6.820 Notes

Jason Chen
Lecturer: Armando Solar-Lezama

Last Updated: February 6, 2022

Contents

1 Introduction

The goal of this course is to give an introduction to the ideas in the programming language community. In particular, we want to understand the tools that help us think about a program's behavior. This has applications in

- finding bugs,
- designing languages to prevent bugs,
- program synthesis, and
- program optimization.

As a prerequisite, we must be able to understand the program behavior ourselves. In particular, we want to learn how to define programs and languages unambiguously, how to prove theorems about the behavior of programs, and how to automate this process.

The big ideas in this class include:

Operational semantics which gives programs meaning via interpreters that have the added feature that we can make mathematical statements while interpreting.

Program proofs as inductive invariants where we take advantage of the structure that programs maintain within a loop

Abstraction where we model programs with specifications, as opposed to the implementation. E.g. considering a program as a bag of statements without order.

Modularity where we split programs up into pieces to analyze smaller parts separately.

For this class, we will be using Haskell, Coq, OCaml, and the Spin model checker.

2 Functional Programming

2.1 Introduction

In functional programming, functions are first-class values. This means that, for instance, you can pass functions as values and return functions in other functions. Another (more defining) feature of functional programming languages are that they are based on lambda calculus.

The following is a function definition

```
   increment x = x + 1
-- [1]      [2]  [ 3 ]
```

1. `increment` is the name of the function we are defining.
2. The first `x` is the parameter of the function.
3. The `x + 1` is the body of the function. It is how the function is evaluated if an argument is supplied.

2.1.1 Scoping

When we want to introduce components of expressions, we can use the `let ... in ...` construction:

```
let dx = 5 - 2
    dy = 3 - 1
 in dx * dx + dy * dy
```

which is equivalent to `(5 - 2) * (5 - 2) + (3 - 1) * (3 - 1)`. The order of the statements inside of the `let` do not matter, even if one depends on the other.

When we have nested `let ... in ...` expressions with the same variable names, the value of the variables are based on the innermost scope. For instance,

```
let x = 3
    y = 4
 in let y = 5
     in x + y
```

and

```
let x = 3
    y = 4
 in x + 5
```

are equivalent.

While the compiler may understand what this program means, and the program only has one interpretation, it is confusing for humans to understand it when different values are bound to the same name. To solve this issue, we can rename some variables (formally called α-renaming). This gives us the equivalent code of

```haskell
let x = 3
    y = 4
 in let y' = 5
     in x + y'
```

Note that the apostrophe (`'`, typically called “prime”) is indeed a valid character to use in variable names.

However, we should be careful when doing this, to make sure we are not overriding another value. In particular, we do not want to rename `y` to `x`, as this changes the behavior of the program.

```haskell
let x = 3
    y = 4
 in let x = 5
     in x + x
```

Of course, the easy solution is to just not code like this in the first place.

2.1.2 Currying

In Haskell, a function can only have one parameter. However, we can still define functions that seem to take in multiple parameters.

```haskell
let plus x y = x + y
 in plus 3 4
-- 7
```

The way we interpret this is `(plus x) y = x + y`. In particular, `plus` is a function that takes in a number `x` and returns another function `plus x`. This new function called `plus x` then takes in another number `y` and adds `x` to it. This idea of transforming a function that takes multiple arguments into one that takes in arguments one at a time is called **currying**. This is useful because it lets us partially apply functions:

```haskell
let plus x y = x + y
    increment = plus 1
 in increment 3
-- 4
```

In general, we interpret `(a b c d)` as `(((a b) c) d)`, i.e. function application associates to the left.

2.1.3 Infix operators

Note that we have been defining `plus x y = x + y`. It would be convenient if we are able to say that `plus` is the same thing as `+`. We can indeed do this. In the expression `(x + y)`, we are using `+` as an infix operator, but we can convert it to a normal function by writing it as `(+)`. In particular, our original definition is equivalent too `plus x y = (+) x y`. We can simplify this further to the code `plus = (+)`.

Conversely, if we have a named function that takes in (at least) two arguments, we can make it infix by surrounding the name in backticks, like ``(x `plus` y)``.

2.2 Lambda Calculus

Lambda calculus gives us a way to write and apply functions without needing to give them names.

The heart of λ-calculus are expressions. They have the grammar

$$E = x \mid \lambda x.E \mid EE.$$

The x is a variable. The more interesting terms are

application E_1E_2 where E_1 is the function and E_2 is the argument. Application is left-associative, so $E_1E_2E_3E_4$ means $(((E_1E_2)E_3)E_4)$.

abstraction $\lambda x.E$ where the x is called the **bound variable** and E is the **lambda term**. The body of abstraction extends as far right as possible.

λ-calculus follows **lexical scoping**, which means that variables are bound based on the closest definition.

A **free variables** are the variables that are not bound by a λ within the expression. In particular, we have the following rules:

- $FV(x) = \{x\}$
- $FV(E_1E_2) = FV(E_1) \cup FV(E_2)$
- $FV(\lambda x.E) = FV(E) \setminus \{x\}$

A **combinator** is a λ-expression with no free variables.

To do calculation, we use a rule called **β-reduction**, which is a reduction rule telling us how to apply functions:

$$(\lambda x.E)E_a \longrightarrow E[E_a/x].$$

The expression $E[E_a/x]$ means we substitute E_a for x in E. More formally:

- If E is a variable y,
$$y[E_a/x] = \begin{cases} E_a & \text{if } x \equiv y \\ y & \text{otherwise.} \end{cases}$$
- If E is an application E_1E_2,
$$(E_1E_2)[E_a/x] = (E_1[E_a/x])(E_2[E_a/x]).$$
- If E is an abstraction $\lambda y.E$,
$$(\lambda y.E)[E_a/x] = \begin{cases} \lambda y.E & \text{if } x \equiv y \\ \lambda z.E[z/y][E_a/x] & \text{otherwise,} \end{cases}$$
where $z \notin FV(E) \cup FV(E_a) \cup FV(x)$. Note that we need the $[z/y]$ step in the case that E has the variable y already, to avoid substitution capture.

The process of the substitution $[z/y]$ has a more general name: **α-substitution**:

$$\lambda x.E \longrightarrow \lambda y.E[y/x],$$

where $y \notin FV(E)$.

There is another rule called **η-reduction**:

$$\lambda x.Ex \longrightarrow E,$$

where $x \notin FV(E)$.

We call the subterms that match the left sides of these rules **redexes** (for reducible expression). We say that a term is in **normal form** if they contain no redexs, i.e. we cannot apply any more rules.

1 **Example**

Consider the λ-expressions

$$C \equiv \lambda x.\lambda y.\lambda f.fxy$$
$$H \equiv \lambda x.\lambda y.x$$
$$T \equiv \lambda x.\lambda y.y$$

These expressions correspond to cons, head, and tail. For instance,

$$\begin{aligned}(Cab)H &\longrightarrow ((\lambda x.\lambda y.\lambda f.fxy)ab)(\lambda x.\lambda y.x)\\ &\longrightarrow (\lambda f.fab)(\lambda x.\lambda y.x)\end{aligned}$$

$$\begin{aligned}
&\to (\lambda x.\lambda y.x)ab \\
&\to a \\
(Cab)T &\to ((\lambda x.\lambda y.\lambda f.fxy)ab)(\lambda x.\lambda y.y) \\
&\to (\lambda f.fab)(\lambda x.\lambda y.y) \\
&\to (\lambda x.\lambda y.y)ab \\
&\to b.
\end{aligned}$$

These combinators can be interpreted multiple ways. Another way of interpreting C, H, and T are as pair, first, and second.

2.2.1 Church encoding

We can also encode the natural numbers with λ-expressions. Let

$$\begin{aligned}
0 &\equiv \lambda f.\lambda x.x \\
1 &\equiv \lambda f.\lambda x.fx \\
2 &\equiv \lambda f.\lambda x.f(fx) \\
3 &\equiv \lambda f.\lambda x.f(f(fx)) \\
&\vdots
\end{aligned}$$

where we interpret the natural number n as applying a function f to some argument x, n times.

Using this interpretation, we can define succ, the successor function. In particular, succ is a function that takes in a natural n and returns another numeral that applies the function f, $n+1$ times. We know that nfx applies the function n times so we apply f to the result of this once more to get the desired expression.

$$\begin{aligned}
\text{succ} &\equiv \lambda n.(\lambda f.\lambda x.f(nfx)) \\
&\equiv \lambda n.\lambda f.\lambda x.f(nfx).
\end{aligned}$$

Since a natural n is encoded as applying a function n times, we can define plus as repeated application of succ:

$$\begin{aligned}
\text{plus} &\equiv \lambda m.\lambda n.m\,\text{succ}\,n \\
&\equiv \lambda m.m\,\text{succ}\,.
\end{aligned}$$

where $m\,\text{succ}\,n$ can be thought of as applying succ, m times to n. The second line is obtained from η-reduction on $\lambda n.m\,\text{succ}\,n$.

We can similarly define times to be

$$\text{times} \equiv \lambda m.\lambda n.m(\text{plus}\,n)0.$$

We can also represent booleans as:

$$\begin{aligned}
\text{true} &\equiv \lambda x.\lambda y.x \\
\text{false} &\equiv \lambda x.\lambda y.y
\end{aligned}$$

If we want to make choices based on a boolean, like the `b?x:y` operator in C, note that true and false select their first and second arguments respectively. Therefore, we have the analogous operator in λ-calculus

$$\text{cond} = \lambda b.\lambda x.\lambda y.bxy.$$

Note that this η-reduces to the identity function, so we can actually just apply the boolean to two values we want to choose betwen.

We can write a function isZero by applying $\lambda x.\,\text{false}$ to true n times. In particular, for any natural other than 0, the result will be false, and otherwise it will be true. Written out,

$$\begin{aligned}\mathsf{isZero} &\equiv \lambda n.n(\lambda x.\,\mathsf{false})\,\mathsf{true} \\ &\equiv \lambda n.n(\mathsf{true}\,\mathsf{false})\,\mathsf{true}\,.\end{aligned}$$

2.2.2 Semantics

Operational semantics talks about how the terms reduce. Operational semantics means that two terms are the same thing if one can be reduced to the other, e.g. PLUS 0 1 and 1 should have the same meaning.

Denotational semantics says what the terms mean. Denotational semantics is mapping lambda terms to meanings like the naturals, which should distinguish between terms that should not be equal.

The goal is to show that operational semantics and denotational semantics agree. A semantics is **fully abstract** if when two terms have different meanings according to the semantics, then there exists a term that can distinguish them.

If we have a lambda term, and replace each redex by $\perp$ (**bottom**), then we get the **instantaneous information** of the term, i.e. "what we already know". Here $\perp$ represents "no information".

2 **Example**

We can calculate the instantaneous information for a few terms as we evaluate

$$\begin{aligned}(\lambda q.\lambda p.p(qa))(\lambda z.z) &\xrightarrow{\text{instantaneous information}} \perp \\ \lambda q.\lambda p.p((\lambda z.z)a) &\xrightarrow{\text{instantaneous information}} \lambda p.p\perp \\ \lambda p.pa &\xrightarrow{\text{instantaneous information}} \lambda p.pa\end{aligned}$$

Note that β-reductions monotonically increase information. We say that the meaning of a term is the maximum amount of information that can be obtained by β-reductions.

One natural definition is to say that the meaning of a term is the normal form. However, a term may not have a normal form, e.g.

$$\Omega = (\lambda x.xx)(\lambda x.xx).$$

If we try to reduce this, we get Ω back. We say that the meaning of Ω is $\perp$. If we have a term like $\lambda x.x\Omega$, then its meaning is $\lambda x.x\perp$.

Also, we may consider the case where a term may have more than one normal form. If this is true, then the term would have multiple meanings. However, this is not possible because of **confluence**.

A term may have multiple redexes, e.g.

$$(\underbrace{(\lambda x.M)A}_{\rho_1})(\underbrace{(\lambda x.N)B}_{\rho_2}) \qquad \underbrace{((\lambda x.M)(\underbrace{(\lambda x.N)B}_{\rho_2}))}_{\rho_1}$$

Note that ρ_1 followed by ρ_2 does not necessarily produce the same thing as ρ_2 followed by ρ_1. However, Church-Rosser property says that this does not matter.

3 **Definition**

The **Church-Rosser property** says that if $E \longrightarrow E_1$ and $E \longrightarrow E_2$, then there exists some E_3 such that $E_1 \longrightarrow E_3$ and $E_2 \longrightarrow E_3$.

4 **Theorem** (Church-Rosser, Martin-Löf & Tate)

λ-calculus has the Church-Rosser property.

This implies that if a normal form exists, then it is unique.

2.2.3 Interpreters

An **interpreter** for the λ-calculus is a program that reduces λ-expressions to "answers". However, to do this we need to know what we mean by "answers", and how to choose redexs.

One way to define "answer" is to think about normal form. However, this may be too restrictive, so we consider the following:

5 **Definition**

A β-redex r is in **head position** in a term t if t is of the form

$$t = \lambda x_1 \ldots \lambda x_n. \underbrace{(\lambda x.A)M_1}_{r} M_2 \ldots M_m$$

where $n \geq 0$ and $m \geq 1$. Recall that normal form is a term in which we cannot apply any more β-reductions,

6 **Definition**

Expressions are in **head normal form** if they do not contain a β-reduction in head position. In particular,

- x is in HNF,
- $(\lambda x.E)$ is in HNF if E is in HNF, and
- $(xE_1 \ldots E_n)$ is in HNF.

They are terms of the following form:

$$\lambda x_1 \ldots \lambda x_n.xM_1M_2 \ldots M_m,$$

where x is a variable, and $n, m \geq 0$. Note that $M_1, \ldots, M_m$ are not necessarily in normal form.

This is semantically the most interesting, because it represents the information content of an expression.

7 **Definition**

Expressions are in **weak head normal form** if the left-most application is not a redex. In particular,

- x is in WHNF,
- $(\lambda x.E)$ is in WHNF, and
- $(xE_1 \ldots E_n)$ is in WHNF.

Equivalently, a term is in WHNF if it is in HNF or a lambda abstraction.

This is the most practical form, because it represents printable forms.

8 **Example**

- $(\lambda x.xy)$, z, and xy are in normal form.
- $(x((\lambda y.y)z))$ and $(\lambda x.(xz))$ are in HNF but not normal form.
- $(\lambda x.(\lambda t.ty)x)$ is in WHNF but not HNF.
- $\Omega = (\lambda x.xx)(\lambda x.xx)$ is not in any normal form.

We have two common reduction strategies.

Applicative order is when we evaluate the right-most innermost redex first. This is also called **call by value** evaluation.

Normal order is when we evaluate the left-most (outermost) redex first. This is also called **call by name** evaluation.

When we compute a normal form, keep in mind that:

- Not every λ-expression has an answer, i.e.

$$\Omega \to \Omega \to \Omega \to \cdots .$$

- Even if an expression has an answer, not all reduction strategies may produce it, e.g. $(\lambda x.\lambda y.y)\Omega$.

9 **Remark**

In most languages, e.g. Java or C, we use call by value evaluation for almost everything. The way we get around it for recursion is that `if` statements do not evaluate both branches.

A reduction strategy is **normalizing** if it terminates and produces an answer of an expression whenever it has an answer.

10 **Theorem**

The normal order reduction strategy is normalizing for the λ-calculus.

A call-by-name interpreter gives the weak head normal form of an expression by evaluating the left-most redex. In particular, we apply the function before evaluating the arguments.

```
cn([[x]]) = x
cn([[λx.E]]) = λx.E
cn([[E1 E2]]) = let f = cn(E1)
                in case f of
                     λx.E3 -> cn(E3[E2/x])
                     _ -> f E2
```

In a call-by-value interpreter, we evaluate the right-most innermost redex not inside of a lambda abstraction. In particular, we evaluate the arguments before applying the function.

```
cv([[x]]) = x
cv([[λx.E]]) = λx.E
cv([[E1 E2]]) = let f = cv(E1)
                    a = cv(E2)
                in case f of
                     λx.E3 -> cv(E3[a/x])
                     _ -> f a
```

2.2.4 Recursion

In λ-calculus, recursive functions can be thought of as solutions of fixed point equations. For instance, we know

$$\mathrm{fact} = \lambda n.\,\mathsf{Cond}(\mathsf{isZero}\,n)(1)(\mathsf{Times}\,n(\mathrm{fact}(\mathsf{Minus}\,n\,1))).$$

If we have the function

$$H = \lambda f.\lambda n.\,\mathsf{Cond}(\mathsf{isZero}\,n)(1)(\mathsf{Times}\,n(f(\mathsf{Minus}\,n\,1))),$$

note that fact is a fixed point of H (i.e. H fact = fact).

Now we need to figure out how to determine the fixed point of a function. If we were doing this in $\mathbb{R}$, it would not be hard to do by iteration, as we have a notion of distance. Here we are working in a space of discrete functions, in which we can use a similar idea, called Scott continuity.

One thing to be cautious about is the existence of more than one fixed point. For instance, consider

$$f = \lambda n.\,\mathrm{if}(n=0)\,\mathrm{then}(1)\,\mathrm{else}(\mathrm{cond}(n=1, f(3), f(n-2)))$$
$$H = \lambda f.\lambda n.\,\mathrm{if}(n=0)\,\mathrm{then}(1)\,\mathrm{else}(\mathrm{cond}(n=1, f(3), f(n-2)))$$

Note that both

$$f_1 n = \begin{cases} 1 & n \text{ is even} \\ \bot & \text{otherwise} \end{cases}$$

$$f_2 n = \begin{cases} 1 & n \text{ is even} \\ 5 & \text{otherwise} \end{cases}$$

are both fixed points of H. However, it feels like f_2 is "making things up" about f. In particular, f_1 contains no arbitrary information, and is called the least fixed point. Moreover, assuming monotonicity and continuity, least fixed points are unique and computable.

We can actually define an order on these, and then compute some optimal fixed point. Recall $\Omega = (\lambda x.xx)(\lambda x.xx)$. If we modify this to $\Omega_F = (\lambda x.F(xx))(\lambda x.F(xx))$, note that this β-reduces to $F((\lambda x.F(xx))(\lambda x.F(xx)))$. In particular, we have $F\Omega_F = \Omega_F$. This means that Ω_F is a fixed point of F! Therefore, $Y = \lambda F.\Omega_F$ is a function that gives the fixed point of F.

2.3 Big Step Semantics

Big step operational semantics models the execution in an abstract machine.

- There are **judgements**, written $\langle\text{configuration}\rangle \longrightarrow \text{result}$, describing how a program configuration is evaluated into a result. The configuration typically includes the program as well as any state.
- There are **inference rules**, which describe how to derive judgements for an arbitrary program. These are sometimes called derivation rules or evaluation rules.

11 **Example**

Call by name evaluation order is described by:

$$\frac{}{\langle x\rangle \longrightarrow x} \qquad \frac{}{\langle \lambda x.e\rangle \longrightarrow \lambda x.e} \qquad \frac{\langle e_1\rangle \longrightarrow \lambda x.e_1' \quad \langle e_1'[e_2/x]\rangle \longrightarrow e_3}{\langle e_1 e_2\rangle \longrightarrow e_3} \qquad \frac{\langle e_1\rangle \longrightarrow x}{\langle e_1 e_2\rangle \longrightarrow x e_2} \qquad \frac{\langle e_1\rangle \longrightarrow e_a e_b}{\langle e_1 e_2\rangle \longrightarrow e_a e_b e_2}$$

Call by value evaluation order is described by:

$$\frac{}{\langle x\rangle \longrightarrow x} \qquad \frac{}{\langle \lambda x.e\rangle \longrightarrow \lambda x.e} \qquad \frac{\langle e_1\rangle \longrightarrow \lambda x.e_1' \quad \langle e_2\rangle \longrightarrow e_2' \quad \langle e_1'[e_2'/x]\rangle \longrightarrow e_3}{\langle e_1 e_2\rangle \longrightarrow e_3}$$

$$\frac{\langle e_1\rangle \longrightarrow x \quad \langle e_2\rangle \longrightarrow e_2'}{\langle e_1 e_2\rangle \longrightarrow x e_2'} \qquad \frac{\langle e_1\rangle \longrightarrow e_a e_b \quad \langle e_2\rangle \longrightarrow e_2'}{\langle e_1 e_2\rangle \longrightarrow e_a e_b e_2'}$$

Often, we drop the angle brackets for convenience.

12 **Example**

To prove $(\lambda x.x)((\lambda y.y)z) \longrightarrow z$, we have the proof tree

$$\frac{\frac{}{\lambda x.x \longrightarrow \lambda x.x} \qquad \frac{\frac{}{\lambda y.y \longrightarrow \lambda y.y} \quad \frac{}{y[z/y] \longrightarrow z}}{(\lambda y.y)z \longrightarrow z}}{(\lambda x.x)((\lambda y.y)z) \longrightarrow z}$$

2.4 Coq

Before we work with big-step semantics, we will first introduce a tool we will use. One good thing about Coq is that it tries to rely on as little as possible.

In Coq, we can introduce definitions and theorem, and we prove them by applying steps called **tactics**.

For instance, we can define the natural numbers:

```
Inductive nat := O | S (n : nat).
Fixpoint plus (n m : nat) : nat :=
  match n with
  | O => m
  | S n' => S (plus n' m)
  end.
```

Here O represents zero, and S represents successor.

To define a theorem and its proof, we can write:

```
Lemma O_plus : forall n, plus O n = n.
Proof.
  (* sequence of tactics *)
Qed.
```

Here, the word **Lemma** can be replaced with any of **Theorem**, **Remark**, **Corollary**, **Fact**, or **Proposition**.

Proving theorems in Coq is a very interactive process, so it is impossible to recreate on paper. It is strongly recommended to try running the Coq code, as just knowing the tactics is only a small part of the tool.

When running Coq code interactively, we can check the status of a proof by running the code up to some point in the proof. There is a horizontal line, above which is a list of hypotheses: objects we know exist or statements we know are true. Below the line is the goal we want to prove.

2.4.1 Tactics

In order to complete proofs, we apply tactics that use the hypotheses to resolve our goal.

- **reflexivity** can be used if our goal is resolved by normalizing terms, e.g. `plus O (S O) = S O`.
- **induction** `x` can be used to prove a goal by induction on a quantified variable `x`. Here, `x` is a inductively defined data type. Note that all variables quantified before `x` will be fixed throughout the induction. More on this in the next section.
- **simpl** will generally simplify your goal. Typically, it will include doing some β-reduction.
- **rewrite** `H` will rewrite the goal using a theorem/hypothesis `H : a = b`, by replacing `a` with `b`. Similarly, **rewrite** `<- H` will replace `b` with `a`.
- **intro** moves a quantified variable or hypothesis above the line. **intros** will do this as many as possible.
- **apply** `thm` will apply a theorem, reducing the goal to a subgoals for each of the theorem's hypotheses. `H : a = b`, by replacing `a` with `b`.
- **assumption** will solve a goal that matches a hypothesis.
- **destruct** `E` will do case analysis on some term `E`.

2.5 Induction

Recall the definition of the naturals

$$\texttt{N := O | S N.}$$

To prove $\forall n \in \texttt{N}, P(n)$ by induction,

- Prove $P(\texttt{O})$.
- Assume $P(n)$ for an arbitrary $n \in \texttt{N}$. Prove $P(\texttt{S}\, n)$.

2.5.1 Structural induction

We can extend this idea to **structural induction**: if we have a tree datatype

$$\texttt{Tree := Leaf | Node Tree Tree,}$$

To prove $\forall t \in \texttt{Tree}.P(t)$ by induction,

- Prove $P(\texttt{Leaf})$.
- Assume $P(t_1)$ and $P(t_2)$ for arbitrary $t_1, t_2 \in \texttt{Tree}$. Prove $P(\texttt{Node}\, t_1\, t_2)$.

We can see another example of this if we consider the following datatype of expressions

```
E := Const N | Plus E E | Times E E.
```

To prove $\forall e \in \texttt{E}.P(e)$ by induction,

- Prove $P(\texttt{Const n})$ forall $n \in \mathbb{N}$.
- Assume $P(e_1)$ and $P(e_2)$ for arbitrary $e_1, e_2 \in \texttt{E}$. Prove $P(\texttt{Plus}\, e_1\, e_2)$.
- Assume $P(e_1)$ and $P(e_2)$ for arbitrary $e_1, e_2 \in \texttt{E}$. Prove $P(\texttt{Times}\, e_1\, e_2)$.

2.5.2 Rule induction

We can extend this even further to **rule induction** (also called induction on the structure of derivations):

Consider proofs that a natural is even. We know that 0 is even, and we know that if n is even, then $n + 2$ is even.

$$\frac{}{\mathsf{even}(0)} \qquad \frac{\mathsf{even}(n)}{\mathsf{even}(n+2)}$$

This corresponds to the (dependent) datatype

```
even := Even0 : even(0)
      | Even2(even(n)) : even(n + 2)
```

To prove $\forall n \in \mathbb{N}.\mathsf{even}(n) \implies P(n)$,

- Prove $P(0)$.
- Assume $P(n)$ for an arbitrary $n \in \mathbb{N}$. Prove $P(n+2)$.

Another example of this is the following structure:

```
eval := EvConst : eval(Const n, n)
      | EvPlus eval(e1, n1) eval(e2, n2) : eval(Plus e1 e2, n1 + n2)
```

corresponding to the inference rules:

$$\frac{}{\mathsf{eval}(\mathsf{Const}\, n, n)} \qquad \frac{\mathsf{eval}(e_1, n_1) \quad \mathsf{eval}(e_2, n_2)}{\mathsf{eval}(\mathsf{Plus}\, e_2\, e_2, n_1 + n_2)}$$

To prove $\forall e \in \mathsf{E}, n \in \mathbb{N}.\mathsf{eval}(e, n) \implies P(e, n)$,

- Prove $P(\mathsf{Const}\, n, n)$.
- Assume $P(e_1, n_1)$ and $P(e_2, n_2)$. Prove $P(\mathsf{Plus}\, e_1\, e_2, n_1 + n_2)$.

One possible P we may use here is if we have some interpreter, and $P(e, n)$ is the proposition that interpreting e gives n.

3 Type Theory

3.1 Type systems

When programming, we want to know which programs are valid. In particular, we want to determine which programs have semantics. For instance, if

```
let f x = if x then 1 else 0
 in f 6
```

this will evaluate to `if 6 then 1 else 0`. There are multiple options for how a language can handle this:

- Leave the behavior up to the implementation of the language. (E.g. C, where there are things with unspecified behavior, like the order in which arguments are evaluated.)

  This is not a good idea.
- Provide a way that identifies and rules out these "bad" programs. In this case, programs will only compile/run if the programmer can prove they will be correct.

  This is the essence of a type system.
- Give every program correct behavior.

  This is the approach that a language like JavaScript takes: the `6` is typecasted into `true`, and the program runs. However, this means that nonsensical things like adding a string to an object is defined.

  Note that λ-calculus also does this!

  Type systems are useful in these cases too.

13 **Sidenote** (Self-application)
In λ-calculus, self application is essential for recursion, but it turns out it also introduces paradoxes.
Consider $u = \lambda y.\, \text{if } (yy) = a \text{ then } b \text{ else } a$. Then, $uu = \text{if } (uu) = a \text{ then } b \text{ else } a$, which is a contradiction.
This was one of the original motivation for types.

In a narrow sense, a type system is a mechanism for ensuring that variables only take values from a certain predefined category. However, it can also be viewed as a proof system or annotation system that lets us check a specific property of interest for a program. In particular, they can even deal with information flow violations, deadlocks, and more.

We write $e : T$ to represent a **value** (or **object**) e of **type** T. When $x : \tau$, then only operations that are appropriate on τ may be performed on x. We say that a program is **type correct** if it never performs a wrong operation on an object.

Note that whether something is appropriate may differ by languages, as in Python we can multiply lists by integers, but in Java we cannot.

A language is **type safe** if only type correct programs can be written in that language. Note that just because a language has types does not mean it has type safe:

- In Fortran, there is equivalence and parameter passing.
- In Pascal, there is variant records and files.
- In C/C++, there are pointers and type casting.

However, Java, Ada, ML, Go, Haskell, Bluespec, etc. are type-safe.

Type declaration and inference In modern languages, we can get all of the benefits of types without explicitly declaring types. A language is called **statically typed** if type checking is done at compile time.

- Languages where users must declare types: CLU, Pascal, Ada, C, C++, Fortran, Java
- Languages where type declaration are reconstructed during runtime: Scheme, Lisp
- Languages where type declaration are not required but allowed, and are reconstructed during runtime: ML, Go, Haskell, pH, Bluespec

Recently, the boundary between the languages that require type declarations and those that don't is becoming fuzzier, as type inference is catching on.

Polymorphism A **polymorphic language** is a language in which there are type variables. For instance, in Pascal, a monomorphic language, there has to be a different length function for each type of list. A non-polymorphic type is called a **simple type**.

In a polymorphic language like ML, there is a polymorphic type `list t`, where `t` is the type variable, and we can define a single function to determine the length.

Most modern functional programming languages have type systems, and follow the Hindley-Milner type system.

3.1.1 Formal definition

A type system of a language is almost never orthogonal to the semantics of a language, as the types might affect the behavior of the language (e.g. operator overloading). When we define a typed language, we need

- the syntax,
- the dynamic semantics (operational semantics),
- the static semantics (typing rules, and how types are assigned), and
- a type soundness argument (relationship between static and dynamic semantics).

In the language, the type system assigns types to elements. For example, `5:Int` and `"Hello":String`. However, the types of some elements (like `x`) depends on the environment. We denote this by $\Gamma \vdash e : T$. The Γ is the environment, and we call the entire statement a **judgement**.

The typing rules tell us how to derive typing judgements. These look very similar to the derivation rules in big step operational semantics.

14 **Example**

For the language of expressions, the typing judgements are

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash N : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}}$$

where x is a variable, and N is an integer in the language.

If we want to prove $x : \mathsf{int}, y : \mathsf{int} \vdash x + (y + 5) : \mathsf{int}$, we have the proof tree

$$\frac{\dfrac{x : \mathsf{int} \in \Gamma}{\Gamma \vdash x : \mathsf{int}} \qquad \dfrac{\dfrac{x : \mathsf{int} \in \Gamma}{\Gamma \vdash y : \mathsf{int}} \qquad \dfrac{}{\Gamma \vdash 5 : \mathsf{int}}}{\Gamma \vdash (y + 5) : \mathsf{int}}}{\underbrace{x : \mathsf{int}, y : \mathsf{int}}_{\Gamma} \vdash x + (y + 5) : \mathsf{int}}$$

3.1.2 Simply typed λ calculus (F_1)

We can introduce typing rules into lambda calculus:

$$\frac{x:\tau\in\Gamma}{\Gamma\vdash x:\tau} \qquad \frac{(\Gamma,x:\tau_1)\vdash e:\tau_2}{\Gamma\vdash(\lambda x:\tau_1.e):\tau_1\to\tau_2} \qquad \frac{\Gamma\vdash e_1:\tau'\to\tau \quad \Gamma\vdash e_2:\tau'}{\Gamma\vdash e_1e_2:\tau}$$

Note that when we have a lambda, we must specify the type of the input as we did in $\lambda x:\tau_1.e$.

The second typing rule states that if we want to prove a lambda $\lambda x:\tau_1.e$ has type $\tau_1\to\tau_2$, then we must show that $e:t_2$ in the original context Γ with the additional information that $x:\tau_1$. Note that if Γ has a type for x already, then in $(\Gamma,x:\tau_1)$ it is replaced with the new information $x:\tau_1$.

The third typing rule states that if $e_1:\tau'\to\tau$ and $e_2:\tau'$ in some context, then $e_1e_2:\tau$ in that context.

We also include a few more rules

$$\frac{}{\Gamma\vdash N:\mathsf{int}} \qquad \frac{\Gamma\vdash e_1:\mathsf{int} \quad \Gamma\vdash e_2:\mathsf{int}}{\Gamma\vdash e_1+e_2:\mathsf{int}} \qquad \frac{\Gamma\vdash e_1:\mathsf{int} \quad \Gamma\vdash e_2:\mathsf{int}}{\Gamma\vdash e_1=e_2:\mathsf{bool}} \qquad \frac{\Gamma\vdash e:\mathsf{bool} \quad \Gamma\vdash e_t:\tau \quad \Gamma\vdash e_f:\tau}{\Gamma\vdash \mathsf{if}\ e\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f:\tau}$$

15 **Example**

If we wanted to prove

$$\vdash(\lambda x:\mathsf{bool}.\lambda y:\mathsf{bool}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ y+1):\mathsf{bool}\to\mathsf{int}\to\mathsf{int},$$

where nothing to the left of $\vdash$ is the empty context, then we have the proof tree

$$\dfrac{\dfrac{\dfrac{\dfrac{x:\mathsf{bool}\in y:\mathsf{int},x:\mathsf{bool}}{y:\mathsf{int},x:\mathsf{bool}\vdash x:\mathsf{bool}} \quad \dfrac{y:\mathsf{int}\in y:\mathsf{int},x:\mathsf{bool}}{y:\mathsf{int},x:\mathsf{bool}\vdash y:\mathsf{int}} \quad \dfrac{\dfrac{y:\mathsf{int}\in y:\mathsf{int},x:\mathsf{bool}}{y:\mathsf{int},x:\mathsf{bool}\vdash y:\mathsf{int}} \quad \dfrac{}{y:\mathsf{int},x:\mathsf{bool}\vdash 1:\mathsf{int}}}{y:\mathsf{int},x:\mathsf{bool}\vdash y+1:\mathsf{int}}}{y:\mathsf{int},x:\mathsf{bool}\vdash \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ y+1:\mathsf{int}}}{x:\mathsf{bool}\vdash(\lambda y:\mathsf{int}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ y+1):\mathsf{int}\to\mathsf{int}}}{\vdash(\lambda x:\mathsf{bool}.\lambda y:\mathsf{int}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ y+1):\mathsf{bool}\to\mathsf{int}\to\mathsf{int}}$$

This is a really strong type system on λ-calculus. It turns out that this is so strong that we can't write non-terminating computation.

3.2 Progress and preservation

We will formalize what we mean when we say "well typed programs never go wrong". In particular, we will prove this by induction:

- If a program is well typed, it will stay well typed in the next step of evaluation.
- If a program is well typed, then either we are done or we can do another step of evaluations.

In order to do this, we need to formalize what we mean by "step of evaluation".

In contextual semantics, we have contexts that represent a 'hole' in which the next evaluation step will occur.

$$H\coloneqq\bullet\mid H\,e_1\mid v\,H\mid H+e\mid v+H\mid \mathsf{if}\ H\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2,$$

where e is any expression and v is a fully evaluated expression. We also have local reduction rules

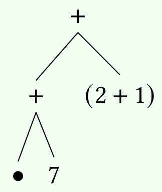
- $n_1+n_2\to\mathsf{plus}\ n_1\ n_2$
- $\mathsf{if}\ \mathsf{true}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\to e_1$
- $\mathsf{if}\ \mathsf{false}\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2\to e_2$
- $(\lambda x:\tau\,.\,e_1)v_2\to e_1[v_2/x]$

and the global reduction rule

$$H[r]\to H[e]\iff r\to e.$$

16 **Example**

If we have $((3+1)+7)+(2+1)$, we have the following context:



Here are some more examples:

Expression	Context	Redex
$(\lambda x : \mathsf{int} \,.\, x+1)(1+1)$	$(\lambda x : \mathsf{int} \,.\, x+1)\bullet$	$1+1$
$(\lambda x : \mathsf{int} \,.\, x+1)2$	$\bullet$	$(\lambda x : \mathsf{int} \,.\, x+1)2$
$2+1$	$\bullet$	$2+1$

For any expression, we can decompose it into a context and a redex. Then, apply the reduction rule to the redex and repeat.

3.2.1 Preservation

When we used big-step semantics, we can prove global preservation. In particular, we can prove this by induction on the structure of derivation of $e_1 \longrightarrow e_2$.

17 **Recall**

To prove $Q(e) \implies P(e)$ by induction on the structure of derivation,

1. Prove the base cases, i.e. prove $P(e)$ holds for any e in which the proof tree for $Q(e)$ has depth 1.
2. Prove the inductive step, i.e. prove $P(e)$ holds for any e in which the proof tree for $Q(e)$ has height $h+1$, assuming that $P(e')$ holds for any e' in which the proof tree of $Q(e')$ has height h.

18 **Proposition** (Global preservation)

$e_1 \longrightarrow e_2 \implies \Gamma \vdash e_1 : \tau \implies \Gamma \vdash e_2 : \tau$.

Proof. We proceed by induction on e_2. Here $Q(e_1, e_2) = (e_1 \longrightarrow e_2)$ and $P(e_1, e_2) = (\Gamma \vdash e_1 : \tau \implies \Gamma \vdash e_2 : \tau)$.

Base cases There are two base cases corresponding to

$$\frac{}{x \longrightarrow x} \qquad \frac{}{\lambda x.e \longrightarrow \lambda x.e}$$

In particular,

$$\Gamma \vdash x : \tau \implies \Gamma \vdash x : \tau$$
$$\Gamma \vdash \lambda x.e : \tau \implies \Gamma \vdash \lambda x.e : \tau$$

are both true.

Inductive case We have the operational rule

$$\frac{e_1 \longrightarrow \lambda x.e_1' \quad e_1'[e_2/x] \longrightarrow e_3}{e_1 e_2 \longrightarrow e_3}$$

We want to prove

$$\Gamma \vdash e_1 e_2 : \tau \implies \Gamma \vdash e_3 : \tau,$$

given the induction hypotheses

$$\forall t.(\Gamma \vdash e_1 : t \implies \Gamma \vdash \lambda x.e_1' : t) \tag{3.1}$$

$$\forall t.(\Gamma \vdash e_1'[e_2/x] : t \implies \Gamma \vdash e_3 : t). \tag{3.2}$$

From the typing rule

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau}$$

so we know there exists some τ' such that $\Gamma \vdash e_1 : \tau' \to \tau$ and $\Gamma \vdash e_2 : \tau'$. By eq. (3.1), we have $\Gamma \vdash \lambda x.e_1' : \tau' \to \tau$. From the typing rule

$$\frac{\Gamma, x : \tau' \vdash e_1' : \tau}{\Gamma \vdash (\lambda x : \tau'.\, e_1') : \tau' \to \tau}$$

we know that $\Gamma, x : t' \vdash e_1' : \tau$.

We use the following lemma:

19 **Lemma**

$$\Gamma, x : \tau' \vdash e_1' : \tau \wedge \Gamma \vdash e_2 : \tau' \implies \Gamma \vdash e_1'[e_2/x] : \tau.$$

The proof is omitted as an exercise. The lemma yields $\Gamma \vdash e_1'[e_2/x] : \tau$, and from eq. (3.2), we have $\Gamma \vdash e_3 : \tau$. □

3.2.2 Progress

20 **Proposition** (Progress)
If $\vdash e : \tau$ and e is not a value, then there exists e' such that $e \to e'$ (equivalently, there exists H and r such that $e = H[r]$).

To prove progress, we want to use small step semantics, because we want to talk about steps of evaluation.

Proof. We prove this by induction on the derivation of $\vdash e : \tau$.

Base cases: We have the base cases

$$\frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash N : \mathsf{int}} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_2 . e) : \tau_1 \to \tau_2}$$

Since these are irreducible values, the condition does not hold, so these vacuously satisfy the proposition.

Inductive cases: We have the inductive cases

$$\frac{\Gamma \vdash e_1 : \tau' \to \tau \quad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 e_2 : \tau} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 = e_2 : \mathsf{bool}} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau}{\Gamma \vdash \mathsf{if}\, e \,\mathsf{then}\, e_t \,\mathsf{else}\, e_f : \tau}$$

We will do the proof for

$$\frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau}{\Gamma \vdash \mathsf{if}\, e \,\mathsf{then}\, e_t \,\mathsf{else}\, e_f : \tau}$$

The rest of the cases are similar. By the inductive hypothesis, e is either a value or can be decomposed into $e = H[r]$.

In the first case, e is just true or false, and the entirety of e is a redex.

In the second case, we have the context $\mathsf{if}\, H \,\mathsf{then}\, e_t \,\mathsf{else}\, e_f$. □

3.3 Type inference

Recall the typing rules of typed lambda calculus:

21 **Typing Rules of $F1$**

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau} \qquad \frac{(\Gamma, x:\tau_1) \vdash e:\tau_2}{\Gamma \vdash (\lambda x:\tau_1 . e):\tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1:\tau' \to \tau \quad \Gamma \vdash e_2:\tau'}{\Gamma \vdash e_1 e_2:\tau}$$

$$\frac{}{\Gamma \vdash N:\text{int}} \qquad \frac{\Gamma \vdash e_1:\text{int} \quad \Gamma \vdash e_2:\text{int}}{\Gamma \vdash e_1 + e_2:\text{int}} \qquad \frac{\Gamma \vdash e_1:\text{int} \quad \Gamma \vdash e_2:\text{int}}{\Gamma \vdash e_1 = e_2:\text{bool}} \qquad \frac{\Gamma \vdash e:\text{bool} \quad \Gamma \vdash e_t:\tau \quad \Gamma \vdash e_f:\tau}{\Gamma \vdash \text{if}\, e\, \text{then}\, e_t\, \text{else}\, e_f:\tau}$$

22 **Example**

Consider whether the expression $(\lambda f:\text{int} \to \text{int} . f5)(\lambda x:\text{int} . x+1)$ is well typed in F_1.

We can construct the following type checking tree from the bottom:

$$\frac{\dfrac{\dfrac{\dfrac{f:\text{int} \to \text{int} \in f:\text{int} \to \text{int}}{f:\text{int} \to \text{int} \vdash f:\text{int} \to \tau}\;\text{(B)} \quad \dfrac{}{f:\text{int} \to \text{int} \vdash 5:\text{int}}}{f:\text{int} \to \text{int} \vdash f5:\tau}}{\vdash (\lambda f:\text{int} \to \text{int} . f5):\tau_1 \to \tau}\;\text{(A)} \quad \dfrac{\dfrac{\dfrac{x:\text{int} \in x:\text{int}}{x:\text{int} \vdash x:\text{int}} \quad \dfrac{}{x:\text{int} \vdash 1:\text{int}}}{x:\text{int} \vdash x+1:\text{int}}}{\vdash (\lambda x:\text{int} . x+1):\tau_1 = \text{int} \to \text{int}}}{\vdash (\lambda f:\text{int} \to \text{int} . f5)(\lambda x:\text{int} . x+1):\tau}$$

At (A), we know that $\tau_1 = \text{int} \to \text{int}$. At (B), we know that $\tau = \text{int}$, so the entire expression is of type int. The rest of the tree typechecks, so this is well typed.

Note that even if we don't have the type annotations, we can still have done the same type derivation:

23 **Example**

We can check the well typed-ness of $(\lambda f . f5)(\lambda x . x+1)$ in F_1:

$$\frac{\dfrac{\dfrac{\dfrac{f:\tau_4 \to \tau \in f:\tau_1}{f:\tau_1 \vdash f:\tau_4 \to \tau}\;\text{(D)} \quad \dfrac{}{f:\tau_1 \vdash 5:\text{int}}}{f:\tau_1 \vdash f5:\tau}}{\vdash (\lambda f . f5):\tau_1 \to \tau} \quad \dfrac{\dfrac{\dfrac{x:\text{int} \in x:\tau_2}{x:\tau_2 \vdash x:\text{int}}\;\text{(C)} \quad \dfrac{}{x:\tau_2 \vdash 1:\text{int}}}{x:\tau_2 \vdash x+1:\tau_3}\;\text{(B)}}{\vdash (\lambda x . x+1):\tau_1}\;\text{(A)}}{\vdash (\lambda f . f5)(\lambda x . x+1):\tau}$$

- From (A), we have $\tau_1 = \tau_2 \to \tau_3$.
- From (B), we have $\tau_3 = \text{int}$.
- From (C), we have $\tau_2 = \text{int}$.
- From (D), we have $\tau_1 = \tau_4 \to \tau$.

This gives $\tau_1 = \text{int} \to \text{int}$, and $\tau_2 = \tau_3 = \tau_4 = \tau = \text{int}$.

This is the idea behind type inference: we can find the types of expressions even if the types are not explicitly given in the program. More explicitly, the strategy we used for type inference is

1. Use the typing rules to find constraints on the types of each subexpression.
2. Solve the resulting system of constraints.

24 **Example**

To do type inference in Haskell, we do a similar process. Suppose we have the function `twice f x = f (f x)`. We can find the most general type for `twice` by doing the following:

1. Assign types to every subexpression:

   ```
   x :: t0     f :: t1     f x :: t2     f (f x) :: t3
   ```

   This implies `twice :: t1 -> t0 -> t3`.

2. Set up constraints. In this case, we have

```
t1 = t0 -> t2     from f x
t1 = t2 -> t3     from f (f x)
```

3. Resolve the constraints. We have `t0 -> t2 = t2 -> t3`, so `t0 = t2` and `t2 = t3`. This gives `twice :: (t0 -> t0) -> t0 -> t0`.

25 **Question**

How do we formalize this process?

3.3.1 Constraints

First, we will formalize what a contraint actually is, talk about how to generate them.

Consider the language of constraints:

$$C \coloneqq \tau_1 = \tau_2 \mid C \vee C \mid \exists \tau . C$$

We can build constraints from the typing rules as we build the type checking tree. The base cases are

$$[\![\Gamma \vdash x : \tau]\!] = (\Gamma(x) = \tau) \qquad [\![\Gamma \vdash N : \tau]\!] = (\mathrm{int} = \tau).$$

The inductive cases are

$$\begin{aligned}
[\![\Gamma \vdash e_1 e_2 : \tau]\!] &= \exists a([\![\Gamma \vdash e_1 : a \longrightarrow \tau]\!] \wedge [\![\Gamma \vdash e_2 : a]\!]) \\
[\![\Gamma \vdash \lambda x.e : \tau]\!] &= \exists a_1 a_2.([\![\Gamma, x : a_1 \vdash e : a_2]\!] \wedge \tau = a_1 \longrightarrow a_2) \\
[\![\Gamma \vdash e_1 + e_2 : \tau]\!] &= \exists a_1 a_2.([\![\Gamma \vdash e_1 : \mathrm{int}]\!] \wedge [\![\Gamma \vdash e_2 : \mathrm{int}]\!] \wedge \tau = \mathrm{int})
\end{aligned}$$

26 **Example**

Suppose we want to determine the type of

$$\lambda x.x + 1.$$

We can recursively evaluate the type judgement

$$\begin{aligned}
[\![\Gamma \vdash (\lambda x\,.\,x + 1) 2 : \tau]\!] &= \exists a_1 .\, [\![\Gamma \vdash (\lambda x\,.\,x+1) : a_1 \longrightarrow \tau]\!] \wedge [\![\Gamma \vdash 2 : a_1]\!] \\
&= \exists a_1 .\, (\exists a_2 a_3 .\, [\![\Gamma, x : a_2 \vdash x + 1 : a_3]\!] \wedge (a_1 \longrightarrow \tau = a_2 \longrightarrow a_3)) \wedge (a_1 = \mathrm{int}) \\
&= \exists a_1 .\, (\exists a_2 a_3 .\, [\![\Gamma, x : a_2 \vdash x : \mathrm{int}]\!] \wedge [\![\Gamma, x : a_2 \vdash 1 : \mathrm{int}]\!] \wedge (a_3 = \mathrm{int}) \wedge (a_1 \longrightarrow \tau = a_2 \longrightarrow a_3)) \wedge (a_1 = \mathrm{int}) \\
&= \exists a_1 .\, (\exists a_2 a_3 .\, (a_2 = \mathrm{int}) \wedge (\mathrm{int} = \mathrm{int}) \wedge (a_3 = \mathrm{int}) \wedge (a_1 \longrightarrow \tau = a_2 \longrightarrow a_3)) \wedge (a_1 = \mathrm{int}).
\end{aligned}$$

and this gives us an expression in the language of constraints.

Now that we have constraints, we want to determine if they are equal, and whether they can be satisfied with an appropriate choice of type variables.

To answer the first question, to determine when two types τ_a and τ_b are equal, we use **structural equality**. In particular, $\tau_a = \tau_b$ when their constructors are the same, and their contents are the same. For instance, if $\tau_a = \tau_1 \longrightarrow \tau_2$ and $\tau_b = \tau_3 \longrightarrow \tau_4$, then $\tau_a = \tau_b \iff \tau_1 = \tau_3 \wedge \tau_2 = \tau_4$.

3.3.2 Unification

The answer to the question of how to solve constraints is more complicated. We define **unification** to be the process of determining whether two types can be made equal by selecting appropriate substitutions for type variables.

First we will formally define types and type substitution. The language of **types** is

$$\begin{aligned} \tau &:= \iota && \text{base types } (\texttt{Int}, \texttt{Bool}, \ldots) \\ &\mid t && \text{type variables} \\ &\mid \tau_1 \to \tau_2 && \text{function types.} \end{aligned}$$

A **type substitution** is a map

$$S\colon \{\text{type variables}\} \to \{\text{types}\}$$
$$S = [\tau_1/t_1, \tau_2/t_2, \ldots, \tau_n/t_n],$$

and when $\tau' = S\tau$, we say τ' is a **substitution instance** of τ.

27 **Example**

If we have $S_1 = [(t \to \text{bool})/t_1]$ and $S_2 = [\text{int}/t]$, then

$$\begin{aligned} S_2 S_1 (t_1 \to t_1) &= S_2((t \to \text{bool}) \to (t \to \text{bool})) \\ &= (\text{int} \to \text{bool}) \to (\text{int} \to \text{bool}). \end{aligned}$$

To determine whether two types are able to be unified, we use the following algorithm:

```
unify($\tau_1$, $\tau_2$) =
  case ($\tau_1$, $\tau_2$) of
    ($\tau_1, t_2$) -> [$\tau_1/t_2$] given $t_2 \notin FV(\tau_1)$
    ($t_1, \tau_2$) -> [$\tau_2/t_1$] given $t_1 \notin FV(\tau_2)$
    ($\iota_1, \iota_2$) -> if (eq $\iota_1$ $\iota_2$) then []
                             else Fail
    ($\tau_{11} \to \tau_{12}, \tau_{21} \to \tau_{22}$) -> let $S_1$ = unify($\tau_{11}, \tau_{21}$)
                                 $S_2$ = unify($S_1\tau_{12}, S_1\tau_{22}$)
                             in $S_2 S_1$
    otherwise -> Fail
```

3.3.3 Putting it together

Now that we have formalized both constraints and unification, we can combine them into a type inference algorithm. In the examples above (e.g. example 23), we collected all of the constraints, and then at the end we solved the system. However in practice, this can be inefficient if your program is large, so we often solve the constraints incrementally, and build the substitution map incrementally.

Let us come up with an inference algorithm W such that $W(\text{TE}, e) = (S, \tau)$, where $S(\text{TE}) \vdash e : \tau$. Here, we are writing TE instead of Γ for convenience. Intuitively, the type environment TE stores the most general type of each identifier while the substitution S stores the changes in type variables.

Naturally, we will do casework on e. We have

$$[\![\Gamma \vdash N : \tau]\!] = (\text{int} = \tau) \qquad [\![\Gamma \vdash x : \tau]\!] = (\Gamma(x) = \tau)$$
$$[\![\Gamma \vdash e_1 e_2 : \tau]\!] = \exists a([\![\Gamma \vdash e_1 : a \to \tau]\!] \land [\![\Gamma \vdash e_2 : a]\!])$$
$$[\![\Gamma \vdash \lambda x.e : \tau]\!] = \exists a_1 a_2.([\![\Gamma, x : a_1 \vdash e : a_2]\!] \land \tau = a_1 \to a_2).$$

This gives us the algorithm

```
w(te, e) = case e of
  c    -> ({}, Int)
  x    -> if (x ∈ te) then ({}, te[x])
                     else Fail
  λx.e -> let ($S_1$, $\tau_1$) = w(te ∪ {x:u}, e)
           in ($S_1$, $S_1$(u) -> $\tau_1$)
  $e_1 e_2$  -> let ($S_1$, $\tau_1$) = w(te, $e_1$)
               ($S_2$, $\tau_2$) = w($S_1$(te), $e_2$)
               $S_3$ = unify($S_2$($\tau_1$), $\tau_2$ -> u)
           in ($S_3 S_2 S_1$, $S_3$(u))
```

28 **Example**

We can even run this algorithm by hand: Let's say we are trying to compute $W(\varnothing, (\lambda f\,.\,f 5)(\lambda x\,.\,x))$.

Computation (Application)

Let $(S_1, \tau_1) = W(\varnothing, \lambda f\,.\,f 5) = (S_1', S_1'(u_1) \to \tau_1') = ([\mathsf{int} \to u_2/u_1], (\mathsf{int} \to u_2) \to u_2)$, with subvariables obtained from the following subcomputation:

Computation (Lambda)

Let $(S_1', \tau_1') = W(\varnothing \cup \{f : u_1\}, f 5) = (S_3'' S_2'' S_1'', S_3'' u_2) = ([\mathsf{int} \to u_2/u_1], u_2)$ with subvariables obtained from the subcomputation:

Computation (Application)

Let $(S_1'', \tau_1'') = W(\{f : u_1\}, f) = (\varnothing, u_1)$.

Let $(S_2'', \tau_2'') = W(\{f : u_1\}, 5) = (\varnothing, \mathsf{int})$.

Let $S_3'' = \mathsf{unify}(S_2''(u_1), \mathsf{int} \to u_2)$.

$$= \mathsf{unify}(u_1, \mathsf{int} \to u_2)$$

$$= [\mathsf{int} \to u_2/u_1]$$

Let $(S_2, \tau_2) = W(S_1'\varnothing, \lambda x\,.\,x) = (S_1', S_1'(u_3) \to \tau_1') = (\varnothing, u_3 \to u_3)$, with subvariables obtained from the subcomputation:

Computation (Lambda)

Let $(S_1', \tau_1') = W(\varnothing \cup \{x : u_3\}, x) = (\varnothing, u_3)$.

Let $S_3 = \mathsf{unify}(S_2(\tau_1), \tau_2 \to u_4) = \mathsf{unify}((\mathsf{int} \to u_2) \to u_2, (u_3 \to u_3) \to u_4) = S_2' S_1' = [\mathsf{int}/u_2, \mathsf{int}/u_3, \mathsf{int}/u_4]$, with subvariables obtained from the subcomputation:

Computation (Unify)

Let $S_1' = \mathsf{unify}(\mathsf{int} \to u_2, u_3 \to u_3) = S_2'' S_1'' = [\mathsf{int}/u_2, \mathsf{int}/u_3]$, with subvariables obtained from:

Computation (Unify)

Let $S_1'' = \mathsf{unify}(\mathsf{int}, u_3) = [\mathsf{int}/u_3]$.

Let $S_2'' = \mathsf{unify}(S_1'' u_2, S_1'' u_3) = \mathsf{unify}(u_2, \mathsf{int}) = [\mathsf{int}/u_2]$.

Let $S_2' = \mathsf{unify}(S_1' u_2, S_1' u_4) = \mathsf{unify}(\mathsf{int}, u_4) = [\mathsf{int}/u_4]$

> This gives $W(\varnothing, (\lambda f\,.\,f 5)(\lambda x\,.\,x)) = (S_3 S_2 S_1, S_3 u_4) = ([\mathsf{int}/u_1, \mathsf{int}/u_2, \mathsf{int}/u_3, \mathsf{int}/u_4], \mathsf{int})$.

We can include the let ... in ... construct with the typing rule

$$\frac{\Gamma, x:\tau' \vdash e_1:\tau' \quad \Gamma, x:\tau' \vdash e_2:\tau}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau}$$

This gives the constraint rule

$$[\![\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2:\tau]\!] = \exists \tau'.[\![\Gamma, x:\tau' \vdash e_1:\tau']\!] \wedge [\![\Gamma, x:\tau' \vdash e_2:\tau]\!].$$

This gives the type inference case:

```
w(te, e) = case e of
  (let $x = e_1$ in $e_2$) -> let ($S_1$, $\tau_1$) = w(te ∪ {x:u}, $e_1$)
                              $S_2$       = unify ($S_1$(u), $\tau_1$)
                              ($S_3$, $\tau_2$) = w($S_2 S_1$(te) ∪ {x:$\tau_1$}, $e_2$)
                           in ($S_3 S_2 S_1$, $\tau_2$)
```

3.4 Polymorphism

Under our current rules, let id = $\lambda x\,.\,x$ in (... (id true) ... (id 1) ...) is not well typed, because id cannot be both int $\longrightarrow$ int and bool $\longrightarrow$ bool. There are a few ways we can try to fix this:

Explicit polymorphism We introduce a type-level lambda expression, formalized with the typing rules

$$\frac{\Gamma \vdash e:\tau}{\Gamma \vdash \Lambda t.\, e:\forall t.\tau} \qquad \frac{\Gamma \vdash e:\forall t.\tau'}{\Gamma \vdash e[\tau]:\tau'[\tau/t]}$$

For instance, we have id = $\Lambda T\,.\,\lambda x:T\,.\,x$, and when we want to use it, we write id[int]5. However, this is annoying because we have to explicitly pass in the type whenever we use it.

Impredicative polymorphism We allow self referential types, formalized by the language of types and expressions:

$$\tau \coloneqq b \mid \tau_1 \longrightarrow \tau_2 \mid T \mid \forall T.\tau$$
$$e \coloneqq x \mid \lambda x:\tau\,.\,e \mid e_1 e_2 \mid \Lambda T.\,e \mid e[\tau]$$

Note that here, we can define a type variable $t = \forall x.x \longrightarrow x$, where x can refer to t. This is powerful, but we cannot express recursion. Moreover, type inference is undecidable.

Predicative polymorphism We restrict impredicativity to get the language of types and expressions:

$$\tau \coloneqq b \mid \tau_1 \longrightarrow \tau_2 \mid T$$
$$\sigma \coloneqq \tau \mid \forall T.\,\tau \mid \sigma_1 \longrightarrow \sigma_2$$
$$e \coloneqq x \mid \lambda\,x:\sigma.\,e \mid e_1 e_2 \mid \Lambda T.\,e \mid e[\tau]$$

This is still very powerful, but we cannot instantiate a polymorphic type. Moreover, type inference is still undecidable.

Prenex predicative polymorphism We restrict so that all quantifiers are at the beginning. This gives us the grammar

$$\tau \coloneqq b \mid \tau_1 \longrightarrow \tau_2 \mid T$$
$$\sigma \coloneqq \tau \mid \forall T.\,\tau$$
$$e \coloneqq x \mid \lambda\,x:\tau.\,e \mid e_1 e_2 \mid \Lambda T.\,e \mid e[\tau]$$

This makes our type inference decidable, but now we cannot pass polymorphic functions as function arguments.

The compromise we will settle on is called let polymorphism. In particular, we introduce the expression type of the form let $x = e_1$ in e_2, which is equivalent to $(\lambda x\,.\,e_2)e_1$, except x can be polymorphic. This gives a good amount of expressiveness, while maintaining decidability. This is called the Hindley-Milner type system.

3.5 Type inference with polymorphism

If we have let $(\mathsf{id} = \lambda x.x)$ in $\ldots (\mathsf{id}\,1) \ldots (\mathsf{id}\,\mathsf{true}) \ldots$, we can come up with multiple constraints

$$\mathsf{id} : t_1 \to t_1 \qquad \mathsf{id} : \mathsf{int} \to t_1 \qquad \mathsf{id} : \mathsf{bool} \to t_1,$$

which do not unify. The solution to this is to generalize the type to

$$\mathsf{id} : \forall t_1.\, t_1 \to t_1.$$

Then, we can instantiate the generalized type variable differently in different contexts.

$$\mathsf{id}_1 : \mathsf{int} \to \mathsf{int} \qquad \mathsf{id}_2 : \mathsf{bool} \to \mathsf{bool},$$

3.5.1 Hindley-Milner types

Consider the language

$E \coloneqq c$	constant
$\mid x$	variable
$\mid \lambda x.E$	abstraction
$\mid (E_1 E_2)$	application
$\mid$ let $x = E_1$ in E_2	let-block

Note that there are no explicit types in the language! All types are inferred according to the **Hindley-Milner type inference algorithm**.

The types are

$\tau \coloneqq \iota$	base types
$\mid t$	type variables
$\mid \tau_1 \to \tau_2$	function types,

and there are type schemes

$$\sigma \coloneqq \tau \mid \forall t.\, \sigma.$$

Note that all of the $\forall$s occur at the beginning of a type scheme, so a type τ cannot contain a type scheme σ.

A **type environment** is a mapping from identifiers to type schemes.

A type scheme $\sigma = \forall t_1 \ldots t_n.\, \tau$ can be **instantiated** into a type τ' by substituting types for the bound variables $t_1, \ldots, t_n$ of σ, i.e. $\tau' = S\tau$ for some substitution S with $\mathrm{Dom}(S) \subseteq \mathrm{BV}(\sigma)$. We then say that τ' is an **instance** of σ, and write $\sigma > \tau'$. We call τ' a **generic instance** of σ if S maps $t_1, \ldots, t_n$ to new type variables.

29 **Example**

If we have $\sigma = \forall t_1.\, t_1 \to t_1$, then

- $t_3 \to t_2$ is a generic instance of σ, and
- $\mathsf{int} \to t_2$ is a non-generic instance of σ.

We can go in the other direction. In particular, we can generalize types with free variables:

$$\mathrm{Gen}(\mathrm{TE}, \tau) = \forall t_1 \ldots t_n.\, \tau \qquad \text{where} \qquad \{t_1, \ldots, t_n\} = \mathrm{FV}(\tau) \setminus \mathrm{FV}(\mathrm{TE}).$$

This captures the notion of *new* type variables in τ, and introduces polymorphism. In particular, we quantify the type variables that are free in τ but not in the type environment TE.

Overall, this gives us the type inference rules:

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1 e_2) : \tau'}\text{(App)} \qquad \frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1 e_2) : \tau'}\text{(Abs)} \qquad \frac{x : \sigma \in \Gamma \quad \sigma \geq \tau}{\Gamma \vdash x : \tau}\text{(Var)}$$

$$\frac{\text{typeof}(c) \geq \tau}{\Gamma \vdash c : \tau}\text{(Const)} \qquad \frac{\Gamma;\{x : \tau\} \vdash e_1 : \tau \quad \Gamma;\{x : \text{Gen}(\Gamma,\tau)\} \vdash e_2 : \tau'}{\Gamma \vdash (\text{let}\, x = e_1 \,\text{in}\, e_2) : \tau'}\text{(Let)}$$

The important rules for polymorphism is in (Var) and (Let). In particular, the type of a variable is generalized in the let expression, and then the type scheme can be specialize to different types with the Var rule.

3.5.2 HM Inference algorithm

Our type inference algorithm changes a bit to account for polymorphism, and we get the following:

```
w (te, e) = case e of
  c    -> ({}, typeof(c))
  x    -> if (x ∉ te) then Fail
                      else let $\forall t_1 \dots t_n . \tau$ = TE[x]
                            in ({}, $[u_i/t_i]\tau$)
  $\lambda$x.e -> let ($S_1$, $\tau_1$) = w(te ∪ {x:u}, e)
           in ($S_1$, $S_1$(u) -> $\tau_1$)
  $e_1 e_2$  -> let ($S_1$, $\tau_1$) = w(te, $e_1$)
              ($S_2$, $\tau_2$) = w($S_1$(te), $e_2$)
              $S_3$ = unify ($S_2(\tau_1)$, $\tau_2$ -> u)
           in ($S_3 S_2 S_1$, $S_3$(u))
  (let $x = e_1$ in $e_2$) -> let ($S_1$, $\tau_1$) = w(te ∪ {x:u}, $e_1$)
                                $S_2$      = unify($S_1$(u), $\tau_1$)
                                $\sigma$      = gen($S_2 S_1$(TE), $S_2(\tau_1)$)
                                ($S_3$, $\tau_2$) = w($S_2 S_1$(te) ∪ {x:$\sigma$}, $e_2$)
                             in ($S_3 S_2 S_1$, $\tau_2$)
```

The only differences are in the variable case and the let case, where we specialize and generalize respectively.

Some important observations about HM type inference:

- We do not generalize type variables that are used elsewhere.
- Let is the only way of defining polymorphic constructs.
- We generalize the types of the let bindings *after* processing their definitions.

HM type inference is sound and generates the most general types of expressions. Therefore, an inferred type is verifiable and any verifiable type is inferred. This complexity of HM type inference is PSPACE-hard, due to the nested let blocks.

30 **Example**

Consider

```
let twice f x = f (f x)
 in twice twice succ 4
```

and

```
let twice f x = f (f x)
    foo g = (g g succ) 4
 in foo twice
```

In the first example, `twice` has a generic type, while in the second example, `g` has a non-generic type. This means that `foo` in the second example is not type correct.

Some extensions that we can think about are:

Type declarations for sanity checks, and for relaxed restrictions.

Incremental type checking if the whole program is not given at the same time, we can still soundly infer the types.

Typing references to mutable objects HM is unsound for a language with refs (mutable locations).

Overloading resolution

3.5.3 Type classes

Hindley-Milner gives us generic functions that make no assumptions about the type, like

```
const :: forall a b. a -> b -> a
const x y = x

const :: forall a b. (a -> b) -> a -> b
apply f x = f x
```

However, what if we *do* need to make assumptions about the types? For instance, if we have the list data type (written two ways for clarity):

```
data List x = Nil | Cons x (List x)
data [x] = [] | x:[x]
```

we can then write a sum function (with an accumulator)

```
sum n [] = n
sum n (x:xs) = sum (n + x) xs
```

The type of `sum` cannot be `a -> [a] -> a` because we have a restriction on the type (namely that we need to be able to add `a`s).

The solution to this is to pass in the notion of plus:

```
sum plus n [] = n
sum plus n (x:xs) = sum (plus n x) xs
```

Now we have a polymorphic type for sum `sum :: (a -> a -> a) -> a -> [a] -> a` When we want to call `sum`, we have to pass in the appropriate function representing addition.

If we want to write other functions like `sum`, we also need to use arithmetic operations. We can generalize if we include a set of functions $\{+, -, \times, \div\}$. In particular, we can create a "class" type:

```
data (Num a) = Num {
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  (*) :: a -> a -> a
  (/) :: a -> a -> a
  fromInteger :: Integer -> a
}
```

Then we can use this group of functions to represent restrictions on type:

```
sum       :: Num a -> a -> [a] -> a
matrixMul :: Num a -> Mat a -> Mat a -> Mat a
dft       :: Num a -> Vec a -> Vec a -> Vec a
```

Then, all of the numeric aspects of the type can be isolated to the `Num` type. For each type that has properties of a number, we can build a `Num` instance and functions that need to use properties of a number will take it as an argument.

We can use the same idea to encompass other ideas, like equality, ordering, and conversion to/from `String`.

However, this is slightly annoying when we have to manually pass in `Num` objects. We have to keep track of the `Num` objects, and make sure that our `Num` instances are consistent with each other (e.g. `Num a` and `Num (Mat a)` are related to each other in a reasonable way.) The way we solve this is we move class objects into a type class.

31 Definition

A **type class** is a group of related functions that are overloaded over types.

We can define the typeclass

```
class Num a where
  (+), (-), (*) :: a -> a -> a
  negate        :: a -> a
  ...
```

and then define instances:

```
instance Num Int where
  x + y = integer_add x y
  x - y = integer_sub x y
  ...

instance Num Float where
  ...
```

We can also define hierarchical data types:

```
class Eq a where
  (==), (/=) :: a -> a -> Bool

class (Eq a) => (Ord a) where
  (<), (<=), (>=), (>) :: a -> a -> Bool
  max, min             :: a -> a -> a
```

Here, `Eq` is a superclass of `Ord`, so every instance of `Ord` is also an instance of `Eq`.

Each type class corresponds to one "concept" and class constraints gives a natural hierarchy on type classes. Often, a type class has laws that are associated with it. For instance, in `Num`, the function `(+)` should be associative and commutative. These aren't checked by the compiler, so the programmer has to ensure that every instance follows these laws.

If we view type classes as predicates, the type checked can deal with all of the passing that we had to manually do before. Note that while this looks like a new feature, the compiler desugars this into pure λ-calculus.

3.5.4 Subtyping

When working with lambda calculus, we may notice that it can be annoyingly rigid. For instance, $(\lambda r:\{x:\mathrm{int}\}\,.\,r[x])\{x=0,y=1\}$ seems well-behaved, but is not well typed, since the argument has type $\{x:\mathrm{int}, y:\mathrm{int}\}$ while the function accepts type $\{x:\mathrm{int}\}$. The goal of subtyping is to refine the typing rules so that they accept terms like the one above.

If we have some operator $[\![\cdot]\!]$ where $[\![\tau]\!]$ represents a set that represents τ, we have e.g. $[\![\mathrm{int}]\!]=\mathbb{Z}$ and $[\![\mathrm{float}]\!]=\mathbb{R}$. Then we may try to define $\tau_1 \le \tau_2$ if and only if $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$. A more useful rule is that $\tau_1 \le \tau_2$ if, we can use a τ_1 everywhere we can use a τ_2. For instance, any integer N can be treated as a real with no loss of meaning.

The typing rule for this, called **subsumption**, is

$$\frac{\Gamma \vdash e:\tau' \quad \tau' \le \tau}{\Gamma \vdash e:\tau}$$

To determine the pairs of subtypes, we can provide a list to the compiler. However, there is a better way to come up with the subtyping rules that does not require the user to explicitly declare them. A **structural subtyping** system includes rules that can analyze the structure of types to deduce subtyping rules, rather than just using the rules that the user explicitly defined.

Product types

Consider the product type $\tau_1 \times \tau_2$.

$$\frac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2'}{\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'}$$

We say that the product type constructor is **covariant**.

Record types

A more general type is a record type $\{a_1 : \tau_1, \ldots, a_N : \tau_N\}$, where for every i, there is a field a_i of type τ_i.

Then there are two types of subtyping:

$$\{a : \mathsf{Int}, b : \mathsf{Alice}\} \leq \{a : \mathsf{Number}, b : \mathsf{Person}\},$$

called depth subtyping, and

$$\{a : \mathsf{Number}, b : \mathsf{Person}, c : \mathsf{Tool}\} \leq \{a : \mathsf{Number}, b : \mathsf{Person}\},$$

called width subtyping.

We can formalize these into the typing rules:

$$\frac{\forall i. \tau_i \leq \tau_i'}{\{a_i : \tau_i\} \leq \{a_i : \tau_i'\}}\text{depth} \qquad \frac{\forall j. \exists i. a_i = a_j' \land \tau_i \leq \tau_j'}{\{a_i : \tau_i\} \leq \{a_j : \tau_j'\}}\text{width}$$

Function types

For function types $\tau_1 \to \tau_2$, we could optimistically guess

$$\frac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

However, we cannot use a $\mathsf{int} \to \mathsf{int}$ anywhere we can use a $\mathsf{float} \to \mathsf{int}$, as our subtype function cannot take all of the inputs that the original function could. The fix to this is to switch the subtyping direction of the input type.

$$\frac{\tau_1 \geq \tau_1' \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

We say that the function type is **contravariant** in the domain and covariant in the codomain.

Array and list types

For for *mutable* arrays $\tau[]$, the operations we can perform on them are both reading or writing to some entry of the array. There are two possible options for a typing rule if we were to have one, a covariant and contravariant rule, which are respectively:

$$\frac{\tau_1 \leq \tau_2}{\tau_1[] \leq \tau_2[]} \qquad \frac{\tau_1 \geq \tau_2}{\tau_1[] \leq \tau_2[]}$$

In the covariant case, note that we cannot write a τ_2 into a $\tau_1[]$, and in the contravariant case, we cannot read a τ_2 from a $\tau_1[]$. Therefore, arrays cannot be subtyped. For this reason, we call arrays **invariant**.

For *immutable* lists $[\tau]$, we have

$$\frac{\tau_1 \geq \tau_2}{[\tau_1] \leq [\tau_2]}$$

In general, most “data structures” are covariant.

3.6 Monads

These notes contain a different explanation of monads than what was presented in lecture. I personally believe that this explanation is a better introduction. Explanations inspired by: `https://mightybyte.github.io/monad-challenges/`

3.6.1 Maybe monad

Suppose we have some functions that may fail: Recall the `Maybe` a type in Haskell:

```haskell
data Maybe a = Nothing | Just a
```

Suppose we have the functions

```haskell
tailMaybe :: [a] -> Maybe [a]
tailMaybe [] = Nothing
tailMaybe (x:xs) = Just xs

maximumMaybe :: Ord a => [a] -> Maybe a
maximumMaybe [] = Nothing
maximumMaybe xs = Just $ foldl1 max xs

reciprocalMaybe :: (Eq a, Fractional a) => a -> Maybe a
reciprocalMaybe 0 = Nothing
reciprocalMaybe n = Just $ 1/n
```

and we want to compose these. The standard way to do this is to case match on each result like so:

```haskell
f :: (Ord a, Fractional a) => [a] -> Maybe a
f xs =
  case tailMaybe xs of
    Nothing -> Nothing
    Just ys -> case maximumMaybe ys of
                 Nothing -> Nothing
                 Just m  -> reciprocalMaybe m
```

However, this gets annoying once we have a lot of functions. We can abstract over this: We define a function to facilitate this composition

```haskell
bind :: Maybe a -> (a -> Maybe b) -> Maybe b
bind Nothing _ = Nothing
bind (Just a) f = f a
```

Then, we can rewrite our function

```haskell
f :: (Ord a, Fractional a) => [a] -> Maybe a
f xs = bind (bind (tailMaybe xs) maximumMaybe) reciprocalMaybe
```

If we define another function to “inject” a value into the `Maybe` context, we can use `bind` with infix notation:

```haskell
pure x = Just x    :: a -> Maybe a
(>>=) = bind       :: Maybe a -> (a -> Maybe b) -> Maybe b

f :: (Ord a, Fractional a) => [a] -> Maybe a
f xs = pure xs >>= tailMaybe >>= maximumMaybe >>= reciprocalMaybe
```

3.6.2 Writer monad

Suppose we have some functions

```
plusOne x = x + 1
square x = x * x
```

that we wanted to augment by letting them append to a log. One idea we may think of is to also return a string that we can look at:

```
plusOne' x = (x + 1, "added one")
square' x = (x * x, "squared")
```

However, this breaks compositionality, because `plusOne' (square' x)` does not typecheck like `plusOne (square x)` does. In particular, composition becomes really annoying:

```
let (y, s) = square' x
    (z, t) = plusOne' y
 in (z, s ++ t)
```

We can abstract over this like we did for `Maybe`. We define a function to facilitate this composition:

```
bind :: (a, String) -> (a -> (b, String)) -> (b, String)
bind (x, y) f = let (u, v) = f x in (u, y ++ v)
(>>=) = bind
```

Then, we can write `square' x >>= plusOne'`. Like before, we can also define

```
pure :: Int -> (Int, String)
pure x = (x, "")
```

3.6.3 Generalized monads

This idea of "passing along computation" seems pretty useful. We can make a typeclass for this!

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  pure :: a -> m a
```

This typeclass actually appears in many other places! Some instances include `Either a b`, `[a]`, and `((->) a b)`.

Monad laws

Note that the type `Monad m => a -> m b` is almost like a function, but it returns something in the monadic context. However, since we have `(>>=)`, we can collapse two layers of the monad into one layer with a function

```
join :: Monad m => m (m a) -> m a
join mma = do ma <- mma; ma
join mma = mma >>= id
```

We can also define a function that is *almost* function composition, but instead of functions, we work with values of type `Monad m => a -> m b`. These are called Kleisli arrows.

```
(.)   ::             (b ->   c) -> (a ->   b) -> (a ->   c)
(<=<) :: Monad m => (b -> m c) -> (a -> m b) -> (a -> m c)
(f <=< g) a = g a >>= f
```

Then, we have the monad laws:

```haskell
-- Left identity
pure <=< f = f
-- Right identity
f <=< pure = f
-- Associativity
(f <=< g) <=< h = f <=< (g <=< h)
```

A more confusing way to write these with the standard function `(>>=)` are:

```haskell
-- Left identity
pure a >>= f = f a
-- Right identity
m >>= pure = m
-- Associativity
m >>= (\x -> f x >>= g) = (m >>= f) >>= g
```

Syntactic sugar (do notation)

In Haskell, there is syntactic sugar for monads, which allows us to think about them more easily.

```haskell
do e                         =>  e
do p <- e ; statements       =>  e >>= \p -> do statements
do e ; statements            =>  e >>= \_ -> do statements
do let p = e ; statements    =>  let p = e in do statements
```

Here, the `<-` can be intuitively thought of as "extracting" the value from the monad, but eventually we must always have a result still within the monad.

32 **Example**

To rewrite the examples from before, we can rewrite

```haskell
f :: (Ord a, Fractional a) => [a] -> Maybe a
f xs = pure xs >>= tailMaybe >>= maximumMaybe >>= reciprocalMaybe

f xs = do
  t <- tailMaybe xs
  m <- maximumMaybe t
  reciprocalMaybe m
```

3.6.4 IO monad

So far, we've only been writing pure functions. But in order for our programs to be useful, we want to be able to perform side effects, like printing to console. The way we do this is with a monad called `IO`. All side effects will be isolated in this monad, so that we still get the benefits of pure functions elsewhere.

Let's say that we have a function `wc :: String -> (Int, Int, Int)`, and we want to write a program that takes in a file and prints the result to `stdout`. We expect there to be types and functions to help us do IO:

```haskell
type Filepath = String
data IOMode = ReadMode | WriteMode | ...
data Handle = ... -- implemented as built-in type

openFile :: FilePath -> IOMode -> Handle
hClose   :: Handle -> ()
hIsEOF   :: Handle -> Bool
hGetChar :: Handle -> Char
```

However, these are not pure, so when we try to write code, there is no way to model or control side effects.

```haskell
getFileContents :: String -> String
getFileContents filename =
  let h = openFile filename ReadMode
      readFromHandle handle =
        if hIsEOF handle
          then ""
          else hGetChar handle : readFromHandle handle
        --                            [???]
      content = readFromHandle h
      () = hClose h
   in content
```

The solution to this is wrap things with an IO monad, to force there to be a single sequence of IO operations, eliminating nondeterminism issues.

```haskell
openFile :: FilePath -> IOMode -> IO Handle
hClose   :: Handle -> IO ()
hIsEOF   :: Handle -> IO Bool
hGetChar :: Handle -> IO Char
```

Then, our function becomes

```haskell
readFromHandle :: Handle -> IO String
readFromHandle h = do
  isEOF <- hIsEOF h
  if isEOF
    then pure ""
    else do x <- hGetChar h
            rest <- readFromHandle h
            -- base case should close handle
            pure $ x : rest

getFileContents :: String -> IO String
getFileContents = do
  h <- openFile filename ReadMode
  content <- readFromHandle h
  hClose h
  return content -- equivalently, `pure content`
```

4 Types for imperative programs

Recall that when evaluating lambda calculus, there is no state to keep track of. We had the following inductive definition

$$\frac{}{x \longrightarrow x} \qquad \frac{e \longrightarrow e'}{\lambda x \,.\, e \longrightarrow \lambda x \,.\, e'} \qquad \frac{e_1 \longrightarrow \lambda x \,.\, e_1' \quad e_1'[e_2/x] \longrightarrow e_3}{e_1 e_2 \longrightarrow e_3}$$

We want to do the same thing for imperative programs. Since we now have state, the configuration must now include it, rather than just the types of each variable.

We will be using the language IMP, which is the imperative equivalent to lambda calculus in the sense that it is the simplest imperative language that we can add things to.

4.1 Introduction

The grammar of IMP is:

$$e \coloneqq n \mid x \mid e_1 + e_2 \mid e_1 == e_2 \mid \mathsf{true} \mid \mathsf{false}$$

$$c \coloneqq (x \coloneqq e) \mid c_1; c_2 \mid \mathsf{if}\, e \,\mathsf{then}\, c_1 \,\mathsf{else}\, c_2 \mid \mathsf{while}\, e \,\mathsf{do}\, c \mid \mathsf{skip}$$

where e represents expressions and c represents commands.

4.1.1 Big step semantics

Since there is a distinction between expressions and commands, we now need two types of judgements:

- expressions that result in a value $\langle e, \sigma \rangle \longrightarrow n$
- commands that change the state $\langle c, \sigma \rangle \longrightarrow \sigma'$

Here, the state σ is a mapping from variables to values.

The rules for expressions are similar to what we had before:

$$\frac{}{\langle N, \sigma \rangle \longrightarrow n} \qquad \frac{\langle e_1, \sigma \rangle \longrightarrow n_1 \quad \langle e_2, \sigma \rangle \longrightarrow n_2 \quad n = n_1 + n_2}{\langle e_2 + e_2, \sigma \rangle \longrightarrow n} \qquad \frac{}{\langle x, \sigma \rangle \longrightarrow \sigma(x)}$$

where the third rule above is to read from variables. For commands, the state is changed:

$$\frac{\langle e, \sigma \rangle \longrightarrow e'}{\langle X \coloneqq e, \sigma \rangle \longrightarrow \sigma[X \rightarrow e']} \qquad \frac{\langle c_1, \sigma \rangle \longrightarrow \sigma'' \quad \langle c_2, \sigma'' \rangle \longrightarrow \sigma'}{\langle c_1; c_2, \sigma \rangle \longrightarrow \sigma'}$$

$$\frac{\langle e, \sigma \rangle \longrightarrow \mathsf{true} \quad \langle c_1, \sigma \rangle \longrightarrow \sigma'}{\langle \mathsf{if}\, e \,\mathsf{then}\, c_1 \,\mathsf{else}\, c_2, \sigma \rangle \longrightarrow \sigma'} \qquad \frac{\langle e, \sigma \rangle \longrightarrow \mathsf{false} \quad \langle c_2, \sigma \rangle \longrightarrow \sigma'}{\langle \mathsf{if}\, e \,\mathsf{then}\, c_1 \,\mathsf{else}\, c_2, \sigma \rangle \longrightarrow \sigma'}$$

$$\frac{\langle e, \sigma \rangle \longrightarrow \mathsf{false} \quad \langle c_2, \sigma \rangle \longrightarrow \sigma}{\langle \mathsf{while}\, e \,\mathsf{do}\, c, \sigma \rangle \longrightarrow \sigma'} \qquad \frac{\langle e, \sigma \rangle \longrightarrow \mathsf{true} \quad \langle c, \sigma \rangle \longrightarrow \sigma'' \quad \langle \mathsf{while}\, e \,\mathsf{do}\, c, \sigma'' \rangle \longrightarrow \sigma'}{\langle \mathsf{while}\, e \,\mathsf{do}\, c, \sigma \rangle \longrightarrow \sigma'}$$

For the $\mathsf{while}\,\mathsf{true}\,\mathsf{do}\, c$ case, note that the last two premises is equivalent to the sequential composition rule. Therefore, the last rule is equivalent to

$$\frac{\langle e, \sigma \rangle \longrightarrow \mathsf{true} \quad \langle c; \mathsf{while}\, e \,\mathsf{do}\, c, \sigma \rangle \longrightarrow \sigma'}{\langle \mathsf{while}\, e \,\mathsf{do}\, c, \sigma \rangle \longrightarrow \sigma'}$$

4.1.2 Small step semantics

Recall that a redex is some rule $r \longrightarrow v$, and a context is an expression with some hole $C = \bullet \mid \cdots$. Since we now have state, our local reductions must be of the form $\langle r, \sigma \rangle \longrightarrow \langle v, \sigma' \rangle$. Our reductions are fairly intuitive:

- $\langle x, \sigma[x = n] \rangle \longrightarrow \langle n, \sigma[x = n] \rangle$
- $\langle n_1 + n_2, \sigma \rangle \longrightarrow \langle n, \sigma \rangle$ where $n = n_1 + n_2$.
- $\langle x := n, \sigma \rangle \longrightarrow \langle \mathsf{skip}, \sigma[x \leftarrow n] \rangle$
- $\langle \mathsf{skip}; c, \sigma \rangle \longrightarrow \langle c, \sigma \rangle$
- $\langle \mathsf{if\ true\ then}\ c_1\ \mathsf{else}\ c_2, \sigma \rangle \longrightarrow \langle c_1, \sigma \rangle$
- $\langle \mathsf{if\ true\ then}\ c_1\ \mathsf{else}\ c_2, \sigma \rangle \longrightarrow \langle c_2, \sigma \rangle$
- $\langle \mathsf{while}\ b\ \mathsf{do}\ c, \sigma \rangle \longrightarrow \langle \mathsf{if}\ b\ \mathsf{then}\ c; \mathsf{while}\ b\ \mathsf{do}\ c\ \mathsf{else\ skip}, \sigma \rangle$

To do global reduction, we repeatedly identify a redex (i.e. write the program as $C[r]$ for some C and r), and reduce it. To do this, we need to identify the next redex, which defines the evaluation order. We define

$$H := \bullet \mid n + H \mid H + e \mid (x := H) \mid \mathsf{if}\ H\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid H; c$$

Note that in $e_1 + e_2$, e_1 is evaluated before e_2. In general, redexes and contexts define evaluation order and short circuit behavior.

33 **Lemma** (Decomposition)
If c is not skip, then there exists a unique H and r such that $c = H[r]$.

This is similar to the progress lemma we had before, but this time, we can guarantee determinism with the uniqueness of H and r.

4.1.3 References

Many imperative programs have a heap, which allows us to maintain unbounded state. We will talk about one approach to the heap, which comes from the ML world. One thing that we have in OCaml is an explicit heap. In particular, we can create heap allocated objects and references them. However, it still tries to preserve much of what we like about functional languages.

To add references into a *functional* language, it helps to think about the type also. We modify lambda calculus like so:

$$\tau := \cdots \mid \tau\ \mathsf{ref}$$
$$e := \cdots \mid \mathsf{ref}\ e \mid (e_1 := e_2) \mid e_1; e_2 \mid !e$$

The idea is that $\mathsf{ref}\ e$ creates a new ref with initial value e, and $!e$ extracts the value from the ref.

To model the heap, we have the following definition:

$$h := \varnothing \mid h, a \longrightarrow (v : \tau)$$

Then, a program is a heap along with an expression, and heap addresses act like bound variables in expressions:

$$p = \mathsf{heap}\ h\ \mathsf{in}\ e$$

The new contexts added to support the heap are:

$$H = \cdots \mid \mathsf{ref}\ H \mid (H := e) \mid (\mathsf{addrs} := H) \mid !H$$

Since the heap is a global object, there are no local reduction rules. We do, however, have new global reduction rules:

- $\mathsf{heap}\ h\ \mathsf{in}\ H[\mathsf{ref}\ v : \tau] \longrightarrow \mathsf{heap}\ (h[a \longrightarrow v : \tau])\ \mathsf{in}\ H[a]$
- $\mathsf{heap}\ h\ \mathsf{in}\ H[!a] \longrightarrow \mathsf{heap}\ h\ \mathsf{in}\ H[v]$ where $a \longrightarrow v \in h$
- $\mathsf{heap}\ h\ \mathsf{in}\ H[a := v] \longrightarrow \mathsf{heap}\ (h[a \longrightarrow v] : \tau)\ \mathsf{in}\ H[\mathsf{unit}]$

The typing rules for each of the ref primitives are

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\mathsf{ref}\, e : \tau) : \tau\ \mathsf{ref}} \qquad \frac{\Gamma \vdash e : \tau\ \mathsf{ref}}{\Gamma \vdash\ !e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau\ \mathsf{ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \mathsf{unit}}$$

One problem with this is how they interact with polymorphism. Consider the following:

$$\begin{aligned} &\mathsf{let}\ x : \forall t.((t \to t)\ \mathsf{ref}) = \Lambda t.\ \mathsf{ref}(\lambda x : t\,.\,x) \\ &\mathsf{in}\ x[\mathsf{bool}] := \lambda x : \mathsf{bool}\,.\,\mathsf{not}\ x \\ &\quad (!x[\mathsf{int}])5 \end{aligned}$$

The solution is to disallow side effects in let.

4.1.4 Typing rules

For imperative programs, the typing rules are not too interesting:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}} \qquad \frac{}{\Gamma \vdash N : \mathsf{int}} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash c_1 : \mathsf{unit} \quad \Gamma \vdash c_2 : \mathsf{unit}}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 : \mathsf{unit}}$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash x : \tau}{\Gamma \vdash x := e : \mathsf{unit}} \qquad \frac{\Gamma \vdash e : \mathsf{bool} \quad \Gamma \vdash c : \mathsf{unit}}{\Gamma \vdash \mathsf{while}\ e\ \mathsf{do}\ c : \mathsf{unit}}$$

To prove progress and preservation, it it similar to what we did for functional programs. The difference is that we need to take into account the mutable state. More specifically, we have

Preservation If the mutable state is well typed, then after evaluating one step it will remain well typed.
Progress If the mutable state is well typed, then we can make progress.

4.2 Types for information flow

Reference: Myers, A. C. "JFlow: practical mostly-static information flow control". In POPL '99

Normally when people talk about type systems in imperative programming, where evaluation involves updating a store of values, standard types place a restriction on the program store and what operations you can use on the values.

Another common feature of imperative languages is public and public variables, which restricts operations in contexts where they do not apply. More generally, these are called information flow properties. We say that code has **confidentiality** if private information is not leaked to public values. Conversely, we say that code has **integrity** if public values do not have control over the internal state of private values.

34 **Example** (Confidentiality)
If we have a private class (in Python, e.g.) `Passwords` and a public class `Wikipedia`, then the following code is not confidential:

```
pws: Passwords = Passwords()
wiki: Wikipedia = Wikipedia()

wiki.add_entry(str(pws))
```

However, not all information flow violations are as explicit. If we see the statement `wiki.write("YES")`, we may think that it does not leak information, but in the right context, it actually does leak information:

```
if len(pws.get_password()) == 10:
  wiki.write("YES")
```

There are even more ambiguous situations:

```
p = pws.get_password()
if q: wiki.write("YES")
```

If p and q are related to each other, then information gets leaks.

35 **Example** (Integrity)

This code does not have integrity:

```
class Passwords:
  change_password(self):
    self.password = wiki.entry[:10]
```

One may think that a solution would just to not have these pieces of code interacting with each other at all, but there are other use cases that have similar requirements. For instance, a medical program may deal with many patients, and we want to make sure there is no information flow between each individual patient. Another example is a browser, where we do not want information from the saved passwords file to be leaked other than a few specific situations in which the user allows it to.

More formally, if we have a program P that has both public (L) and private (H) inputs and outputs, Then for all L_i, H_i, H_i', then $F(L_i, H_i) = (L_o, H_o) \implies F(L_i, H_i') = (L_o, H_o')$, i.e. changing the private input will not change the public output. Note that we cannot determine this from a single execution of the program. Because of this, it turns out our type based approach will work fairly well.

4.2.1 Strategy

We will first define a checkable property that implies that information flow is secure. Note that this may be stronger than what we actually need, e.g. a program may be insecure in an unreachable branch, but we will still reject the program.

This property will be defined with a *dynamic* labeling scheme. We will attach a label to every private value, and then we will propagate it to other values that use it. This turns a global property about executions with all possible inputs to a local property.

36 **Example**

If x has a private label, and we have $z = a \cdot x$, then we make z also have a private label. Note that even if there is no information flow (like when $a = 0$), z will still have the label.

It turns out that $\text{NaN} \cdot 0 = \text{NaN}$, so it pays to be conservative in some of these cases.

We then will define a *static* type system that allows us to approximate the set of labels that values can have.

4.2.2 Labels

Before, our examples just had one high confidentiality channel and one public channel. In more complicated programs, there are often different private channels, so we need a scheme that supports this.

We have principals, representing users or other identities, such as groups or roles. We then define a *policy* to be a principal called the *owner*, and a set containing principals called the *readers*. Policies will be of the form Owner: (reader1, reader2, reader3). Then, a label will be a set of policies. The reason why we want owners is so that permissions are able to change in the future if an owner wants to update it.

Note that there is a partial ordering on these labels. In particular, we write $L_1 \leq L_2$ if L_2 is more restrictive than L_1. Note that this is slightly counterintuitive, because L_2 actually has fewer readers than L_1. This partial order forms a **lattice**, meaning

- we can take the least upper bound ($\sqcup$) of two elements,
- there exists a least fixed point
- there exists a minimum element, called bottom.

This partial order is similar to subtyping: if a variable is trusted to handle a value with an L_2 label, then it is also trusted to handle a value with a label of L_1.

37 **Example**

$$\{A : (B, C)\} \leq \{A : (B)\}$$

$$\{A : (B, C), C : (B)\} \leq \{A : (B, C), C : (B)\}$$

$$\{A : (B), C : (B)\} \geq \{A : (B, C)\}$$

where the last one is because on the right side, C does not have any privacy concerns, i.e. it allows everything.

Now when we have some assignment `x{L2} := v{L1}`, it is valid only if `L1 <= L2`, i.e. the variable being assigned to is more restrictive. When we have a binary operation like `x{L1} + y{L2}`, then the result has label $L_1 \sqcup L_2$. However, if we consider the following code:

```
int{A: everyone} a, b, c;
int{A: A} p;
c = 0;
if (p) { c = a + b; }
```

Note that the information leaked into `c` is actually given by the label of `p`, not just the join of the two labels. The solution to this is to also consider the program counter PC. In particular, we need to keep track of the information leaked from knowing that the program is at a certain position. Therefore, as we run the program, we also label the PC.

4.2.3 Type system for information flow

Our basic judgements will be of the form $A \vdash E : X$. Here, the type environment A will carry a lot more information. In particular, it will have information about the program counter. Additionally, the type X will be a map containing several values:

- $X[\underline{\text{nv}}]$ will be the label of E if it terminates normally.
- $X[\underline{\text{n}}]$ will be the label that will be leaked if execution terminated after evaluating this expression.

We have the following rules:

$$\frac{}{A \vdash \text{literal} : X_\varnothing[\underline{\text{n}} := A[\underline{\text{pc}}], \underline{\text{nv}} := A[\underline{\text{pc}}]]}$$

$$\frac{}{A \vdash \text{skip} : X_\varnothing[\underline{\text{n}} := A[\underline{\text{pc}}]]}$$

$$\frac{A[v] = \langle \text{var } [\text{final}]\ \tau\{L\}\ uid\rangle \qquad X = X_\varnothing[\underline{\text{n}} := A[\underline{\text{pc}}], \underline{\text{nv}} := L \sqcup A[\underline{\text{pc}}]]}{A \vdash v : X}$$

$$\frac{A \vdash E : X \qquad A[v] = \langle \text{var } \tau\{L\}\ uid\rangle \qquad A \vdash X[\underline{\text{nv}}] \sqsubseteq L}{A \vdash v = E : X}$$

$$\frac{A \vdash E : X_E \qquad A[\underline{\text{pc}} := X_E[\underline{\text{nv}}]] \vdash S_1 : X_1 \qquad A[\underline{\text{pc}} := X_E[\underline{\text{nv}}]] \vdash S_2 : X_2 \qquad X = X_E[\underline{\text{n}} := \underline{\varnothing}] \oplus X_1 \oplus X_2}{A \vdash \text{if } E \text{ then } S_1 \text{ else } S_2 : X}$$

$$\frac{A \vdash S_1 : X_1 \qquad \text{extend}(A, S_1)[\underline{\text{pc}} := X_1[\underline{\text{n}}]] \vdash S_2 : X_2 \qquad X = X_1[\underline{\text{n}} := \underline{\varnothing}] \oplus X_2}{A \vdash S_1 ; S_2 : X}$$

TODO: The first two rules are:

- If the expression is a literal, then the only information leaked is the program counter at that point.
- If the expression is a literal, then the only information leaked is the program counter at that point.