet v
(2) If v is Nil, evaluate expression e2.
(3) If v is $v_1 :: v_2$, then bind v1 to variable x, v2 to variable y, and then evaluate e3.

Consider the following example of a function on lists:

```
fun rec length with l  =
    match l with
    | Nil -> 0
    | h::t -> (length t) + 1
    end
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 rec add with l, n =
  match l with
  | Nil -> Nil
  | h::t -> (h + n) :: (add t n)
  end
in add (1 :: 2 :: 3 :: Nil) 2
```

This program evaluates to the list $3 :: 4 :: 5 :: \mathsf{Nil}$.

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.**

If you'd like to write down λ^+ programs in a text editor, please refer to the context-free grammar presented in Figure 1.

$$
\begin{array}{lcl}
Expr & ::= & let\ ID\ =\ Expr\ in\ Expr \\
 & | & fun\ ID\ with\ Idlist\ =\ Expr\ in\ Expr \\
 & | & fun\ rec\ ID\ with\ Idlist\ =\ Expr\ in\ Expr \\
 & | & lambda\ Idlist.\ Expr \\
 & | & fix\ ID\ is\ Expr \\
 & | & if\ Expr\ then\ Expr1\ else\ Expr2 \\
 & | & Expr1 \oplus Expr2 \qquad (\oplus \in \{+, -, *, =, >, <\}) \\
 & | & \mathsf{Nil} \\
 & | & Expr1 :: Expr2 \\
 & | & match\ Expr\ with\ |\ Nil \rightarrow Expr\ |\ ID :: ID\ \rightarrow Expr\ end \\
 & | & \mathsf{INT\_CONST} \\
 & | & ID
\end{array}
$$

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 parsed as (2*3)+4, instead of 2*(3+4). Similarly, since ::has higher precedence than anything else, this means the expression x::y+1 should be understood as (x ::y) + 1 rather than (x ::y) + 1.

Observe that precedence declarations are not sufficient to resolve all ambiguities concerning these operators; we also need associativity declarations. For example, precedence declarations

$*$
$+, -$
$<, >, =$
$::$

Figure 2. Operator precedence

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.

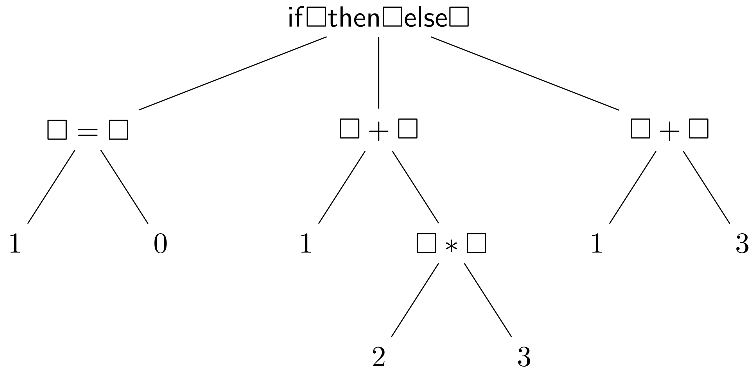
In the λ^+ language, the binary operators +, -, *, =, <, and > are all left-associative; the only right-associative operator is ::. This indicates that the expression 1+2+3 should be parsed as (1+2)+3, while the expression $1 :: 2 :: 3$ should be parsed as $1 :: (2 :: 3)$.

3.2. **Abstract Syntax.** The abstract syntax is a mathematical description of how to inductively construct an abstract syntax tree (AST). The abstract syntax of λ^+ is specified by the context-free grammar presented in Figure 3. Note that syntactic sugars, such as multi-argument lambdas, are no longer present.

Syntactic construct			Description
$\oplus \in \mathit{ArithOp}$	$::=$	$+ \mid - \mid *$	
$\odot \in \mathit{Pred}$	$::=$	$= \mid < \mid >$	
$\diamond \in \mathit{BinOp}$	$::=$	$\oplus \mid \odot$	
$e \in \mathit{Expr}$	$::=$	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
	$\mid$	$\mathsf{fix}\ f\ \ \mathsf{is}\ e$	fixed-point operator
	$\mid$	$e_1\ e_2$	function application
	$\mid$	Nil	empty list
	$\mid$	$e_1 :: e_2$	cons cell
	$\mid$	$\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \to e_2 \mid x :: y \to e_3\ \mathsf{end}$	list pattern-match

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

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.
- Any boolean constant is a value
- Nil is a value.
- If v_1, v_2 are values, then $v_1 :: v_2$ are values.
- No other expression is a value.

For example, the following expressions are all values:

$$\begin{aligned}&10\\&\mathsf{lambda}\,x.\ 1+2\\&\mathsf{true}\\&\mathsf{Nil}\\&10 :: \mathsf{lambda}\,y.\ y\end{aligned}$$

The following expressions are **<u>not</u>** values:

$$\begin{aligned}&1+2\\&(\mathsf{lambda}\,x.\ 1+2)10\\&\mathsf{if}\,\mathsf{Nil}\,\mathsf{then}\,10\,\mathsf{else}\,20\\&(1+2) :: \mathsf{Nil}\end{aligned}$$

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 $\textsc{Int}$ 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:

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

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 $\textsc{Add}$ rule, so then we recursively proceed to evaluate $1 + 2$. But again only the $\textsc{Add}$ rule matches, so we evaluate 1 to 1 by the $\textsc{Int}$ rule (and similarly for 2). Overall, $1 + 2$ evaluates to 3 under $\textsc{Add}$, so then we continue through with the use of the $\textsc{Add}$ 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^+:

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad \oplus \in \{+, -, *\} \qquad (i_3 = i_1 \oplus i_2)}{e_1 \oplus e_2 \Downarrow i_3}\ \textsc{Arith}$$

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 false if the predicate does not hold and to true otherwise, as stated in the following operational semantic rules:

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad (i_1 \odot i_2 \text{ holds})}{e_1 \odot e_2 \Downarrow \mathsf{true}} \ \textsc{PredTrue}$$

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad (i_1 \odot i_2 \text{ does not hold})}{e_1 \odot e_2 \Downarrow \mathsf{false}} \ \textsc{PredFalse}$$

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 true, and another in which it evaluates to false:

$$\frac{e_1 \Downarrow \mathsf{true} \qquad e_2 \Downarrow v}{\mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 \Downarrow v} \ \textsc{IfTrue}$$

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

In the first rule IfTrue, if the expression e_1 evaluates to true, then the if-then-else expression should evaluate e_2 which is the then branch, and yields the value v. If e_1 evaluates to false, the expression e_3 is never evaluated. The second rule IfFalse is similar to first one, except that when e_1 evaluates to false, 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.

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

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:

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

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.** Giving a precise meaning to recursive functions can be quite tricky. In λ^+, recursive function definitions are desugared to use the fixed-point operator fix, so the meaning of recursive functions depends on the meaning of fix. The fix operator behaves like various fixed-point operators in untyped lambda calculus: it evaluates a recursive function by unrolling its definition, while replacing any recursive call with a copy of itself:

$$\frac{e[f \mapsto \mathsf{fix}\, f \ \mathsf{is}\ e] \Downarrow v}{\mathsf{fix}\, f \ \mathsf{is}\ e \Downarrow v}\ \textsc{Fix}$$

That is, to evaluate a fixed-point expression $\mathsf{fix}\, f \ \mathsf{is}\ e$, we substistute any occurrence of f in e – which you can think of as recursive calls – with the $\mathsf{fix}\, f \ \mathsf{is}\ e$ itself. Essentially, we're unrolling the body of the recursive function once. We then evaluate the result of the substitution into a value v.

4.6. **List Operators.** Recall that a list is either the empty list Nil, or it is a *cons cell* $(e_1 :: e_2)$ 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:

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

Now that we have defined the meaning of Nil, we can now state the semantics of the pattern-matching on lists. Since any list value can either be Nil or a (::) constructor, we have two cases:

$$\frac{e_1 \Downarrow \mathsf{Nil} \qquad e_2 \Downarrow v}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3\ \mathsf{end} \Downarrow v}\ \textsc{MatchNil}$$

$$\frac{e_1 \Downarrow v_1 :: v_2 \qquad e_3[x \mapsto v_1][y \mapsto v_2] \Downarrow v_3}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3\ \mathsf{end} \Downarrow v_3}\ \textsc{MatchCons}$$

The $\mathsf{MatchNil}$ rule says that if e_1 evaluates to Nil, then we branch to the Nilcase, and evaluate e_2. The $\mathsf{MatchCons}$ rule says that if e_1 evaluates to a (::) constructor whose head value is v_1 and tail value is v_2, then we branch into the (::)-case of the match, and evaluate e_3. Since inside e_3, we may have referred to the head using name x and the tail using the name y, we substistute x with v_1 and y with v_2 in e_3 before evaluating it.

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.

$$\frac{}{i \Downarrow i}\ \textsc{Int} \qquad \frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad \oplus \in \{+, -, *\} \qquad (i_3 = i_1 \oplus i_2)}{e_1 \oplus e_2 \Downarrow i_3}\ \textsc{Arith}$$

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad (i_1 \odot i_2 \text{ holds})}{e_1 \odot e_2 \Downarrow \mathsf{true}}\ \textsc{PredTrue}$$

$$\frac{e_1 \Downarrow i_1 \qquad e_2 \Downarrow i_2 \qquad (i_1 \odot i_2 \text{ does not hold})}{e_1 \odot e_2 \Downarrow \mathsf{false}}\ \textsc{PredFalse}$$

$$\frac{e_1 \Downarrow \mathsf{true} \qquad e_2 \Downarrow v}{\mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \Downarrow v}\ \textsc{IfTrue} \qquad \frac{e_1 \Downarrow \mathsf{false} \qquad e_3 \Downarrow v}{\mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \Downarrow v}\ \textsc{IfFalse}$$

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

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

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

$$\frac{e[f \mapsto \mathsf{fix}\, f\ \mathsf{is}\ e] \Downarrow v}{\mathsf{fix}\, f\ \mathsf{is}\ e \Downarrow v}\ \textsc{Fix}$$

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

$$\frac{e_1 \Downarrow \mathsf{Nil} \qquad e_2 \Downarrow v}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3\ \mathsf{end} \Downarrow v}\ \textsc{MatchNil}$$

$$\frac{e_1 \Downarrow v_1 :: v_2 \qquad e_3[x \mapsto v_1][y \mapsto v_2] \Downarrow v_3}{\mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \rightarrow e_2 \mid x :: y \rightarrow e_3\ \mathsf{end} \Downarrow v_3}\ \textsc{MatchCons}$$

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.

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

Figure 5. Types of the λ^+ language.

5.2. **Language Extension.** We shall update the syntax of *expressions* in λ^+ with type annotations:

$$\begin{array}{rcll} e & ::= & \ldots & \text{(as before)} \\ & \mid & \mathsf{Nil}[\mathsf{T}] & \text{empty list} \\ & \mid & \mathsf{lambda}\, x : \mathsf{T}.\ e & \text{lambda abstraction} \\ & \mid & \mathsf{fix}\, f : \mathsf{T}\ \mathsf{is}\ e & \text{fixed-point operator} \\ & \mid & (e \mathbin{@} \mathsf{T}) & \text{type annotation} \end{array}$$

That is, we require type annotations on the Nil constructor, function parameters, and the name bound by the fixed-point operator. We also introduce a new syntax "$(e \mathbin{@} \mathsf{T})$" that annotates the expression e with the type T.

Although type annotations are important for type checking, they are transparent to the operational semantics of the language. In other words, the expression $(e \mathbin{@} \mathsf{T})$ behaves exactly like e in terms of evaluation.

$$\frac{e \Downarrow v}{(e \mathbin{@} \mathsf{T}) \Downarrow v}\ \textsc{Annot}$$

The same goes for $\mathsf{Nil}[\mathsf{T}]$, $\mathsf{lambda}\, x : \mathsf{T}.\ e$, and $\mathsf{fix}\, f : \mathsf{T}\ \mathsf{is}\ e$.

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 $x : \mathsf{T}, \Gamma$ 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_1. That is, the type of a variable is looked up in a typing environment from left to right.

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.

5.3. **Explanation of the typing rules.** 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.

5.3.1. *Integers and Booleans.* Integer constants have type Int:

$$\dfrac{}{\Gamma \vdash i : \mathsf{Int}}\ \text{T-Int}$$

and boolean constants have type Bool:

$$\dfrac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}\ \text{T-True} \qquad\qquad \dfrac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}\ \text{T-False}$$

An integer arithmetic operation has type Int if both its operands have type Int:

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

An integer comparison operation has type Bool if both its operands have type Int:

$$\dfrac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int} \qquad \square \in \{<, >, =\}}{\Gamma \vdash e_1 \mathbin{\square} e_2 : \mathsf{Bool}}\ \text{T-Rel}$$

If either operand of an arithmetic/comparison operation are not integers, then the operation will clearly get stuck during evaluation. Thus, the type system preemptively checks for this and prevents such operations from being typed. This is reflected by the fact that for those kinds of expressions, the typing will get stuck, which means that no typing derivation tree can be constructed for the expression.

For example, we can derive the following typing relation:

$$\vdash 1 > 2 * (4 - 6) : \mathsf{Bool}$$

with the following derivation tree:

$$\dfrac{\dfrac{}{\vdash 1 : \mathsf{Int}}\ \text{T-Int} \qquad \dfrac{\dfrac{}{\vdash 2 : \mathsf{Int}}\ \text{T-Int} \qquad \dfrac{\dfrac{}{\vdash 4 : \mathsf{Int}}\ \text{T-Int} \qquad \dfrac{}{\vdash 6 : \mathsf{Int}}\ \text{T-Int}}{\vdash 4 - 6 : \mathsf{Int}}\ \text{T-Arith}}{\vdash 2 * (4 - 6) : \mathsf{Int}}\ \text{T-Arith}}{\vdash 1 > 2 * (4 - 6) : \mathsf{Bool}}\ \text{T-Rel}$$

In contrast, the relation $\vdash 1 > \mathsf{true} : \mathsf{Bool}$ is not derivable:

$$\dfrac{\dfrac{}{\vdash 1 : \mathsf{Int}}\ \text{T-Int} \qquad \dfrac{}{\vdash \mathsf{true} : \mathsf{Int}}\ ???}{\vdash 1 > \mathsf{true} : \mathsf{Bool}}\ \text{T-Rel}$$

because we will get stuck when trying to type true as an integer.

Finally, for if-then-else expressions, the type of the condition must be boolean, and the type of the overall expression is the same as the type of the then-branch and the else-branch:

$$\dfrac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : \mathsf{T}_1 \qquad \Gamma \vdash e_3 : \mathsf{T}_2 \qquad (\mathsf{T}_1 = \mathsf{T}_2)}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : \mathsf{T}_1}\ \text{T-If}$$

Note that we use the relation $\mathsf{T}_1 = \mathsf{T}_2$ to check if two types are equal. We elide the rules for this relation, but they are straightforward. You are encouraged to formalize them as a set of inference rules as an exercise.

5.3.2. *Lists.* The empty list has type $\mathsf{List[T]}$ if it is annotated with the type of its elements:

$$\frac{}{\Gamma \vdash \mathsf{Nil[T]} : \mathsf{List[T]}}\ \textsc{T-Nil}$$

Note that the type annotation associated with Nil indicates the type of the elements of the list, not the type of the list itself. For example, $\mathsf{Nil[Int]}$ has type $\mathsf{List[Int]}$, and $\mathsf{Nil[List[Int]]}$ has type $\mathsf{List[List[Int]]}$!

The cons is typed as follows:

$$\frac{\Gamma \vdash e_1 : \mathsf{T} \qquad \Gamma \vdash e_2 : \mathsf{List[T]}}{\Gamma \vdash e_1 :: e_2 : \mathsf{List[T]}}\ \textsc{T-Cons}$$

That is, the cons cell $e_1 :: e_2$ has type $\mathsf{List[T]}$ if the head e_1 has type T, and that the tail e_2 is another list of type $\mathsf{List[T]}$.

5.3.3. *Typing of Binding Constructs.* Recall that we use a typing environment Γ to keep track of the types of variables. This environment comes into play when we have variable declaration (as in let-expressions, lambda abstractions, pattern matches, and the fixed-point operator), or when we have variable references.

When we have a variable reference, we simply look up the type of the variable in the typing environment:

$$\frac{\Gamma(x) = \mathsf{T}}{\Gamma \vdash x : \mathsf{T}}\ \textsc{T-Var}$$

Constructs that declare variables extend the typing environment with the type of the variable being declared. For example, according to the T-Let rule:

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

a let-expression has type T_2 if the type of the expression bound to x is T_1, and that that the body of the let-expression has type T_2 under the extended typing environment $x : \mathsf{T}_1, \Gamma$.

Similarly, in the T-Lambda rule:

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

a lambda abstraction has type $\mathsf{T}_1 \to \mathsf{T}_2$ if the body of the lambda abstraction returns type T_2 under the extended typing environment $x : \mathsf{T}_1, \Gamma$, where T_1 is an annotation supplied by the programmer.

The rule for typing a list pattern match might look a bit daunting, but it is actually what you would expect:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathsf{List}[\mathsf{T}_1] \qquad \Gamma \vdash e_2 : \mathsf{T}_2 \\ x : \mathsf{T}_1, y : \mathsf{List}[\mathsf{T}_1], \Gamma \vdash e_3 : \mathsf{T}_3 \qquad (\mathsf{T}_2 = \mathsf{T}_3)\end{array}}{\Gamma \vdash \mathsf{match}\ e_1\ \mathsf{with\ Nil} \to e_2 \mid x :: y \to e_3\ \mathsf{end} : \mathsf{T}_2}\ \textsc{T-Match}$$

To unpack this rule a bit, it says that the overall pattern match has type T_2 if

- the thing being matched is some list of type $\mathsf{List}[\mathsf{T}_1]$
- the Nil branch e_2 and the cons branch e_3 have the same type

- the cons branch e_3 needs to be typed under the extended typing environment $x : \mathsf{T}_1, y : \mathsf{List}[\mathsf{T}_1], \Gamma$, since x will be mapped to the head of the list, which has type T_1, and y will be mapped to the tail of the list, which has type $\mathsf{List}[\mathsf{T}_1]$.

In a fixed-point operator:

$$\frac{f : \mathsf{T}, \Gamma \vdash e : \mathsf{T}}{\Gamma \vdash (\mathsf{fix}\, f : \mathsf{T}\ \mathsf{is}\ e) : \mathsf{T}}\ \textsc{T-Fix}$$

the fixed-point operator has type T, if the body of the fixed-point operator has type T under the extended typing environment $f : \mathsf{T}, \Gamma$. In case the fixed-point operator defines a recursive function, then T will be some function type $\mathsf{T}_1 \to \mathsf{T}_2$, although in the general case there is no restriction on whether T should be a function or not.

5.3.4. *Miscellaneous.* The rule for function application $(e_1 e_2)$ is as follows:

$$\frac{\Gamma \vdash e_1 : \mathsf{T}_1 \to \mathsf{T}_2 \qquad \Gamma \vdash e_2 : \mathsf{T}_3 \qquad (\mathsf{T}_1 = \mathsf{T}_3)}{\Gamma \vdash (e_1\ e_2) : \mathsf{T}_2}\ \textsc{T-App}$$

That is, the function application will have result type T_2 if the function e_1 has type $\mathsf{T}_1 \to \mathsf{T}_2$, i.e., it takes an argument of type T_1 and returns a result of type T_2, and if the actual argument e_2 has type T_1.

Finally, we have the rule for type annotations:

$$\frac{\Gamma \vdash e : \mathsf{T}_2 \qquad \mathsf{T}_1 = \mathsf{T}_2}{\Gamma \vdash (e \,@\, \mathsf{T}_1) : \mathsf{T}_1}\ \textsc{T-Annot}$$

That is, the type annotation expression $(e \,@\, \mathsf{T}_1)$ has type T_1 if we can show expression e indeed has the same type.

5.4. **Summary of Typing Rules.** The typing rules for λ^+ are summarized in Figure 6.

$$\frac{\Gamma \vdash e : \mathsf{T}_2 \qquad \mathsf{T}_1 = \mathsf{T}_2}{\Gamma \vdash (e \,@\, \mathsf{T}_1) : \mathsf{T}_1}\;\textsc{T-Annot}$$

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

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

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

$$\frac{\Gamma \vdash e_1 : \mathsf{T}_1 \to \mathsf{T}_2 \qquad \Gamma \vdash e_2 : \mathsf{T}_3 \qquad (\mathsf{T}_1 = \mathsf{T}_3)}{\Gamma \vdash (e_1\ e_2) : \mathsf{T}_2}\;\textsc{T-App}$$

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{Bool}}\;\textsc{T-True} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{Bool}}\;\textsc{T-False}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : \mathsf{T}_1 \qquad \Gamma \vdash e_3 : \mathsf{T}_2 \qquad (\mathsf{T}_1 = \mathsf{T}_2)}{\Gamma \vdash \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 : \mathsf{T}_1}\;\textsc{T-If}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int} \qquad \Box \in \{<, >, =\}}{\Gamma \vdash e_1 \;\Box\; e_2 : \mathsf{Bool}}\;\textsc{T-Rel}$$

$$\frac{f : \mathsf{T}, \Gamma \vdash e : \mathsf{T}}{\Gamma \vdash (\mathsf{fix}\, f : \mathsf{T} \;\mathsf{is}\; e) : \mathsf{T}}\;\textsc{T-Fix}$$

$$\frac{}{\Gamma \vdash \mathsf{Nil}[\mathsf{T}] : \mathsf{List}[\mathsf{T}]}\;\textsc{T-Nil} \qquad \frac{\Gamma \vdash e_1 : \mathsf{T} \qquad \Gamma \vdash e_2 : \mathsf{List}[\mathsf{T}]}{\Gamma \vdash e_1 :: e_2 : \mathsf{List}[\mathsf{T}]}\;\textsc{T-Cons}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathsf{List}[\mathsf{T}_1] \qquad \Gamma \vdash e_2 : \mathsf{T}_2 \\ x : \mathsf{T}_1, y : \mathsf{List}[\mathsf{T}_1], \Gamma \vdash e_3 : \mathsf{T}_3 \qquad (\mathsf{T}_2 = \mathsf{T}_3)\end{array}}{\Gamma \vdash \mathsf{match}\ e_1\ \mathsf{with}\ \mathsf{Nil} \to e_2 \mid x :: y \to e_3\ \mathsf{end} : \mathsf{T}_2}\;\textsc{T-Match}$$

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 intersperse a program with type annotations to guide the type checker. In this section, we will explore how to write an algorithm to *infer* the types of programs while requiring zero type annotations.

One popular paradigm for type inference is *constraint-based* type inference. Such an algorithm will have two phases:

- **Constraint generation** It will first traverse the AST of a program and use the typing rules to collect a set of *constraints* in the form of equations relating known types and unknown *type variables*
- **Constraint solving** Then, it will *solve* the set of constraints to obtain a solution that maps from unknown type variables to 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 > 3
```

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. *Constraint generation.* In our standard type checking algorithm, we would not be able to determine the type of `lambda x. x > 3`. This is because the T-Lambda rule requires that we already know what the type of `x` is before we recursively check the function body. In type checking, the type of `x` will have been given by the programmer using type annotation (for example, using `lambda (x : int). x > 3`).

In contrast, type inference allows zero annotation for `x`. So instead of mapping `x` to the annotated type, we will use an unknown type variable X 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 > 3` is well-typed is when the T-Rel rule is used, so we can deduce the following information. The T-Rel says that the left and right operands must both be of type `int`, so we can generate two equations, one for the left operand, and one of the right operand:

$$\begin{cases} \mathsf{X} = \mathsf{Int} \\ \mathsf{Int} = \mathsf{Int} \end{cases}$$

The above system of equations will be the output of the constraint generation phase of our type checking algorithm.

6.1.2. *Solving constraints.* Once we collected a set of equations, we can ask whether there exists a *solution* to the system of equations. For example, for the above equations, we can ask whether there exists a *concrete* type (e.g., Int, Bool, $\mathsf{List}[\mathsf{Int}]$, and so on) such that substituting the concrete type for the unknown variable X_0 in the above system of equations would make all of the equations true simultaneously.

In this case, the answer is relatively obvious: we can take $\mathsf{X} = \mathsf{Int}$ to be our solution. Note that our solution is a *substitution*: it specifies what X should be replaced with, and the thing it should be replaced with must be a concrete type that does not contain any more unknowns.

Now, once we substituted Int for X, the first equation becomes Int = Int, which is clearly true, while the second equation Int = Int remains true since it doesn't care about what X is. Thus, we have found a solution to the system of equations, and we can conclude that the program is well-typed.

6.1.3. *Analogy with high school math.* Our set up is very similar to when you're solving a system of linear equations in high school math. The first phase, constraint generation, may pose the following set of equations:

$$\begin{cases} 3 = x + 1 \\ y + x = 5 \end{cases}$$

although usually your math teacher is responsible for generating the equations for you.

The second phase, constraint solving (usually the students' responsibility), then proceeds to find two concrete numbers such that substituting them for x and y would make the equations true For example, we can solve the above system of equations by substituting $x = 2$ and $y = 3$ in the equations, which gives us

$$\begin{cases} 3 = 2 + 1 \\ 3 + 2 = 5 \end{cases}$$

To help you arrive at such a solution, your high school math teacher probably taught you some algebra tricks that "massage" the original system of constraints into simpler and simpler forms, until you can simply read off the solution. In the same way, we will develop a system of "algebraic tricks" for manipulating type constraints into a form where the solution is obvious.

As a final note, recall that a system of equations may not be solvable in the first place. For example, the system

$$\begin{cases} 3 = x + 1 \\ 4 = 2x + 1 \end{cases}$$

has no solution at all, since the first equation forces $x = 2$, but the second equation requires $x = 1$. In the same way, a system of type constraints may not be solvable, maybe because the original program is inherently buggy and has lead to an unsolvable system. In that case, we will proclaim that that the program is ill-typed, and reject it just like we rejected programs that violated the type checking rules.

6.2. **A more involved example.** The previous example was meant to illustrate the basic idea behind constraint-based type inference. However, to illustrate the details of both constraint generation and solving, we will consider a more involved example:

```
lambda x. match x with Nil -> 0 | y::z -> y end
```

6.2.1. *Generating constraints.* Again, the above program is missing a type annotation for `x`, so type checking would fail. However, in type inference, we will generate a set of constraints that we will solve to determine the type of `x`, as well as the type of the entire program.

Let's first generate the constraints for this program using our intuition from the type checking rules.

First of all, the program is a lambda expression. If we assign type variable X_0 to the parameter `x`, then the overall lambda expression must have the function type $\mathsf{X}_0 \to ?$, where ? is the return type. Let's not worry about ? – we will figure out what ? should be by recursively inferring the type of the body of the lambda.

The body of the lambda expression is a list pattern-match:

```
match x with Nil -> Nil | y::z -> 0::x end
```

The only way to type check a list pattern-match is to use the T-Match rule.

- The T-Match rule first requires the scrutinee (i.e., the thing being matched) to be a list. Since we're matching `x`, this puts a constraint on what the type of `x` should be:

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

  The problem is, we don't know what the type of the elements of the list is. But this is not a problem in type inference, since we can simply create a new type variable X_1 to represent the type of the elements of the list, and defer the problem of determining what X_1 is to the solving phase. Thus, we generate the following constraint due to `match x with ..`:

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

  The T-Match rule then recursively type checks the branches of the match expression.
- To type check the first branch body, which is `Nil`, we must use the T-Nil rule. However, T-Nil requires a type annotation that we don't have, so we employ the old trick: we generate a new type variable X_2 to represent the type of the elements of the list, and we return $\mathsf{List}[\mathsf{X}_2]$ as the type of the nil branch.
- The second branch `y::z -> 0::x` is more interesting.
  - To type check the body expression `0::x`, T-Match says that the typing environment will be updated to map `y` to the type of the elements of the list, and `z` to the type of the list itself. Since we said `x` was a list of type $\mathsf{List}[\mathsf{X}_1]$, the typing environment must map `y` to X_1 and `z` to $\mathsf{List}[\mathsf{X}_1]$ (in addition to mapping `x` to $\mathsf{List}[\mathsf{X}_0]$).
  - Under this environment, we recursively type check the body expression `0::x`. The T-Cons rule requires the head to be of a known type T, and the tail to be $\mathsf{List}[\mathsf{T}']$, and that $\mathsf{T} = \mathsf{T}'$. In this case, we do know that the type of `0` is $\mathsf{Int} = \mathsf{T}$ using the T-Int rule, and that the type of `x` is $\mathsf{List}[\mathsf{X}_1] = \mathsf{List}[\mathsf{T}']$. Since T-Cons requires $\mathsf{T} = \mathsf{T}'$, we generate the constraint

    $$\mathsf{Int} = \mathsf{X}_1$$

    Finally, the T-Cons rule says the type of the overall list is the same as the type of the tail, so we can return X_0 as the result.
- The last constraint induced by T-Match is that the two branches must have the same type. Since the type of the first branch is $\mathsf{List}[\mathsf{X}_2]$ and the type of the second branch is X_0, we generate the constraint

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

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

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

6.2.2. *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 $\mathsf{X}_0, \mathsf{X}_1$ and X_2 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, slightly reordered (this is ok since the constraints form a set, so order doesn't matter):

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

We would like to find a solution to the above constraints, which will be some 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:

$$\sigma = \left\{\mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1]\right. \qquad \begin{cases}\mathsf{X}_0 = \mathsf{List}[\mathsf{X}_1]\\ \mathsf{List}[\mathsf{X}_2] = \mathsf{X}_0\\ \mathsf{Int} = \mathsf{X}_1\\ \mathsf{X}_2 = \mathsf{Int}\end{cases}$$

Next, we can eliminate the first constraint, now that we have extracted all the information from it:

$$\sigma = \left\{\mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1]\right. \qquad \begin{cases}\cancel{\mathsf{X}_0 = \mathsf{List}[\mathsf{X}_1]}\\ \mathsf{List}[\mathsf{X}_2] = \mathsf{X}_0\\ \mathsf{Int} = \mathsf{X}_1\\ \mathsf{X}_2 = \mathsf{Int}\end{cases}$$

Finally, we need to perform the substitution in the remaining constraints to "eliminate" X_0. This gives us

$$\sigma = \left\{\mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1]\right. \qquad \begin{cases}\cancel{\mathsf{X}_0 = \mathsf{List}[\mathsf{X}_1]}\\ \mathsf{List}[\mathsf{X}_2] = \mathsf{List}[\mathsf{X}_1]\\ \mathsf{Int} = \mathsf{X}_1\\ \mathsf{X}_2 = \mathsf{Int}\end{cases}$$

Now we repeat the same process. The first constraint now says $\mathsf{List}[\mathsf{X}_2] = \mathsf{List}[\mathsf{X}_1]$. Unfortunately, we cannot immediately add it to σ, because it doesn't have the form *Some type variable* X = *Some type* T. However, we can *simplify* this constraint: the only way for two list types to be the same is if they contain the same type of elements. Thus, in the above system of equations, we can replace the first equation by a simpler one:

$$\sigma = \left\{ \mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1] \right. \qquad \left\{ \begin{array}{l} \cancel{\mathsf{X}_0 = \mathsf{List}[\mathsf{X}_1]} \\ \mathsf{X}_2 = \mathsf{X}_1 \\ \mathsf{Int} = \mathsf{X}_1 \\ \mathsf{X}_2 = \mathsf{Int} \end{array} \right.$$

Now that the first equation has the type variable X_2 on the left hand side, we can add it to our substitution:

$$\sigma = \left\{ \begin{array}{l} \mathsf{X}_2 \mapsto \mathsf{X}_1 \\ \mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1] \end{array} \right. \qquad \left\{ \begin{array}{l} \cancel{\mathsf{X}_0 = \mathsf{List}[\mathsf{X}_1]} \\ \mathsf{X}_2 = \mathsf{X}_1 \\ \mathsf{Int} = \mathsf{X}_1 \\ \mathsf{X}_2 = \mathsf{Int} \end{array} \right.$$

All information has now been extracted from the first constraint, so we can remove it, and eliminate X_2 from the remaining constraints:

$$\sigma = \left\{ \begin{array}{l} \mathsf{X}_2 \mapsto \mathsf{X}_1 \\ \mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1] \end{array} \right. \qquad \left\{ \begin{array}{l} \cancel{\mathsf{X}_0 = \mathsf{List}[\mathsf{X}_1]} \\ \cancel{\mathsf{X}_2 = \mathsf{X}_1} \\ \mathsf{Int} = \mathsf{X}_1 \\ \mathsf{X}_1 = \mathsf{Int} \end{array} \right.$$

Then we repeat the same process. The first constraint is $\mathsf{Int} = \mathsf{X}_1$, which is the same as $\mathsf{X}_1 = \mathsf{Int}$, which we add to our substitution, remove this constraint, and eliminate X_1 from the remaining constraints:

$$\sigma = \left\{ \begin{array}{l} \mathsf{X}_1 \mapsto \mathsf{Int} \\ \mathsf{X}_2 \mapsto \mathsf{X}_1 \\ \mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1] \end{array} \right. \qquad \left\{ \begin{array}{l} \cancel{\mathsf{X}_0 = \mathsf{List}[\mathsf{X}_1]} \\ \cancel{\mathsf{X}_2 = \mathsf{X}_1} \\ \cancel{\mathsf{Int} = \mathsf{X}_1} \\ \mathsf{Int} = \mathsf{Int} \end{array} \right.$$

We're not done yet! We still need to process the last constraint, which says $\mathsf{Int} = \mathsf{Int}$. This is always true, so we can simply remove it:

$$\sigma = \left\{ \begin{array}{l} \mathsf{X}_1 \mapsto \mathsf{Int} \\ \mathsf{X}_2 \mapsto \mathsf{X}_1 \\ \mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1] \end{array} \right. \qquad \left\{ \begin{array}{l} \cancel{\mathsf{X}_0 = \mathsf{List}[\mathsf{X}_1]} \\ \cancel{\mathsf{X}_2 = \mathsf{X}_1} \\ \cancel{\mathsf{Int} = \mathsf{X}_1} \\ \cancel{\mathsf{Int} = \mathsf{Int}} \end{array} \right.$$

We have processed all the constraints, so it is tempting to say that we are done, and simply return σ as our solution. However, although σ tells us what type each variable should be replaced with, the right-hand side of each substitution may still contain type variables. So we perform what is called *back substitution* within σ to make sure every line has the form $\mathsf{X} \mapsto \mathsf{T}$, where T does not contain any type variables:

$$\sigma = \left\{ \begin{array}{l} \mathsf{X}_1 \mapsto \mathsf{Int} \\ \mathsf{X}_2 \mapsto \mathsf{Int} \\ \mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{Int}] \end{array} \right.$$

In the subsequent sections, we will consider a more generalized form of this constraint generation+solving procedure that can be used to type check arbitrary λ^+ programs with no type annotations.

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

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

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:

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

6.4. **(Non-Polymorphic) 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 type inference 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.

$$\frac{}{\Gamma \vdash i : \textsf{Int}}\ \textsc{CT-Int} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \textsf{T}_1 \qquad \Gamma \vdash e_2 : \textsf{T}_2 \qquad \Box \in \{+, -, *\} \\ \textsf{T}_1 == \textsf{Int} \qquad \textsf{T}_2 == \textsf{Int}\end{array}}{\Gamma \vdash e_1 \Box e_2 : \textsf{Int}}\ \textsc{CT-Arith}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \textsf{T}_1 \qquad \Gamma \vdash e_2 : \textsf{T}_2 \qquad \Box \in \{<, >, =\} \\ \textsf{T}_1 == \textsf{Int} \qquad \textsf{T}_2 == \textsf{Int}\end{array}}{\Gamma \vdash e_1 \Box e_2 : \textsf{Bool}}\ \textsc{CT-Rel}$$

$$\frac{}{\Gamma \vdash \textsf{true} : \textsf{Bool}}\ \textsc{CT-True} \qquad \frac{}{\Gamma \vdash \textsf{false} : \textsf{Bool}}\ \textsc{CT-False}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \textsf{T}_1 \qquad \Gamma \vdash e_2 : \textsf{T}_2 \qquad \Gamma \vdash e_3 : \textsf{T}_3 \\ \textsf{T}_1 == \textsf{Bool} \qquad \textsf{T}_2 == \textsf{T}_3\end{array}}{\Gamma \vdash \textsf{if}\, e_1\, \textsf{then}\, e_2\, \textsf{else}\, e_3 : \textsf{T}_2}\ \textsc{CT-If}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \textsf{T}_1 \qquad \Gamma \vdash e_2 : \textsf{T}_2 \\ \textsf{X}_1, \textsf{X}_2\ \textsf{fresh} \qquad \textsf{T}_1 == \textsf{X}_1 \to \textsf{X}_2 \qquad \textsf{T}_2 == \textsf{X}_1\end{array}}{\Gamma \vdash (e_1\ e_2) : \textsf{X}_2}\ \textsc{CT-App}$$

$$\frac{\begin{array}{c}\textsf{X fresh} \qquad x : \textsf{X}, \Gamma \vdash e : \textsf{T} \\ \textsf{T} == \textsf{X}\end{array}}{\Gamma \vdash \textsf{fix}\, x\ \textsf{is}\ e : \textsf{X}}\ \textsc{CT-Fix}$$

$$\frac{\Gamma(x) = \textsf{T}}{\Gamma \vdash x : \textsf{T}}\ \textsc{CT-Var} \qquad \frac{\Gamma \vdash e_1 : \textsf{T}_1 \qquad \Gamma, x : \textsf{T}_1 \vdash e_2 : \textsf{T}_2}{\Gamma \vdash \textsf{let}\, x = e_1\, \textsf{in}\, e_2 : \textsf{T}_2}\ \textsc{CT-Let}$$

$$\frac{\textsf{X fresh}}{\Gamma \vdash \textsf{Nil} : \textsf{List[X]}}\ \textsc{CT-Nil} \qquad \frac{\begin{array}{c}\Gamma \vdash e_1 : \textsf{T}_1 \qquad \Gamma \vdash e_2 : \textsf{T}_2 \\ \textsf{T}_2 == \textsf{List}[\textsf{T}_1]\end{array}}{\Gamma \vdash e_1 :: e_2 : \textsf{List}[\textsf{T}_1]}\ \textsc{CT-Cons}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \textsf{T}_1 \qquad \textsf{X fresh} \qquad \textsf{T}_1 == \textsf{List[X]} \\ \Gamma \vdash e_2 : \textsf{T}_2 \qquad x : \textsf{X}, y : \textsf{List[X]}, \Gamma \vdash e_3 : \textsf{T}_3 \qquad \textsf{T}_2 == \textsf{T}_3\end{array}}{\Gamma \vdash \textsf{match}\ e_1\ \textsf{with Nil} \to e_2 \mid x :: y \to e_3\ \textsf{end} : \textsf{T}_2}\ \textsc{CT-Match}$$

$$\frac{\Gamma \vdash e : \textsf{T}_2 \qquad \textsf{T}_1 == \textsf{T}_2}{\Gamma \vdash (e@\textsf{T}_1) : \textsf{T}_1}\ \textsc{CT-Annot}$$

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

Algorithm 1 Unification Algorithm

```
procedure UNIFY($C$)                    ▷ Returns a substitution that solves the constraints
    if $C = \emptyset$ then
        []                                          ▷ return the empty substitution
    else
        let $\{\mathsf{T}_1 = \mathsf{T}_2\} \cup C' = C$     ▷ split $C$ into its first constraint union the remaining $C'$
        if $\mathsf{T}_1 = \mathsf{T}_2$ then
            unify($C'$)
        else if $\mathsf{T}_1 = \mathsf{X} \wedge \mathsf{X} \notin FV(\mathsf{T}_2)$ then
            unify($C'[\mathsf{X} \mapsto \mathsf{T}_2]$) $\circ[\mathsf{X} \mapsto \mathsf{T}_2]$
        else if $\mathsf{T}_2 = \mathsf{X} \wedge \mathsf{X} \notin FV(\mathsf{T}_1)$ then
            unify($C'[\mathsf{X} \mapsto \mathsf{T}_1]$) $\circ[\mathsf{X} \mapsto \mathsf{T}_1]$
        else if $(\mathsf{T}_1 = \mathsf{T}_{11} \to \mathsf{T}_{12}) \wedge (\mathsf{T}_2 = \mathsf{T}_{21} \to \mathsf{T}_{22})$ then
            unify($C' \cup \{\mathsf{T}_{11} = \mathsf{T}_{21}, \mathsf{T}_{12} = \mathsf{T}_{22}\}$)
        else
            fail
```

6.5. **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 and product 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.

6.5.1. *Back Substitution.* The solution σ to the type equations computed by the unification algorithm may still contain type variables on the right-hand side of the substitution. This is acceptable in some cases, where a type variable is genuinely free and should be left as is. However, in all other cases, we should make sure that the types in σ is substituted "as much as possible," according to the following criterion:

Definition 6.5. A substitution σ is *transitively closed* if every type variable X that is in the domain of σ does *not* appear as free type variables in the range of σ.

Informally, if you think of σ as a dictionary, then the domain of σ is the set of all keys (type variables) in the dictionary, and the range as the set of all values (types) in the dictionary. The above criterion says that if a substitution knows what a type variable X should be replaced with, then X should not appear anywhere else in the dictionary.

For example, the following substitution is *not* transitively closed:

$$\sigma = \begin{cases} \mathsf{X}_3 \mapsto (\mathsf{X}_1 \to \mathsf{X}_4) \\ \mathsf{X}_2 \mapsto \mathsf{X}_1 \\ \mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_2] \end{cases}$$

This is because σ knows any occurrence of X_2 can be substituted with X_1, but there is still a left-over X_2 on the right-hand-side of the third line. However, this can be easily fixed by applying the substitution itself to the right-hand side of the third line:

$$\sigma' = \begin{cases} \mathsf{X}_3 \mapsto (\mathsf{X}_1 \to \mathsf{X}_4) \\ \mathsf{X}_2 \mapsto \mathsf{X}_1 \\ \mathsf{X}_0 \mapsto \mathsf{List}[\mathsf{X}_1] \end{cases}$$

where σ' is now transitively closed. This process of turning an arbitrary substitution into a transitively closed one is called *back substitution*.

6.6. **Polymorphism.**

6.6.1. *Motivation.* The current type inference rules is able to infer types for programs with no type annotations, thus alleviating a lot of burden from the programmer. However, the type system is still very restrictive. For example, consider the `tail` function:

```
fun tail with xs =
  match xs with
  | Nil -> Nil
  | _::ys -> ys
  end
in ...
```

which returns the tail of a list. Expressions like `tail (1::2::3::Nil)` and `tail (true::false::Nil)` are well-typed now. So it may seem that `tail` is generic over all types of lists. However, if we write

```
...
let t1 = tail (1::2::3::Nil) in
let t2 = tail (true::false::Nil) in
...
```

Then the unification algorithm will fail! Say we assigned $\mathsf{X} \to \mathsf{X}$ as the type of `tail`.[3]

Then the two applications of `tail` would generate the constraints

$$\begin{gathered} \mathsf{X} == \mathsf{List}[\mathsf{Int}] \\ \mathsf{X} == \mathsf{List}[\mathsf{Bool}] \\ \vdots \end{gathered}$$

since we first called `tail` with a list of integers and then with a list of booleans. The unification algorithm would then fail when trying to unify:

$$\mathsf{List}[\mathsf{Int}] == \mathsf{List}[\mathsf{Bool}]$$

which is clearly unsolvable.

The root cause the problem is that, although `tail` has the seemingly generic type $\mathsf{X} \to \mathsf{X}$, where X is a type variable and hence can be substituted with any type, **any particular solution** σ

[3] This is a slight simplification of the reality, since we will actually assign $\mathsf{X}_1 \to \mathsf{X}_2$ as the type of `tail`, and generate a constraint that says $\mathsf{X}_1 == \mathsf{X}_2$, but the idea is the same.

to the type equations can only assign exactly one type to X, simply by virtue of the fact that σ is a *function* that maps every unknown variable to one concrete type.

For certain kinds of functions, however, this is not what we want; we want to be able to assign different types to X in different contexts, all within the same solution. This notion, that the same function can be used with different types in different contexts, is called *polymorphism* or *generic programming*, is an extremely powerful idea that enables us to write more reusable and expressive code. Almost any modern programming language has some form of polymorphism/generics. For example, in Java, the `List` interface is polymorphic over the type of its elements:

```
interface List<T> {
  T get(int index);
  void add(T element);
  ...
}
```

6.6.2. *Polymorphism and Type Schemes.* To enable polymorphism in our language, we need to augment our types with something called *type schemes*. The idea is that, instead of assigning a single type to a function, we assign a template that can be instantiated with different types in different contexts.

Consider our previous example. When we call the `tail` function with a list of integers, we really want to assign $\mathsf{X}' \mapsto \mathsf{Int}$, and when we call it with a list of booleans, we want to assign $\mathsf{X}'' \mapsto \mathsf{Bool}$, where X' and X'' are conceptually different "versions" of the same type variable X.

In some sense, the type "$\mathsf{X} \to \mathsf{X}$" should really be a template, or a factory, that allows us to replace "X" — which can be thought of as some kind of "template variable" — with actual type variables $\mathsf{X}', \mathsf{X}'', \mathsf{X}_1, \mathsf{X}_{42}$ or types $\mathsf{Int}, \mathsf{Bool}$, etc., and yield infinitely many instantiations of the same template:

$$\begin{gathered}
\mathsf{X}' \to \mathsf{X}' \\
\mathsf{X}'' \to \mathsf{X}'' \\
\mathsf{X}_1 \to \mathsf{X}_1 \\
\mathsf{X}_{42} \to \mathsf{X}_{42} \\
\mathsf{Int} \to \mathsf{Int} \\
\mathsf{Bool} \to \mathsf{Bool} \\
\vdots
\end{gathered}$$

The key is that during constraint solving, each instantiation will be distinct from each other, allowing the unification algorithm to succeed by assigning different types to X', X'', etc.

Hence, we introduce the notion of *type schemes* to distinguish them from our usual, *monomorphic* types (which include type variables, Int, Bool, etc.):

$$\begin{array}{rcll}
\rho & ::= & \forall \overline{\mathsf{X}}.\ \mathsf{T} & \text{type scheme} \\
 & | & \mathsf{T} & \\
\mathsf{T} & ::= & & \text{good ol' monomorphic types} \\
 & | & \mathsf{T}_1 \to \mathsf{T}_2 & \\
 & | & \mathsf{Int} & \\
 & | & \mathsf{Bool} & \\
 & | & \mathsf{List}[\mathsf{T}] & \\
 & | & \mathsf{X} &
\end{array}$$

The new case "$\forall\overline{\mathsf{X}}.\ \mathsf{T}$" represents a type scheme. The notation $\overline{\mathsf{X}}$ denotes a list of *quantified variables* in the type scheme, which are variables in the template that can be replaced with monomorphic types. For example, for the `tail` function, instead of assigning monomorphic type $\mathsf{X} \to \mathsf{X}$ where X can only be replaced with one type in any particular solution, we will assign to it the type scheme

$$\forall \mathsf{X}.\ \mathsf{X} \to \mathsf{X}.$$

Each time we call `tail`, we will instantiate the above template with a fresh type variable. In the previous example where we called `tail` twice, we will produce the instantiations:

$$\begin{aligned} &\mathsf{X}_1 \to \mathsf{X}_1 && \text{first call to } \texttt{tail} \\ &\mathsf{X}_2 \to \mathsf{X}_2 && \text{second call to } \texttt{tail} \end{aligned}$$

and we can constrain

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

which allows the unification algorithm to succeed. Compare this to the previous situation:

$$\begin{aligned} \mathsf{X} &== \mathsf{List[Int]} \\ \mathsf{X} &== \mathsf{List[Bool]} \\ &\vdots \end{aligned}$$

where we were forced to assign distinct types to X due to the two calls to `tail`, which caused the unification algorithm to fail.

6.6.3. *Let-Polymorphism.* Several questions arise when we introduce type schemes:

(1) **Generalization** When and how should we "lift" a usual, monomorphic type into a type scheme?
(2) **Instantiation** Given a type scheme, when and how should we instantiate the template to produce a monomorphic type?

There are multiple ways to answer these questions, and the answers will lead to languages with different type systems with varying degrees of expressiveness and complexity. We will present the most practical and widely-used approach, which is known as *let-polymorphism.* This idea was first introduced in the ML programming language, and is now used in many other languages, including OCaml and many other functional languages.

Let-polymorphism answers the first question by performing generalization when a function is defined for the first time, i.e., at let-bindings. It answers the question by always instantiating type schemes whenever a function is used, i.e., when we use a variable to refer to a previously defined function. These give rise to the following updated rules for let-bindings and variable references:

$$\frac{\Gamma(x) = \rho \qquad \mathsf{T} = \mathsf{instantiate}(\rho)}{\Gamma \vdash x : \mathsf{T}}\ \text{CT-V}\textsc{ar}^*$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathsf{T}_1 \\ \sigma = \mathsf{solve}(C) \qquad \mathsf{T}_1' = \sigma(\mathsf{T}_1) \qquad \Gamma' = \sigma(\Gamma) \qquad \rho = \mathsf{generalize}(\mathsf{T}_1', \Gamma') \\ x : \rho, \Gamma' \vdash e_2 : \mathsf{T}_2\end{array}}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \mathsf{T}_2}\ \text{CT-L}\textsc{et}^*$$

The CT-Var* rule is almost the same as the original CT-Var rule, except that once we queried the type of a variable from the typing environment, we call a new function instantiate to instantiate the type scheme to a monomorphic type. The instantiate function is defined as follows:

```
procedure INSTANTIATE($\rho$)                                  ▷ Instantiates a type scheme
    if $\rho = \forall \mathsf{X}_1, \mathsf{X}_2, \cdots .\ \mathsf{T}$ then        ▷ $\rho$ is a type scheme with quantified variables $\mathsf{X}_1, \mathsf{X}_2, \cdots$
        $\mathsf{X}_1', \mathsf{X}_2', \cdots \leftarrow \mathsf{fresh}(), \mathsf{fresh}(), \cdots$    ▷ Generate fresh variables for each quantified variable
        return $\mathsf{T}[\mathsf{X}_1 \mapsto \mathsf{X}_1'][\mathsf{X}_2 \mapsto \mathsf{X}_2'][\cdots]$        ▷ Replace each quantified var with the fresh var
    else if $\rho = \mathsf{T}$ then                              ▷ $\rho$ is already a monomorphic type
        return $\mathsf{T}$                                          ▷ Return the type as is
```

The CT-Let* is also similar to the original CT-Let rule. The difference is that once we obtain T_1 by recursively typing e_1, we do not directly add x, T_1 to Γ and immediately infer a type for e_2. Instead, we do the following:

(1) Stop the world, and solve all constraints accumulated so far to obtain a substitution σ.
(2) Apply the substitution σ to T_1 to obtain T_1', and to Γ to obtain Γ'.
(3) Lift T_1' from a monomorphic type into a type scheme by calling the generalize function, which is defined as follows:

```
procedure GENERALIZE($\mathsf{T}, \Gamma$)          ▷ Generalizes a monomorphic type into a type scheme
    $\overline{\mathsf{X}} \leftarrow \mathsf{FV}(\mathsf{T}) \setminus \mathsf{FV}(\Gamma)$               ▷ Find all type variables in $\mathsf{T}$ that are not free in $\Gamma$
    return $\forall \overline{\mathsf{X}}.\ \mathsf{T}$               ▷ Return a type scheme with the free type vars now quantified
```

(4) Use the resulting type scheme ρ by adding x, ρ to the typing environment when inferring a type for e_2.