Inductive definitions and proofs

June 26, 2025

1 Inductively defined sets

We represent inductively defined sets in the form of a set of inference rules. Suppose we want to define the set S, each inference rule takes the following general form

$$\frac{P_0 \qquad P_1 \qquad \dots \qquad P_n}{a \in S}\ R_0$$

Each premise P_i above the horizontal line is a logical formula that either does not involve S at all or a logical formula of the form $b \in S$ where S does not appear in b.

Each inference rule can be read as an implication that describes the elements that inhabit the set S. Rule R_0, for example, can be read as the proposition $(P_0 \wedge P_1 \dots \wedge P_n) \to a \in S$. When the premise is empty, the statement simply states that the conclusion is true. Note that shuffling the order of $P_0 \dots P_n$ doesn't change the definition of the set S since the underlying proposition should remain equivalent.

Variables that appear free in each rule are implicitly quantified. As an example, recall the following rules we have seen in class.

$$\frac{}{0 \in S}\ R_0$$

$$\frac{n \in S}{n+4 \in S}\ R_4$$

$$\frac{n \in S}{n+6 \in S}\ R_6$$

Here, the proposition corresponding to rule R_4 is $\forall n \in S, n+4 \in S$ where the variable n is universally quantified.

An element a is in the set S if and only if there exists a proof of $a \in S$ using the rules. While it's possible to describe the proofs in English, we can write

proofs in the form of *derivation trees*, a formal mathematical object. Here is an example of a derivation tree that shows $10 \in S$.

$$\dfrac{\dfrac{\dfrac{}{0 \in S}\, R_0}{4 \in S}\, R_4}{10 \in S}\, R_6$$

The derivation tree for $10 \in S$ looks more like a sequence than a tree because the rules for the set S only contain at most one premise.

Consider the following inductive definition of the set *beautiful*.

$$\dfrac{}{0 \in \textit{beautiful}}\, B_0$$

$$\dfrac{}{3 \in \textit{beautiful}}\, B_3$$

$$\dfrac{}{5 \in \textit{beautiful}}\, B_5$$

$$\dfrac{n \in \textit{beautiful} \qquad m \in \textit{beautiful}}{m + n \in \textit{beautiful}}\, B_n$$

Here is a derivation tree for $11 \in \textit{beautiful}$.

$$\dfrac{\dfrac{\dfrac{}{3 \in \textit{beautiful}}\, B_3 \quad \dfrac{}{3 \in \textit{beautiful}}\, B_3}{6 \in \textit{beautiful}}\, B_n \quad \dfrac{}{5 \in \textit{beautiful}}\, B_5}{11 \in \textit{beautiful}}\, B_n$$

Since B_n has two premises, invoking B_n to prove $11 \in \textit{beautiful}$ requires us to provide two subproofs/subtrees $6 \in \textit{beautiful}$ and $5 \in \textit{beautiful}$.

Exercise 1. Write a derivation tree for $9 \in \textit{beautiful}$ using only rules B_3 and B_n.

Exercise 2. Write a derivation tree for $9 \in \textit{beautiful}$ that involves at least one usage of rule B_0.

2 Inductive proofs

Given a set S defined by some inference rules, rule induction says that to show that S is a subset of some set R, it suffices to show for each inference rule defining S, the proposition corresponds to the inference rule after replacing S by R holds.

For example, given some set R, to prove that *beautiful* $\in R$, it suffices to show that the propositions correspond to the following rules hold:

$$\frac{}{0 \in R}\ B_0$$

$$\frac{}{3 \in R}\ B_3$$

$$\frac{}{5 \in R}\ B_5$$

$$\frac{n \in R \qquad m \in R}{m + n \in R}\ B_n$$

Thus, to show that $S \subseteq R$, the rule of induction says it is sufficient to prove the following statements.

- $0 \in R$
- $3 \in R$
- $5 \in R$
- $\forall n\, m, n \in R \wedge m \in R \implies m + n \in R$

Exercise 3. Prove by induction that S only contains natural numbers. For each rule, explicitly write down the statement you need to prove and then show why it's true.

Exercise 4. Try proving that S only contains odd numbers, which is a false statement as $3 + 3 = 6 \in S$. Again, for each rule, write down the statement you need to prove. Which rule fails to hold?

Suppose we want to prove the statement $\forall a \in S, P(a)$ where P is a predicate over objects. We can prove the statement through the induction principle by instantiate R with the set $\{a \mid P(a)\}$. Thus, to prove $\forall a \in S, P(a)$, it suffices to show the following statements.

- $P(0)$ is true.
- $P(3)$ is true.
- $P(5)$ is true.
- $\forall n\, m, P(n) \wedge P(m) \implies P(m + n)$ is true.

For example, to prove that all elements in *beautiful* are linear combinations of 3 and 5, we can instantiate P with $P(a) := \exists m\ n \in \mathbb{N}, a = 3m + 5n$. Then by induction, it suffices to show that the following statements hold.

- 0 is a linear combination of 3 and 5.
- 3 is a linear combination of 3 and 5.
- 5 is a linear combination of 3 and 5.
- If n and m are both linear combinations of 3 and 5, then $m+n$ is a linear combination of 3 and 5.

Don't forget that we are not done yet! The above process helps us find what needs to be proven by invoking the induction principle. We still need to check that all the propositions hold.

As you get more familiar with inductive proofs, you should be able to perform the rewriting from *beautiful* to R in your head and directly prove the statement that corresponds to each rule. With more complicated definitions, however, it is sometimes useful to explicitly write down the induction principle.

Of course, so far we are only talking about how to obtain the induction principle, but we never asked *why* this style of reasoning is correct. Justifying the validity of induction is a topic we will cover later in class.

3 Inductively defined data types

In this section, we use the knowledge developed in section 1 to define various data types that will be useful when defining languages. As we will see, inductively defined data types really are just an instance of the inductively defined sets we have already seen, except that we expect them to satisfy some constraints so we can define recursive functions over them.

3.1 Sets of Symbols

In set theory, the notion of strings or symbols is not a primitive concept. However, as we are defining programming languages, it is convenient to assume that there exists a set named *Symbol* that consist of strings from the English and Greek alphabet.

It is important to distinguish between symbols and metavariables that we can quantify over. Therefore, to avoid ambiguity, we will sometimes preceed strings from the set *Symbol* with a single quote $'$. Using this notation, we have $'\lambda, 'foo, 'bar, 'plus, '+ \in Symbol$. Importantly, the symbol $'+$ is really a string and is not to be confused with the mathematical function + over numbers.

Here is an example of why the distinction between symbols and metavariables are important. Consider the following two rules.

$$\frac{n \in \mathbb{N}}{n \in S}\ R_n$$

$$\frac{}{'n \in S}\ R'_n$$

Rule R_n corresponds to the universally quantified statement $\forall n \in \mathbb{N}, n \in S$, which says all natural numbers n are in the set S. Rule R'_n, on the other hand, corresponds to $'n \in S$, the proposition that says S contains the single symbol $'n$.

3.2 Inductively Defined Lists and Recursive Functions

Given a set A, we inductively define the set $\mathbf{List}_A$ with the following two inference rules.

$$\frac{}{'\mathbf{Nil} \in \mathbf{List}_A}\ L_0$$

$$\frac{a \in A \qquad l \in \mathbf{List}_A}{(a, l) \in \mathbf{List}_A}\ L_1$$

Thus, an element in $\mathbf{List}_A$ should take the general form $(a_0, (a_1, \ldots, (a_n, '\mathbf{Nil})))$ where $a_0, \ldots, a_n \in A$. We write $(a_0\ a_1 \ldots\ a_n)$ as a shorthand notation for the same list to avoid nested parentheses. Note that when the list is empty, we use the notation $()$ to denote the symbol $'\mathbf{Nil}$. For example, we write $(1\ 2)$ for the 2-element list of natural numbers instead of $(1, (2, '\mathbf{Nil}))$.

Given a non-empty list $(a, l) \in \mathbf{List}_A$, applying the projection operator π_1 gives us $\pi_1(a, l) = a \in A$, the head of the list, whereas applying the projection operator π_2 gives us $\pi_2(a, l) = l$, the remainder of the list.

However, there are more interesting functions that we'd like to define over lists. For example, an operation that computes the length of a list, or a function that appends two lists. That's where recursive definitions become useful.

The principle of recursion says to define a function f from $\mathbf{List}_A$ to some set T, it suffices to provide two equations about f, one for rule L_0 and one for rule L_1.

$$f('\mathbf{Nil}) = e_0$$
$$f((a, l)) = e_1$$

The expression e_0 must not contain any reference to f. The expression e_1 can refer to its arguments a and l and $f(l)$.

Following the recursion principle, we can define the length function as follows.

$$\mathbf{length}('\mathbf{Nil}) = 0$$

$$\mathbf{length}((a, l)) = 1 + \mathbf{length}(l)$$

Exercise 5. Recall that (1 2 3) is a shorthand for $(1, (2, (3, {}'\mathbf{Nil})))$. Following the equations of the **length** function, show that $\mathbf{length}((1\ 2\ 3)) = 3$.

Exercise 6. Consider the following equations.

$$g({}'\mathbf{Nil}) = 0$$

$$g((a, l)) = 1 + g((a, l))$$

Explain why these two equations *do not* induce a valid recursive definition for a function $g : \mathbf{List}_A \to \mathbb{N}$.

In particular, explain intuitively why applying g to the singleton list (1) fail to produce a natural number.

Exercise 7. The **stutter** function takes a list as input and produces a list with double the size by duplicating each element. For example, given the list (1 2 3) as input, we want **stutter** to produce the list (1 1 2 2 3 3) as its output.

Write a recursive definition of $\mathbf{stutter} : \mathbf{List}_A \to \mathbf{List}_A$ by filling out the right hand side of the following equations.

$$\mathbf{stutter}({}'\mathbf{Nil}) = \ldots$$

$$\mathbf{stutter}((a, l)) = \ldots$$

Exercise 8. By rule induction, show that for all $a \in \mathbf{List}_A$, $\mathbf{length}(\mathbf{stutter}(a)) = 2 \cdot \mathbf{length}(a)$.

In Exercises 1 and 2, we have seen that there can be multiple derivation trees that prove $9 \in$ *beautiful*. The definition of $\mathbf{List}_A$, on the other hand, satisfies the following uniqueness property about derivation trees.

Proposition 1. For all $a \in \mathbf{List}_A$, there exists a unique derivation tree with a conclusion that $a \in \mathbf{List}_A$.

Proof. We proceed by rule induction over $\mathbf{List}_A$, which requires us to show the following two statements.

- There is a unique derivation tree for ${}'\mathbf{Nil} \in \mathbf{List}_A$.
- If $a \in A$ and there is a unique derivation tree for $l \in \mathbf{List}_A$, then there exists a unique derivation tree for $(a, l) \in \mathbf{List}_A$.

□

Exercise 9. Finish off the proof above by showing that both statements hold.

Remark. In most cases, the uniqueness derivation tree property can be quickly checked by looking at the conclusion of each rule. In the case of $\mathbf{List}_A$, ${}'\mathbf{Nil}$ and (a, l) are clearly different objects so given $a \in \mathbf{List}_A$ there will be no ambiguity which rule can fire.

The uniqueness of derivation trees is essential for the recursion principle to work. Consider, for example, the following set of recursive equations for the set *beautiful*.

$$h(0) = 1$$

$$h(3) = 1$$

$$h(5) = 1$$

$$h(m + n) = h(m) + h(n)$$

The equations do not induce a function because a single input can correspond to multiple different outputs: $h(3) = h(0) + h(3) = 1 + 1 = 2$, $h(3) = 1$.

Exercise 10. Consider the following definition of the set *Even*.

$$\frac{}{0 \in \mathit{Even}}\ E_0$$

$$\frac{n \in \mathit{Even}}{n + 2 \in \mathit{Even}}\ E_1$$

- First, show that there can be only one derivation tree for every proof of $a \in \mathit{Even}$.
- Next, write a recursive function p from *Even* to $\mathbb{N}$ that divides the input by 2.
- Finally, use rule induction to show the correctness of p by proving $\forall a \in \mathit{Even}, 2 \cdot p(a) = a$

4 An Arithmetic Language

In this section, we use the concepts we have learned in previous sections to define a simple language that describes arithmetic.

4.1 Syntax

The arithmetic language allows us to write expressions that you can write in a simple calculator:

$$4 + (9 * 7)$$

$$(3 * 3) * 3$$

However, instead of working with strings of characters, we skip the parsing step completely and define the *abstract syntax tree* (AST) of the language as an inductively defined sets. By working with ASTs, we no longer have to worry about operator precedence and parentheses while reasoning about the language. In practice, a parser is needed to turn well-formed strings representing algorithmic expressions into their corresponding ASTs.

The set **Arith** is defined by the following set of rules. For simplicity, we consider multiplication and addition as our only operators.

$$\frac{n \in \mathbb{N}}{n \in \textbf{Arith}}\ A_0$$

$$\frac{a \in \textbf{Arith} \qquad b \in \textbf{Arith}}{('\textbf{Add}\ a\ b) \in \textbf{Arith}}\ A_1$$

$$\frac{a \in \textbf{Arith} \qquad b \in \textbf{Arith}}{('\textbf{Mult}\ a\ b) \in \textbf{Arith}}\ A_2$$

The arithmetic expression is either a number (by rule A_0), or a 3-element list containing a symbol $'\textbf{Add}$ or $'\textbf{Mult}$ and two subexpressions a and b.

For consicion, I'll ignore the $'$ symbol and the parentheses when there's no ambiguity. For example, we write **Add** (**Mult** 1 2) 3 as an arithmetic expression that corresponds to $(1 \cdot 2) + 3$.

Are **Add** (**Mult** 1 2) 3 and 5 equal arithmetic expressions? The answer is that it depends. Without first assigning meaning to our language, we can only revert back to the set-theoretic equality, where the number 5 is distinct from **Add** (**Mult** 1 2) 3, which is really a list of three elements.

However, morally, we'd like to define an equivalence relation that identifies these two terms since they both represent the number 5. Such an equivalence relation can be specified by defining *semantics* for our language.

In the following sections, we will explore two different ways of assigning meaning to our language.

4.2 Denotational Semantics

Denotational semantics consist of a function that maps the syntactic terms into something that we already understand and have an existing notion of equality. In the case of **Arith**, we can assign a denotational semantics by defining a recursive function from **Arith** to $\mathbb{N}$.

Of course, before we start writing the recursive equations, we should check that **Arith** has the property that there is at most one derivation tree for each element in it. This is trivially the case since the number n and the lists **Add** a b and **Mult** a b are clearly distinct from one another. Now we define the function $I : \textbf{Arith} \to \mathbb{N}$ with the following equations.

$$I(n) = n$$

$$I(\textbf{Add}\ a\ b) = I(a) + I(b)$$
$$I(\textbf{Mult}\ a\ b) = I(a) \cdot I(b)$$

From the definition of I, we define the relation $\simeq \subseteq \textbf{Arith} \times \textbf{Arith}$ as follows.

$$a \simeq b := I(a) = I(b)$$

Note that the $=$ in $I(a) = I(b)$ is the set-theoretic equality, which behaves the way we expect over natural numbers.

Exercise 11. Show that **Add** (**Mult** 1 2) $3 \simeq 5$.

Exercise 12. Let **mirror** be a recursive function from **Arith** to **Arith** defined as follows.

$$\begin{aligned}
&\textbf{mirror}(n) = n \\
&\textbf{mirror}(\textbf{Add}\ a\ b) = \textbf{Add}\ (\textbf{mirror}(b))\ (\textbf{mirror}(a)) \\
&\textbf{mirror}(\textbf{Mult}\ a\ b) = \textbf{Mult}\ (\textbf{mirror}(b))\ (\textbf{mirror}(a))
\end{aligned}$$

Given an arithmetic expression representing $4 + (7 \cdot 8)$ as input, what does the **mirror** function return? Try out a few more examples to get a better understanding of what the function does, then prove through induction that **mirror** is idempotent, meaning that $\forall a \in \textbf{Arith}, \textbf{mirror}(\textbf{mirror}(a)) = a$.

Finally, show that **mirror** preserves the meaning of arithmetic expressions, meaning that $\forall a \in \textbf{Arith}, a \simeq \textbf{mirror}(a)$.

Exercise 13. We can extend our arithmetic language with the minus operator by adding the following rule.

$$\frac{a \in \textbf{Arith} \qquad b \in \textbf{Arith}}{\textbf{Minus}\ a\ b \in \textbf{Arith}}\ A_3$$

Extend the interpretation function I with an equation for this new rule. Prove that the **mirror** function Exercise 12 is still idempotent. Next, find an arithmetic expression a such that $\textbf{mirror}(a) \simeq a$ doesn't hold.

4.3 Small-Step Operational Semantics

We define the small-step operational semantics for **Arith** by defining the following inductively defined relation $\textbf{Step} \subseteq \textbf{Arith} \times \textbf{Arith}$.

$$\frac{m \in \mathbb{N} \qquad n \in \mathbb{N}}{(\textbf{Add}\ m\ n, m+n) \in \textbf{Step}}\ S_0$$

$$\frac{m \in \mathbb{N} \qquad n \in \mathbb{N}}{(\textbf{Mult}\ m\ n, m \cdot n) \in \textbf{Step}}\ S_1$$

$$\frac{a \notin \mathbb{N} \qquad (a, a') \in \textbf{Step}}{(\textbf{Add}\ a\ b, \textbf{Add}\ a'\ b) \in \textbf{Step}}\ S_2$$

$$\frac{a \in \mathbb{N} \qquad b \notin \mathbb{N} \qquad (b, b') \in \textbf{Step}}{(\textbf{Add}\ a\ b, \textbf{Add}\ a\ b') \in \textbf{Step}}\ S_3$$

$$\frac{a \notin \mathbb{N} \qquad (a, a') \in \textbf{Step}}{(\textbf{Mult}\ a\ b, \textbf{Mult}\ a'\ b) \in \textbf{Step}}\ S_4$$

$$\frac{a \in \mathbb{N} \qquad b \notin \mathbb{N} \qquad (b, b') \in \textbf{Step}}{(\textbf{Mult}\ a\ b, \textbf{Mult}\ a\ b') \in \textbf{Step}}\ S_5$$

Exercise 14. Show that (**Add** (**Mult** 3 4) (**Add** 2 3), **Add** 12 (**Add** 2 3)) $\in$ *Step* by explicitly writing down the derivation tree.

Next, explain why (**Add** (**Mult** 3 4) (**Add** 2 3), **Add** (**Mult** 3 4) 5) $\in$ *Step* is *not* true by showing that none of the rules can be applied.

In PL papers, a directed arrow such as $\rightsquigarrow$ is used to name the stepping relation. Furthermore, the infix notation $a \rightsquigarrow b$ is used to denote $(a, b) \in \rightsquigarrow$. Using this new notation, we can write the operational semantics in a more readable format. The induction principles should work the same way, but if you ever get confused, you can always rewrite $a \rightsquigarrow b$ into $(a, b) \in \rightsquigarrow$, which can be

useful when you are trying to figure out the induction principles.

$$\frac{m \in \mathbb{N} \qquad n \in \mathbb{N}}{\textbf{Add}\ m\ n \rightsquigarrow m + n}\ S_0$$

$$\frac{m \in \mathbb{N} \qquad n \in \mathbb{N}}{\textbf{Mult}\ m\ n \rightsquigarrow m \cdot n}\ S_1$$

$$\frac{a \notin \mathbb{N} \qquad a \rightsquigarrow a'}{\textbf{Add}\ a\ b \rightsquigarrow \textbf{Add}\ a'\ b}\ S_2$$

$$\frac{a \in \mathbb{N} \qquad b \notin \mathbb{N} \qquad b \rightsquigarrow b'}{\textbf{Add}\ a\ b \rightsquigarrow \textbf{Add}\ a\ b'}\ S_3$$

$$\frac{a \notin \mathbb{N} \qquad a \rightsquigarrow a'}{\textbf{Mult}\ a\ b \rightsquigarrow \textbf{Mult}\ a'\ b}\ S_4$$

$$\frac{a \in \mathbb{N} \qquad b \notin \mathbb{N} \qquad b \rightsquigarrow b'}{\textbf{Mult}\ a\ b \rightsquigarrow \textbf{Mult}\ a\ b'}\ S_5$$

4.4 A more concise representation

In this section, we present a more concise way of presenting the rules. First, we define **Op** to be the finite set of symbols $\{\textbf{Add}, \textbf{Mult}\}$ (the $'$ mark is omitted for concision). Of course, we can also present it inductively, though it's not quite necessary since the set is finite.

$$\frac{}{\textbf{Mult} \in \textbf{Op}} \qquad \frac{}{\textbf{Add} \in \textbf{Op}}$$

The syntax of the arithmetic language is then defined as follows.

$$\frac{n \in \mathbb{N}}{n \in \textbf{Arith}'}\ A'_0$$

$$\frac{op \in \textbf{Op} \qquad a \in \textbf{Arith}' \qquad b \in \textbf{Arith}'}{op\ a\ b \in \textbf{Arith}'}\ A'_1$$

It is not hard to show that **Arith**$'$ defines the exact same set elements as **Arith** through induction (you would need to do it twice for each direction).

Denotational semantics for **Arith**$'$ can be broken in two steps. First, we define the function I_o that maps symbols in the set **op** to their corresponding binary functions over natural numbers.

$$I_o(\textbf{Mult}) = x\ y \mapsto x \cdot y$$
$$I_o(\textbf{Add}) = x\ y \mapsto x + y$$

Then we can define our interpretation function I' with the following recursive equations:

$$I'(n) = n, \text{ where } n \in \mathbb{N}$$
$$I'(op\ a\ b) = f(I'(a), I'(b)), \text{ where } f = I_o(op)$$

Exercise 15. Show that $I'(\textbf{Add}\ (\textbf{Mult}\ 1\ 2)\ 3) = 5$ by carefully unfolding the definitions of I' and I_o.

Remark. The function $I_o : \textbf{Op} \to (\mathbb{N} \times \mathbb{N} \to \mathbb{N})$ takes an element from the set **Op** and returns a binary operator over numbers. Functions that takes another function as input or returns another function as output are referred to as *higher-order functions*. In the definition of $I'(op\ a\ b)$, I introduced the local definition f so the whole concept of functions returning other functions feel less intimidating, but if you are comfortable, you can always inline the definition of f and directly write the recursive case as:

$$I'(op\ a\ b) = (I_o(op))(I'(a), I'(b))$$

The operational semantics can benefit the most from the concise definition as we can group multiple similar cases together.

$$\frac{op \in \textbf{Op} \qquad m \in \mathbb{N} \qquad n \in \mathbb{N}}{op\ m\ n \rightsquigarrow (I_o(op))(m, n)}\ S'_0$$

$$\frac{op \in \textbf{Op} \qquad a \notin \mathbb{N} \qquad a \rightsquigarrow a'}{op\ a\ b \rightsquigarrow op\ a'\ b}\ S'_1$$

$$\frac{op \in \textbf{Op} \qquad a \in \mathbb{N} \qquad b \notin \mathbb{N} \qquad b \rightsquigarrow b'}{op\ a\ b \rightsquigarrow op\ a\ b'}\ S_2',$$

Extending our language with the minus operator is now as simple as extending the set **Op** as $\{\textbf{Mult}, \textbf{Minus}, \textbf{Add}\}$ and adding a new case for $I_o(\textbf{Minus}) = x\ y \mapsto x - y$. The operational semantics and the denotational semantics for $\textbf{Arith}'$ are now capable of handling the minus operation.

4.5 BNF notation

A different way of specifying the syntax of **Arith** is by using the Backus-Naur form (a.k.a the BNF notation). I won't go over what precisely constitutes a BNF grammar specification, but with the tools we have developed so far, we can view BNF notation as a concise way of writing inductively defined data types / languages.

Here is how **Arith** (the concise version in the previous section) can be specified using BNF notation.

$$
\begin{aligned}
& n \in \mathbb{N} \\
& op := \mathbf{Mult} \mid \mathbf{Add} \mid \mathbf{Minus} \\
& a := op\ a\ a \mid n
\end{aligned}
$$

Each clause $t := t_0 \mid t_1 \ldots \mid t_n$ can be read as t is a metavariable ranging over an inductive set whose elements are of the form one of the t_is.

For example, the metavariable op in this case ranges over the finite set containing $\{\mathbf{Mult}, \mathbf{Add}, \mathbf{Minus}\}$.

The metavariable a ranges over the set of arithmetic terms, which is either an operator op followed by two expressions a of the same set, or a natural number.

The choice of metavariable is important because of the implicit information associated with the set that the metavariable ranges over. For example, the definition $a := \mathbf{op}\ b\ a$ does not make sense because we don't have a clause of the form $b := \ldots$.

It is possible, however, to assign multiple metavariable names to the same clause. For example, the clause $a, b := op\ a\ b \mid n$ is equivalent to the clause $a := op\ a\ a \mid n$. In the former, it's understood that both a and b range over the same set of terms that we are defining.

Exercise 16. Write the inductive definition that corresponds to the following BNF grammar for the untyped lambda calculus, which we will cover later in class.

$$
\begin{aligned}
& x \in \mathbf{Symbol} \\
& t := x \mid \mathbf{Lam}\ x\ t \mid \mathbf{App}\ t\ t
\end{aligned}
$$

Explicitly spelling out the names of the operators as **Mult** or **Add** is very tedious. In some textbooks, you might see a BNF grammar for the arithmetic language presented as follows.

$$
\begin{aligned}
& n \in \mathbb{N} \\
& a := a + a \mid a \cdot a \mid n
\end{aligned}
$$

The same notation for usual arithmetic is reused for specifying the grammar of the language we are trying to define. This style of definition is quite common in practice, but can be very confusing.

For example, we can immediately tell that **Mult** 1 2 is a syntax tree of the language we are defining, and it is a distinct tree from **Mult** 2 1. However, if we use the notation from above, how do we know if $1 \cdot 2$ and $2 \cdot 1$ are equal? If we read $1 \cdot 2$ and $2 \cdot 1$ as syntax trees, then they are not the same. However, if we read $1 \cdot 2$ and $2 \cdot 1$ as applications of the multiplication operator on the numbers 1 and 2, then they both are the number 2. This ambiguity issue can be avoided by choosing the syntax carefully to not collide with language from

the metatheory (set theory or Rocq), or simply clarifying whenever ambiguity arises.

As an example of notation usage, the untyped lambda calculus from Exercise 16 can be written as follows.

$$x \in \textbf{Symbol}$$
$$t := x \mid \lambda x.t \mid t\ t$$

Here, the . that appears in $\lambda x.t$ does not have any meaning. It's simply there to make lambda abstractions look prettier. In fact, both . and λ are implicitly assumed to be symbols as there are no clauses of the form $. := \ldots$ or $\lambda := \ldots$.

Back to the example with arithmetic expressions. We can rewrite the BNF grammar more concisely as follows.

$$n \in \mathbb{N}$$
$$\circ := + \mid \cdot$$
$$a := a \circ a \mid n$$

In this definition, $\cdot$ is not just a symbol, but a metavariable that ranges over $-$ and $+$. Thus, the language contains expressions of the form $3 + 4$, $9 \cdot 3$ but *not* $3 \circ 4$.

Suppose instead we chose to define the grammar as follows.

$$n \in \mathbb{N}$$
$$a := a \circ a \mid n$$

Then $\circ$ is implicitly treated as a symbol, and our language only contains expressions of the shape $3 \circ (4 \circ 5)$ but not $3 + (4 + 5)$.

4.6 Transitive reflexive closure

The small-step semantics only allows us to take one step at a time. By taking the transitive and reflexive closure of the stepping relation, we can talk about how an arithmetic expression can step into another expression with 0 or more steps.

We consider more generally a relation $R \subseteq A \times A$ for some arbitrary set A. The transitive reflexive closure of R, which we denote R^*, is defined inductively as follows.

$$\frac{a \in A}{(a, a) \in R^*}\ \textsc{Refl}$$

$$\frac{(a, b) \in R \qquad (b, c) \in R^*}{(a, c) \in R^*}\ \textsc{Step}$$

Exercise 17. Prove that R^* is transitive. That is, for all a b and c, if $(a, b) \in R^*$ and $(b, c) \in R^*$, then $(a, c) \in R^*$.

The proof should proceed by induction over the derivation of $(a, b) \in R^*$. However, one thing that is left ambiguous is whether c is fixed.

Formally, the statement takes the form:

$$\forall a\ b\ c.(a, b) \in R^* \wedge (b, c) \in R^* \to (a, c) \in R^*$$

To invoke the induction principle, the statement must be of the form $\forall a\ b.(a, b) \in R^* \to \ldots$.

Here, one way to proceed is to first assume $c \in A$, and then apply induction hypothesis on the statement $\forall a\ b.(a, b) \in R^* \to (b, c) \in R^* \to (a, c) \in R^*$.

Alternatively, we leave the for all quantified c in our statement, and invoke the induction principle on the statement $\forall a\ b.(a, b) \in R^* \to \forall c.(b, c) \in R^* \to (a, c) \in R^*$

Try applying the induction principle in the two different ways described above. Notice how the second version gives a more flexible induction hypothesis, even though the induction hypotheses for both proofs are sufficient to prove the desired property.

By taking the transitive closure of the reduction relation $\rightsquigarrow$, we can now relate the denotational semantics and operational semantics.

Definition 1. Let $a \in$ **Arith**. We define the binary relation $\Downarrow \subseteq$ **Arith** $\times \mathbb{N}$ such that $a \Downarrow n := a \rightsquigarrow^* n$.

Exercise 18. Prove that if $a \rightsquigarrow b$, then $I(a) = I(b)$. There are multiple ways to prove this property. For this exercise, you are specifically asked to prove by induction over the derivation of $a \rightsquigarrow b$.

Exercise 19. Prove that if $a \rightsquigarrow^* b$, then $I(a) = I(b)$, using the result of Exercise 18 as a lemma. Conclude that if $a \Downarrow n$, then $a \simeq n$. (Recall that given $a, b \in$ **Arith**, $a \simeq b := I(a) = I(b)$)

Exercise 20 (Proof by Inversion)**.** First, show that it is not the case that $n \rightsquigarrow a$ for any $n \in \mathbb{N}$ and $a \in$ **Arith**.

Next, show that given $op\ a\ b \rightsquigarrow c$ for some arbitrary terms a, b, and c, precisely one of the three cases hold.

- We have $a, b \in \mathbb{N}$ and $c = I_o(op)(a, b)$.
- There exists some a_0 such that $a \rightsquigarrow a_0$ and $c = op\ a_0\ b$.
- $a \in \mathbb{N}$ and there exists some b_0 such that $b \rightsquigarrow b_0$ and $c = op\ a\ b_0$.

Exercise 21. Show that the stepping relation is deterministic, which can be written formally as follows.

$$\forall a\ b\ c.a \rightsquigarrow b \wedge a \rightsquigarrow c \implies b = c$$

In other words, the stepping relation is a partial function from **Arith** to **Arith**.

First, make sure you have completed Exercise 20 as it teaches you how to do *proof by inversion*, which is crucial for completing the proof.

Next, prove the statement by inducting on the derivation of a.

Finally, prove the statement by inducting on the derivation of $a \rightsquigarrow b$. Note that this requires you to massage the statement into the following form before applying the induction principle.

$$\forall a\ b. a \rightsquigarrow b \implies (\forall c. a \rightsquigarrow c \implies b = c)$$

In practice, it is usually nicer to induct over the derivation of $a \rightsquigarrow b$, but figuring out the induction hypotheses can be tricky.

Exercise 22. Redo Exercise 18 but this time prove the statement by inducting on the the derivation of $a \in \mathbf{Arith}$. Which proof do you find easier?

5 Untyped Lambda Calculus

5.1 Working with anonymous functions

The untyped lambda calculus provides a language where the user can define anonymous functions and perform function applications. To motivate the untyped lambda calculus, it is useful to first get familiar with the concept of anonymous functions.

Consider the following definition of the function f.

$$f : \mathbb{N} \to \mathbb{N}$$
$$f(x) = x \cdot x + 1$$

The function f takes as its input a natural number x and returns as output the number $x \cdot x + 1$. We can use the anonymous function notation $x \mapsto x \cdot x + 1$ to fully capture the behavior of the function without assigning it any particular name. Using the anonymous function notation, every function definition of the form $f(x) = a$ can be rewritten as $f = x \mapsto a$.

Anonymous functions can be applied as normal functions. For example, $(x \mapsto x \cdot x + 1)(2) = 2 \cdot 2 + 1 = 5$. In general, to apply an anonymous function $x \mapsto a$ to some argument b, we replace every occurrence of x with the argument b in the exact same way we apply a named function.

For clarity, sometimes we can add annotations to our anonymous function to indicate its domain. For example, we can write $x \in \mathbb{N} \mapsto x + 1$ to indicate that the function takes natural numbers as its input.

Exercise 23. Consider the following two anonymous functions.

$$x \in \mathbb{N} \mapsto x + x$$

$$y \in \mathbb{N} \mapsto y + y$$

Explain why these functions are equal.

Exercise 24. Suppose we have some known constant $n \in \mathbb{N}$. We define the following two functions.

$$x \in \mathbb{N} \mapsto n$$

$$n \in \mathbb{N} \mapsto n$$

Explain why these two functions are different.

The anonymous function notation can be used to define higher-order functions. The function $x \mapsto y \mapsto y^x$ takes a number x as input, and returns a function $y \mapsto y^x$ which lifts its input y to the xth power. More concretely, applying $x \mapsto y \mapsto y^x$ to 2 gives us the function $y \mapsto y^2$, the square function.

Exercise 25. Suppose $n \in \mathbb{N}$ is a known constant. Consider the following anonymous function from $\mathbb{N}$ to $\mathbb{N} \to \mathbb{N}$ (i.e. for each number $n \in \mathbb{N}$, it outputs a function of type $\mathbb{N} \to \mathbb{N}$).

$$m \mapsto n \mapsto n^m$$

Explain why this function is equal to the $x \mapsto y \mapsto y^x$ function we have discussed earlier.

Applying $x \mapsto y \mapsto y^x$ to the constant n, we get the result $y \mapsto y^n$, a function that maps its input to the nth power.

Now apply $m \mapsto n \mapsto n^m$ to the constant n, and explain why the result is *not* $n \mapsto n^n$.

Exercise 26. Suppose we are given sets A, B, and C. We define the **compose** function as follows.

$$\begin{aligned}&\textbf{compose} : (A \to B) \times (B \to C) \to (A \to C)\\ &\textbf{compose}(f, g) = x \mapsto f(g(x))\end{aligned}$$

Instantiate A, B, and C with the set $\mathbb{N}$. Show that

$$\textbf{compose}(z \mapsto 2z + 1, z \mapsto 3z) = z \mapsto 6z + 1$$

5.2 Syntax

The syntax for the untyped lambda calculus can be represented as the following BNF grammar.

$$\begin{aligned}&x \in \textbf{Symbol}\\ &t := x \mid \textbf{Lam}\ x\ t \mid \textbf{App}\ t\ t\end{aligned}$$

We use the notation $\lambda x.t$ to represent **Lam x t** and $t_0 \# t_1$ to represent $\textbf{App}\ t_0\ t_1$.

The pure untyped lambda calculus is a language about anonymous functions. The lambda form $\lambda x.t$ corresponds to an anonymous function with an argument x and the application form $t_0 \# t_1$ our usual function applications. As a concrete example, the term $\lambda x.x$ corresponds to an identity function that returns its input unchanged.

The beauty of the untyped lambda calculus is that more advanced data structures such as pairs, lists, and numbers can be encoded in the form of functions, making it a suitable model for computation. However, instead of convincing you that the lambda term $\lambda f.\lambda x.f\#x$ represents the number 1, let's simply consider an extended untyped lambda calculus with the grammar from our **Arith** language. This extended lambda calculus corresponds more closely to how programming languages are implemented in practice, where numbers are treated as their own primitives rather than syntax sugar over functions.

Here is the extended grammar of untyped lambda calculus, which is a fusion of the grammar of **Arith** and the pure untyped lambda calculus.

$$\begin{aligned} op &\in \{\textbf{Mult}, \textbf{Add}\} \\ n &\in \mathbb{N} \\ x &\in \textbf{Symbol} \\ t &::= x \mid \textbf{Lam}\ x\ t \mid \textbf{App}\ t\ t \mid op\ t\ t \mid n \end{aligned}$$

In our extended language, we can define anonymous functions that perform arithmetic operations on numbers. For example, the lambda term $\lambda x.\textbf{Mult}\ x\ x$ corresponds to the anonymous function $x \in \mathbb{N} \mapsto x \cdot x$, which is really just the square function.

5.3 Small-step operational semantics

Unlike the **Arith** section, we skip the denotational semantics for untyped lambda calclus since it is surprisingly tricky to define for languages that support functions.

We write Λ to denote the inductively defined set of lambda terms corresponding to the BNF grammar. Before we can specify the operational semantics, we need to define the substitution function, specified below.

Let $t \in \Lambda$ and $x \in \mathit{Symbol}$, we define $\textbf{subst}_{t,x} : \Lambda \to \Lambda$ recursively over its input in Figure 1.

We use the notation $a\{b/x\}$ as a shorthand for $\textbf{subst}_{b,x}(a)$.

Exercise 27. Show that the following statements hold, assuming that x and y are distinct symbols.

- $(\textbf{Add}\ x\ 3)\{4/x\} = \textbf{Add}\ 4\ 3$
- $(\lambda x.x)\{4/x\} = (\lambda x.x)\{3/x\} = \lambda x.x$
- $(\lambda y.x)\{4/x\} = \lambda x.4$
- $(\lambda x.y)\{y/x\} = \lambda x.x$

The operational semantics is specified as an inductively defined binary relation $\rightsquigarrow$ over lambda terms.

$$\mathbf{subst}_{t,x}(y) = \begin{cases} t & \text{if } x = y \\ y & \text{if } x \neq y \end{cases}$$

$$\mathbf{subst}_{t,x}(\lambda y.t_0) = \begin{cases} \lambda y.t_0 & \text{if } x = y \\ \lambda y.(\mathbf{subst}_{t,x}(t_0)) & \text{if } x \neq y \end{cases}$$

$$\mathbf{subst}_{t,x}(t_0 \# t_1) = \mathbf{subst}_{t,x}(t_0) \# \mathbf{subst}_{t,x}(t_1)$$

$$\mathbf{subst}_{t,x}(n) = n$$

$$\mathbf{subst}_{t,x}(op\ t_0\ t_1) = op\ \mathbf{subst}_{t,x}(t_0)\ \mathbf{subst}_{t,x}(t_1)$$

Figure 1: Definition of the substitution function

$$\frac{}{(\lambda x.a)\#b \rightsquigarrow a\{b/x\}}\ L_0$$

$$\frac{a_0 \rightsquigarrow a_1}{a_0\#b \rightsquigarrow a_1\#b}\ L_1$$

$$\frac{op \in \mathbf{Op} \qquad m, n \in \mathbb{N}}{op\ m\ n \rightsquigarrow (I_o(op))(m,n)}\ L_2$$

$$\frac{op \in \mathbf{Op} \qquad a \rightsquigarrow a'}{op\ a\ b \rightsquigarrow op\ a'\ b}\ L_3$$

$$\frac{op \in \mathbf{Op} \qquad a \in \mathbb{N} \qquad b \notin \mathbb{N} \qquad b \rightsquigarrow b'}{op\ a\ b \rightsquigarrow op\ a\ b'}\ L_4$$

Note that rules L_0 and L_1 are the new rules for handling functions. Rules L_2, L_3, and L_4 are already present in our compact presentation of the arithmetic language.

Rule L_0 says if we are applying an anonymous function $(\lambda x.a)$ to an argument b, then we can make progress by replacing the occurrences of x in the body a with the argument b. The operational semantics precisely capture the substitution operation required when we apply functions.

When the left hand side of the application is not yet of the form $\lambda x.a$, the rule L_1 allows us to step the function a_0 so we can eventually reach a lambda term where we can apply rule L_0. This is analogous to rules L_3 and L_4, which step both arguments of an arithmetic operator to a natural number until we can apply rule L_2 to perform the actual operation.

Exercise 28. Recall that $\rightsquigarrow^*$ is the transitive and reflexive closure of $\rightsquigarrow$, such that $a \rightsquigarrow^* b$ holds precisely when a steps to b with $\rightsquigarrow$ for 0 or more times.

Write the full reduction sequence for

- $(\lambda x.x)\#(\mathbf{Add}\ 3\ 4) \rightsquigarrow^* 7$
- $(\lambda x.\mathbf{Mult}\ x\ x)\#(((\lambda x.\lambda y.x)\#3)\#4) \rightsquigarrow^* 9$

Exercise 29. Explain why the expressions $\mathbf{Add}\ 4\ (\lambda x.x)$ and $4\#1$ fail to take any steps.

Exercise 30. Consider the following recursive function $\mathbf{free} : \Lambda \to \mathcal{P}(Symbol)$, which takes a lambda term and returns a set of symbols.

$$\begin{aligned}
&\mathbf{free}(x) = \{x\} \\
&\mathbf{free}(\lambda y.t_0) = \mathbf{free}(t_0) - \{y\} \\
&\mathbf{free}(t_0\#t_1) = \mathbf{free}(t_0) \cup \mathbf{free}(t_1) \\
&\mathbf{free}(n) = \emptyset \\
&\mathbf{free}(op\ t_0\ t_1) = \mathbf{free}(t_0) \cup \mathbf{free}(t_1)
\end{aligned}$$

In the $\lambda y.t_0$ case, the symbol $-$ is the set difference operation.

We say that a symbol x appears free in a term a if $x \in \mathbf{free}(a)$. Prove by induction that if x does not appear free in a, that for every term b, we have $a\{b/x\} = a$.

5.4 Parallel substitution

In Figure 1, the substitution function takes the form $\mathbf{subst}_{b,x}(a)$, which substitutes the term b for the variable x in a. However, what if we want to substitute two variables at once? One approach is to simply perform the substitution operation twice. For example, the term $\mathbf{subst}_{c,y}(\mathbf{subst}_{b,x}(a))$ is obtained by taking the term a, replacing x by b, then replacing y by c.

However, the issue with chaining two substitution operations is that it matters which operation happens first. In general, it is not the case that $\mathbf{subst}_{c,y}(\mathbf{subst}_{b,x}(a))$ is equal to $\mathbf{subst}_{b,x}(\mathbf{subst}_{c,y}(a))$, where the ordering of substitution is flipped.

Exercise 31. Suppose $x \neq y$, find an instantiation of a, b, and c such that $\mathbf{subst}_{c,y}(\mathbf{subst}_{b,x}(a)) \neq \mathbf{subst}_{b,x}(\mathbf{subst}_{c,y}(a))$. To construct the counterexample, it is useful to think about the case where b contains y as a free variable.

The notion of *parallel substitution*, on the other hand, allows us to describe the operation of substituting multiple variables at once more naturally without thinking about the order in which the variables are substituted as they all happen at once.

There is nothing fundamentally wrong with single substitution. However, in practice, I find parallel substitution much easier to work with, and it is easy

to recover single substitution from parallel substitution, but not the other way around.

Definition 2 (Substitution map)**.** We refer to the set of functions $\mathbf{Symbols} \to \Lambda$ as *substitution maps*. Often, we use the symbol ρ or γ to denote such functions.

Definition 3 (Identity substitutions)**.** Recall that $\mathbf{Symbols} \subseteq \Lambda$. We define

$$\mathbf{id} : \mathbf{Symbols} \to \Lambda$$

$$\mathbf{id}(x) = x$$

We refer to **id** as the identity substitution, which maps symbols to variables in the lambda calculus.

Definition 4 (Renamings)**.** A substitution map is a *renaming* if only maps from symbols to variables. For example, the map **id** is a renaming.

There are different operations we can perform on a substitution map ρ. The extension operator allows us to modify a single entry of the substitution map.

Definition 5 (Extension operator)**.** Let ρ be a substitution map. Given $x \in \mathbf{Symbols}$ and $a \in \Lambda$, we define a new substitution map $\rho\{x \mapsto a\}$ as follows.

$$\rho\{x \mapsto a\}(y) = \begin{cases} a & \text{if } x = y \\ \rho(y) & \text{if } x \neq y \end{cases}$$

Let ρ be a substitution map, we define the parallel substitution function $\mathbf{subst}_\rho(a)$ recursively over the syntax of lambda terms in Figure 2.

$$\begin{aligned}
&\mathbf{subst}_\rho(x) = \rho(x) \\
&\mathbf{subst}_\rho(\lambda y.t_0) = \lambda y.\mathbf{subst}_{\rho\{y \mapsto y\}}(t_0) \\
&\mathbf{subst}_\rho(t_0 \# t_1) = \mathbf{subst}_\rho(t_0) \# \mathbf{subst}_\rho(t_1) \\
&\mathbf{subst}_\rho(n) = n \\
&\mathbf{subst}_\rho(op\ t_0\ t_1) = op\ \mathbf{subst}_\rho(t_0)\ \mathbf{subst}_\rho(t_1)
\end{aligned}$$

Figure 2: Parallel substitution

In the λ case, by modifying the substitution map from ρ to $\rho\{y \mapsto y\}$ when making the recursive call, we ensure that the bound variable y doesn't get replaced by the parallel substitution operator.

We use the notation $a\{\rho\}$ as a shorthand for $\mathbf{subst}_\rho(a)$. We can easily substitute the single term b for x by instantiating ρ with $\mathbf{id}\{x \mapsto b\}$, a function that maps x to b, but every other symbol $y \neq x$ to y itself.

Exercise 32. Prove that $a\{\mathbf{id}\} = a$.

Exercise 33. Prove that $a\{\mathbf{id}\{x \mapsto b\}\} = a\{b/x\}$. That is, the single substitution operator $\mathbf{subst}_{b,x}$ is really just a special instance of the parallel substitution operator $\mathbf{subst}_\rho$ with $\rho = \mathbf{id}\{x \mapsto b\}$.

Thus, from now on, we only think of single substitution as an instance of parallel substitution. Instead of defining $a\{b/x\}$ as a shorthand for $\mathbf{subst}_{b,x}(a)$, we can define it as a shorthand for the parallel substitution $a\{\mathbf{id}\{x \mapsto b\}\}$.

The benefit of working with parallel substitution is that we can talk about replacing multiple variables at once without any specific ordering. For example, replacing x by b and y by c can be expressed by the substitution map $\mathbf{id}\{x \mapsto b\}\{y \mapsto c\}$. Of course, we could have written the substitution map as $\mathbf{id}\{y \mapsto c\}\{x \mapsto b\}$, but the following exercise shows that the order doesn't really matter as long as $x \neq y$.

Exercise 34. By unfolding Definition 5, prove that $\rho\{x \mapsto a\}\{y \mapsto b\} = \rho\{y \mapsto b\}\{x \mapsto a\}$ assuming that x and y are distinct symbols.

As a corollary, show that $a\{\rho\{x \mapsto a\}\{y \mapsto b\}\} = a\{\rho\{y \mapsto b\}\{x \mapsto a\}\}$ if $x \neq y$.

Exercise 35. Consider the lambda term $y\#x$ where the symbols $x \neq y$. Simplify and compare the expressions $((y\#x)\{(y\#x)/x\})\{x/y\}$ and $(y\#x)\{\mathbf{id}\{x \mapsto (y\#x)\}\{y \mapsto x\}\}$. The former has a single free vairiable y whereas the latter has both x and y as its free variables.

Parallel substitution will become useful when we talk about type soundness and proof by logical relation after we introduce a type system to our lambda calculus. However, since our variables are still represented as symbols, our naively defined substitution operation still fails to preserve the binding structure of terms, as the following example shows.

Exercise 36 (Capturing of free variables)**.** Consider the lambda term $\lambda y.(x\#y)$ where $y \neq x$. First, show that $(\lambda y.(x\#y))\{y/x\} = \lambda y.(y\#y)$.

Next, show that $(\lambda x.\lambda y.(x\#y))\#y \rightsquigarrow \lambda y.(y\#y)$.

Explain why reducing from $(\lambda x.\lambda y.(x\#y))\#y$ to $\lambda y.(y\#y)$ is wrong, and why it makes more sense to have $(\lambda x.\lambda y.(x\#y))\#y \rightsquigarrow \lambda z.(y\#z)$ for any symbol $z \neq y$. You can use your intuition about function arguments and drawings of the binding structure (i.e. arrows from variables to the location they refer to) as part of your explanation.

To prevent a free variable x from being "captured" by a $\lambda x. \ldots$ after performing substitution, PL researchers refine the substitution function into a capture-avoiding substitution operator such that when performing a substitution over a lambda term $\lambda x.a$, the substitution operator first picks a fresh symbol z and renames the lambda term into $\lambda z.(a\{z/x\})$ before performing the substitution. Thus, if the term we are replacing x with contains the symbol x, it would not be bound by the λx form as we've replaced λx with λz for some z that has never appeared before in the text.

Recall that the functions $x \mapsto x + 1$ and $z \mapsto z + 1$ are the same function. Thus, the systematic renaming of bound variables do not change the meaning of the function only serves the purpose of preserving the binding structure.

However, the capture-avoiding substitution operator is difficult to specify formally. Instead, in next section, we give our encoding of binding structure a final upgrade by introducing de bruijn representation with parallel substitution.

5.5 De Bruijn Representation

The de Bruijn representation requires us to formulate the syntax of the lambda calculus slightly differently.

$$
\begin{aligned}
op \in& \{\mathbf{Mult}, \mathbf{Add}\} \\
n \in& \mathbb{N} \\
x \in& \mathbb{N} \\
t :=& x \mid \mathbf{Lam}\ t \mid \mathbf{App}\ t\ t \mid op\ t\ t \mid \mathbf{Num}\ n
\end{aligned}
$$

The first difference is that the variables x are no longer symbols, but natural numbers. The second difference is that the syntax for function abstractions no longer contains a variable for the function argument; now it simply takes the form $\mathbf{Lam}\ t$, which we can also denote as $\lambda.t$. Since variables are now represented as numbers, we want to be able to distinguish between numbers representing variables and numbers representing numeric constants. We achieve that distinction by explicitly tagging numbers with the $'Num$ symbol, so numeric constants now take the form $\mathbf{Num}\ n$. If we see a number on its own, then it unambiguously represents a variable. For example, $\lambda.0$ is an identity function whereas $\lambda.\mathbf{Num}\ 0$ a constant function that always returns the number 0.

Without variables for functions arguments, how do we know which lambda a variable is supposed to refer to? It turns out that the number contains all the information we need to find corresponding lambda form. For example, given $\lambda.\lambda.x$. We start from the variable x, and traverse toward the root of abstract syntax tree and find the xth (0-indexed) closest λ. For example, $\lambda.\lambda.0$, in nominal representation, corresponds to $\lambda x.\lambda y.y$, whereas $\lambda.\lambda.1$ correspond to $\lambda x.\lambda y.x$.

In de Bruijn representation, the same number might refer to a different function argument depending on its location in the term. Consider the expression $\lambda.(0\#(\lambda.0))$. The first 0 refers to the first λ whereas the second 0 refers to the second λ. In nominal representation, we can write this term as $\lambda x.(x\#(\lambda y.y))$.

Likewise, different numbers might refer to the same function argument. For example, the expression $\lambda.(0\#(\lambda.1))$ corresponds to $\lambda x.(x\#(\lambda y.x))$.

Exercise 37. Consider the following lambda terms in nominal representation.

- $\lambda x.x$
- $\lambda y.y$

- $\lambda x.\lambda y.x$
- $\lambda x.\lambda x.x$
- $\lambda x.\lambda z.x$

Write down their corresponding de Bruijn representation. Notice how some of these expressions have the exact same de Bruijn representation.

Exercise 38. Convert the following terms from de Bruijn representation to their corresponding nominal representation. Is the nonimal representation unique?

- $\lambda.(\lambda.1\#(0\#0))\#(\lambda.1\#(0\#0))$
- $\lambda.\lambda.0$

What about free variables? In the exercises, we have only considered lambda terms that do not contain free variables. However, what should be the de Bruijn representation of $\lambda x.y$ where $y \neq x$? What number should we pick for y? Of course, one thing we know for sure is that the number must be greater than 0 since $\lambda.0$ represents the identity function $\lambda x.x$. However, there is no information about what happens outside the lambda terms.

Nevertheless, we can distinguish between variables that are bound and free. A variable is free if it is too large to be bound to any of the λs and is bound otherwise. Furthermore, we can tell which numbers refer to the same free variable.

Given expressions 0 and 1, we ought to know that 0 and 1 refer to different free variables. We can assume that there exists an infinite list of free variables around. If a variable x is too large, then instead of seeking for more λs, we continue to index into this infinite list. For example, the variable 0 refers to the first position of the list, whereas the variable 1 refers to the second position of the list.

Consider the expressions $\lambda.1$ and 0. The variable 1 and 0 should both refer to the same free variable. Why? Because 1 needs to first leap over the λ that wraps around it, after which we seek the first (i.e. 0th) available free variable.

Applying the idea we have discussed so far about free de Bruijn variables, we can define a mapping between de Bruijn terms and nominal terms as long as we pick a bijection $F : \mathbb{N} \to \textbf{Symbols}$. For example, if we have $F(0) =' x$, $F(1) =' y$, and $F(2) =' z$, then the de Bruijn term $\lambda.2\#0$ would correspond to the nominal term $\lambda x.y\#x$ or $\lambda z.y\#z$, whereas the nominal term $\lambda x.z\#x$ would uniquely correspond to the de Bruijn term $\lambda.3\#0$. Of course, we could have picked the bijection F differently, in which case a de Bruijn term would correspond to a different nominal term.

5.6 Renaming (de Bruijn)

In this section, we define the renaming function over de Bruijn terms.

Definition 6 (Renaming)**.** We say that ξ is a renaming for de Bruijn terms if it is a function of type $\mathbb{N} \to \mathbb{N}$.

Definition 7 (Some useful renamings)**.** We write $\uparrow$ as a shorthand for the renaming that increments its input by 1.

We write **id** to denote the identity renaming that leaves its input unchanged.

Definition 8 (Extension (renaming))**.** Given a renaming ξ and a number x, we define the renaming $\xi : x$ as follows.

$$(\xi : x)(n) = \begin{cases} x & \text{if } n = 0 \\ \xi(n-1) & \text{if } n > 0 \end{cases}$$

Definition 9 (Composition)**.** Given two renamings ξ_0 and ξ_1, we can use the usual function composition operator $\circ$ to obtain a new renaming $\xi_0 \circ \xi_1$ such that $(\xi_0 \circ \xi_1)(x) = \xi_0(\xi_1(x))$.

Definition 10 (Lifting)**.** We define the lifting operator $\Uparrow$ as follows.

$$\Uparrow \xi = (\uparrow \circ \xi) : 0$$

Exercise 39. In class, we define the lifting operator $\Uparrow$ as a function that satisfies the following equations for each ξ.

$$\Uparrow \xi(x) = \begin{cases} 0 & \text{if } x = 0 \\ 1 + \xi(x-1) & \text{if } x > 0 \end{cases}$$

Prove that the lifting operator defined in Definition 10 satisfies these equations.

The recursive definition of the renaming function is given below.

$$\begin{aligned} &\mathbf{ren}_\xi(x) = \xi(x) \\ &\mathbf{ren}_\xi(a \# b) = \mathbf{ren}_\xi(a) \# \mathbf{ren}_\xi(b) \\ &\mathbf{ren}_\xi(\lambda.a) = \lambda.\mathbf{ren}_{\Uparrow\xi}(a) \end{aligned}$$

We use the shorthand $a\langle\xi\rangle$ to denote $\mathbf{ren}_\xi(a)$.

Exercise 40. Prove by induction over a that the following properties hold.

- $a\langle\mathbf{id}\rangle = a$
- $a\langle\xi_0 \circ \xi_1\rangle = a\langle\xi_1\rangle\langle\xi_0\rangle$

At a high-level, given a term a, the term $\mathbf{ren}_\xi(a)$ is obtained by replacing every *free* variable x of a with $\xi(x)$.

However, the concept of free variables is quite tricky in de Bruijn representation. Similar to how two distinct numbers can refer to the same boudn variable (e.g. the 0 and 1 in $\lambda.0\#(\lambda.1)$), two distinct numbers can also refer to the same

free variable (e.g. the 0 and 1 in $0\#(\lambda.1)$). However, since the same free variable can correspond to different numbers, which number should we pick for the free variable? For example, in $0\#(\lambda.1)$, the numbers 0 and 1 both refer to the first free variable as the 1 needs to leap through the λ first. Should the free variable of the term be 0 or 1? It turns out we always take the perspective of being outside the lambda term, and therefore the only free variable of $0\#(\lambda.1)$ is 0.

Let's now formally define the **free** function for de Bruijn terms. We start by defining the following auxiliary function, which will be useful in the λ case.

Definition 11 (Downshifting)**.** Let $A \subseteq \mathbb{N}$, we define $\downarrow A$ as follows.

$$\downarrow A = \{x - 1 \mid x \in A, x > 0\}$$

Note that $\downarrow A$ effectively removes the element 0 from A and decrements all other numbers by 1.

We then define **free** recursively as follows.

$$\begin{aligned}
&\mathbf{free}(x) = \{x\} \\
&\mathbf{free}(a\#b) = \mathbf{free}(a) \cup \mathbf{free}(b) \\
&\mathbf{free}(\lambda.a) = \downarrow \mathbf{free}(a)
\end{aligned}$$

Exercise 41. Prove that $\mathbf{free}(\lambda.\lambda.(0\ 1)) = \emptyset$ and $\mathbf{free}(\lambda.1\ (\lambda.1\ 3)) = \{0, 1\}$ and observe the effect of $\downarrow$ in the λ case. Try convincing yourself that the result makes sense.

Exercise 42. Prove by induction over a that the following equality holds.

$$\mathbf{free}(a\langle\xi\rangle) = \{\xi(x) \mid x \in \mathbf{free}(a)\}$$

Exercise 43 (A little challenging!)**.** We want to show that if a is closed (i.e. $\mathbf{free}(a) = \emptyset$), then $a\langle\xi\rangle = a$ for all ξ.

Try proving the statement by induction over a. If done correctly, you'll realize the induction hypothesis is not usable in the λ case and the proof won't actually go through. (If somehow you managed to prove it, then something must have gone wrong!)

Instead, prove the following strengthened statement, which says if $(\forall x \in \mathbf{free}(a), \xi(x) = x)$, then $a\langle\xi\rangle = a$.

5.7 Substitution (de Bruijn)

Definition 12 (Substitution maps)**.** We say that ρ is a substitution map if it's a function of type $\mathbb{N} \to \Lambda$.

Note that every renaming is a special instance of a substitution map.

Definition 13 (Identity substitution)**.** We use the same symbol **id** to represent an identity substitution that maps each variable to itself.

Definition 14 (Extension)**.** Given a substitution map ρ and a term a, we define the extended substitution map $\rho : a$ as follows.

$$(\rho : a)(x) = \begin{cases} a & \text{if } x = 0 \\ \rho(x-1) & \text{if } x > 0 \end{cases}$$

Definition 15 (Composition (with renaming))**.** Given a renaming ξ, we write $\mathbf{ren}_\xi$ as the function (which has type $\Lambda \to \Lambda$) $a \mapsto a\langle\xi\rangle$. Thus, we can compose a renaming ξ and a substitution map ρ by writing the substitution map $\mathbf{ren}_\xi \circ \rho$, which satisfies the equation $(\mathbf{ren}_\xi \circ \rho)(x) = \rho(x)\langle\xi\rangle$.

Definition 16 (Lifting)**.** Given a substitution ρ, we overload the symbol $\Uparrow$ for renamings and define the substitution $\Uparrow \rho$ as follows.

$$\Uparrow \rho = (\mathbf{ren}_\uparrow \circ \rho) : 0$$

Exercise 44. Show that $\Uparrow \rho(x) = \begin{cases} 0 & \text{if } x = 0 \\ \rho(x-1)\langle\uparrow\rangle & \text{if } x > 0 \end{cases}$

Prove as a corollary that when ρ happens to be a renaming, there is no ambiguity whether $\Uparrow$ refers to Definition 10 or Definition 16 as the results are equal.

We now define the substitution function $\mathbf{subst}_\rho(a)$ recursively over the syntax of a.

$$\begin{aligned} &\mathbf{subst}_\rho(x) = \rho(x) \\ &\mathbf{subst}_\rho(a\#b) = \mathbf{subst}_\rho(a)\#\mathbf{subst}_\rho(b) \\ &\mathbf{subst}_\rho(\lambda.a) = \lambda.\mathbf{subst}_{\Uparrow\rho}(a) \end{aligned}$$

Note that the definition of $\mathbf{subst}_\rho$ closely mirrors the definition of $\mathbf{ren}_\xi$. We use the shorthand $a\{\rho\}$ to denote $\mathbf{subst}_\rho(a)$.

Definition 17 (Single substitution)**.** Similar to the parallel substitution operation for nominal terms, we can recover single substitution by composing the identity substitution and the extension operator.

Let a and b be lambda terms. We write $a\{b\}$ as a shorthand for $a\{\mathbf{id} : b\}$.

We can now finally recover the small-step operational semantics using our newly defined substitution operator.

$$\frac{}{(\lambda.a)\#b \rightsquigarrow a\{b\}}\ D_0$$

$$\frac{a_0 \rightsquigarrow a_1}{a_0\#b \rightsquigarrow a_1\#b}\ D_1$$

Exercise 45. Show that the following statements hold.

- $\forall a, (\lambda.0)\#a \rightsquigarrow a$.
- $\forall a, (\lambda.\lambda.1)\#a \rightsquigarrow \lambda.(a\langle\uparrow\rangle)$.
- $(\lambda.\lambda.1)\#0\#1 \rightsquigarrow 0$
- $(\lambda.\lambda.0)\#0\#1 \rightsquigarrow 1$
- $(\lambda.0\#2)\#0 \rightsquigarrow 0\#1$

For the second statement, explain why it makes sense that the result has a shifted by the $\uparrow$ operator.

For the last statement, explain why the free variable 2 is decremented 1 and why it would be wrong to leave it as 2.

Exercise 46. Consider the nominal term $(\lambda x.\lambda y.x)\#y$. If we reduce naively, we end up with the wrong term $\lambda y.y$ where the free variable y becomes captured.

Suppose we map the nominal free variable y to the free de Bruijn variable 0, confirm that the lambda term can be written as $(\lambda.\lambda.1)\#0$.

Show that $(\lambda.\lambda.1)\#0 \rightsquigarrow \lambda.1$. Explain why the $\mathbf{ren}_\uparrow$ that appears in Definition 16 is crucial for avoiding the capturing of free variables.

Now that we have the definition of the substitution function, we can compose two substitutions as follows.

Definition 18 (Composition)**.** We define $\mathbf{subst}_\rho := a \mapsto a\{\rho\}$. We can then combine two substitution maps ρ_0 and ρ_1 by writing $\mathbf{subst}_{\rho_0} \circ \rho_1$, which is a new substitution that maps each variable i to $\rho_1(i)\{\rho_0\}$.

For the following exercises, you will show that the operational semantics respect substitution.

Exercise 47. Prove that if $a \rightsquigarrow b$, then $a\{\rho\} \rightsquigarrow b\{\rho\}$.

Exercise 48. Prove that if $a \rightsquigarrow b$ and $a = a_0\langle\xi\rangle$ for some a_0 and ξ, then there exists some b_0 such that $b = b_0\langle\xi\rangle$ and $a_0 \rightsquigarrow b_0$.

The proof should proceed by induction over the derivation of $a \rightsquigarrow b$. You need to be extra careful when writing down the induction hypotheses for the cases.

6 The simply typed lambda calculus

Consider the following extended grammar of the lambda calculus with booleans. For readability, we revert back to the nominal representation for now.

$$x \in \mathbf{Symbol}$$
$$t, a, b, c := x \mid \lambda x.t \mid t\ t \mid \mathbf{true} \mid \mathbf{false} \mid \mathbf{if}\ t\ t\ t$$

Our operational semantics need to take into account the new boolean constructs.

$$\frac{}{\textbf{if true } b\ c \rightsquigarrow b} \qquad \frac{}{\textbf{if false } b\ c \rightsquigarrow c} \qquad \frac{a_0 \rightsquigarrow a_1}{\textbf{if } a_0\ b\ c \rightsquigarrow \textbf{if } a_1\ b\ c}$$

In the **if** a b c form, the term a is expected to step to a boolean, whereas b and c correspond to the then and else branches respectively.

Consider the term **if** $(\lambda x.x)$ a b, where a function appears in the position where a boolean value is expected, causing our operational semantics to get stuck. In real world languages, this kind of stuck state correspond to run-time errors. For example, in Python or Scheme, passing a function to an if statement or expression would cause the program to crash.

In this section, we learn how to use a type system to specify the set of well-behaved terms, which, when executed, will never step into one of those stuck states.

Exercise 49. Given another example of a stuck term that involves the application form and boolean values (**true** and **false**).

6.1 Nominal specification of the type system

A type system allows us to associate terms with their *types*, similar to how we associate mathematical objects with the sets they belong to.

Types, similar to terms, are syntactic entities that are defined inductively.

$$\tau, A, B ::= \textbf{bool} \mid \textbf{Arr } \tau\ \tau$$

The symbol **bool** denotes the type for boolean expressions. The **Arr** τ_0 τ_1 denotes the type for functions that map expressions of type τ_0 to expressions of type τ_1. We write $\tau_0 \to \tau_1$ as a shorthand for **Arr** τ_0 τ_1, though it is important to keep in mind that the $\to$ here is just a piece of syntax and should be distinguished from the arrow used for set-theoretic functions. In PL papers, it is more common to simply write the grammar as follows with the understanding that the symbols appearing in the grammar are syntax rather than semantic objects from the metatheory (e.g. set theory or Rocq).

$$\tau ::= \textbf{bool} \mid \tau \to \tau$$

It should be quite obvious that we can associate **true** and **false** to **bool**. However, given a variable x, without further information, we are unable to determine its type.

In mathematics, we are able to determine the set that a variable a belongs to from a previous definition or declaration (e.g. "Suppose $a \in \mathbb{N}$.."). In a type system, we use a typing context to bookkeep the variables that have been declared so far and their associated types.

The typing context is inductively defined as follows.

$$\Gamma ::= \cdot \mid \Gamma, x : A$$

In other words, a typing context is either an empty context $\cdot$, or an existing context Γ extended with a variable x and its associated type A, which we denote as $\Gamma, x : A$. Note that in $\Gamma, x : A$, the , and the : are both notations; the only data required to extend a context is an existing context Γ, a variable x, and a type A.

As a shorthand, we write the context $\cdot, x : A, y : B, \ldots$ as $x : A, y : B, \ldots$, omitting the $\cdot$ at the very beginning.

Under the typing context $x : \tau_0, y : \tau_1$, we can say that x has type τ_0 and y has type τ_1. However, what if the context has the same variable appearning twice? For example, what is type of x if the context is $x : \tau_0, x : \tau_1$? We always favor the declaration closer the tail of the context. In this case, we relate x to the type τ_1, shawdoing the declaration $x : \tau_0$.

We can make the idea of variable lookup in a typing context precise by defining the ternary relation $x : A \in \Gamma$, which holds precisely when x and A appear in Γ without being shadowed by a later declaration.

$$\frac{}{x : A \in \Gamma, x : A} \qquad \frac{x \neq y \qquad x : A \in \Gamma}{x : A \in \Gamma, y : B}$$

The $x \neq y$ side condition in the second rule prevents us from looking up further if we have around found a declaration for it.

Exercise 50. Show that it is not the case that $x : \mathbf{bool} \in x : \mathbf{bool}, x : \mathbf{bool} \to \mathbf{bool}$

Exercise 51. Show that if $x : A \in \Gamma$ and $x : B \in \Gamma$, then $A = B$. In other words, the lookup relation can be viewed as a partial function that takes x and Γ as inputs and returns a unique A as output if a declaration for x exists in Γ.

You can prove this statement by induction on either Γ or $x : A \in \Gamma$.

Now we have all the ingredients needed to specify the typing judgment $\Gamma \vdash a : A$, a ternary relation that relates the term a to the type A under the typing context Γ. Again, the $\vdash$ and : in $\Gamma \vdash a : A$ are notations and bear no special meaning. The relation is defined inductively as follows, with the rule names annotated on top of each rule.

$$\text{T-Var}\ \frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \text{T-Bool}\ \frac{a \in \{\mathbf{true}, \mathbf{false}\}}{\Gamma \vdash a : \mathbf{bool}} \qquad \text{T-If}\ \frac{\Gamma \vdash a : \mathbf{bool} \qquad \Gamma \vdash b : A \qquad \Gamma \vdash c : A}{\Gamma \vdash \mathbf{if}\ a\ b\ c : A}$$

$$\text{T-Lam}\ \frac{\Gamma, x : A \vdash a : B}{\Gamma \vdash \lambda x.a : A \to B} \qquad \text{T-App}\ \frac{\Gamma \vdash b : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash b\ a : B}$$

Exercise 52. Prove that $\cdot \vdash \lambda x.\lambda y.x\ y : (\mathbf{bool} \to \mathbf{bool}) \to \mathbf{bool} \to \mathbf{bool}$.

Exercise 53. Let A, B, and C be arbitrary types. Find a lambda term of the type $(A \to B) \to (B \to C) \to A \to C$.

Exercise 54. Let A, B, and C be arbitrary propositions. Prove that if $A \to B$ and $B \to C$, then $A \to C$.

Compare your proof of this exercise to the lambda term from the previous exercise and convince yourself that the lambda term can be viewewd as a formal encoding of the proof you have written for this exercise.

What should we expect from the typing relation? We can view the typing relation as a way of carving out a set of well-behaved lambda terms that satisfy some desired properties. These properties may vary depend on the use case. For our simple system, we want to ensure that well-typed terms do not step into the stuck state we have discussed informally at the beginning of this section. Here, we make the notion of stuckness precise.

Definition 19 (Values)**.** We say that a lambda term is a value if it is of the form $\lambda x.a$, **true**, or **false**.

Definition 20 (Stuck terms)**.** A lambda term a is stuck if it is *not* a value and there is no term b such that $a \rightsquigarrow b$.

Our goal is to show the following proposition, which we refer to as the type safety property.

Definition 21 (Type safety)**.** Our language is type safe if given $\Gamma \vdash a : A$ and $a \rightsquigarrow^* b$, the term b is not stuck.

Exercise 55. Prove that the term $(\lambda x.x\ x)\ (\lambda x.x\ x)$ can only step into itself and therefore can never be in a stuck state.

The previous exercise shows that type safety says nothing about termination. A language can include infinite loops and still be type safe.

Proving type safety using the nominal representation is quite painful. In the next section, we present the type system using the de Bruijn representation and prove the type safety result without the headache of dealing with named variables.