EECS 3/496 Type Systems – Notes

Jesse A. Tov

Winter 2018

Contents

1 The `let-zl` language

1.1 Syntax

The `let-zl` language has expressions e defined as follows:

$$
\begin{aligned}
e &::= z \\
&\quad \mid \texttt{nil} \\
&\quad \mid (\texttt{cons}\ e\ e) \\
&\quad \mid (\texttt{+}\ e\ e) \\
&\quad \mid (\texttt{*}\ e\ e) \\
&\quad \mid (\texttt{car}\ e) \\
&\quad \mid (\texttt{cdr}\ e) \\
&\quad \mid x \\
&\quad \mid (\texttt{let}\ x\ e\ e) \\
z &::= \textit{integer} \\
x, y &::= \textit{variable-not-otherwise-mentioned}
\end{aligned}
$$

There are two kinds of literal expressions, integers z and the empty list `nil`. Additionally, we build longer lists with $(\texttt{cons}\ e_1\ e_2)$, which is our traditional cons that creates a linked list node with first and rest. We have two elimination forms for integers, $(\texttt{+}\ e_1\ e_2)$ and $(\texttt{*}\ e_1\ e_2)$. Additionally, we have elimination forms for lists, $(\texttt{car}\ e)$ and $(\texttt{cdr}\ e)$. Finally, we have variables x, and we have a form of sharing in $(\texttt{let}\ x\ e_1\ e_2)$, which binds x to the value of e_1 in e_2.

1.2 Dynamic semantics

We might have a decent guess as to what this language means, but to be precise, we will define its dynamic semantics using a rewriting system, which registers computation by rewriting expressions to expressions and eventually (hopefully) to values:

$$
\begin{aligned}
v &::= z \\
&\quad \mid \texttt{nil} \\
&\quad \mid (\texttt{cons}\ v\ v)
\end{aligned}
$$

We define values—final results—to include numbers z, the empty list `nil`, and pairs of values $(\texttt{cons}\ v_1\ v_2)$.

The reduction relation describes a single computation step, and has a case for each kind of basic computation step that our language performs. For example, here is how we perform addition:

$$E[(\texttt{+}\ z_1\ z_2)] \longrightarrow E[z_1 + z_2\ ] \text{ [plus]}$$

The [plus] rule says that to reduce an addition expression where both parameters are already reduced to numbers, we add the numbers in the metalanguage. The E portion of each term is the evaluation context, which means that addition can be performed not just on whole terms, but within terms according to a grammar given below.

The multiplication is similar, also allowing multiplication within any evaluation context:

$$E[(* \; z_1 \; z_2)] \longrightarrow E[z_1 \times z_2 \;] \text{ [times]}$$

We have two rules for getting the first and rest of a list:

$$E[(\texttt{car} \; (\texttt{cons} \; v_1 \; v_2))] \longrightarrow E[v_1] \text{ [car]}$$

$$E[(\texttt{cdr} \; (\texttt{cons} \; v_1 \; v_2))] \longrightarrow E[v_2] \text{ [cdr]}$$

These say that if we have a cons (pair) of values $(\texttt{cons} \; v_1 \; v_2)$ then `car` extracts the first value v_1 and `cdr` extracts the second value v_2.

Finally (for now), the rule for `let` involves substituting the value for the variable in the body:

$$E[(\texttt{let} \; x \; v \; e)] \longrightarrow E[e[x{:=}v]] \text{ [let]}$$

In order to describe where evaluation can happen when when it is finished, we extend our syntax with values v and evaluation contexts E:

$$\begin{aligned} E ::= \; & \texttt{[]} \\ & \mid (\texttt{cons} \; E \; e) \\ & \mid (\texttt{cons} \; v \; E) \\ & \mid (+ \; E \; e) \\ & \mid (+ \; v \; E) \\ & \mid (* \; E \; e) \\ & \mid (* \; v \; E) \\ & \mid (\texttt{car} \; E) \\ & \mid (\texttt{cdr} \; E) \\ & \mid (\texttt{let} \; x \; E \; e) \end{aligned}$$

Evaluation context E give a grammar for where evaluation can take place. For example, suppose we want to reduce the term $(* \; (+ \; 1 \; 2) \; (+ \; 3 \; 4))$. We need to decompose that term into an evaluation context and a redex, so that they match one of the reduction rules above. We can do that: the evaluation context is $(* \; \texttt{[]} \; (+ \; 3 \; 4))$, which matches the grammar of e, and the redex in the hole is thus $(+ \; 1 \; 2)$. This decomposition matches rule [plus], which converts it to $(* \; 3 \; (+ \; 3 \; 4))$. Then to perform another reduction, we decompose again, into evaluation context $(* \; 3 \; \texttt{[]})$ and redex $(+ \; 3 \; 4)$. That converts to 7 plugged back into the evaluation context, for $(* \; 3 \; 7)$. Then to perform one more reduction step, we decompose into the evaluation context `[]` and the redex $(* \; 3 \; 7)$, which converts to 21.

We define `-->*` to be the reflexive, transitive closure of `-->`. That is, $e_1 \longrightarrow^*_{e2}$ means that e_1 reduces to e_2 in zero or more steps.

The dynamic semantics of `let-zl` is now given by the evaluation function *eval*, defined as:

$\text{eval}(e) = v$ if $e \longrightarrow^* v$

As we discuss below, *eval* is partial for `let-zl` because there are errors that cause reduction to get "stuck."

Exercise 1. *Extend the language with Booleans. Besides Boolean literals, what do you think are essential operations? Extend the dynamic semantics with the necessary reduction rule(s) and evaluation context(s).*

Later we're going to do induction on *the size of terms* rather than the structure of terms, and we're going to use a particular size function, defined as:

$$
\begin{array}{ll}
|z| & = 0 \\
|\texttt{nil}| & = 0 \\
|(\texttt{cons}\ e_1\ e_2)| & = |e_1| + |e_2| \\
|(\texttt{+}\ e_1\ e_2)| & = 1 + |e_1| + |e_2| \\
|(\texttt{*}\ e_1\ e_2)| & = 1 + |e_1| + |e_2| \\
|(\texttt{car}\ e_1)| & = 1 + |e_1| \\
|(\texttt{cdr}\ e_1)| & = 1 + |e_1| \\
|x| & = 0 \\
|(\texttt{let}\ x\ e_1\ e_2)| & = 1 + |e_1| + |e_2|
\end{array}
$$

Exercise 2. *Prove that for all values, $|v| = 0$.*

1.2.1 Errors

Can `let-zl` programs experience errors? What does it mean for a reduction semantics to have an error? Right now, there are no explicit, checked errors, but there are programs that don't make sense. For example, $(\texttt{car}\ 5)$. What do these non-sense terms do right now? They get *stuck*! That is, a term that has $(\texttt{car}\ 5)$ in the hole won't reduce any further.

Indeed, there several classes of terms that get stuck in our definition of `let-zl` thus far:

- $(\texttt{car}\ \texttt{nil})$ and $(\texttt{cdr}\ \texttt{nil})$.
- $(\texttt{car}\ z)$ and $(\texttt{cdr}\ z)$, where z is an integer.
- $(\texttt{+}\ v_1\ v_2)$ or $(\texttt{*}\ v_1\ v_2)$ where v_1 or v_2 is not an integer.
- Any open term, that is, a term with a variable that is not bound by `let`.

What do these stuck states mean? They might correspond to a real language executing an invalid instruction or some other kind of undefined behavior. This is no good, but there are several ways we could solve the problem.

First, we could make such programs defined by adding transition rules. For example, we could add a rule that the car of a number is 0. Another way to make the programs defined, without sanctioning nonsense, is to add an error state. We do this by extending terms e to configurations C:

$$\begin{aligned} C ::= & \ e \\ | & \ \texttt{WRONG} \end{aligned}$$

Then we add transition rules that detect all bad states and transition them to `WRONG`, thus flagging them as errors.

$$E[(\texttt{car nil})] \longrightarrow \texttt{WRONG} \ \text{[car-nil]}$$

$$E[(\texttt{cdr nil})] \longrightarrow \texttt{WRONG} \ \text{[cdr-nil]}$$

This approach is equivalent to adding errors or exceptions to our programming language.

We now update our evaluation function *eval* to take these errors into account:

$\text{eval}(e) = v$ if $e \longrightarrow^* v$
$\text{eval}(e) = \texttt{WRONG}$ if $e \longrightarrow^* \texttt{WRONG}$

Alas, *eval* is still partial, because there are stuck states that we haven't converted to wrong states. (The other reason that *eval* could be partial is non-termination, but as we will prove, we don't have that.) A second way to rule out stuck states is to impose a type system, which rules out programs with some kinds of errors. We can then prove that no programs admitted by the type system get stuck, which will make *eval* total for this language.

1.3 Static semantics

With a type system, we assign types to (some) terms to classify them by what kind of value they compute. In our first, simple type system, we will have only two types:

$$\begin{aligned} \tau ::= & \ \texttt{int} \\ | & \ \texttt{list} \end{aligned}$$

To keep things simple, we will limit `list` to be lists of integers.

We then define a relation that assigns types to terms. For example, integer literals always have type `int`:

$$\frac{}{z : \texttt{int}} \ \text{[int]}$$

Similarly, the literal empty list has type `list`:

$$\frac{}{\texttt{nil} : \texttt{list}}\,[\text{nil}]$$

To type check an addition or multiplication, we check that the operands are both integers, and then the whole thing is an integer:

$$\frac{e_1 : \texttt{int} \quad e_2 : \texttt{int}}{(+\, e_1\, e_2) : \texttt{int}}\,[\text{plus}]$$

$$\frac{e_1 : \texttt{int} \quad e_2 : \texttt{int}}{(*\, e_1\, e_2) : \texttt{int}}\,[\text{times}]$$

To type check a `cons`, we require that the first operand be an integer and the second be a list, and then the whole thing is a list:

$$\frac{e_1 : \texttt{int} \quad e_2 : \texttt{list}}{(\texttt{cons}\, e_1\, e_2) : \texttt{list}}\,[\text{cons}]$$

To type check `car` and `cdr`, we require that the operand be a list; the result for `car` is an integer, and the result for `cdr` is another list:

$$\frac{e : \texttt{list}}{(\texttt{car}\, e) : \texttt{int}}\,[\text{car}]$$

$$\frac{e : \texttt{list}}{(\texttt{cdr}\, e) : \texttt{list}}\,[\text{cdr}]$$

But when we come to check a variable x, we get stuck. What's the type of a variable? To type check variables, we introduce type environments, which keep track of the type of each `let`-bound variable:

$$\begin{aligned} \Gamma ::= {} & \bullet \\ \mid {} & \Gamma, x{:}\tau \end{aligned}$$

We then retrofit all our rules to carry the environment Γ through. For example, the rule for `car` becomes

$$\frac{\Gamma \vdash e : \texttt{list}}{\Gamma \vdash (\texttt{car}\, e) : \texttt{int}}\,[\text{car}]$$

and similarly for the other rules we've seen so far.

Then we can write the rules for variables and for `let`. To type check a variable, look it up in the environment:

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\,[\text{var}]$$

If it isn't found, then the term is open and does not type.

Finally, to type check (`let` x e_1 e_2), we first type check e_1, yielding some type τ_1. We then type check e_2 with an environment extended with x bound to τ_1. The resulting type, τ_2, is the type of the whole expression:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{let}\ x\ e_1\ e_2) : \tau_2}\,[\text{let}]$$

Exercise 3. *Extend the type system to your language with Booleans.*

Exercise 4 (Generic lists). *Modify the type system as follows: instead of a single type* `list` *for lists of* `int`*s, allow* (`list int`)*,* (`list` (`list int`))*,* (`list` (`list` (`list int`))) *and so on. How do you have to change the syntax of* τ*? The typing rules?*

1.3.1 Type safety

The goal of our type system is to prevent undetected errors—that is, stuck terms—in our programs. To show that it does this, we will prove *type safety*: if a term e has a type τ, then one of:

- It will reduce in some number of steps to a value v that also has type τ.
- It will reduce in some number of steps to `WRONG`.
- It will reduce forever.

The last case cannot happen with this language, but it will be possible with languages we study in the future.

It is conventional to prove this theorem in terms of two lemmas, progress and preservation:

- Preservation: if e_1 has type τ and converts in one step to e_2, then e_2 also has type τ.
- Progress: if e has a type τ, then either e takes a conversion step or e is a value.

Preservation

Before we start, we make an observation about how typing derivations must be formed.

Lemma (Inversion). If $\Gamma \vdash e : \tau$ then,

- If the term is a variable x then $\Gamma(x) = \tau$.
- If the term is an integer z then $\tau =$ `int`.

- If the term is `nil` then τ = `list`.
- If the term is $(+\ e_1\ e_2)$ or $(*\ e_1\ e_2)$ then τ = `int` and $\Gamma \vdash e_1 : \texttt{int}$ and $\Gamma \vdash e_2 : \texttt{int}$.
- If the term is $(\texttt{cons}\ e_1\ e_2)$, then τ = `list` and $\Gamma \vdash e_1 : \texttt{int}$ and $\Gamma \vdash e_2 : \texttt{list}$
- If the term is $(\texttt{let}\ x\ e_1\ e_2)$ then there is some type τ_1 such that $\Gamma \vdash e_1 : \tau_1$ and $\Gamma, x{:}\tau_1 \vdash e_2 : \tau$.

Proof. By inspection of the typing rules.

We want to prove that if a term has a type and takes a step, the resulting term also has a type. We can do this be considering the cases of the reduction relation and showing that each preserves the type. Alas, each rule involves evaluation contexts E in the way of the action. Consequently, we'll have to prove a lemma about evaluation contexts.

Lemma (Replacement). *If* $\bullet \vdash E[e_1] : \tau$*, then there exists some type* τ_e *such that* $\bullet \vdash e_1 : \tau_e$*. Furthermore, for any other term* e_2 *such that* $\bullet \vdash e_2 : \tau_e$*, it is the case that* $\bullet \vdash E[e_2] : \tau$.

Proof. By induction on the structure of E:

- If E is `[]`, then $e = E[e_1]$, so τ_e must be τ. Then since $\bullet \vdash e_2 : \tau_e$, we have that $\bullet \vdash E[e_2] : \tau_e$.
- If $(\texttt{cons}\ E_1\ e_{22})$, then the only typing rule that applies is [cons], which means that τ must be `list`. Furthermore, by inversion of that rule it must be the case that $\bullet \vdash E_1[e_1] : \texttt{int}$ and $\bullet \vdash e_{22} : \texttt{list}$. By the induction hypothesis on the former, e_1 has some type τ_e, and furthermore, for any term e_2 that also has type τ_e, we have that $\bullet \vdash E_1[e_2] : \texttt{int}$. Then by applying rule [list], we have that $\bullet \vdash (\texttt{cons}\ E_1\ e_{22})[e_2] : \texttt{list}$.
- If $(\texttt{cons}\ e_{11}\ E_2)$, then as in the previous case, the only typing rule that applies is [cons], which means that τ must be `list`. It also means that $E_2[e_1]$ must have type `list` and e_{11} must have type `int`. Then by IH on the former, e_1 has a type τ_e, and furthermore, for any e_2 having type τ_e, $\bullet \vdash E_2[e_2] : \tau_e$. Then by reapplying rule [cons], we have that $\bullet \vdash E[e_2] : \texttt{list}$.
- If $(+\ E_1\ e_{22})$, then the only typing rule that applies is [plus], which means that τ is `int`. It also requires that $E_1[e_1]$ and e_{22} both have type `int`. Then apply IH to the former, yielding that e_1 has some type τ_e. Furthermore, by the IH, for any other e_2 having type τ_e, we have that $\bullet \vdash E_1[e_2] : \tau_e$. Then reapplying rule [plus], we have that $\bullet \vdash E[e_2] : \texttt{int}$.
- If $(+\ v_1\ E_2)$ or $(*\ E_1\ e_2)$ or $(*\ v_1\ E_2)$, as in the previous case, m.m.

- If $(\texttt{car}\ E_1)$ (or $(\texttt{cdr}\ E_1)$) then the only typing rule that applies is [car] (resp. [cdr]), which means that τ is `int` (resp. `list`). Furthermore, rule [car] (resp. [cdr]) requires that $E_1[e_1]$ must have type `list`. Then apply IH to get that $\bullet \vdash E_1[e_2] : \texttt{list}$ as well. Then $\bullet \vdash E[e_2] : \texttt{list}$ as well. Then apply rule [car] (resp. [cdr]) to get that $E[e_2]$ has type `int` (resp. `list`).

- If $(\texttt{let}\ x\ E_1\ e_{22})$, then the only rule that applies is [let]. By that rule, $E_1[e_1]$ must have some type τ_1, and $([x\ \tau_1]) \vdash e_{22} : \tau$. Then by the IH on the former, $\bullet \vdash e_1 : \tau_e$ for some τ_e. Furthermore, for any other e_2 having type τ_e, the IH tells us that $\bullet \vdash E_1[e_2] : \tau_1$ as well. Then we can reapply rule [let] to get $\bullet \vdash (\texttt{let}\ x\ E_1\ e_{22})[e_2] : \tau$.

QED.
There's one more standard lemma we need before we can prove preservation:
Lemma (Substitution). *If* $\Gamma, x{:}\tau_x \vdash e : \tau$ *and* $\Gamma \vdash v : \tau_x$ *then* $\Gamma \vdash e[x{:=}v] : \tau$.
Proof. By induction on the typing derivation for e; by cases on the conclusion:

- $\Gamma, x{:}\tau_x \vdash z : \texttt{int}$: Then $z[x{:=}v]$ is z, and $\Gamma \vdash z : \texttt{int}$.

- $\Gamma, x{:}\tau_x \vdash \texttt{nil} : \texttt{list}$: Then $\texttt{nil}[x{:=}v]$ is `nil`, and $\Gamma \vdash \texttt{nil} : \texttt{list}$.

- $\Gamma, x{:}\tau_x \vdash (\texttt{cons}\ e_1\ e_2) : \texttt{list}$: Then we know that $\Gamma, x{:}\tau_x \vdash e_1 : \texttt{int}$ and $\Gamma, x{:}\tau_x \vdash e_2 : \texttt{list}$. Then by the induction hypothesis, $\Gamma \vdash e_1[x{:=}v] : \texttt{int}$ and $\Gamma \vdash e_2[x{:=}v] : \texttt{list}$. Then by rule [cons], we have that $\Gamma \vdash (\texttt{cons}\ e_1[x{:=}v]\ e_2[x{:=}v]) : \texttt{list}$. But $(\texttt{cons}\ e_1[x{:=}v]\ e_2[x{:=}v])$ is $(\texttt{cons}\ e_1\ e_2)[x{:=}v]$, so $\Gamma \vdash (\texttt{cons}\ e_1\ e_2)[x{:=}v] : \texttt{list}$.

- $\Gamma, x{:}\tau_x \vdash (+\ e_1\ e_1) : \texttt{int}$: Then we know that $\Gamma, x{:}\tau_x \vdash e_1 : \texttt{int}$ and $\Gamma, x{:}\tau_x \vdash e_2 : \texttt{int}$. Then by the induction hypothesis, $\Gamma \vdash e_1[x{:=}v] : \texttt{int}$ and $\Gamma \vdash e_2[x{:=}v] : \texttt{int}$. Then apply rule [plus].

- $\Gamma, x{:}\tau_x \vdash (*\ e_1\ e_2) : \texttt{int}$: as in the previous case.

- $\Gamma, x{:}\tau_x \vdash (\texttt{car}\ e_1) : \texttt{int}$: Then we know that $\Gamma, x{:}\tau_x \vdash e_1 : \texttt{list}$. Then by IH, $\Gamma \vdash e_1[x{:=}v] : \texttt{list}$. And then by rule [car], $\Gamma \vdash (\texttt{car}\ e_1)[x{:=}v] : \texttt{int}$.

- $\Gamma, x{:}\tau_x \vdash (\texttt{cdr}\ e_1) : \texttt{list}$: As in the previous case.

- $\Gamma, x{:}\tau_x \vdash (\texttt{let}\ y\ e_1\ e_2) : \tau$: There are two possibilities, whether $x = y$ or not:

  - First, consider the case where $y \neq x$. Then we know that $\Gamma, x{:}\tau_x \vdash e_1 : \tau_e$ for some τ_e, and that $\Gamma, x{:}\tau_x, y{:}\tau_e \vdash e_2 : \tau$. Then by the induction hypothesis,

$\Gamma \vdash e_1[x{:=}v] : \tau_e$. Because $x \neq y$, $\Gamma, x{:}\tau_x, y{:}\tau_e = \Gamma, y{:}\tau_e, x{:}\tau_x$. (This reordering could be proved correct in an "exchange" lemma, but we take it to be obviously correct from the typing rules. Exchange will be of more interest when linear type systems force us to get serious about contexts.) So we have that $\Gamma, y{:}\tau_e, x{:}\tau_x \vdash e_2 : \tau$. Then by the induction hypothesis, $\Gamma, y{:}\tau_e \vdash e_2[x{:=}v] : \tau$. Then $\Gamma \vdash (\texttt{let}\ y\ e_1\ y_2)[x{:=}v] : \tau$ by rule [let].

- If $x = y$ then, as before, the induction hypothesis gives us that $\Gamma \vdash e_1[x{:=}v] : \tau_e$. By the assumption we know that $\Gamma, x{:}\tau_x \vdash (\texttt{let}\ x\ e_1\ e_2) : \tau$. By inversion, we know that $\Gamma, x{:}\tau_x, x{:}\tau_e \vdash e_2 : \tau$. But from the way environments work, we know that $\Gamma, x{:}\tau_x, x{:}\tau_e$ is the same as $\Gamma, x{:}\tau_e$. Thus we know $\Gamma, x{:}\tau_e \vdash e_2 : \tau$, which gives us the pieces to use the let rule to conclude that $\Gamma \vdash (\texttt{let}\ x\ e_1[x{:=}v]\ e_2) : \tau$, which is almost what we need to finish this case. Consider what the substituion function does when the variables are equal: $(\texttt{let}\ y\ e_1\ e_2)[x{:=}v] = (\texttt{let}\ x\ e_1\ e_2)[x{:=}v] = (\texttt{let}\ x\ e_1[x{:=}v]\ e_2)$. That means, that the typing derivation we just proved, namely $\Gamma \vdash (\texttt{let}\ x\ e_1[x{:=}v]\ e_2) : \tau$ is the same as the one that finishes this case, and thus $\Gamma \vdash (\texttt{let}\ y\ e_1\ e_2)[x{:=}v] : \tau$.

- $\Gamma, x{:}\tau_x \vdash y : \Gamma, x{:}\tau_x(y)$: There are two possibilities, whether $x = y$ or not:
  - If $x = y$, then $y[x{:=}v]$ is v. Furthermore, this means that $\tau = \tau_x$. And we have from the premise that $\Gamma \vdash v : \tau_x$.
  - If $x \neq y$, then $y[x{:=}v]$ is y. Furthermore, we know that $\Gamma, x{:}\tau_x(y) = \Gamma(y) = \tau$. Then $\Gamma \vdash y : \Gamma(y)$.

QED.
Now we are ready to prove preservation:

Lemma (Preservation). If $\bullet \vdash e_1 : \tau$ and $e_1 \longrightarrow e_2$ then $\bullet \vdash e_2 : \tau$.
Proof. By cases on the reduction relation:

- $E[(+\ z_1\ z_2)] \longrightarrow E[z_1 + z_2\ ]$: By the replacement lemma, $(+\ z_1\ z_2)$ must have some type, and by inversion, that type must be `int`. The result of the addition metafunction is also an integer with type `int`. Then by replacement, $\bullet \vdash E[z_1 + z_2\ ] : \tau$.

- $E[(*\ z_1\ z_2)] \longrightarrow E[z_1 \times z_2\ ]$: as in the previous case.

- $E[(\texttt{car}\ (\texttt{cons}\ v_1\ v_2))] \longrightarrow E[v_1]$: By the replacement lemma, $\bullet \vdash (\texttt{car}\ (\texttt{cons}\ v_1\ v_2)) : \tau_e$ for some type τ_e. The only rule that applies is [car], which requires that $\tau_e = \texttt{int}$ and $\bullet \vdash (\texttt{cons}\ v_1\ v_2) : \texttt{list}$. This types only by rule [cons], which requires that $\bullet \vdash v_1 : \texttt{int}$. Then by replacement, $\bullet \vdash E[v_1] : \tau$.

- $E[(\texttt{cdr}\ (\texttt{cons}\ v_1\ v_2))] \longrightarrow E[v_2]$: By the replacement lemma, $\bullet \vdash (\texttt{cdr}\ (\texttt{cons}\ v_1\ v_2)) : \tau_e$ for some type τ_e. The only rule that applies is [cdr], which requires that $\tau_e = \texttt{list}$ and $\bullet \vdash (\texttt{cons}\ v_1\ v_2) : \texttt{list}$. This types only by rule [cons], which requires that $\bullet \vdash v_2 : \texttt{list}$. Then by replacement, $\bullet \vdash E[v_2] : \tau$.

- $E[(\texttt{let}\ x\ v_1\ e_{22})] \longrightarrow E[e_{22}[x{:=}v_1]]$: By the replacement lemma, $\bullet \vdash (\texttt{let}\ x\ v_1\ e_{22}) : \tau_e$ for some types τ_e. The only rule that applies is [let], which requires that $\bullet \vdash v_1 : \tau_x$ for some τ_x such that $([x\ \tau_x]) \vdash e_{22} : \tau_e$. Then by the substitution lemma, $\bullet \vdash e_{22}[x{:=}v_1] : \tau_e$. Then by replacement, $\bullet \vdash E[e_{22}[x{:=}v_1]] : \tau$.

QED.

Progress

Before we can prove progress, we need to classify values by their types.

Lemma (Canonical forms).
If v has type τ, then:

- If τ is `int` then v is an integer literal z.
- If τ is `list`, then either $v = \texttt{nil}$ or $v = (\texttt{cons}\ v_1\ v_2)$ where v_1 has type `int` and v_2 has type `list`.

Proof. By induction on the typing derivation of $\bullet \vdash v : \tau$:

- $\bullet \vdash z : \texttt{int}$: Then v is an integer literal.
- $\bullet \vdash \texttt{nil} : \texttt{list}$: Then v is the empty list.
- $\bullet \vdash (\texttt{cons}\ e_1\ e_2) : \texttt{list}$: By the syntax of values it must be the case that e_1 is a value v_1 having type `int`, and e_2 is a value v_2 having type `list`.
- $\bullet \vdash (+\ e_1\ e_2) : \texttt{int}$: Vacuous, because not a value.
- The remaining cases are all vacuous because they do not allow for value forms.

QED.

Lemma (Context replacement). *If* $e_1 \longrightarrow e_2$ *then* $E[e_1] \longrightarrow E[e_2]$. *If* $e_1 \longrightarrow \texttt{WRONG}$ *then* $E[e_1] \longrightarrow \texttt{WRONG}$.

Proof. If $e_1 \longrightarrow e_2$ then e_1 must be some redex in a hole: $E_1[e_{11}]$. Furthermore, it must take a step to some $E_1[e_{22}] = e_2$. Then the same redex e_{11} converts to the same contractum e_{22} in any evaluation context, including $E[E_1]$.

If $e_1 \longrightarrow \texttt{WRONG}$ then e_1 must be some redex in a hole: $E_1[e_{11}]$ which converts to `WRONG`. Then that same redex converts to `WRONG` in any evaluation context, including $E[E_1]$.

Lemma (Progress). *If* $\bullet \vdash e : \tau$ *then term* e *either converts or is a value.*

Proof. By induction on the typing derivation; by cases on the conclusion:

- $\bullet \vdash z : \texttt{int}$: Then e is a value.

- $\bullet \vdash \texttt{nil} : \texttt{list}$: Then e is a value.

- $\bullet \vdash (\texttt{cons}\ e_1\ e_2) : \texttt{list}$: Then $\bullet \vdash e_1 : \texttt{int}$ and $\bullet \vdash e_2 : \texttt{list}$. By the induction hypothesis, term e_1 either converts, or is a value. If e_1 converts to some term e_{11}, then $(\texttt{cons}\ e_1\ e_2) \longrightarrow (\texttt{cons}\ e_{11}\ e_2)$ by the context replacement lemma. If e_1 converts to WRONG, then $(\texttt{cons}\ e_1\ e_1) \longrightarrow \texttt{WRONG}$ by the context replacement lemma. If e_1 is a value v_1, then consider e_2, which by the induction hypothesis either converts or is a value. If e_2 converts to a term e_{22}, then $(\texttt{cons}\ v_1\ e_2) \longrightarrow (\texttt{cons}\ v_1\ e_{22})$ by the context replacement lemma. If e_2 converts to WRONG, then $(\texttt{cons}\ v_1\ e_2) \longrightarrow \texttt{WRONG}$ by the context replacement lemma. Finally, if e_2 is a value v_2 then e is a value $(\texttt{cons}\ v_1\ v_2)$.

- $\bullet \vdash (+\ e_1\ e_2) : \texttt{int}$: Then $\bullet \vdash e_1 : \texttt{int}$ and $\bullet \vdash e_2 : \texttt{int}$. By the induction hypothesis, e_1 either converts or is a value. If e_1 converts to a term e_{11}, then $(+\ e_1\ e_2) \longrightarrow (+\ e_{11}\ e_2)$ by the context replacement lemma. If e_1 converts to WRONG then $(+\ e_1\ e_2) \longrightarrow \texttt{WRONG}$ by the context replacement lemma. If e_1 is a value v_1, then consider e_1, which by the induction hypothesis either converts or is a value. If e_2 converts to a term e_{22}, then $(+\ v_1\ e_2) \longrightarrow (+\ v_1\ e_{22})$ by the context replacement lemma. If e_2 converts to WRONG, then $(+\ v_1\ e_2) \longrightarrow \texttt{WRONG}$ by the context replacement lemma. Otherwise, e_2 is a value v_2. By the canonical forms lemma, v_1 is an integer z_1 and v_2 is an integer z_2. Thus, we can take the step $(+\ z_1\ z_2) \longrightarrow z_1 + z_2$.

- $\bullet \vdash (*\ e_1\ e_2) : \texttt{int}$: As in the previous case.

- $\bullet \vdash (\texttt{car}\ e_1) : \texttt{int}$: Then $\bullet \vdash e_1 : \texttt{list}$. By the induction hypothesis, e_1 either converts or is a value. If it converts to a term e_{11}, then $(\texttt{car}\ e_1) \longrightarrow (\texttt{car}\ e_{11})$ by the context replacement lemma. If it converts to WRONG, then $(\texttt{car}\ e_1) \longrightarrow \texttt{WRONG}$ by the context replacement lemma. Otherwise, e_1 is a value. By the canonical forms lemma, it has the form $(\texttt{cons}\ v_1\ v_2)$, so we can take a step $(\texttt{car}\ (\texttt{cons}\ v_1\ v_2)) \longrightarrow v_1$.

- $\bullet \vdash (\texttt{cdr}\ e_1) : \texttt{list}$: As in the previous case, but reducing to v_2.

- $\bullet \vdash x : \tau$: Vacuous.

- $\bullet \vdash (\texttt{let}\ x\ e_1\ e_2) : \tau$: Then $\bullet \vdash e_1 : \tau_x$ and $([x\ \tau_x]) \vdash e_2 : \tau$ for some τ_x. Then by the induction hypothesis, e_1 either converts or is a value. If e_1 converts to a term e_{11}, then $(\texttt{let}\ x\ e_1\ e_2) \longrightarrow (\texttt{let}\ x\ e_{11}\ e_2)$ by the context replacement lemma. If e_1 converts to WRONG then $(\texttt{let}\ x\ e_1\ e_2) \longrightarrow \texttt{WRONG}$ by the context replacement lemma. Otherwise, e_1 is a value v_1, and $(\texttt{let}\ x\ v_1\ e_2) \longrightarrow e_2[x{:=}v_1]$.

QED.

Exercise 5. *Prove progress and preservation for your language extended with Booleans.*

Exercise 6. *Prove progress and preservation for your language extended with generic lists.*

Exercise 7. *Are the previous two exercises orthogonal? How do they interact or avoid interaction?*

1.4 Termination

Now let's prove a rather strong property about a rather weak language.

Theorem (Size is work). *Suppose* $\bullet \vdash e : \tau$ *and* $|e| = k$. *Then* e *either reduces to a value or goes wrong in* k *or fewer steps.*

Proof. This proof uses induction, but it uses induction on the set $\times$, using a lexicographic ordering. That is, we consider the first natural number to be the number of nodes in the given e (when viewed as a tree) and the second one to be $|e|$. The lexicographic order is well-founded, and so we can use induction when we have a term where the $|e|$ is strictly less than the given one, or when $|e|$ is the same as the given one, but the number of nodes is strictly smaller.

- z: Then k = 0, and e reduces to value z in 0 steps.
- `nil`: Also k = 0.
- (`cons` e_1 e_2). Then by inversion of [cons], $\bullet \vdash e_1 :$ `int` and $\bullet \vdash e_2 :$ `list`. Let j be the size of e_1; then the size of e_2 is k – j. We can use induction on both e_1 and e_2, because they both have strictly fewer nodes that (`cons` e_1 e_2), and $|e_1|$ and $|e_2|$ are both less than or equal to $|$(`cons` e_1 e_2)$|$. (With the exception of the `let` case, the justification for induction will be the same as this one in all the other cases.) Then by induction, e_1 reduces to a value v_1 or to `WRONG` in j or fewer steps. If it reduces to `WRONG` then by the context replacement lemma, (`cons` e_1 e_2) also reduces to `WRONG` in j or fewer steps. Otherwise, consider the induction hypothesis on e_2 (size k – j); it must reduce to a value v_2 or to `WRONG` in k – j or fewer steps. If `WRONG`, then the whole thing goes wrong by context replacement. Otherwise, (`cons` e_1 e_2) goes to (`cons` v_1 v_2) in k or fewer steps.
- (+ e_1 e_2). Then by inversion of the typing rule `int`, both subterms have type `int`. Let j be the size of e_1; then the size of e_2 is k – j – 1. Then by the induction hypothesis, each reduces to a value or goes wrong, in at most j and k – j – 1 steps respectively. If either goes wrong, then the whole term goes wrong because both (+ `[]` e_2) and (+ v_1 `[]`) are evaluation contexts. Otherwise, by the canonical values lemma both values must be numbers z_1 and z_2. Because $e_1 \longrightarrow^* z_1$ in j or fewer steps, by context replacement (+ e_1 e_2) $\longrightarrow^*$ (+ z_1 e_2) in j or fewer steps. And because $e_2 \longrightarrow^* z_2$ in k – j – 1 or fewer steps, by context replacement again (+ z_1 e_2) $\longrightarrow^*$ (+ z_1 z_2) in k – j – 1 or

fewer steps. Then in one more step $(\texttt{+}\ z_1\ z_2) \longrightarrow z_1 + z_2$, which is a value. The total number of steps has been k or fewer.

- $(\texttt{*}\ e_1\ e_2)$: As in the previous case, m.m.
- $(\texttt{car}\ e_1)$ and $(\texttt{cdr}\ e_1)$: In either case, the subterm e_1 must have type `list` by inversion of the typing rule. Furthermore, the size of e_1 must be k – 1. Then by the induction hypothesis, e_1 either reduces to a value or goes wrong in k – 1 or fewer steps. If it goes wrong then the whole term goes wrong. If it reduces to a value, then by preservation, that value also has type `list`. (Note also that it also reduces to a value in the evaluation context $(\texttt{car}\ [])$.) Then by the canonical values lemma, that value must be either `nil` or $(\texttt{cons}\ v_1\ v_2)$ for some values v_1 and v_2. If the former then the whole term goes to `WRONG` in one more step by rule [car-nil] or rule [cdr-nil], respectively. If the latter, then it take one more step to v_1 or v_2, respectively. In either case, k steps have transpired.
- x: Vacuous because open terms don't type.
- $(\texttt{let}\ x\ e_1\ e_2)$: By inversion, we know that $\bullet \vdash e_1 : \tau_x$ for some type τ_x. And we know that $([x\ \tau_x]) \vdash e_2 : \tau$. Let j be the size of e_1; then the size of e_2 is k – j – 1. We can use induction on e_1 because $|e_1|$ is less than $|(\texttt{let}\ x\ e_1\ e_2)|$ and there are strictly few nodes. Thus, by induction on e_1, we have that e_1 reduces to a value or goes wrong in j or fewer steps. If it goes wrong then the whole term goes wrong. If it reduces to a value v_1, then by context replacement (and induction on the length of the reduction sequence), the whole term reduces $(\texttt{let}\ x\ e_1\ e_2) \longrightarrow^* (\texttt{let}\ x\ v_1\ e_2)$ in j or fewer steps. Then in one more step, $(\texttt{let}\ x\ v_1\ e_2) \longrightarrow e_2[x{:=}v_1]$. Note that because the size of a variable is 0 and so is the size of a value, the size of $e_2[x{:=}v_1]$ is the same as the size of e_2, k – j – 1, which is strictly less than the size of $(\texttt{let}\ x\ e_1\ e_2)$. In this case, the number of nodes in $e_2[x{:=}v_1]$ might be many more than the number of nodes in e_2 because v_1 might be a long list. But we set up our induction using the lexicgraphic order so that we only need to consider the relative sizes of $e_2[x{:=}v_1]$ and $(\texttt{let}\ x\ e_1\ e_2)$, not the number of nodes them to justify induction. Now, by preservation, $\bullet \vdash e_1[x{:=}v_1] : \tau$. So we an apply the induction hypothesis to $e_1[x{:=}v_1]$, learning that it goes wrong or reaches a value in k – j – 1 or fewer steps. This yields a total of k or fewer steps.

QED.

2 The simply-typed lambda calculus λ-st

2.1 Syntax

The λ-st language has types τ and terms e defined as follows:

$$
\begin{array}{rl}
\tau ::= & \texttt{nat} \\
& \mid (\rightarrow \tau\ \tau) \\
e ::= & x \\
& \mid \texttt{z} \\
& \mid (\texttt{s}\ e) \\
& \mid (\lambda\ x\ \tau\ e) \\
& \mid (\texttt{ap}\ e\ e) \\
x, y ::= & \textit{variable-not-otherwise-mentioned}
\end{array}
$$

Types include the natural numbers `nat` and function types $(\rightarrow \tau_1\ \tau_2)$. Terms include variables, Peano naturals (`z` for zero and `s` for successor), lambda abstractions, and applications.

2.2 Dynamic semantics

To define the dynamic semantics of λ-`st`, we give syntax for values and evaluation contexts:

$$
\begin{array}{rl}
v ::= & \texttt{z} \\
& \mid (\texttt{s}\ v) \\
& \mid (\lambda\ x\ \tau\ e) \\
E ::= & [\,] \\
& \mid (\texttt{s}\ E) \\
& \mid (\texttt{ap}\ E\ e) \\
& \mid (\texttt{ap}\ v\ E)
\end{array}
$$

Values include natural numbers and lambda abstractions. We evaluate under `s` and we evaluate both the operator and operand in an application.

Then the reduction relation consists of one rule:

$$E[(\texttt{ap}\ (\lambda\ x\ \tau\ e)\ v)] \longrightarrow E[e[x{:=}v]] \ \text{[}\beta\text{-val]}$$

The dynamic semantics of λ-`st` is given by the evaluation function *eval*:

$\text{eval}(e) = v$ if $e \longrightarrow^* v$

As defined, *eval* could be partial, but we will prove it total on well typed terms, first by proving that well typed terms don't go wrong, and then by proving that well typed terms don't diverge.

2.3 Static semantics

To type λ-`st`, we define typing contexts mapping variables to types:

$$
\begin{array}{rl}
\Gamma ::= & \bullet \\
& \mid \Gamma, x{:}\tau
\end{array}
$$

Then the rules are as follows. There are two constructors for the naturals, and they type as such:

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat}}\,[\text{zero}]$$

$$\frac{\Gamma \vdash e : \mathtt{nat}}{\Gamma \vdash (\mathtt{s}\ e) : \mathtt{nat}}\,[\text{succ}]$$

That is, `z` is a natural, and for any term e of type `nat`, $(\mathtt{s}\ e)$ has type `nat` as well.

Variables type by looking them up in the typing context:

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\,[\text{var}]$$

Lambda abstractions type by extending the typing context with the bound variable and checking the body:

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda\ x\ \tau_1\ e) : (\rightarrow\ \tau_1\ \tau_2)}\,[\text{abs}]$$

And finally, applications require the domain of the operator to match the type of the operand:

$$\frac{\Gamma \vdash e_1 : (\rightarrow\ \tau_2\ \tau) \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\mathtt{ap}\ e_1\ e_2) : \tau}\,[\text{app}]$$

Exercise 8. *Extend* λ`-st` *with a product type* $(*\ \tau_1\ \tau_2)$. *You will need a form for constructing products and projections for getting the components out. Add the necessary reduction and typing rules.*

Exercise 9. *Extend* λ`-st` *with a sum type* $(+\ \tau_1\ \tau_2)$. *You will need two injection forms* $(\mathtt{inl}\ e)$ *and* $(\mathtt{inr}\ e)$ *to create sums, and one case analysis form to eliminate them,* $(\mathtt{match}\ e\ [x\ e_l]\ [y\ e_r])$. *The case analysis form takes a step once* e *reduces to a sum value:* $(\mathtt{match}\ (\mathtt{inl}\ v)\ [x\ e_l]\ [y\ e_r]) \longrightarrow e_l[x{:=}v]$, *and similarly for* $(\mathtt{inr}\ v)$. *Add the necessary reduction and typing rules.*

2.3.1 Type safety

Before we can prove type safety, we need to prove several standard lemmas.

We use the judgment $e : \tau$ with no context to mean that e types in an empty context: $\bullet \vdash e : \tau$.

Lemma (Replacement). *If* $E[e] : \tau$ *then* $e : \tau_e$ *for some* τ_e. *Furthermore, for any other term* $e_{new} : \tau_e$, $E[e_{new}] : \tau$.

Proof. By induction on the structure of E:

- `[]`: Then trivially, with $\tau_e = \tau$.
- $(\mathtt{s}\ E_1)$: By inspection of the typing rules, we know that if $(\mathtt{s}\ E_1)[e]$ has a type, that type is `int`. By inversion, we know that $E_1[e]$ has type `int` as well. Then by the induction hypothesis, e has some type τ_e, and for any e_{new} having type τ_e, $E_1[e_{new}] : \mathtt{int}$. Then by rule [succ], $(\mathtt{s}\ E_1[e_{new}]) : \mathtt{int}$.
- $(\mathtt{ap}\ E_1\ e_1)$: The whole term has a type τ only if there is some type τ_1 such that $E_1[e] : \tau_1 \longrightarrow \tau$ and $e_1 : \tau_1$. Then by the induction hypothesis, $e : \tau_e$, and for any term e_{new} having type τ_e, $E_1[e_{new}] : \tau_1 \longrightarrow \tau$. Then $(\mathtt{ap}\ E_1[e_{new}]\ e_1) : \tau$.
- $(\mathtt{ap}\ v_1\ E_1)$: The whole term has a type τ only if there is some type τ_1 such that $v_1 : \tau_1 \longrightarrow \tau$ and $E_1[e] : \tau_1$. Then by the induction hypothesis, $e : \tau_e$, and for any term e_{new} having type τ_e, $E_1[e_{new}] : \tau_1$. Then $(\mathtt{ap}\ v_1\ E_1[e_{new}]) : \tau$.

Lemma (Substitution). *If* $\Gamma \vdash e_1 : \tau_1$ *and* $\Gamma, x{:}\tau_1 \vdash e_2 : \tau_2$ *then* $\Gamma \vdash e_2[x{:=}e_1] : \tau_2$.

Proof. By induction on the type derivation for e_2:

- $\Gamma, x{:}\tau_1 \vdash y : \tau_2$: If $x = y$ then $\tau_1 = \tau_2$, and $y[x{:=}e_1] = e_1$, which has type τ_1. If $x \neq y$, then $y[x{:=}e_1] = y$, which must type in Γ.
- $\Gamma, x{:}\tau_1 \vdash \mathtt{z} : \mathtt{nat}$: This types in an environment.
- $\Gamma, x{:}\tau_1 \vdash (\mathtt{s}\ e) : \mathtt{nat}$: Then it must be the case that $\Gamma, x{:}\tau_1 \vdash e : \mathtt{nat}$. Then by induction, $\Gamma \vdash e[x{:=}e_1] : \mathtt{nat}$, and reapplying rule [succ], we have that $\Gamma \vdash (\mathtt{s}\ e)[x{:=}e_1] : \mathtt{nat}$.
- `ap` cases are similar.
- $\Gamma, x{:}\tau_1 \vdash (\lambda\ y\ \tau_{21}\ e) : (\rightarrow\ \tau_{21}\ \tau_{22})$: This can be the case only if $\Gamma, x{:}\tau_1, y{:}\tau_{21} \vdash e : \tau_{22}$. Without loss of generality, $x \neq y$, so we can swap them, yielding $\Gamma, y{:}\tau_{21}, x{:}\tau_1 \vdash e : \tau_{22}$. Then by induction $\Gamma, y{:}\tau_{21} \vdash e[x{:=}e_1] : \tau_{22}$. Then apply rule [abs], to get $\Gamma \vdash (\lambda\ y\ \tau_{21}\ e)[x{:=}e_1] : (\rightarrow\ \tau_{21}$

Lemma (Preservation). *If* $e_1 : \tau$ *and* $e_1 \longrightarrow e_2$ *then* $e_2 : \tau$.

Proof. By cases on the reduction relation. There is one case:

- $E[(\mathtt{ap}\ (\lambda\ x\ \tau_x\ e_{11})\ v_{12})] \longrightarrow E[e_{11}[x{:=}v_{12}]]$. By the replacement lemma, we know that $(\mathtt{ap}\ (\lambda\ x\ \tau_x\ e_{11})\ v_{12}) : \tau_1$. This only types if $(\lambda\ x\ \tau_x\ e_{11}) : (\rightarrow\ \tau_x\ \tau_1)$ and $v_{12} : \tau_x$. The former is only the case if $\bullet, x{:}\tau_x \vdash e_{11} : \tau_1$. Then by the substitution lemma, $e_{11}[x{:=}v_{12}] : \tau_1$, and by replacement, $E[e_{11}[x{:=}v_{12}]] : \tau$.

QED.

Lemma (Canonical forms).
If $v : \tau$ then:

- If τ is `nat`, then v is either `z` or or $(\texttt{s}\ v_1)$ for some v_1.
- If τ is $(\rightarrow\ \tau_1\ \tau_2)$, then v is some lambda abstraction $(\lambda\ x\ \tau_1\ e_1)$.

Proof. By induction on the structure of the typing derivation. Only three rules form values, and those three rules correspond to the conditions of the lemma.

Lemma (Progress). *If $e_1 : \tau$ then either e_1 is a value or $e_1 \longrightarrow e_2$ for some e_2.*

Proof. By induction on the structure of the typing derivation, considering the terms:

- x: Vacuous, open terms don't type.
- `z`: A value.
- $(\texttt{s}\ e)$: By induction, e steps or is a value of type `nat`. If the former then the whole term steps; if the latter then the whole term is a value.
- $(\texttt{ap}\ e_{11}\ e_{12})$: By induction, each subterm steps or is a value. If the first subterm steps, then the whole term steps. If the first subterm is a value and the second steps, then the whole thing steps. If both are values, then by inversion of the [app] rule, e_{11} has a function type, and by the canonical forms lemma, that means it is a lambda abstraction $(\lambda\ x\ e)$. Then the whole term steps to $e[x{:=}e_{12}]$.
- $(\lambda\ x\ \tau_1\ e)$: A value.

Theorem (Safety). *1) If $e_1 : \tau$ and $e_1 \longrightarrow e_2$ then $e_2 : \tau$. 2) If $e_1 : \tau$ then either e_1 is a value or $e_1 \longrightarrow e_2$ for some e_2.*

Exercise 10. *Extend the type safety theorem to cover product and/or sum types.*

2.4 An extension

As it stands, we can't do much with natural numbers. Inspired by Gödel's system T, we add a limited, terminating form of recursion on natural numbers. We extend the syntax of terms and evaluation contexts as follows:

$$
\begin{array}{rl}
e ::= & \ldots \\
 & \mid (\texttt{rec}\ e\ [e_z]\ [x_{\text{pre}}\ y_{\text{rec}}\ e_s]) \\
E ::= & \ldots \\
 & \mid (\texttt{rec}\ E\ [e_z]\ [x_{\text{pre}}\ y_{\text{rec}}\ e_s])
\end{array}
$$

The new form is the recursor, which works as follows. First, it evaluates e to a value, which must be a natural number. If that number is zero, then it evaluates e_z. Otherwise, if that term is a successor $(\texttt{s}\ v)$, it recurs on v, binding the recursive result to y_{rec} and the predecessor v to x_{pre} in e_s.

There are no new types. We extend the reduction relation with two cases representing the just-described dynamics:

$$E[(\texttt{rec z}\ [e_z]\ [x_{\text{pre}}\ y_{\text{rec}}\ e_s])] \longrightarrow E[e_z] \quad \text{[rec-zero]}$$

$$E[(\texttt{rec}\ (\texttt{s}\ v)\ [e_z]\ [x_{\text{pre}}\ y_{\text{rec}}\ e_s])] \longrightarrow E[e_s[x_{\text{pre}}{:=}v][y_{\text{rec}}{:=}(\texttt{rec}\ v\ [e_z]\ [x_{\text{pre}}\ y_{\text{rec}}\ e_s])]] \quad \text{[rec-succ]}$$

There is one rule for typing the new form:

$$\frac{\Gamma \vdash e : \texttt{nat} \qquad \Gamma \vdash e_z : \tau \qquad \Gamma, x_{\text{pre}}{:}\texttt{nat}, y_{\text{rec}}{:}\tau \vdash e_s : \tau}{\Gamma \vdash (\texttt{rec}\ e\ [e_z]\ [x_{\text{pre}}\ y_{\text{rec}}\ e_s]) : \tau}\ \text{[rec]}$$

Here is predecessor expressed using the recursor:

$$\text{pred} = (\lambda\ \texttt{n nat}\ (\texttt{rec n}\ [\texttt{z}]\ [x_{\text{pre}}\ y_{\text{rec}}\ x_{\text{pre}}]))$$

For zero it returns zero, and for any other number it returns the predecessor, ignoring the recursive result.

And here is addition expressed using the recursor:

$$\text{add} = (\lambda\ \texttt{n nat}\ (\lambda\ \texttt{m nat}\ (\texttt{rec n}\ [\texttt{m}]\ [x_{\text{pre}}\ y_{\text{rec}}\ (\texttt{s}\ y_{\text{rec}})])))$$

Exercise 11. *Implement multiplication using the recursor.*

Exercise 12. *Implement factorial using the recursor.*

Exercise 13. *Implement a function that divides a natural number by two (rounding down).*

Exercise 14. *Extend the type safety theorem for the recursor.*

Exercise 15. *The recursor is currently call-by-name, in the sense that it substitutes the whole recursive expression of* $(\texttt{rec}\ v\ [e_z]\ [x_{\text{pre}}\ y_{\text{rec}}\ e_s])$ *for* y_{rec} *in the non-zero case. The call-by-value form would compute the value of that subterm first and then subtitute the value, but making it call-by-value requires introducing an additional form.* `let` *will do. (Why can't we just use* λ*?) Show how to add* `let` *to* λ`-st` *and how that can be used to make the recursor call-by-value.*

2.5 Normalization

A normal form is a form that doesn't reduce any further, which for our purposes (since we have eliminated stuck states) is a value. A term e normalizes, written $(\Downarrow e)$ if it reduces to a normal form, that is, a value.

Historically, when working with lambda calculi that allow free conversion (that is, redexes can by identified anywhere in a term, without a notion of evaluation contexts) authors have distinguished weak from strong normalization. A term weakly normalizes if it has some reduction sequence reaching a normal form; a term strongly normalizes if every of its reduction sequences reaches a normal form. However, we've defined our language to be deterministic, which causes weak and strong normalization to coincide.

We wish to show that all terms that have a type reduce to a value. It is insufficient to do induction on typing derivations. (Shall we try it?) What we end up needing is a (unary) relation on terms, indexed by types, and defined by induction on types, of the form $(\text{SN}\ \tau\ e)$, as follows:

- $(\text{SN}\ \texttt{nat}\ e)$ iff $e : \texttt{nat}$ and $(\Downarrow e)$.
- $(\text{SN}\ (\rightarrow \tau_1\ \tau_2)\ e)$ iff $e : (\rightarrow \tau_1\ \tau_2)$ and $(\Downarrow e)$ and for all e_1 such that $(\text{SN}\ \tau_1\ e_1)$, $(\text{SN}\ \tau_2\ (\texttt{ap}\ e\ e_1))$.

Exercise 16. *How would we extend* SN *for product and/or sum types?*

Lemma (SN preserved by conversion).
Suppose that $e_1 : \tau$ and $e_1 \longrightarrow e_2$. Then:

- $(\text{SN}\ \tau\ e_2)$ implies $(\text{SN}\ \tau\ e_1)$.
- $(\text{SN}\ \tau\ e_1)$ implies $(\text{SN}\ \tau\ e_2)$.

Proof. By induction on τ:

- nat: If e_2 normalizes then e_1 normalizes by the same sequence because e_1 takes a step to e_2, which then normalizes. We know this because our formulation of λ-st is determistic, and there is no other way to reduce the term. (We could prove this but haven't.) Since it has type τ, we have $(\text{SN}\ \texttt{nat}\ e_1)$

  If e_1 normalizes then it does so via e_2, so e_2 normalizes as well and by preservation it has the same type, so $(\text{SN}\ \texttt{nat}\ e_2)$.

- $(\rightarrow \tau_1\ \tau_2)$: If $(\text{SN}\ (\rightarrow \tau_1\ \tau_2)\ e_2)$ then we know that e_2 normalizes and when applied to a good term, that normalizes too. We need to show that e_1 does that same, that is, that $(\text{SN}\ \tau_1\ e_{\text{arb}})$ implies that $(\text{SN}\ \tau_2\ (\texttt{ap}\ e_1\ e_{\text{arb}}))$ for arbitrary term e_{arb}. We know that $(\text{SN}\ \tau_2\ (\texttt{ap}\ e_2\ e_{\text{arb}}))$. Since $e_1 \longrightarrow e_2$, we know that $(\texttt{ap}\ e_1\ e_{\text{arb}}) \longrightarrow (\texttt{ap}\ e_2\ e_{\text{arb}})$. Since τ_2 is a subexpression of $(\rightarrow \tau_1\ \tau_2)$, we can apply the induction hypothesis at that type, yielding $(\text{SN}\ \tau_2\ (\texttt{ap}\ e_1\ e_{\text{arb}}))$ as desired.

If $(\text{SN}\ (\to\ \tau_1\ \tau_2)\ e_1)$ then we know that e_1 normalizes and when applied to a good term, that normalizes too. We need to know that e_2 does the same, that is, that $(\text{SN}\ \tau_1\ e_{\text{arb}})$ implies that $(\text{SN}\ \tau_2\ (\texttt{ap}\ e_2\ e_{\text{arb}}))$ for arbitrary term e_{arb}. We know that $(\text{SN}\ \tau_2\ (\texttt{ap}\ e_1\ e_{\text{arb}}))$. Since $e_1 \longrightarrow e_2$, we know that $(\texttt{ap}\ e_1\ e_{\text{arb}}) \longrightarrow (\texttt{ap}\ e_2\ e_{\text{arb}})$. Then by induction, $(\text{SN}\ \tau_2\ (\texttt{ap}\ e_2\ e_{\text{arb}}))$.

QED.

Next, we define substitutions, and what it means for a substitution to satisfy a typing environment. A substitution associates some variables with values to substitute them:

$$\begin{aligned}\gamma ::= &\bullet \\ \mid &\ \gamma[x{:=}v]\end{aligned}$$

To apply a substitution to a term, written $e\gamma$, is to substitute in the term the values of the substitution for their variables.

A substitution satisfies a typing environment if they have the same domains (sets of variables) and every value in the substitution not only has the type given for the corresponding variable in the type environment, but is SN for that type:

$$\frac{}{\bullet \vDash \bullet}\ [\text{nil}]$$

$$\frac{(\text{SN}\ \tau\ v) \qquad \gamma \vDash \Gamma}{\gamma[x{:=}v] \vDash \Gamma,\ x{:}\tau}\ [\text{cons}]$$

Note that if a substitution satisfies a type environment, this means that it contains values that typed in the empty type environment, meaning they are closed. Thus, the order of substitution doesn't matter, as no variable in the substitution will interfere with any other.

Now we can prove a lemma that if we apply a substitution to a term that types in an environment consistent with the substitution, then the substituted term types in the empty environment:

Lemma (Mass substitution). *If* $\Gamma \vdash e : \tau$ *and* $\gamma \vDash \Gamma$ *then* $\bullet \vdash e\gamma : \tau$.

Proof. By induction on the length of γ. If empty, then Γ is empty, and the substitution has no effect. Otherwise, $\gamma = \gamma_1[x{:=}v_x]$, where $\Gamma = \Gamma_1,\ x{:}\tau_x$ and $\gamma_1 \vDash \Gamma_1$ and $(\text{SN}\ \tau_x\ v_x)$. Then by the substitution lemma, $\Gamma_1 \vdash e[x{:=}v_x] : \tau$. Then by induction, $\bullet \vdash e[x{:=}v_x]\gamma_1 : \tau$. But that is $e\gamma$.

Lemma (Every typed term is good). *If* $\Gamma \vdash e : \tau$ *and* $\gamma \vDash \Gamma$ *then* $(\text{SN}\ \tau\ e\gamma)$.

Proof. By induction on the typing derivation:

- $\Gamma \vdash x : \Gamma(x)$: Appling γ to x gets us a v such that $(\text{SN}\ \Gamma(x)\ v)$.
- $\Gamma \vdash \texttt{z} : \texttt{nat}$: Since $\texttt{z} = \texttt{z}\gamma$, and $\bullet \vdash \texttt{z} : \texttt{nat}$ and $(\Downarrow \texttt{z})$, we have that $(\text{SN}\ \texttt{nat}\ \texttt{z})$.

- $\Gamma \vdash (\mathtt{s}\ e_1) : \mathtt{nat}$: By inversion, we know that $\Gamma \vdash e_1 : \mathtt{nat}$. Then by induction, we have that $(\mathrm{SN}\ \mathtt{nat}\ e_1\gamma)$. By the definition of NT for `nat`, we have that $e_1\gamma$ types in the empty context and reduces to a natural number. Then $(\mathtt{s}\ e_1)\gamma$ does as well.

- $\Gamma \vdash (\mathtt{ap}\ e_1\ e_2) : \tau$: By inversion, we know that $\Gamma \vdash e_1 : (\to \tau_2\ \tau)$ and $\Gamma \vdash e_2 : \tau_2$. By induction, we know that $(\mathrm{SN}\ (\to \tau_2\ \tau)\ e_1)$ and $(\mathrm{SN}\ \tau_2\ e_2)$. The former means that for any e_{arb} such that $(\mathrm{SN}\ \tau_2\ e_{\mathrm{arb}})$, we have $(\mathrm{SN}\ \tau\ (\mathtt{ap}\ e_1\ e_{\mathrm{arb}}))$. Let e_{arb} be e_2. Then $(\mathrm{SN}\ \tau\ (\mathtt{ap}\ e_1\ e_2))$.

- $\Gamma \vdash (\lambda\ x\ \tau_1\ e_2) : (\to \tau_1\ \tau_2)$: Without loss of generality, let x be fresh for γ. So then that term equals $(\lambda\ x\ \tau_1\ e_1\gamma)$. We need to show that $(\mathrm{SN}\ (\to \tau_1\ \tau_2)\ (\lambda\ x\ \tau_1\ e_2\gamma))$. To show this, we need to show three things:

  - To show $\bullet \vdash (\lambda\ x\ \tau_1\ e_2\gamma) : (\to \tau_1\ \tau_2)$. It suffices to show that $\Gamma \vdash (\lambda\ x\ \tau_1\ e_2) : (\to \tau_1\ \tau_2)$ for some Γ such that $\gamma \vDash \Gamma$, by the mass substitution lemma. That is what we have.
  - To show $(\Downarrow\ (\lambda\ x\ \tau_1\ e_2\gamma))$. This is clear, because it is a value.
  - To show that for any e_1 such that $(\mathrm{SN}\ \tau_1\ e_1)$, $(\mathrm{SN}\ \tau_2\ (\mathtt{ap}\ (\lambda\ x\ \tau_1\ e_2\gamma)\ e_1))$. By the definition of SN, we know that $e_1 \longrightarrow^* v_1$ for some value v_1. Then we can take an additional step, $(\mathtt{ap}\ (\lambda\ x\ \tau_1\ e_2\gamma)\ v_1) \longrightarrow e_2\gamma[x{:=}v_1]$. Because SN is preserved by backward conversion, it suffices to show that this term is SN for τ_2.

    By the lemma that SN is preserved by forward conversion, we know that $(\mathrm{SN}\ \tau_1\ v_1)$.

    So then we can say that $\gamma[x{:=}v_1] \vDash \Gamma, x{:}\tau_1$. Now consider inverting the judgment that $\Gamma \vdash (\lambda\ x\ \tau_1\ e_2) : (\to \tau_1\ \tau_2)$. From this, we know that $\Gamma, x{:}\tau_1 \vdash e_2 : \tau_2$. Then applying the induction hypothesis, we have that $(\mathrm{SN}\ \tau_2\ e_2\gamma[x{:=}v_1])$. This is what we sought to show.

QED.

Strong normalization follows as a corollary.

Exercise 17. *Extend the normalization proof to cover products and/or sums.*

Exercise 18. *Show that λ-`st` with the recursor still enjoys normalization.*

2.6 Adding nontermination

We can add unrestricted recursion to λ-`st` by adding a fixed-point operator. We will also add an `if0` construct, which lets us discriminate between zero and non-zero and extracts the predecessor from a natural. (This is redundant with the recursor, but easier to use. The resulting language is equivalent to the classic PCF.)

The new expression forms are $(\mathtt{fix}\ e)$ and $(\mathtt{if0}\ e\ e\ [x\ e])$:

$$
\begin{array}{rl}
e ::= & \texttt{....} \\
& \mid (\texttt{fix}\ e) \\
& \mid (\texttt{if0}\ e\ e\ [x\ e])
\end{array}
$$

What `fix` does at run time is apply its argument, which must be a function, to the `fix` of itself, thus implementing recursion. What `if0` does is discriminate between `z` and (`s` v), evaluating to its then-branch if zero, and its else-branch if not.

$$E[(\texttt{fix}\ (\lambda\ x\ \tau\ e))] \longrightarrow E[e[x{:=}(\texttt{fix}\ (\lambda\ x\ \tau\ e))]] \quad \text{[fix]}$$

$$E[(\texttt{if0}\ \texttt{z}\ e_z\ [x\ e_s])] \longrightarrow E[e_z] \quad \text{[if0-z]}$$

$$E[(\texttt{if0}\ (\texttt{s}\ v)\ e_z\ [x\ e_s])] \longrightarrow E[e_s[x{:=}v]] \quad \text{[if0-s]}$$

To type `fix`, we type its argument, which must be a function from the desired type to itself. To type `if0`, the natural position must type as `nat`, and the then- and else-branches must have the same type, where the else-branch has the predecessor bound.

$$\frac{\Gamma \vdash e : (\to\ \tau\ \tau)}{\Gamma \vdash (\texttt{fix}\ e) : \tau}\ \text{[fix]}$$

$$\frac{\Gamma \vdash e : \texttt{nat} \qquad \Gamma \vdash e_z : \tau \qquad \Gamma, x{:}\texttt{nat} \vdash e_s : \tau}{\Gamma \vdash (\texttt{if0}\ e\ e_z\ [x\ e_s]) : \tau}\ \text{[if0]}$$

Exercise 19. *Define addition, multiplication, and factorial using* `fix` *and* `if0`*. How does it compare to using the recursor?*

Exercise 20. *Extend type safety for* `fix`*.*

Exercise 21. *Where does the normalization proof break down if we add* `fix`*?*

Exercise 22. *Implement a union-find in STLC with records and vectors.*

3 λ-sub: subtyping with records

3.1 Syntax

Extending STLC with records is straightforward. First, we extend the syntax of types and terms, using ℓ for record field labels:

$$
\begin{array}{rl}
\tau ::= & \texttt{....} \\
& \mid (\texttt{Record}\ [\ell\ \tau]\ ...) \\
e ::= & \texttt{....} \\
& \mid (\texttt{record}\ [\ell\ e]\ ...) \\
& \mid (\texttt{project}\ e\ \ell) \\
\ell, m ::= & \textit{variable-not-otherwise-mentioned}
\end{array}
$$

A record type lists field names with their types; assume the field names are not repeated within a record. A record expression lists field names with expressions whose values will fill the fields. A projection expression projects the value of the named field from a record.

3.2 Dynamic semantics

The dynamics are straightforward. We extend values to include records where every field contains a value. We extend evaluation contexts to evaluate the fields of a record from left to right.

$$
\begin{array}{l}
v ::= \texttt{....} \\
\quad \mid (\texttt{record}\ [\ell\ v]\ ...) \\
E ::= \texttt{....} \\
\quad \mid (\texttt{record}\ [\ell\ v]\ ...\ [\ell\ E]\ [\ell\ e]\ ...)
\end{array}
$$

.

Then we add one reduction rule, for projecting the field from a record:

$$E[(\texttt{project}\ (\texttt{record}\ [\ell_i\ v_i]\ ...\ [\ell\ v]\ [\ell_j\ v_j]\ ...)\ \ell)] \longrightarrow E[v]\ \text{[prj]}$$

3.3 Static semantics

The simplest way to type records is to add one rule for each new expression form and keep the rest of the language the same:

$$\frac{\Gamma \vdash e_i : \tau_i \quad ...}{\Gamma \vdash (\texttt{record}\ [\ell_i\ e_i]\ ...) : (\texttt{Record}\ [\ell_i\ \tau_i]\ ...)}\ \text{[record]}$$

$$\frac{\Gamma \vdash e : (\texttt{Record}\ [\ell_i\ \tau_i]\ ...\ [\ell\ \tau]\ [\ell_j\ \tau_j]\ ...)}{\Gamma \vdash (\texttt{project}\ e\ \ell) : \tau}\ \text{[project]}$$

This works, but it's not as expressive as we might like. Consider a function $(\lambda\ x\ (\texttt{Record}\ [\ell\ \texttt{nat}])\ (\texttt{proje}$ It takes a record of one field ℓ and projects out that field. But is there any reason we shouldn't be able to use this function on a record with *more* fields than ℓ? Subtyping captures that intuition, allowing us to formalize it and prove it sound.

3.3.1 Subtyping

To do this, we define the subtype relation `<:`, which related pairs of types. Intuititively $\tau_1 <: \tau_2$ means that a τ_1 may be used wherever a τ_2 is required.

First, `nat` is a subtype of itself:

$$\frac{}{\texttt{nat} <: \texttt{nat}}\ \text{[nat]}$$

Second, function types are contravariant in the domain and covariant in the arguments:

$$\frac{\begin{array}{c}\tau_{21} <: \tau_{11} \\ \tau_{12} <: \tau_{22}\end{array}}{(\to\ \tau_{11}\ \tau_{12}) <: (\to\ \tau_{21}\ \tau_{22})}\ [\text{arr}]$$

Exercise 23. *Suppose that* `Int` <: `Real`. *Consider the types* (→ `Int` `Int`), (→ `Real` `Real`), (→ `Int` `Real`), *and* (→ `Real` `Int`). *Which of these are subtypes of which others? Does this make sense?*

Finally, records provide subtyping by allowing the forgetting of fields (this is called width subtyping) and by subtyping within individual fields (depth subtyping). We can express this with three rules:

$$\frac{}{(\texttt{Record}) <: (\texttt{Record})}\ [\text{rec-empty}]$$

$$\frac{(\texttt{Record}\ [m_\text{i}\ \tau_\text{i}]\ ...) <: (\texttt{Record}\ [m_\text{j}\ \tau_\text{j}]\ ...)}{(\texttt{Record}\ [\ell\ \tau]\ [m_\text{i}\ \tau_\text{i}]\ ...) <: (\texttt{Record}\ [m_\text{j}\ \tau_\text{j}]\ ...)}\ [\text{rec-width}]$$

$$\frac{\begin{array}{c}\tau_1 <: \tau_\text{r} \\ (\texttt{Record}\ [m_\text{i}\ \tau_\text{i}]\ ...) <: (\texttt{Record}\ [m_\text{j}\ \tau_\text{j}]\ ...\ [m_\text{k}\ \tau_\text{k}]\ ...)\end{array}}{(\texttt{Record}\ [\ell\ \tau_\text{l}]\ [m_\text{i}\ \tau_\text{i}]\ ...) <: (\texttt{Record}\ [m_\text{j}\ \tau_\text{j}]\ ...\ [\ell\ \tau_\text{r}]\ [m_\text{k}\ \tau_\text{k}]\ ...)}\ [\text{rec-depth}]$$

Rule [rec-empty] says that the empty record is a subtype of itself; we need this as a base case. Rule [rec-width] says that supertype records may have fields that are missing from their subtypes. Rule [rec-depth] says that when records have a common member then the types of the fields must be subtypes.

Exercise 24. *Prove that* `<:` *is a preorder, that is, reflexive and transitive.*

The idea of subtyping is that we can apply it everywhere. If we can conclude that $\Gamma \vdash e : \tau_1$ and $\tau_1 <: \tau_2$ then we should be able to conclude that $\Gamma \vdash e : \tau_2$. It's possible to add such a rule, and it works fine theoretically, but because the rule is not *syntax directed*, it can be difficult to implement. In fact, the only place in our current language that we need subtyping is in the application rule, so we replace the STLC application rule with this:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (\to\ \tau_1\ \tau) \\ \Gamma \vdash e_2 : \tau_2 \\ \tau_2 <: \tau_1\end{array}}{\Gamma \vdash (\texttt{ap}\ e_1\ e_2) : \tau}\ [\text{app}]$$

3.3.2 Type safety

Subtyping changes our preservation theorem somewhat, because reduction can cause type refinement. (That is, we learn more type information.) Here is the updated preservation theorem:

Theorem (Preservation). *If* $\bullet \vdash e_1 : \tau_1$ *and* $e_1 \longrightarrow e_2$ *then there exists some* τ_2 *such that* $\bullet \vdash e_2 : \tau_2$ *and* $\tau_2 <: \tau_1$.

Before we can prove it, we update the replacement and substitution lemmas as follows:

Lemma (Replacement). *If* $\Gamma \vdash E[e] : \tau$, *then* $\Gamma \vdash e : \tau_e$ *for some type* τ_e. *Furthermore, for any* e_{new} *such that* $\Gamma \vdash e_{new} : \tau_{new}$ *for* $\tau_{new} <: \tau_e$, $\Gamma \vdash E[e_2] : \tau_{out}$ *for some* τ_{out} *such that* $\tau_{out} <: \tau$.

Proof. By induction on E. The interesting cases are for application:

- If E is $(\mathsf{ap}\ E_1\ e_2)$ then the whole term has a type τ only if there are some types τ_1 and τ_2 such that $\bullet \vdash E_1[e] : (\to \tau_1\ \tau)$ and $\bullet \vdash e_2 : \tau_2$ where $\tau_2 <: \tau_1$. Then by induction, e has a type, and if we replace e with e_{new} having a subtype of that, then $\bullet \vdash E_1[e_{new}] : \tau^\dagger$ for $\tau^\dagger <: (\to \tau_1\ \tau)$. The subtyping relation relates arrows only to other arrows, so $\tau^\dagger = (\to \tau_1^\dagger\ \tau_2^\dagger)$ with $\tau_1 <: \tau_1^\dagger$ and $\tau_2^\dagger <: \tau$. Then by transitivity, $\tau_2 <: \tau_1^\dagger$. This means that we can reform the application $\bullet \vdash (\mathsf{ap}\ E_1[e_{new}]\ e_2) : \tau_2^\dagger$, which has a subtype of τ.

- If E is $(\mathsf{ap}\ e_1\ E_2)$, then the whole term has a type τ only if there are some types τ_1 and τ_2 such that $\bullet \vdash e_1 : (\to \tau_1\ \tau)$ and $\bullet \vdash E_2[e] : \tau_2$ where $\tau_2 <: \tau_1$. Then by induction, e has a type, and if we replace e with e_{new} having a subtype of that, then $\bullet \vdash E_2[e_{new}] : \tau_2^\dagger$ where $\tau_2^\dagger <: \tau_2$. Then by transitivity, $\tau_2^\dagger <: \tau_1$, so we can reform the application having the same type τ.

Lemma (Substitution). *If* $\Gamma \vdash e_1 : \tau_1$ *and* $\Gamma, x{:}\tau_1^\dagger \vdash e_2 : \tau_2$ *where* $\tau_1 <: \tau_1^\dagger$ *then* $\Gamma \vdash e_2[x{:=}e_1] : \tau_2^\dagger$ *for* $\tau_2^\dagger <: \tau_2$.

Proof. By induction on the derivation of the typing of e_2:

- $\Gamma, x{:}\tau_1^\dagger \vdash y : \tau_2$.

  If $x = y$, then $\tau_2 = \tau_1^\dagger$. Then $e_2[x{:=}e_1] = e_1$, which has type τ_1. Let $\tau_2^\dagger$ be τ_1. Then the subtyping holds.

  If $x \neq y$, then $\Gamma, x{:}\tau_1^\dagger \vdash y : \Gamma(y)$, as before the substitution.

- $\Gamma, x{:}\tau_1^\dagger \vdash \mathsf{z} : \mathsf{nat}$, then substitution has no effect and it types in any environment.

- $\Gamma, x{:}\tau_1^\dagger \vdash (\mathsf{s}\ e) : \mathsf{nat}$, then by induction $\Gamma \vdash e[x{:=}e_1] : \mathsf{nat}$, which relates only to nat. Then reapply s.

- $\Gamma, x{:}\tau_1^\dagger \vdash (\lambda\ y\ \tau_{11}\ e) : (\to\ \tau_{11}\ \tau_{12})$, then by inversion we know that $\Gamma, x{:}\tau_1^\dagger, y{:}\tau_{11} \vdash e : \tau_{12}$. Then by the induction hypothesis, $\Gamma, y{:}\tau_{11} \vdash e[x{:=}e_1] : \tau_{12}^\dagger$ for some $\tau_{12}^\dagger <: \tau_{12}$. Then by [abs], $\Gamma \vdash (\lambda\ y\ \tau_{11}\ e)[x{:=}e_1] : (\to\ \tau_{11}\ \tau_{12}^\dagger)$, which is a subtype of $(\to\ \tau_{11}\ \tau_{12})$.

- $\Gamma, x{:}\tau_1^\dagger \vdash (\mathtt{ap}\ e_{11}\ e_{12}) : \tau_1$, then by inversion we know that $\Gamma, x{:}\tau_1^\dagger \vdash e_{11} : (\to\ \tau_{11}\ \tau_1)$ and $\Gamma, x{:}\tau_1^\dagger \vdash e_{12} : \tau_{12}$ where $\tau_{12} <: \tau_{11}$. Then by induction (twice), we have that $\Gamma \vdash e_{11}[x{:=}e_1] : \tau_{11}^\dagger$ where $\tau_{11}^\dagger <: (\to\ \tau_{11}\ \tau_1)$ and that $\Gamma \vdash e_{12}[x{:=}e_1] : \tau_{12}^\dagger$ where $\tau_{12}^\dagger <: \tau_{12}$. By inspection of the subtype relation, the only types related to arrow types are arrow types, so $\tau_{11}^\dagger$ must be an arrow type $(\to\ \tau_{111}^\dagger\ \tau_{112}^\dagger)$ where $\tau_{11} <: \tau_{111}^\dagger$ and $\tau_{112}^\dagger <: \tau_1$. Then by transitivity (twice), $\tau_{12}^\dagger <: \tau_{111}^\dagger$. This means we can apply $(\mathtt{ap}\ e_{11}[x{:=}e_1]\ e_{12}[x{:=}e_1])$ yielding type $\tau_{112}^\dagger$, which is a subtype of τ_1.

- The record construction and projection cases are straightforward.

Proof (of preservation). By cases on the reduction relation. There are two cases:

- If $E[(\mathtt{ap}\ (\lambda\ x\ \tau_1\ e_1)\ v_2)] \longrightarrow E[e_1[x{:=}v_2]]$, then by replacement, $(\mathtt{ap}\ (\lambda\ x\ \tau_1\ e_1)\ v_2)$ has a type, and it suffices to show that this type is preserved. Then by inversion (twice), we know that $\bullet, x{:}\tau_1 \vdash e_1 : \tau$ and $\bullet \vdash v_2 : \tau_2$ where $\tau_2 <: \tau_1$. Then by the substitution lemma, $\bullet \vdash e_1[x{:=}v_2] : \tau^\dagger$ where $\tau^\dagger <: \tau$.

- If $E[(\mathtt{project}\ (\mathtt{record}\ [\ell_i\ v_i]\ ...\ [\ell\ v]\ [\ell_j\ v_j]\ ...)\ \ell)] \longrightarrow E[v]$, this case is straightforward.

QED.

Lemma (Canonical forms).
If $\bullet \vdash v : \tau$, then:

- If τ is $\mathtt{nat}$, then v is either $\mathtt{z}$ or $(\mathtt{s}\ v_1)$.

- If τ is $(\to\ \tau_1\ \tau_2)$, then v has the form $(\lambda\ x\ \tau_1\ e)$.

- If τ is $(\mathtt{Record}\ [\ell\ \tau_1]\ ...)$, then v is a record with at least the fields ℓ.

Proof. By induction on the structure of the typing derivation. Only four rules form values, and those rules correspond to the conditions of the lemma.

Lemma (Progress). *If $\bullet \vdash e_1 : \tau$ then either e_1 is a value or $e_1 \longrightarrow e_2$ for some term e_2.*
Proof. By induction on the typing derivation:

- $\bullet \vdash x : \tau$ is vacuous.

- $\bullet \vdash \mathtt{z} : \mathtt{nat}$ is a value.

- If $\bullet \vdash (\texttt{s}\ e) : \texttt{nat}$ then by inversion, $\bullet \vdash e : \texttt{nat}$. Then by induction, e either takes a step or is a value. If it's a value, then $(\texttt{s}\ e)$ is a value; if it takes a step to $e^\dagger$ then $(\texttt{s}\ e)$ takes a step to $(\texttt{s}\ e^\dagger)$.

- If $\bullet \vdash (\texttt{ap}\ e_{11}\ e_{12}) : \tau$ then by inversion, $\bullet \vdash e_{11} : (\rightarrow \tau_{11}\ \tau)$ and $\bullet \vdash e_{12} : \tau_{12}$ for some types τ_{11} and τ_{12} such that $\tau_{12} <: \tau_{11}$. Then by induction, each of e_{11} and e_{12} either is a value or takes a step. If e_{11} takes a step to $e_{11}^\dagger$, then the whole term takes a step to $(\texttt{ap}\ e_{11}^\dagger\ e_{12})$. If e_{11} is a value v_{11} and e_{12} takes a step to $e_{12}^\dagger$, then the whole term takes a step to $(\texttt{ap}\ v_{11}\ e_{12}^\dagger)$. Otherwise, e_{12} is a value v_{12}. By the canonical forms lemma, v_{11} has the form $(\lambda\ x\ \tau_{11}\ e_{111})$. Then the whole term takes a step to $e_{111}[x{:=}v_{12}]$.

- $\bullet \vdash (\lambda\ x\ \tau_1\ e) : (\rightarrow \tau_1\ \tau_2)$ is a value.

- If $\bullet \vdash (\texttt{record}\ [\ell_\text{i}\ e_\text{i}]\ ...) : (\texttt{Record}\ \ell_\text{i}\ \tau_\text{i})$ then by inversion, $\bullet \vdash e_\text{i} : \tau_\text{i}$ for all e_i. Then by induction, each of those takes a step or is a value. If any takes a step, then the whole term steps by the leftmost e_i to take a step. Otherwise, they are all values, and the whole term is a value.

- If $\bullet \vdash (\texttt{project}\ e\ \ell) : \tau_\text{f}$ then by inversion, e has a record type with a field ℓ having type τ_f. By induction, e either takes a step or is a value v. If it takes a step then the whole term takes a step. If it's a value, then by the canonical forms lemma, it's a value $(\texttt{record}\ [\ell_\text{i}\ v_\text{i}]\ ...\ [\ell\ v_\text{f}]\ [\ell_\text{j}\ v_\text{j}]\ ...)$. Then the whole term takes a step to v_f.

Theorem (Type safety). λ-sub is type safe.
Proof. By progess and preservation.

3.4 Compiling with coercions

To say that $\tau_1 <: \tau_2$ is to say that a τ_1 can be used wherever a τ_2 is expected, but do our run-time representations actually make that true? In some languages yes, but in many languages no. We might not want, for example, for record operations to have to do a (linear) search of field names at run time, but instead to fix the offset at compile time. Such a representation choice is not incompable with subtyping, if we are willing to interpret subtyping as a coercion between potentially different underlying representation types. For example, record type $(\texttt{Record}\ [\texttt{a}\ \tau_\text{a}]\ [\texttt{b}\ \tau_\text{b}]\ [\texttt{c}\ \tau_\text{c}])$ is a subtype of record type $(\texttt{Record}\ [\texttt{a}\ \tau_\text{a}]\ [\texttt{c}\ \tau_\text{c}])$. The former is represented by a 3-element vector containing the values of fields a, b, and c, whereas the rather is represented as a 2-element vector containing the values of fields a and c. We cannot use an instance of the former as the latter directly, but we can coerce it. The coercion between two types in the subtype relationship is witnessed by the function converting the subtype to the supertype.

In particular, the witness to the fact that $\texttt{nat} <: \texttt{nat}$ is the identity function on type nat:

$$\frac{}{\texttt{nat} <: \texttt{nat} \to (\lambda\ \texttt{n}\ \texttt{nat}\ \texttt{n})}\ [\text{nat}]$$

To witness an arrow subtyping, we build a function that applies the witness to the domain coercion to the argument and the witness to the codomain coercion to the result of the coerced function:

$$\frac{\tau_{21} <: \tau_{11} \to e_1 \qquad \tau_{12} <: \tau_{22} \to e_2}{(\to\ \tau_{11}\ \tau_{12}) <: (\to\ \tau_{21}\ \tau_{22}) \to (\lambda\ \texttt{h}\ (\to\ \tau_{11}\ \tau_{12})\ (\lambda\ \texttt{n}\ \tau_{21}\ (\texttt{ap}\ e_2\ (\texttt{ap}\ \texttt{h}\ (\texttt{ap}\ e_1\ \texttt{n})))))}\ [\text{arr}]$$

The empty record is a supertype of itself, by the identity coercion:

$$\frac{}{(\texttt{Record}) <: (\texttt{Record}) \to (\lambda\ \texttt{r}\ (\texttt{Record})\ \texttt{r})}\ [\text{rec-empty}]$$

In subtyping records, we can skip fields and not include them in the supertype:

$$\frac{(\texttt{Record}\ [m_\text{i}\ \tau_\text{i}]\ ...) <: (\texttt{Record}\ [m_\text{j}\ \tau_\text{j}]\ ...) \to e}{(\texttt{Record}\ [\ell\ \tau]\ [m_\text{i}\ \tau_\text{i}]\ ...) <: (\texttt{Record}\ [m_\text{j}\ \tau_\text{j}]\ ...) \to (\lambda\ \texttt{r}\ (\texttt{Record}\ [\ell\ \tau]\ [m_\text{i}\ \tau_\text{i}]\ ...)\ (\texttt{ap}\ e\ (\texttt{record}\ [m_\text{i}\ (\texttt{pr}$$

The depth-subyping record case is hairy. We convert record types by converting one element and then recursively converting the rest of the record, and then reassembling the desired result:

$$\frac{\tau_\text{l} <: \tau_\text{r} \to e_1 \qquad (\texttt{Record}\ [m_\text{i}\ \tau_\text{i}]\ ...) <: (\texttt{Record}\ [m_\text{j}\ \tau_\text{j}]\ ...\ [m_\text{k}\ \tau_\text{k}]\ ...) \to e_2}{\begin{array}{l}(\texttt{Record}\ [\ell\ \tau_\text{l}]\ [m_\text{i}\ \tau_\text{i}]\ ...) <: (\texttt{Record}\ [m_\text{j}\ \tau_\text{j}]\ ...\ [\ell\ \tau_\text{r}]\ [m_\text{k}\ \tau_\text{k}]\ ...) \to (\lambda\ \texttt{r}\ (\texttt{Record}\ [\ell\ \tau_\text{r}]\ [m_\text{i}\ \tau_\text{i}]\ ...) \\ \qquad (\texttt{ap}\ (\lambda\ \texttt{s}\ (\texttt{Record}\ [m_\text{j}\ \tau_\text{j}]\ ...\ [m_\text{k}\ \tau \\ \qquad\quad (\texttt{record}\ [m_\text{j}\ (\texttt{project}\ \texttt{s}\ m \\ \qquad\qquad [\ell\ \ (\texttt{ap}\ e_1\ (\texttt{proje} \\ \qquad\qquad [m_\text{k}\ (\texttt{project}\ \texttt{s} \\ \qquad (\texttt{ap}\ e_2\ (\texttt{record}\ [m_\text{i}\ (\texttt{project}\end{array}}$$

The typing rules now translate from a language with subtyping to a language that doesn't use subtyping. All of the rules except [app] just translate each term by homomorphically translating the subterms:

$$\frac{}{\Gamma \vdash x : \Gamma(x) \rightsquigarrow x}\ [\text{var}]$$

$$\frac{}{\Gamma \vdash \mathtt{z} : \mathtt{nat} \rightsquigarrow \mathtt{z}}\ [\text{zero}]$$

$$\frac{\Gamma \vdash e : \mathtt{nat} \rightsquigarrow e_1}{\Gamma \vdash (\mathtt{s}\ e) : \mathtt{nat} \rightsquigarrow (\mathtt{s}\ e_1)}\ [\text{succ}]$$

$$\frac{\Gamma \vdash e : \tau \rightsquigarrow e_1 \quad \ldots}{\Gamma \vdash (\mathtt{record}\ [\ell\ e]\ \ldots) : (\mathtt{Record}\ [\ell\ \tau]\ \ldots) \rightsquigarrow (\mathtt{record}\ [\ell\ e_1]\ \ldots)}\ [\text{record}]$$

$$\frac{\Gamma \vdash e : (\mathtt{Record}\ [\ell_\mathrm{i}\ \tau_\mathrm{i}]\ \ldots\ [\ell\ \tau]\ [\ell_\mathrm{j}\ \tau_\mathrm{j}]\ \ldots) \rightsquigarrow e_1}{\Gamma \vdash (\mathtt{project}\ e\ \ell) : \tau \rightsquigarrow (\mathtt{project}\ e_1\ \ell)}\ [\text{project}]$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2 \rightsquigarrow e_1}{\Gamma \vdash (\lambda\ x\ \tau_1\ e) : (\rightarrow\ \tau_1\ \tau_2) \rightsquigarrow (\lambda\ x\ \tau_1\ e_1)}\ [\text{abs}]$$

The only interesting rule is [app], which includes subtyping. It generates the coercion for the particular subtyping used, and then applies that to coerce the argument to the function:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (\rightarrow\ \tau_1\ \tau) \rightsquigarrow e_{11} \\ \Gamma \vdash e_2 : \tau_2 \rightsquigarrow e_{21} \\ \tau_2 <: \tau_1 \rightsquigarrow e_\mathrm{c}\end{array}}{\Gamma \vdash (\mathtt{ap}\ e_1\ e_2) : \tau \rightsquigarrow (\mathtt{ap}\ e_{11}\ (\mathtt{ap}\ e_\mathrm{c}\ e_{21}))}\ [\text{app}]$$

Exercise 25. *Define a Point as a record with fields x and y, which are integers. Define a ColorPoint as a Point with an additional field, the color, which is a string. Define a function that takes a Point. Show that your function can be used on a ColorPoint.*

4 The polymorphic lambda calculus λ-2

Suppose we want to write the composition function in the simply-typed lambda calculus. What does it look like? Well, it depends on the types of the functions. We can compose two $(\rightarrow\ \mathtt{nat}\ \mathtt{nat})$ functions with this:

$$(\lambda\ x_1\ (\rightarrow\ \mathtt{nat}\ \mathtt{nat})\ (\lambda\ x_2\ (\rightarrow\ \mathtt{nat}\ \mathtt{nat})\ (\lambda\ y\ \mathtt{nat}\ (\mathtt{ap}\ x_1\ (\mathtt{ap}\ x_2\ y)))))$$

But if the functions have different types, we will need to define a different composition function. This is awkward!

Polymorphism lets us write one composition function that works for any types. We introduce type variables a_i and abstract over them with Λ:

$$(\Lambda\ a_1\ (\Lambda\ a_2\ (\Lambda\ a_3\ (\lambda\ x_1\ (\to\ a_2\ a_3)\ (\lambda\ x_2\ (\to\ a_1\ a_2)\ (\lambda\ y\ a_1\ (\mathtt{ap}\ x_1\ (\mathtt{ap}\ x_2\ y)))))))))$$

We model polymorphism with λ-2, also known as System F.

4.1 Syntax

$$\begin{aligned}
\tau &::= a \\
&\quad \mid (\mathtt{all}\ a\ \tau) \\
&\quad \mid (\to\ \tau\ \tau) \\
e &::= x \\
&\quad \mid (\lambda\ x\ \tau\ e) \\
&\quad \mid (\mathtt{ap}\ e\ e) \\
&\quad \mid (\Lambda\ a\ e) \\
&\quad \mid (\mathtt{Ap}\ e\ \tau) \\
x, y &::= \textit{variable-not-otherwise-mentioned} \\
a, b &::= \textit{variable-not-otherwise-mentioned}
\end{aligned}$$

4.2 Dynamic semantics

To give the dynamic semantics of λ-2, we first define values and the evaluation contexts:

$$\begin{aligned}
v &::= (\lambda\ x\ \tau\ e) \\
&\quad \mid (\Lambda\ a\ e) \\
E &::= [\,] \\
&\quad \mid (\mathtt{ap}\ E\ e) \\
&\quad \mid (\mathtt{ap}\ v\ E) \\
&\quad \mid (\mathtt{Ap}\ E\ \tau)
\end{aligned}$$

Then the reduction relation has two rules, one for value abstraction applications, and one for type abstraction applications:

$$E[(\mathtt{ap}\ (\lambda\ x\ \tau\ e)\ v)] \longrightarrow E[e[x{:=}v]] \ \text{[}\beta\text{-val]}$$

$$E[(\mathtt{Ap}\ (\Lambda\ a\ e)\ \tau)] \longrightarrow E[e[a{:=}\tau]] \ \text{[inst]}$$

The dynamic semantics of λ-2 is given by the evaluation function *eval*:

$\text{eval}(e) = v$ if $e \longrightarrow^* v$

As defined, *eval* could be partial, as with STLC, it is total on well typed terms.

4.3 Static semantics

To give the static semantics of λ-2, we have both type variable environments (which tell us which type variables are in scope) and typing environments (which map variables to their types):

$$\begin{aligned}\Delta &::= \bullet \\ &\mid \Delta, a \\ \Gamma &::= \bullet \\ &\mid \Gamma, x{:}\tau\end{aligned}$$

The main typing judgment relies on two auxiliary judgments. The first tells us whether a type is well formed (which for this language just means closed):

$$\frac{a \in \Delta}{\Delta \vdash a}\,[\text{var}]$$

$$\frac{\Delta \vdash \tau_1 \quad \Delta \vdash \tau_2}{\Delta \vdash (\to\ \tau_1\ \tau_2)}\,[\text{arr}]$$

$$\frac{\Delta, a \vdash \tau}{\Delta \vdash (\texttt{all}\ a\ \tau)}\,[\text{all}]$$

A typing environment is well formed when all the types in it are well formed:

$$\frac{}{\Delta \vdash \bullet}\,[\text{nil}]$$

$$\frac{\Delta \vdash \tau \quad \Delta \vdash \Gamma}{\Delta \vdash \Gamma, x{:}\tau}\,[\text{cons}]$$

Finally, we give the typing judgments for λ-2:

$$\frac{\Delta \vdash \Gamma}{\Delta; \Gamma \vdash x : \Gamma(x)}\,[\text{var}]$$

$$\frac{\Delta \vdash \tau_1 \qquad \Delta; \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Delta; \Gamma \vdash (\lambda\ x\ \tau_1\ e) : (\to\ \tau_1\ \tau_2)}\,[\text{abs}]$$

$$\frac{\Delta; \Gamma \vdash e_1 : (\to\ \tau_2\ \tau) \qquad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash (\texttt{ap}\ e_1\ e_2) : \tau}\,[\text{app}]$$

$$\frac{\Delta, a; \Gamma \vdash e : \tau}{\Delta; \Gamma \vdash (\Lambda\ a\ e) : (\texttt{all}\ a\ \tau)}\,[\text{t-abs}]$$

$$\frac{\Delta \vdash \tau \qquad \Delta; \Gamma \vdash e : (\texttt{all}\ a\ \tau_1)}{\Delta; \Gamma \vdash (\texttt{Ap}\ e\ \tau) : \tau_1[a{:=}\tau]}\,[\text{t-app}]$$

Strictly speaking, every Δ and every type well-formedness premiss can go away and it all still works, with type variables acting like constants, but I like knowing where my free type variables are.

4.4 Church data

λ-2, made of lambdas big and small, may seem to lack much in the way of data, but in fact it is very rich. Alonzo Church showed how to represent natural numbers and datatypes in the untyped lambda calculus. STLC is too weak for those encodings to be meaningful, but they work beautifully in λ-2.

4.4.1 Natural numbers

The natural numbers can be defined as functions that iterate a function. In particular, define type `Nat` to be $(\texttt{all}\ a\ (\to\ (\to\ a\ a)\ (\to\ a\ a)))$, with

- c0 = $(\Lambda\ a\ (\lambda\ \texttt{f}\ (\to\ a\ a)\ (\lambda\ x\ a\ x)))$,
- c1 = $(\Lambda\ a\ (\lambda\ \texttt{f}\ (\to\ a\ a)\ (\lambda\ x\ a\ (\texttt{ap}\ \texttt{f}\ x))))$,

- c2 = (Λ a (λ f (→ a a) (λ x a (ap f (ap f x))))),
- and in general, cn as the function that for any type a, iterates an (→ a a) function n times.

Exercise 26. *Define the successor function* succ *of type* (→ Nat Nat).
Exercise 27. *Define addition, multiplication, and exponentiation.*
Exercise 28 (hard). *Define the predecessor function.*

Once we have predecessor we can define subtraction, equality, less-than, and more, but we need a bit more technology before we can define predecessor.

4.4.2 Booleans

The Booleans can be defined as their own elimination rule. In particular, let type Bool = (all a (→ a (→ a a))). Then define:

- tru = (Λ a (λ x a (λ y a x))), and
- fls = (Λ a (λ x a (λ y a y))).

There's no need for if-then-else—just apply the Boolean.
Exercise 29. *Define not, and, and or.*
Exercise 30. *Define* zero? : (→ Nat Bool).

4.4.3 Products

In general, we can represent datatypes by their elimination principles. For example, we represent the product (∗ τ_1 τ_2) as a function of type (all a (→ τ_1 (→ τ_2 a)) a). That is, a pair of a τ_1 and a τ_2 is a function that, for any type a, you can give it a function that turns a τ_1 and τ_2 into an a, and it gives back that a.

The pair value (pair v_1 v_2) of type (∗ τ_1 τ_2) is represented as (Λ a (λ y (→ τ_1 (→ τ_2 a)) ((y v_1) v_2))).
Exercise 31. *How can we write the selectors fst and snd?*
Exercise 32. *Now predecessor becomes possible. The idea is to apply the Nat to count upward, but using a pair to institute a delay.*
Exercise 33. *Define the recursor from our STLC extension.*

4.4.4 Sums

We can represent the sum type (+ τ_1 τ_2) as its elimination rule (all a (→ (→ τ_1 a) (→ (→ τ_2 a) a))). (Write that out in infix.)
Exercise 34. *How can we construct sum values? How do we use them?*

4.4.5 Lists

We can represent a list using its elimination rule, in particular, the type of its fold. Let $(\texttt{List}\ \tau) = (\texttt{all}\ a\ (\to a\ (\to\ (\to\ \tau\ (\to a\ a))\ a)))$.

Exercise 35. *Define cons.*

Exercise 36. *Define* $\texttt{sum} : (\to (\texttt{List}\ \texttt{Nat})\ \texttt{Nat})$.

Exercise 37. *Define empty?, first, and rest.*

4.4.6 Existentials

We can encode existential types in λ-2. An existential type lets us hide part of the representation of a type, and then safely use it without revealing the representation.

Define $(\texttt{ex}\ a\ \tau)$ to be $(\texttt{all}\ b\ (\to (\texttt{all}\ a\ (\to \tau\ b))\ b))$.

To create an existential, we must have a value v_{rep} that has some actual type τ_{act}, but we wish to view as type $(\texttt{ex}\ a\ \tau)$. There must be some type τ_{rep} such that $\tau[a{:=}\tau_{\text{rep}}] = \tau_{\text{act}}$. Then to create the existential, we write:

$$(\Lambda\ b\ (\lambda\ y\ (\texttt{all}\ a\ (\to \tau\ b))\ (\texttt{ap}\ (\texttt{Ap}\ y\ \tau_{\text{rep}})\ v_{\text{rep}})))$$

To use the existential, apply it!

For example, suppose we want to create a value of type $(\texttt{ex}\ a\ (*\ a\ (*\ (\to a\ a)\ (\to a\ \texttt{Nat}))))$ that lets us work abstractly with the naturals. (In particular, we represent a triple containing zero of abstract type, the successor function of abstract type, and a projection that reveals the underlying natural.) We could pack that up as:

$$\begin{aligned}\text{Counter} = \quad & (\Lambda\ b\ (\lambda\ y\ (\texttt{all}\ a\ (\to (*\ a\ (*\ (\to a\ a)\ (\to a\ \texttt{Nat})))\ b)) \\ & \quad (\texttt{ap}\ (\texttt{Ap}\ (y\ (*\ \texttt{Nat}\ (*\ (\to \texttt{Nat}\ \texttt{Nat})\ (\to \texttt{Nat}\ \texttt{Nat})))))) \\ & \quad\quad (\texttt{pair}\ \texttt{c0}\ (\texttt{pair}\ \texttt{succ}\ (\lambda\ x\ \texttt{Nat}\ x))))))\end{aligned}$$

Then to count to 2, we might write:

$$\begin{aligned}& (\texttt{ap}\ (\texttt{Ap}\ \texttt{Counter}\ \texttt{Nat}) \\ & \quad (\Lambda\ a\ (\lambda\ \texttt{counter}\ (*\ a\ (*\ (\to a\ a)\ (\to a\ \texttt{Nat})))) \\ & \quad\quad ((\texttt{snd}\ (\texttt{snd}\ \texttt{counter})) \\ & \quad\quad\ ((\texttt{fst}\ (\texttt{snd}\ \texttt{counter})) \\ & \quad\quad\quad ((\texttt{fst}\ (\texttt{snd}\ \texttt{counter})) \\ & \quad\quad\quad\ (\texttt{fst}\ \texttt{counter})))))))\end{aligned}$$

This is the basis of abstract types as the appear in module and object systems. Of course, it gets a bit easier to read if we add record types and make existentials primitive or, better yet, hidden.

Exercise 38. *Add existentials to* λ-2 *without encoding as universals. In particular, you will need forms for packing and unpacking whose statics and dynamics agree with the encoding above.*

Exercise 39. *Prove type safety for* λ-2.

Exercise 40 (Very difficult). *Prove normalization for* λ-2.

Consider this alternate definition of Counter:

$$\begin{array}{rl} & (\Lambda\, b\, (\lambda\, y\, (\texttt{all}\, a\, (\to (* a\, (* (\to a\, a)\, (\to a\, \texttt{Nat})))\, b)) \\ & \qquad (\texttt{ap}\, (\texttt{Ap}\, (y\, (* \texttt{Nat}\, (* (\to \texttt{Nat}\, \texttt{Nat})\, (\to \texttt{Nat}\, \texttt{Nat})))))) \\ \text{Counter} = & \qquad\quad (\texttt{pair}\, \texttt{c1}\, (\texttt{pair}\, \texttt{succ}\, \texttt{pred}))))) \end{array}$$

It should be indistinguishable from the original definition in all contexts.

Exercise 41. *Can we prove it?*

Exercise 42. *Write a generic sorting function that takes a vector and a comparison function and sorts the vector in place.*

5 The higher-order lambda calculus λ-ω

5.1 Syntax

The higher-order calculus λ-ω extends System F with type operators, that is, functions at the type level:

$$\begin{array}{rcl} \tau & ::= & a \\ & | & (\to\, \tau\, \tau) \\ & | & (\texttt{all}\, a\, \kappa\, \tau) \\ & | & (\lambda\, a\, \kappa\, \tau) \\ & | & (\texttt{ap}\, \tau\, \tau) \end{array}$$

In addition to type variables, functions, and universal types (all from System F), we now have abstraction of types over types ($(\lambda\, a\, \kappa\, \tau)$) and application of types to types ($(\texttt{ap}\, \tau_1\, \tau_2)$).

In order to avoid errors at the type computation level, we now have "types of types," known as *kinds*:

$$\begin{array}{rcl} \kappa & ::= & * \\ & | & (\Rightarrow \kappa\, \kappa) \end{array}$$

There are two kind constructors. Kind $*$ is the kind of proper types, that is, types that can be the types of terms. Kind $(\Rightarrow \kappa_1\, \kappa_2)$ is the kind of a type operator that consumes types of kind κ_1 and produces types of kind κ_2. For example, the type constructor `List` has kind $(\Rightarrow * *)$—apply it to a term with a type of kind $*$, like `Int`, and you get `(List Int)`, a type of kind $*$, back.

Note that types in λ-ω are a copy of the terms of STLC in the type level, with one base kind $*$.

Terms in λ-ω are the same as terms in System F, except that the type variable in $(\Lambda\, a\, \kappa\, e)$ is decorated with a kind:

$$
\begin{array}{rl}
e ::= & x \\
| & (\lambda\ x\ \tau\ e) \\
| & (\mathtt{ap}\ e\ e) \\
| & (\Lambda\ a\ \kappa\ e) \\
| & (\mathtt{Ap}\ e\ \tau)
\end{array}
$$

In fact, all three forms that bind type variables decorate them with a kind.

5.2 Dynamic semantics

The dynamics for λ-ω are straightforward. Values and evaluation contexts are as in System F:

$$
\begin{array}{rl}
v ::= & (\lambda\ x\ \tau\ e) \\
| & (\Lambda\ a\ \kappa\ e) \\
E ::= & [\,] \\
| & (\mathtt{ap}\ E\ e) \\
| & (\mathtt{ap}\ v\ E) \\
| & (\mathtt{Ap}\ E\ \tau)
\end{array}
$$

There are two reduction rules, as in System F, for applications of λs and instantiations of Λs:

$$E[(\mathtt{ap}\ (\lambda\ x\ \tau\ e)\ v)] \longrightarrow E[e[x{:=}v]] \text{ [}\beta\text{-val]}$$

$$E[(\mathtt{Ap}\ (\Lambda\ a\ \kappa\ e)\ \tau)] \longrightarrow E[e[a{:=}\tau]] \text{ [inst]}$$

5.3 Static semantics

λ-ω's type system is more complex than what we've seen before for two reasons: we are now doing computation at the type level and we now need to kind-check types.

Kind checking tells us that a type is well-formed in a context that binds type variables to their kinds. However, we will use the same contexts for type checking, so we will bind term variables to types and type variables to kind in the same context:

$$
\begin{array}{rl}
\Gamma ::= & \bullet \\
| & \Gamma, x{:}\tau \\
| & \Gamma, a{:}\kappa
\end{array}
$$

Then the kinding rules are like STLC at the type level:

$$\frac{}{\Gamma \vdash a :: \Gamma(a)}\,[\text{var}]$$

$$\frac{\Gamma \vdash \tau_1 :: *\qquad \Gamma \vdash \tau_2 :: *}{\Gamma \vdash (\to\ \tau_1\ \tau_2) :: *}\,[\text{arr}]$$

$$\frac{\Gamma, a{:}\kappa \vdash \tau :: *}{\Gamma \vdash (\texttt{all}\ a\ \kappa\ \tau) :: *}\,[\text{all}]$$

$$\frac{\Gamma, a{:}\kappa_1 \vdash \tau :: \kappa_2}{\Gamma \vdash (\lambda\ a\ \kappa_1\ \tau) :: (\Rightarrow\ \kappa_1\ \kappa_2)}\,[\text{abs}]$$

$$\frac{\Gamma \vdash \tau_1 :: (\Rightarrow\ \kappa_2\ \kappa)\qquad \Gamma \vdash \tau_2 :: \kappa_2}{\Gamma \vdash (\texttt{ap}\ \tau_1\ \tau_2) :: \kappa}\,[\text{app}]$$

Note that $\to$ and `all` form types of kind $*$ from types of kind $*$.

In order to type check terms, we need a notion of type equivalence that includes computation in types. The usual theoretical way to do this is to define equivalence as a congruence that includes beta reduction:

$$\frac{}{\tau \equiv \tau}\,[\text{refl}]$$

$$\frac{\tau_2 \equiv \tau_1}{\tau_1 \equiv \tau_2}\,[\text{sym}]$$

$$\frac{\tau_1 \equiv \tau_2 \quad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}\,[\text{trans}]$$

$$\frac{\tau_1 \equiv \tau_3 \quad \tau_2 \equiv \tau_4}{(\to\ \tau_1\ \tau_2) \equiv (\to\ \tau_3\ \tau_4)}\,[\text{arr}]$$

$$\frac{\tau_1 \equiv \tau_2}{(\texttt{all}\ a\ \kappa\ \tau_1) \equiv (\texttt{all}\ a\ \kappa\ \tau_2)}\,[\text{all}]$$

$$\frac{\tau_1 \equiv \tau_2}{(\lambda\ a\ \kappa\ \tau_1) \equiv (\lambda\ a\ \kappa\ \tau_2)}\,[\text{abs}]$$

$$\frac{\tau_1 \equiv \tau_3 \quad \tau_2 \equiv \tau_4}{(\texttt{ap}\ \tau_1\ \tau_2) \equiv (\texttt{ap}\ \tau_3\ \tau_4)}\,[\text{app}]$$

$$\frac{}{(\texttt{ap}\ (\lambda\ a\ \kappa\ \tau_1)\ \tau_2) \equiv \tau_1[a{:=}\tau_2]}\,[\beta]$$

Then in the type rules, we would check that types that in other systems need to be equal are equivalent. However, to type check algorithmically, we instead define type computation in terms of type evaluation contexts and then β reduction over types:

$$\begin{aligned} TE ::= {} & [\,] \\ & \mid (\to\ TE\ \tau) \\ & \mid (\to\ \tau\ TE) \\ & \mid (\texttt{all}\ a\ \kappa\ TE) \\ & \mid (\lambda\ a\ \kappa\ TE) \\ & \mid (\texttt{ap}\ TE\ \tau) \\ & \mid (\texttt{ap}\ \tau\ TE) \end{aligned}$$

$$TE[(\texttt{ap}\ (\lambda\ a\ \kappa\ \tau_1)\ \tau_2)] \longrightarrow TE[\tau_1[a{:=}\tau_2]]\ [\beta\text{-type}]$$

Then to compare two types, we fully reduce both of them and compare for equality. This happens in the rules for application and instantiation.

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\,[\text{var}]$$

$$\frac{\Gamma \vdash \tau_1 :: *\qquad \Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda\ x\ \tau_1\ e) : (\to\ \tau_1\ \tau_2)}\,[\text{abs}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1\qquad \Gamma \vdash e_2 : \tau_2\qquad \tau_1 \longrightarrow^* (\to\ \tau_{2^*}\ \tau)\qquad \tau_2 \longrightarrow^* \tau_{2^*}}{\Gamma \vdash (\mathtt{ap}\ e_1\ e_2) : \tau}\,[\text{app}]$$

$$\frac{\Gamma, a{:}\kappa \vdash e : \tau}{\Gamma \vdash (\Lambda\ a\ \kappa\ e) : (\mathtt{all}\ a\ \kappa\ \tau)}\,[\text{tabs}]$$

$$\frac{\Gamma \vdash \tau :: \kappa\qquad \Gamma \vdash e : \tau_1\qquad \tau_1 \longrightarrow^* (\mathtt{all}\ a\ \kappa\ \tau_{1^*})}{\Gamma \vdash (\mathtt{Ap}\ e\ \tau) : \tau_{1^*}[a{:=}\tau]}\,[\text{tapp}]$$

6 ML type inference

6.1 STLC revisited

We revisit the simply-typed λ calculus, but with several twists:

- We remove base types such as `nat` (for now) in favor of uninterpreted type variables a.
- We add `let` expressions.
- Most importantly, we leave types *implicit*.

Here is the syntax of the resulting system:

$$
\begin{aligned}
e &::= x \\
&\quad \mid (\lambda\ x\ e) \\
&\quad \mid (\texttt{ap}\ e\ e) \\
&\quad \mid (\texttt{let}\ x\ e\ e) \\
\tau &::= a \\
&\quad \mid (\rightarrow\ \tau\ \tau) \\
x, y &::= \textit{variable-not-otherwise-mentioned} \\
a, b &::= \textit{variable-not-otherwise-mentioned}
\end{aligned}
$$

6.1.1 Dynamic semantics

The only values are λ abstractions:

$$
\begin{aligned}
v &::= (\lambda\ x\ e) \\
E &::= [\,] \\
&\quad \mid (\texttt{ap}\ E\ e) \\
&\quad \mid (\texttt{ap}\ v\ E) \\
&\quad \mid (\texttt{let}\ x\ E\ e)
\end{aligned}
$$

The dynamic semantics of this language includes two reduction rules:

$$E[(\texttt{ap}\ (\lambda\ x\ e)\ v)] \longrightarrow E[e[x{:=}v]] \text{ [}\beta\text{-val]}$$

$$E[(\texttt{let}\ x\ v\ e)] \longrightarrow E[e[x{:=}v]] \text{ [let]}$$

Note this means that dynamically we can consider $(\texttt{let}\ x\ e_1\ e_2)$ as syntactic sugar for $(\texttt{ap}\ (\lambda\ x\ e_2)\ e_1)$.

6.1.2 Static semantics

The static semantics should be familiar from the simply-typed lambda calculus, since it's the same but for one thing: the rule for λ has to "guess" the domain type τ_1.

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\,[\text{var}]$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda\ x\ e) : (\to\ \tau_1\ \tau_2)}\,[\text{abs}]$$

$$\frac{\Gamma \vdash e_1 : (\to\ \tau_2\ \tau) \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{ap}\ e_1\ e_2) : \tau}\,[\text{app}]$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma, x{:}\tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{let}\ x\ e_1\ e_2) : \tau_2}\,[\text{let}]$$

Here's how it works: To type a λ expression, choose any type—made out of type variables and arrows—for the parameter that lets you type the body. For example, these are all valid judgments for the identity function:

- $\bullet \vdash (\lambda\ x\ x) : (\to\ a\ a)$
- $\bullet \vdash (\lambda\ x\ x) : (\to\ (\to\ a\ a)\ (\to\ a\ a))$
- $\bullet \vdash (\lambda\ x\ x) : (\to\ (\to\ a\ b)\ (\to\ a\ b))$

Whatever type it's given, it returns the same type. How a variable is used may constrain its type. For example, to type $(\lambda\ x\ (\lambda\ y\ (\texttt{ap}\ y\ x)))$ we have to guess types for x and y such that y can be applied to x. Suppose we guess a for x. Then we are faced with choosing a type for y that can be applied to that, like say $(\to\ a\ b)$. Then we get the type $(\to\ a\ (\to\ (\to\ a\ b)\ b))$ for the whole term. Those aren't the only types we could have chosen, however, For example, we could choose $(\to\ a\ b)$ for x; then y could be any arrow type $(\to\ (\to\ a\ b)\ \tau)$ for any type τ.

Exercise 43. *Find types for these terms:*

- $(\lambda\ x\ (\lambda\ y\ x))$
- $(\lambda\ \texttt{f}\ (\lambda\ \texttt{g}\ (\lambda\ x\ (\texttt{ap}\ \texttt{f}\ (\texttt{ap}\ \texttt{g}\ x)))))$
- $(\texttt{let}\ \texttt{f}\ (\lambda\ x\ x)\ (\texttt{ap}\ \texttt{f}\ (\lambda\ x\ (\lambda\ y\ x))))$

Exercise 44. *Find a closed term that has no type. What is the only cause of type errors in this system?*

6.1.3 Adding a base type

Let's make things a bit more interesting by introducing the potential for more type errors. We add Booleans to the language. We add `true` and `false`, and `if` expressions for distinguishing between the two:

$$
\begin{array}{rl}
e ::= & \texttt{....} \\
& \mid \texttt{true} \\
& \mid \texttt{false} \\
& \mid (\texttt{if}\ e\ e\ e) \\
v ::= & \texttt{....} \\
& \mid \texttt{true} \\
& \mid \texttt{false} \\
E ::= & \texttt{....} \\
& \mid (\texttt{if}\ E\ e\ e) \\
\tau ::= & \texttt{....} \\
& \mid \texttt{bool}
\end{array}
$$

There are two new reduction rules, for reducing `if` in the true case and in the false case:

$$E[(\texttt{if true}\ e_1\ e_2)] \longrightarrow E[e_1] \text{ [if-true]}$$

$$E[(\texttt{if false}\ e_1\ e_2)] \longrightarrow E[e_2] \text{ [if-false]}$$

The type rules assign type `bool` to both Boolean expressions. An `if` expression types if the condition is a `bool` and if the branches have the same type as each other:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}}\ [\text{true}]$$

$$\frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}\ [\text{false}]$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{if}\ e_1\ e_2\ e_3) : \tau}\ [\text{if}]$$

Exercise 45. *Extend* `λ-ml` *with one of these features: products, sums, numbers, records, recursion, references.*

Exercise 46. *Show that the term* `(let f (λ x x) (if true (ap f true) (ap (ap f (λ y y)) false)))` *has no type.*

6.1.4 Introducing `let` polymorphism

By why not? The term reduces to a Boolean, so shouldn't it have type `bool`? It doesn't because `f` is used two different ways. When applied to `true`, it needs to have type (→ `bool` `bool`), but when applied to (λ y y) it needs to have type (→ (→ `bool` `bool`) (→ `bool` `bool`)) (because the result of that application is applied to a `bool`).

However, if we were to reduce the `let`, we would get (`if` `true` (`ap` (λ x x) `true`) (`ap` (`ap` (λ x x) (λ y y
which types fine.

So this suggests a different way to type (`let` x e_1 e_2): copy e_1 into each occurrence of x in e_2:

$$\frac{\Gamma \vdash e_2[x{:=}e_1] : \tau_2}{\Gamma \vdash (\texttt{let}\ x\ e_1\ e_2) : \tau_2}\ \text{[let-copy/wrong]}$$

with this rule, the example from the exercise types correctly. However, other things that shouldn't type also type. In particular, a term like (`let` `f` (`ap` `true` `true`) `true`) has type `bool`, even though the subterm (`ap` `true` `true`) has no type. To remedy this, we ensure that e_1 has a type, even though we don't restrict it to have that particular type in e_2:

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2[x{:=}e_1] : \tau_2}{\Gamma \vdash (\texttt{let}\ x\ e_1\ e_2) : \tau_2}\ \text{[let-copy]}$$

This works! But it has two drawbacks:

- Now we are typechecking term e_1 multiple times, once for each occurrence of x in e_2. We can actually construct a family of terms that grow exponentially as a result of this copying.
- In a real programming system, we want to be able to give a type to x because they often allow bindings with open scopes: (`let` x e_1) for the future. This only makes sense if we can say what type x has. This is essential for separate or incremental compilation.

6.2 Type schemes in λ-`ml`

In the exercise above, x is used at two different types: (→ `bool` `bool`) and (→ (→ `bool` `bool`) (→ `bool` `bool`)
In fact, if we consider it carefully, it's safe to use x on an argument of any type τ, and we get that same τ back. So we could say that x has the *type scheme* (→ a a) for all types a.

We will write type schemes with the universal quantifier to indicate which type variables are free to be instantiated in the scheme:

$$\begin{array}{rl}\sigma ::= & \tau \\ | & (\texttt{all}\ a\ \sigma)\end{array}$$

Note that types in λ-`ml` (and real ML) do not contain `all` like System F types do—`all` just happens in the front of type schemes. (This is called a prenex type.) This is key to making type inference possible, since we cannot in general infer System F types.

6.3 Statics

For λ-`ml`'s statics, we allow type environments Γ to bind variables to type schemes:

$$\begin{array}{rl}\Gamma ::= & \bullet \\ | & \Gamma, x{:}\sigma\end{array}$$

6.3.1 The logical type system

We first give a logical type system, which says which terms has a type but is not very much help in finding the type. The four rules for the four expression forms in our language are nearly the same as before; the only difference is in rule [let-poly], allows the bound variable to have a type scheme instead of a mere monomorphic type ("monotype"):

$$\frac{}{\Gamma \vdash x : \Gamma(x)}\ [\text{var}]$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda\ x\ e) : (\rightarrow\ \tau_1\ \tau_2)}\ [\text{abs}]$$

$$\frac{\Gamma \vdash e_1 : (\rightarrow\ \tau_2\ \tau) \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{ap}\ e_1\ e_2) : \tau}\ [\text{app}]$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma, x{:}\sigma_1 \vdash e_2 : \sigma_2}{\Gamma \vdash (\texttt{let}\ x\ e_1\ e_2) : \sigma_2}\ [\text{let-poly}]$$

On the other hand, notice that the domain type inferred for λ is still required to be a monotype.

There are two initial rules, which are not syntax directed, but which are used to instantiate type schemes to types and generalize types to type schemes. To instantiate a type scheme, we can replace its bound variable with any type whatsoever:

$$\frac{\Gamma \vdash e : (\texttt{all}\ a\ \sigma)}{\Gamma \vdash e : \sigma[a{:=}\tau]}\,[\text{inst}]$$

Finally, we can generalize any type variable that is not free in the environment Γ:

$$\frac{\begin{array}{c}\Gamma \vdash e : \sigma \\ a = \#\ \Gamma\end{array}}{\Gamma \vdash e : (\texttt{all}\ a\ \sigma)}\,[\text{gen}]$$

This is because any type variable that is not mentioned in Γ is unconstrained, but type variables that are mentioned might have requirements imposed on them.

Exercise 47. *Derive a type for* (`let f` (λ x x) (`if true` (`ap f true`) (`ap` (`ap f` (λ y y)) `false`))).

Exercise 48. *What types can you derive for* (λ x (λ y (`ap` x y)))*? What do they have in common? What type scheme instantiates to all of them?*

6.3.2 The syntax-directed type system

The system presented above allows generalization and instantiation anywhere, but in fact, these rules are only useful in certain places, because we do not allow polymorphic type schemes as the domains of functions. The only place that generalization is useful is when binding the right-hand side of a `let`, and instantiation is only useful when we lookup a variable with a type scheme and want to use it. It's not necessary to apply the rules anywhere else, so we can combine rule `var` with rule `inst` into a new rule [var-inst]:

$$\frac{\Gamma(x) > \tau}{\Gamma \vdash x : \tau}\,[\text{var-inst}]$$

The rule uses a relation $>$ for instantiating a type scheme into an arbitrary monotype:

$$\frac{}{\tau > \tau}\,[\text{mono}]$$

$$\frac{\sigma[a{:=}\tau_1] > \tau}{(\texttt{all}\ a\ \sigma) > \tau}\,[\text{all}]$$

Similarly, we combine rule [let-poly] with rule [gen] to get rule [let-gen], which generalizes the right-hand side of the `let`:

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \\ \sigma_1 = \text{gen}[\![\text{ftv}(\tau_1) \setminus \text{ftv}(\Gamma),\ \tau_1]\!] \\ \Gamma,\ x{:}\sigma_1 \vdash e_2 : \tau\end{array}}{\Gamma \vdash (\texttt{let}\ x\ e_1\ e_2) : \tau}\,[\text{let-gen}]$$

The metafunction gen simply generalizes the type into a type scheme with the given bound variables:

$$\begin{aligned}
&\text{gen} : (a \ ...)\ \tau \to \sigma \\
&\text{gen}[\![(), \tau]\!] &&= \tau \\
&\text{gen}[\![(a\ a_i\ ...), \tau]\!] &&= (\texttt{all}\ a\ \text{gen}[\![(a_i\ ...), \tau]\!])
\end{aligned}$$

The syntax-directed type system presented in this section admits exactly the same programs as the logical type system from the previous section. Unlike the logical system, it tells us exactly when we need to apply instantiation and generalization. But it still does not tell us what types to instantiate type schemes to in rule [var-inst], and it does not tell us what type to use for the domain in rule `abs`. To actually type terms, we will need an algorithm.

Exercise 49. *Extend the syntax-directed type system for your extended language.*

6.4 Type inference algorithm

The type inference rules presented above yield many possible typings for terms. For example, the identity function might have type $(\to a\ a)$ or $(\to \texttt{bool}\ \texttt{bool})$ or $(\to (\to \texttt{bool}\ a)\ (\to \texttt{bool}\ a))$ and so on. The most general type, however, is $(\to a\ a)$, since all other terms are instances of that. The algorithm presented in this section always finds the most general type (if a typing exists).

6.4.1 Unification

To perform type inference, we need a concept of a type substitution, which substitutes some monotypes for type variables:

$$\begin{aligned}
S ::= &\ \bullet \\
\mid &\ S[a{:=}\tau]
\end{aligned}$$

Exercise 50. *Give a type substitution S such that $S(\to a\ (\to a\ b)) = (\to \texttt{bool}\ (\to \texttt{bool}\ (\to \texttt{c}\ \texttt{c})))$*

Type inference will hinge on the idea of unification: Given two types τ_1 and τ_2, is there a substitution S that makes them equal: $S\tau_1 = S\tau_2$? We will use this, for example, if we want to apply a function with type $(\to \tau_1\ \tau)$ to argument of type τ_2. Type variables represent unknown parts of the types at question, and unification tells us if the types might be made, by filling in missing information, the same.

The unification procedure takes two types and either produces the unifying substitution, or fails. In particular, any variable unifies with itself, producing the empty substitution:

$$\frac{}{a \sim a \rightsquigarrow \bullet}\ [\text{var-same}]$$

A variable a unifies with any other type τ by extending the substitution to map a to τ, *provided that a is not free in τ*:

$$\frac{a \notin \text{ftv}(\tau)}{a \sim \tau \rightarrow \bullet[a{:=}\tau]}\,[\text{var-left}]$$

(If $a \in \text{ftv}(\tau)$ then they won't unify and we have a type error. This is the only kind of type error in a system without base types.)

If a variable appears on the right, we swap it to the left and unify:

$$\frac{\tau \text{ is not a type variable} \qquad a \sim \tau \rightarrow S}{\tau \sim a \rightarrow S}\,[\text{var-right}]$$

Type `bool` unifies with itself:

$$\frac{}{\texttt{bool} \sim \texttt{bool} \rightarrow \bullet}\,[\text{bool}]$$

Finally, two types unify if their domains unify and their codomains unify:

$$\frac{\tau_{11} \sim \tau_{21} \rightarrow S_1 \qquad S_1\tau_{12} \sim S_1\tau_{22} \rightarrow S_2}{(\rightarrow \tau_{11}\ \tau_{12}) \sim (\rightarrow \tau_{21}\ \tau_{22}) \rightarrow S_2 \circ S_1}\,[\text{arr}]$$

Note that after $\tau_{11} \sim \tau_{21} \rightarrow S$ produces a substitution S, we apply that substitution to τ_{12} and τ_{22} before unifying, in order to propagate the information that we've collected. Further, the result of unifying the arrow types is the composition of the substitutions S_1 and S_2. In general, when we work with substitutions, we will see that we accumulate and compose them.

Unification has an interesting property: It finds the *most general* unifier for any pair of unifiable types. A substitution S is more general than a substitution S_1 if there exists a substitution S_2 such that $S_1 = S_2 \circ S$. That is, if S_1 does more substitution than S. So suppose that τ_1 and τ_2 are two types, and suppose that $S_1\tau_1 = S_1\tau_2$. Then the S given by unifying τ_1 and τ_2 will be more general than (or equal to) S_1.

6.4.2 Algorithm W

Now we are prepared to give the actual inference algorithm. It uses one metafunction, inst, which takes a type scheme and instantiates its bound variables with fresh type variables:

$$\begin{aligned}
&\text{inst} : (a\ ...)\ \sigma \rightarrow \tau \\
&\text{inst}[\![(a\ ...),\ \tau]\!] &&= \tau \\
&\text{inst}[\![(a\ ...),\ (\texttt{all}\ b\ \sigma)]\!] &&= \text{inst}[\![(a\ ...\ b_1),\ \sigma[b{:=}b_1]]\!] \\
&\quad \text{where } b_1 = \#\ (a\ ...)
\end{aligned}$$

The inst metafunction is given a list of type variables to avoid.

Then we have the inference algorithm itself. The algorithm takes as parameters a type environment and a term to type; if it succeeds, it returns both a type for the term and a substitution making it so. Let's start with the simplest rules.

To type check a Boolean, we return `bool` with the empty substitution:

$$\frac{}{\mathrm{W}(\Gamma;\ \texttt{true}) = (\bullet;\ \texttt{bool})}\ [\text{true}]$$

$$\frac{}{\mathrm{W}(\Gamma;\ \texttt{false}) = (\bullet;\ \texttt{bool})}\ [\text{false}]$$

To type check a variable, we look up the variable in the environment and instantiate the resulting type scheme with fresh type variables:

$$\frac{\tau = \mathrm{inst}[\![\mathrm{ftv}(\Gamma),\ \Gamma(x)]\!]}{\mathrm{W}(\Gamma;\ x) = (\bullet;\ \tau)}\ [\text{var}]$$

To type check a λ abstraction, we create a fresh type variable a to use as its domain type, and we type check the body assuming that the formal parameter has that type a:

$$\frac{a = \#\ \Gamma \qquad \mathrm{W}(\Gamma,\ x{:}a;\ e) = (S;\ \tau)}{\mathrm{W}(\Gamma;\ (\lambda\ x\ e)) = (S;\ (\rightarrow\ S a\ \tau))}\ [\text{abs}]$$

Note that while we "guess" a type variable a for the domain, it will be refined (via unification) based on how it's used in the body.

To type check an application is more involved than the other rules we have seen, but the key operation is unifying the domain type of the operator with the type of the operand. First, we infer types for e_1 and e_2, using substitution S_1 (the result of typing e_1) for typing e_2. Then, we get a fresh type variable a to stand for the result type of the application. We unify the type of the operator, $S_2\tau_1$ with the type we need it to have, $(\rightarrow\ \tau_2\ a)$, yielding substitution S_3. Then the composition of the three substitutions, along with result type $S_3 a$, is our result:

$$\frac{\begin{array}{c}\mathrm{W}(\Gamma;\ e_1) = (S_1;\ \tau_1) \\ \mathrm{W}(S_1\Gamma;\ e_2) = (S_2;\ \tau_2) \\ a = \#\ (\Gamma\ S_1\ S_2\ \tau_1\ \tau_2) \\ S_2\tau_1 \sim (\rightarrow\ \tau_2\ a) \rightarrow S_3\end{array}}{\mathrm{W}(\Gamma;\ (\texttt{ap}\ e_1\ e_2)) = (S_3 \circ S_2 \circ S_1;\ S_3 a)}\ [\text{app}]$$

Note again how the substitutions are threaded through: Substitutions must be applied to any types or environments that existed before that substitution was created.

For the `let` rule, we first infer a type for e_1, and we generalize that type with respect to the (updated-by-substitution) type environment $S_1\Gamma$. Then we bind the resulting type scheme in the environment for type checking e_2:

$$\frac{\begin{array}{c}\mathrm{W}(\Gamma;\, e_1) = (S_1;\, \tau_1) \\ \sigma = \mathrm{gen}[\![\mathrm{ftv}(\tau_1) \setminus \mathrm{ftv}(S_1\Gamma),\, \tau_1]\!] \\ \mathrm{W}(S_1\Gamma,\, x{:}\sigma;\, e_2) = (S_2;\, \tau_2)\end{array}}{\mathrm{W}(\Gamma;\, (\texttt{let}\ x\ e_1\ e_2)) = (S_2 \circ S_1;\, \tau_2)}\ [\mathrm{let}]$$

Finally, the rule for `if` works by first type checking its three subterms, threading the substitutions through. Then it needs to unify the type of e_1 with `bool`, and it needs to unify the types of e_2 and e_3 with each other. Either of those is then the type of the result.

$$\frac{\begin{array}{c}\mathrm{W}(\Gamma;\, e_1) = (S_1;\, \tau_1) \\ \mathrm{W}(S_1\Gamma;\, e_2) = (S_2;\, \tau_2) \\ \mathrm{W}(S_2 \circ S_1\Gamma;\, e_3) = (S_3;\, \tau_3) \\ S_3 \circ S_2\tau_1 \sim \texttt{bool} \to S_4 \\ S_4 \circ S_3\tau_2 \sim S_4\tau_3 \to S_5 \\ S = S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1\end{array}}{\mathrm{W}(\Gamma;\, (\texttt{if}\ e_1\ e_2\ e_3)) = (S;\, S_5 \circ S_4\tau_3)}\ [\mathrm{if}]$$

Theorem (Soundness and completeness of W).

- Soundness: If $\mathrm{W}(\bullet;\, e) = (S;\, \tau)$ then $\bullet \vdash e : \tau$.
- Completeness: If $\bullet \vdash e : \tau$ then $\mathrm{W}(\bullet;\, e) = (S;\, \tau_1)$ for some τ_1 that τ is a substitution instance of. (That is, there is some substitution S_1 such that $S_1\tau_1 = \tau$.)

Exercise 51. *Extend unification and Algorithm W for your extended language.*

6.5 Constraint-based type inference

Algorithm W interleaves walking the term and unification. There's another approach based on *constraints*, where we generate a constraint that tells us what has to be true for a term to type, and then we solve the constraint. This technique is important mostly because it allows us to extend our type system in particular ways by adding new kinds of constraints.

Our language of constraints C has the trivial true constraint $\top$, the conjunction of two constraints $(\wedge\ C_1\ C_2)$, a constraint that two types be equal $(=\ \tau_2\ \tau_2)$, and a constraint $(\exists\ a\ C_1)$ that introduces a fresh type variable for the subconstraint C_1. Here is the syntax of contraints:

$$
\begin{aligned}
C ::= {} & \top \\
\mid {} & (\wedge\ C\ C) \\
\mid {} & (=\ \tau\ \tau) \\
\mid {} & (\exists\ a\ C)
\end{aligned}
$$

Then we can write a judgment that takes a constraint and, if possible, solves it, producing a substitution:

$$\frac{}{\text{solve}(\top) \rightarrow \bullet}\ [\text{true}]$$

$$\frac{\begin{array}{c}\text{solve}(C_1) \rightarrow S_1 \\ \text{solve}(S_1 C_2) \rightarrow S_2\end{array}}{\text{solve}((\wedge\ C_1\ C_2)) \rightarrow S_2 \circ S_1}\ [\text{and}]$$

$$\frac{\tau_1 \sim \tau_2 \rightarrow S}{\text{solve}((=\ \tau_1\ \tau_2)) \rightarrow S}\ [\text{equals}]$$

$$\frac{\begin{array}{c}b = \#\ C \\ \text{solve}(C[a{:=}b]) \rightarrow S\end{array}}{\text{solve}((\exists\ a\ C)) \rightarrow S}\ [\text{exists}]$$

Now that we know how to solve contraints, it remains to generate them for a given term. We do that with the metafunction $[\![\Gamma \vdash e : \tau]\!]$, which, given an environment, a term, and a type, generates the constraints required for the typing judgment $\Gamma \vdash e : \tau$ to hold:

generate : $\Gamma\ e\ \tau \to C$

$[\![\Gamma \vdash x : \tau]\!] = (= \tau\ \tau_1)$
where $\tau_1 = \text{inst}[\![\text{ftv}(\Gamma), \Gamma(x)]\!]$

$[\![\Gamma \vdash (\lambda\ x\ e) : \tau]\!] = (\exists\ a\ (\exists\ b\ (\wedge\ (= \tau\ (\to\ a\ b))\ [\![\Gamma, x{:}a \vdash e : b]\!])))$
where $a = \#\ (\Gamma\ \tau)$, $b = \#\ (\Gamma\ \tau\ a)$

$[\![\Gamma \vdash (\texttt{ap}\ e_1\ e_2) : \tau]\!] = (\exists\ a\ (\wedge\ [\![\Gamma \vdash e_1 : (\to\ a\ \tau)]\!]\ [\![\Gamma \vdash e_2 : a]\!]))$
where $a = \#\ (\Gamma\ \tau)$

$[\![\Gamma \vdash (\texttt{let}\ x\ e_1\ e_2) : \tau]\!] = [\![\Gamma, x{:}\sigma_1 \vdash e_2 : \tau]\!]$
where $a = \#\ \Gamma$, $C_1 = [\![\Gamma \vdash e_1 : a]\!]$, $\text{solve}(C_1) \rightsquigarrow S$, $\tau_1 = Sa$, $\sigma_1 = \text{gen}[\![\text{ftv}(\tau_1) \setminus \text{ftv}(\Gamma), \tau_1]\!]$

$[\![\Gamma \vdash \texttt{true} : \tau]\!] = (= \tau\ \texttt{bool})$

$[\![\Gamma \vdash \texttt{false} : \tau]\!] = (= \tau\ \texttt{bool})$

$[\![\Gamma \vdash (\texttt{if}\ e_1\ e_2\ e_3) : \tau]\!] = (\wedge\ [\![\Gamma \vdash e_1 : \texttt{bool}]\!]\ (\wedge\ [\![\Gamma \vdash e_2 : \tau]\!]\ [\![\Gamma \vdash e_3 : \tau]\!]))$

How can we use this to type a term if we don't know its type to begin with? Suppose we want to type a term e in the empty environment. Then we choose a fresh type variable a, generate the constraint that e has type type, and solve the constraint, yielding a substitition:

$$\text{solve}([\![\bullet \vdash e : a]\!]) \rightsquigarrow S$$

Then we look up the type of a in the resulting substiution: Sa.

Note that for `let`-free programs, constraint generation is completely separated from solving. However, when we encounter a `let`, we still interleave solving to get the generalized type of the let-bound variable.

Exercise 52. *Extend constraint generation for your extended language.*

7 Qualified types

In the previous lecture, we saw how ML infers types for programs that lack type annotations. In this lecture, we see how to extend ML with a principled form of overloading, similar to how it appears in Haskell and Rust. In particular, we will extend type schemes to a form $(\texttt{all}\ (a\ ...)\ (\Rightarrow P\ \tau))$, where P is a logical formula over types that must be satisfied to use a value having that type scheme.

7.1 Syntax

Our language includes the usual variables, lambda abstractions, applications, and let from ML, as well as some constants, a condition form, and pairs:

$$
\begin{array}{rl}
e ::= & x \\
& \mid (\lambda\ x\ e) \\
& \mid (\texttt{ap}\ e\ e) \\
& \mid (\texttt{let}\ x\ e\ e) \\
& \mid c \\
& \mid (\texttt{if0}\ e\ e\ e) \\
& \mid (\texttt{pair}\ e\ e)
\end{array}
$$

The constants include integers, and functions for projecting from pairs, subtraction, equality, and less-than:

$$
\begin{array}{rl}
c ::= & z \\
& \mid \texttt{fst} \\
& \mid \texttt{snd} \\
& \mid \texttt{-} \\
& \mid \texttt{=} \\
& \mid \texttt{<} \\
z ::= & \mathit{integer}
\end{array}
$$

7.2 Dynamic semantics

Values include constants, lambdas, and pairs of values:

$$
\begin{array}{rl}
v ::= & c \\
& \mid (\lambda\ x\ e) \\
& \mid (\texttt{pair}\ v\ v)
\end{array}
$$

Evaluation contexts are standard, performing left-to-right evaluation for applications and pairs:

$$
\begin{array}{rl}
E ::= & \texttt{[]} \\
& \mid (\texttt{ap}\ E\ e) \\
& \mid (\texttt{ap}\ v\ E) \\
& \mid (\texttt{pair}\ E\ e) \\
& \mid (\texttt{pair}\ v\ E) \\
& \mid (\texttt{if0}\ E\ e\ e) \\
& \mid (\texttt{let}\ x\ E\ e)
\end{array}
$$

We give a reduction relation that includes rules for application and let, two rules for if0 (true and false), and delta, which handles applications of constants by delegating to a metafunction:

$$E[(\texttt{ap}\ (\lambda\ x\ e)\ v)] \longrightarrow E[e[x{:=}v]] \quad [\beta\text{-val}]$$

$$E[(\texttt{let}\ x\ v\ e)] \longrightarrow E[e[x{:=}v]] \quad [\text{let}]$$

$$E[(\texttt{if0}\ 0\ e_1\ e_2)] \longrightarrow E[e_1] \quad [\text{if-true}]$$

$$E[(\texttt{if0}\ z\ e_1\ e_2)] \longrightarrow E[e_2] \quad [\text{if-false}]$$
$$\text{where } z \neq 0$$

$$E[(\texttt{ap}\ c\ v)] \longrightarrow E[\delta[\![c, v]\!]] \quad [\text{delta}]$$

The metafunction δ gives the results for applying constants to values:

$$\begin{aligned}
&\delta : c\ v \to v \\
&\delta[\![\texttt{fst}, (\texttt{pair}\ v_1\ v_1)]\!] &&= v_1 \\
&\delta[\![\texttt{snd}, (\texttt{pair}\ v_1\ v_2)]\!] &&= v_2 \\
&\delta[\![\texttt{-}, (\texttt{pair}\ z_1\ z_2)]\!] &&= z_1 - z_2 \\
&\delta[\![\texttt{=}, (\texttt{pair}\ v_1\ v_2)]\!] &&= v_1 == v_2\ ?\ 0 : 1 \\
&\delta[\![\texttt{<}, (\texttt{pair}\ v_1\ v_2)]\!] &&= v_1 < v_2\ ?\ 0 : 1
\end{aligned}$$

Note that the functions represented by constants are uncurried, taking pairs of values—this simplifies our presentation somewhat.

7.3 Static semantics

As in ML the static semantics assigns prenex type schemes to let-bound values, but type schemes now have an additional component. The syntax of types is as follows.

7.3.1 Syntax of types

Monotypes include type variables, the base type `Int`, product types, and function types:

$$\begin{aligned}
\tau ::= &\ a \\
&\mid \texttt{Int} \\
&\mid (\texttt{Prod}\ \tau\ \tau) \\
&\mid (\to\ \tau\ \tau)
\end{aligned}$$

To represent overloading, we define a fixed set of *type classes* C, which are used to construct *predicates* on types π:

$$\begin{aligned}
C ::= &\ \texttt{Eq} \\
&\mid \texttt{Ord} \\
\pi ::= &\ (C\ \tau)
\end{aligned}$$

For any type τ, the predicate $(\texttt{Eq}\ \tau)$ means that type τ supports equality, and the predicate $(\texttt{Ord}\ \tau)$ means that type τ supports less-than. In a real system, the set of type classes (and thus the possible predicates) would be extensible by the user.

A *predicate context* P is a collection of predicates:

$$P ::= [\pi\ ...]$$

Then a *qualified type* ρ is a monotype qualified by some predicate context:

$$\rho ::= (\Rightarrow P\ \tau)$$

Then a type scheme is a qualified type generalized over some quantified set of type variables:

$$\begin{aligned} as &::= (a\ ...) \\ \sigma &::= (\texttt{all}\ as\ \rho) \end{aligned}$$

For example, type scheme $(\texttt{all}\ (a)\ (\Rightarrow\ [(\texttt{Eq}\ a)]\ (\rightarrow\ a\ (\rightarrow\ a\ \texttt{Int}))))$ describes a function that takes (curried) two arguments of any type a supporting equality and returns an integer.

As in ML, typing environments map variable names to type schemes:

$$\begin{aligned} \Gamma ::= &\bullet \\ &\mid \Gamma, x{:}\sigma \end{aligned}$$

7.3.2 The types of constants

We can now define the `type-of` metafunction, which gives type schemes for the constants:

$$\begin{aligned} &\text{type-of} : c \rightarrow \sigma \\ &\text{type-of}[\![z]\!] = (\texttt{all}\ ()\ (\Rightarrow\ []\ \texttt{Int})) \\ &\text{type-of}[\![\texttt{fst}]\!] = (\texttt{all}\ (a\ b)\ (\Rightarrow\ []\ (\rightarrow\ (\texttt{Prod}\ a\ b)\ a))) \\ &\text{type-of}[\![\texttt{snd}]\!] = (\texttt{all}\ (a\ b)\ (\Rightarrow\ []\ (\rightarrow\ (\texttt{Prod}\ a\ b)\ b))) \\ &\text{type-of}[\![\texttt{-}]\!] = (\texttt{all}\ ()\ (\Rightarrow\ []\ (\rightarrow\ (\texttt{Prod}\ \texttt{Int}\ \texttt{Int})\ \texttt{Int}))) \\ &\text{type-of}[\![\texttt{=}]\!] = (\texttt{all}\ (a)\ (\Rightarrow\ [(\texttt{Eq}\ a)]\ (\rightarrow\ (\texttt{Prod}\ a\ a)\ \texttt{Int}))) \\ &\text{type-of}[\![\texttt{<}]\!] = (\texttt{all}\ (a)\ (\Rightarrow\ [(\texttt{Ord}\ a)]\ (\rightarrow\ (\texttt{Prod}\ a\ a)\ \texttt{Int}))) \end{aligned}$$

Note that `=` and `<` are overloaded.

7.3.3 Instantiation and entailment

Before we can give our main typing relation, we need two auxiliary judgments. The first, as in ML, relates a type scheme to its instantiations as qualified types:

$$\frac{}{(\texttt{all}\ ()\ \rho) > \rho}\ [\text{mono}]$$

$$\frac{(\texttt{all}\ (a_i\ ...)\ \rho_0)[a{:=}\tau] > \rho}{(\texttt{all}\ (a\ a_i\ ...)\ \rho_0) > \rho}\ [\text{all}]$$

The second relation is entailment for predicate contexts. This is not strictly necessary (and omitted from Jones's paper), but allows us to make predicate contexts smaller when they are redundant. The first two rules rule say that a predicate context entails itself and that entailment is transitive.

$$\frac{}{P \Vdash P}\ [\text{refl}]$$

$$\frac{P_1 \Vdash P_2 \quad P_2 \Vdash P_3}{P_1 \Vdash P_3}\ [\text{trans}]$$

The next rule says that we can remove duplicate predicates from a context:

$$\frac{}{[\pi_i\ ...\ \pi\ \pi_j\ ...\ \pi_k\ ...] \Vdash [\pi_i\ ...\ \pi\ \pi_j\ ...\ \pi\ \pi_k\ ...]}\ [\text{dup}]$$

The next two rules say that integers support equality and ordering, and that fact need not be recorded in the context to prove it:

$$\frac{}{[\pi_i\ ...\ \pi_j\ ...] \Vdash [\pi_i\ ...\ (\texttt{Eq}\ \texttt{Int})\ \pi_j\ ...]}\ [\text{eq-int}]$$

$$\frac{}{[\pi_i\ ...\ \pi_j\ ...] \Vdash [\pi_i\ ...\ (\texttt{Ord}\ \texttt{Int})\ \pi_j\ ...]}\ [\text{ord-int}]$$

Finally, equality works on pairs if it works on both components of the pair:

$$\frac{}{[\pi_i\ ...\ (\texttt{Eq}\ \tau_1)\ (\texttt{Eq}\ \tau_2)\ \pi_j\ ...] \Vdash [\pi_i\ ...\ (\texttt{Eq}\ (\texttt{Prod}\ \tau_1\ \tau_2))\ \pi_j\ ...]}\ [\text{eq-prod}]$$

7.3.4 Syntax-directed typing

The typing judgment is of the form $P \mid \Gamma \vdash e : \tau$, where P gives constraints on the types in Γ. Even though it appears on the left, P should be thought of as an out-parameter.

The rule for typing a variable says to look up its type scheme in the environment and then instantiate the bound variables of the type scheme. The predicate context P from the instantiated type scheme becomes the predicate context for the judgment:

$$\frac{\Gamma(x) > (\Rightarrow P\ \tau)}{P \mid \Gamma \vdash x : \tau}\ [\text{var-inst}]$$

Typing a constants is substantially the same, except we get its type scheme using the `type-of` metafunction:

$$\frac{\text{type-of}[\![c]\!] > (\Rightarrow P\ \tau)}{P \mid \Gamma \vdash c : \tau}\ [\text{const-inst}]$$

The rules for lambda abstractions, applications, conditionals, and pairs, as the same as they would be in ML, except that we thread through and combine the predicate contexts:

$$\frac{P \mid \Gamma, x{:}\tau_1 \vdash e : \tau_2}{P \mid \Gamma \vdash (\lambda\ x\ e) : (\rightarrow \tau_1\ \tau_2)}\ [\text{abs}]$$

$$\frac{\begin{array}{c} P_1 \mid \Gamma \vdash e_1 : (\rightarrow \tau_2\ \tau) \\ P_2 \mid \Gamma \vdash e_2 : \tau_2 \end{array}}{P_1 \cup P_2 \mid \Gamma \vdash (\texttt{ap}\ e_1\ e_2) : \tau}\ [\text{app}]$$

$$\frac{\begin{array}{c} P_1 \mid \Gamma \vdash e_1 : \texttt{Int} \\ P_2 \mid \Gamma \vdash e_2 : \tau \\ P_3 \mid \Gamma \vdash e_3 : \tau \end{array}}{P_1 \cup P_2 \cup P_3 \mid \Gamma \vdash (\texttt{if0}\ e_1\ e_2\ e_3) : \tau}\ [\text{if0}]$$

$$\frac{\begin{array}{c} P_1 \mid \Gamma \vdash e_1 : \tau_1 \\ P_2 \mid \Gamma \vdash e_2 : \tau_2 \end{array}}{P_1 \cup P_2 \mid \Gamma \vdash (\texttt{pair}\ e_1\ e_2) : (\texttt{Prod}\ \tau_1\ \tau_2)}\ [\text{pair}]$$

Finally, the let rule is where the action is:

$$\frac{\begin{array}{c} P_1 \mid \Gamma \vdash e_1 : \tau_1 \\ P \Vdash P_1 \\ \sigma = (\texttt{all}\ ((\text{ftv}(P) \cup \text{ftv}(\tau_1)) \setminus \text{ftv}(\Gamma))\ (\Rightarrow P\ \tau_1)) \\ P_2 \mid \Gamma, x{:}\sigma \vdash e_2 : \tau \end{array}}{P_2 \mid \Gamma \vdash (\texttt{let}\ x\ e_1\ e_2) : \tau}\ [\text{let-gen}]$$

First we type e_1, which produces a predicate context P_1. Then we apply the entailment relation to reduce P_1 to a context that entails it, P. (This step can be omitted, but it reflects the idea that we probably want to simplify predicate contexts before including them in type schemes.) Then we build a type scheme σ by generalizing all the type variables in P and τ_1 that do not appear in Γ, and bind that σ in the environment to type e_2. Note that the resulting predicate context for the judgment is only P_2, the constraints required by e_2, since the constraints required by e_1 are carried by the resulting type scheme.

Alternatively, we could split the predicates of P_1 (or P) into those relevant to τ_1, which we would package up in the type scheme, and those irrelevant to τ_2, which we would propogate upward.

Exercise 53. *Use Haskell's type classes to implement bijections between the natural numbers and lists.*

To get started, install `ghc` (and be sure that QuickCheck is installed, perhaps by issuing the command `cabal install quickcheck`). Put your code in `XEnum.hs` and use `ghc -o XEnum XEnum.hs && ./XEnum` to run your code.

Here are some declarations to get started, along with an explanation of them.

Because Haskell is whitespace-sensitive, copying code from webpages is fraught; accordingly the declarations in the code below are all in XEnum.hs

```
{-# LANGUAGE ScopedTypeVariables #-}
import Test.QuickCheck
import Numeric.Natural

class XEnum a where
  into  :: a -> Natural
  outof :: Natural -> a

instance XEnum Natural where
  into n = n
  outof n = n
```

The `class` declaration introduces a new predicate `XEnum` that supports two operations, `into` and `outof`. These are two functions that realize a bijection between the type `a` and the natural numbers.

The `instance` declaration says that the type `Natural` supports enumeration by giving the functions that translate from the naturals to the naturals (i.e., the identity function).

For our first substantial instance, fill in the `into` and `outof` functions to define a bijection between the natural numbers and the integers:

```
instance XEnum Integer where
  into x = error "not implemented"
  outof n = error "not implemented"
```

There is more than one way to do this, but it is also easy to make arithmetic errors when doing it. So we can use Quick Check to help find those errors. Add this declaration to the end of your program:

```
prop_inout :: (Eq a, XEnum a) => a -> Bool
```

```
prop_inout x = outof (into x) == x
main = quickCheck (prop_inout :: Integer -> Bool)
```
If you do not see output like `+++ OK, passed 100 tests.`, then you have a bug in your bijections.

Once you have finished that, add the support for (disjoint) unions. To do that we need to assume we have two enumerable things and then we are going to add a bijection using the `Either` type:
```
instance (XEnum a , XEnum b) => XEnum (Either a b) where
  into (Left x)  = error "not implemented"
  into (Right x) = error "not implemented"
  outof n = error "not implemented"
```
The idea of this bijection is to use the odd numbers for either `Left` or `Right` values, and use the even numbers for the other. So we can embed two enumerable values into one. Also test this one with Quick Check, using `prop_inout :: (Either Integer Natural) -> Bool`.

Next up, pairs.
```
instance (XEnum a , XEnum b) => XEnum (a , b) where
  into (a , b) = error "not implemented"
  outof n = error "not implemented"
```
The formulas for these ones are more complex; I recommend using Szudzik's "elegant" pairing function, found on page 8 of `https://pdfs.semanticscholar.org/68e8/7ad59107481bc3cfdf1`
`pdf`. Page 9 shows the geometric intuition for the bijection. To implement it, you will need an exact square root function; see `https://stackoverflow.com/questions/19965149/integer-square-root-function-in-haskell` for two definitions.

Once you have that all working, define enumerations for lists.
```
instance XEnum a => XEnum [a] where
  into l = error "not implemented"
  outof n = error "not implemented"
```
Be aware that the formulas and the bijections you've built work only for infinite sets (i.e., the naturals, the integers, pairs of them, etc.) If you want to use these bijections on sets that are finite, you need to add a `size` operation:
```
data ENatural = Fin Natural | Inf

class XEnum a where
  into  :: a -> Natural
  outof :: Natural -> a
  size  :: ENatural
```
The size of an `Either` is the sum of the sizes and the size of a pair enumeration is the product of the sizes. Also note that the corresponding formulas will need adjustment to handle the case where one of sides is finite. If you get stuck trying to figure out the formulas, look in this paper: `https://www.eecs.northwestern.edu/~robby/pubs/papers/jfp2017-nfmf.pdf`,

7.4 Type inference algorithm

The above type system provides a satisfactory account of which terms type and which do not, but it does not give us an algorithm that we can actually run. In this section, we extend ML's Algorithm W for qualified types.

First, we give a helper metafunction for instantiating a type scheme with fresh type variables:

$$
\begin{aligned}
&\text{inst} : (a \ldots)\ \sigma \to \rho \\
&\text{inst}[\![(a \ldots), (\texttt{all}\ ()\ \rho)]\!] &&= \rho \\
&\text{inst}[\![(a \ldots), (\texttt{all}\ (b\ b_i \ldots)\ \rho)]\!] &&= \text{inst}[\![(a \ldots\ b_1), (\texttt{all}\ (b_i \ldots)\ \rho)[b{:=}b_1]]\!] \\
&\quad \text{where } b_1 = \#\ (b_i \ldots\ a \ldots)
\end{aligned}
$$

Again we use unification. Because unification is applied to monotypes, it is the same as in ML (except now we have to handle product types as well):

$$\frac{}{a \sim a \rightsquigarrow \bullet}\ \text{[var-same]}$$

$$\frac{a \notin \text{ftv}(\tau)}{a \sim \tau \rightsquigarrow \bullet[a{:=}\tau]}\ \text{[var-left]}$$

τ is not a type variable

$$\frac{a \sim \tau \rightsquigarrow S}{\tau \sim a \rightsquigarrow S}\ \text{[var-right]}$$

$$\frac{}{\texttt{Int} \sim \texttt{Int} \rightsquigarrow \bullet}\ \text{[int]}$$

$$\frac{\begin{array}{c}\tau_{11} \sim \tau_{21} \rightsquigarrow S_1 \\ S_1\tau_{12} \sim S_1\tau_{22} \rightsquigarrow S_2\end{array}}{(\texttt{Prod}\ \tau_{11}\ \tau_{12}) \sim (\texttt{Prod}\ \tau_{21}\ \tau_{22}) \rightsquigarrow S_2 \circ S_1}\ \text{[prod]}$$

$$\frac{\begin{array}{c}\tau_{11} \sim \tau_{21} \rightsquigarrow S_1 \\ S_1\tau_{12} \sim S_1\tau_{22} \rightsquigarrow S_2\end{array}}{(\to\ \tau_{11}\ \tau_{12}) \sim (\to\ \tau_{21}\ \tau_{22}) \rightsquigarrow S_2 \circ S_1}\ \text{[arr]}$$

Algorithm W for qualified types takes a type environment and a term, and returns a substitution, a type, and a predicate context: $\text{W}(\Gamma; e) = (S; \tau; P)$.

To infer the type of a variable or constant, we look up its type scheme (in the environment or the `type-of` metafunction, respectively) and instantiate it with fresh type variables, yielding a qualified type $(\Rightarrow P\ \tau)$. The τ is the type of the variable or constant, and P is the predicate context that must be satisfied:

$$\frac{(\Rightarrow P\ \tau) = \text{inst}[\![\text{ftv}(\Gamma), \Gamma(x)]\!]}{\text{W}(\Gamma; x) = (\bullet; \tau; P)}\ [\text{var}]$$

$$\frac{(\Rightarrow P\ \tau) = \text{inst}[\![\text{ftv}(\Gamma), \text{type-of}[\![c]\!]]\!]}{\text{W}(\Gamma; c) = (\bullet; \tau; P)}\ [\text{const}]$$

Lambda abstraction, application, pairing, and the conditional are as before, merely propagating and combining predicate contexts:

$$\frac{\begin{array}{c}\text{W}(\Gamma; e_1) = (S_1; \tau_1; P_1) \\ \text{W}(S_1\Gamma; e_2) = (S_2; \tau_2; P_2) \\ a = \#\ (\Gamma\ S_1\ S_2\ \tau_1\ \tau_2\ P_1\ P_2) \\ S_2\tau_1 \sim (\rightarrow \tau_2\ a) \rightarrow S_3\end{array}}{\text{W}(\Gamma; (\texttt{ap}\ e_1\ e_2)) = (S_3 \circ S_2 \circ S_1; S_3 a; S_3(S_2 P_1 \cup P_2))}\ [\text{app}]$$

$$\frac{\begin{array}{c}a = \#\ \Gamma \\ \text{W}(\Gamma, x{:}(\texttt{all}\ ()\ (\Rightarrow []\ a)); e) = (S; \tau; P)\end{array}}{\text{W}(\Gamma; (\lambda\ x\ e)) = (S; (\rightarrow S a\ \tau); P)}\ [\text{abs}]$$

$$\frac{\begin{array}{c}\text{W}(\Gamma; e_1) = (S_1; \tau_1; P_1) \\ \text{W}(S_1\Gamma; e_2) = (S_2; \tau_2; P_2)\end{array}}{\text{W}(\Gamma; (\texttt{pair}\ e_1\ e_2)) = (S_2 \circ S_1; (\texttt{Prod}\ S_2\tau_1\ \tau_2); S_2 P_1 \cup P_2)}\ [\text{pair}]$$

$$\frac{\begin{array}{c}\text{W}(\Gamma; e_1) = (S_1; \tau_1; P_1) \\ \text{W}(S_1\Gamma; e_2) = (S_2; \tau_2; P_2) \\ \text{W}(S_2 S_1 \Gamma; e_3) = (S_3; \tau_3; P_3) \\ S_3 S_2 \tau_1 \sim \texttt{Int} \rightarrow S_4 \\ S_4 S_3 \tau_2 \sim S_4 \tau_3 \rightarrow S_5 \\ S = S_5 \circ S_4 \circ S_3 \circ S_2 \circ S_1\end{array}}{\text{W}(\Gamma; (\texttt{if}\ e_1\ e_2\ e_3)) = (S; S_5 S_4 \tau_3; S_5 S_4 (S_3(S_2 P_1 \cup P_2) \cup P_3))}\ [\text{if0}]$$

Note how substitutions must be applied to predicate contexts, just as we apply them to type environments and types.

Finally, the let rule follows the let rule from the previous section, packaging up the predicate context generated for e_1 in the type scheme assigned to x. We (optionally) assume a metafunction r:qreduce$[\![P]\!]$ that simplifies the predicate context before constructing the type scheme.

$$\frac{\begin{array}{c}\mathrm{W}(\Gamma;\, e_1) = (S_1;\, \tau_1;\, P_1) \\ P = \mathrm{qreduce}[\![P_1]\!] \\ \sigma = (\texttt{all}\ ((\mathrm{ftv}(P) \cup \mathrm{ftv}(\tau_1)) \setminus \mathrm{ftv}(S_1\Gamma))\ (\Rightarrow P\ \tau_1)) \\ \mathrm{W}(S_1\Gamma,\, x{:}\sigma;\, e_2) = (S_2;\, \tau_2;\, P_2)\end{array}}{\mathrm{W}(\Gamma;\, (\texttt{let}\ x\ e_1\ e_2)) = (S_2 \circ S_1;\, \tau_2;\, P_2)}\ [\mathrm{let}]$$

7.5 Evidence translation

Exercise 54. *What is the most general type scheme of the term* $(\lambda\ (x)\ (\lambda\ (y)\ (\texttt{if0}\ (=\ (\texttt{pair}\ x\ y))\ x\ y)))$*?*

How would you implement such a function—in particular, how does it figure out the equality for a generic/unknown type parameter? Well, our operational semantics cheated by relying on Racket's underlying polymorphic `equal?` function. Racket's `equal?` relies on Racket's object representations, which include tags that distinguish number from Booleans from pairs, etc. But what about in a typed language that does not use tags and thus cannot support polymorphic equality?

One solution is called evidence passing, wherein using a qualified type requires passing evidence that it is inhabited, where this evidence specifies some information about how to perform the associated operations. In our type classes example, the evidence is the equality or less-than function specialized to the required type. (In a real evidence-passing implementation such as how Haskell is traditionally implemented, the evidence is a dictionary of methods.)

We can translate implicitly-typed λ-`qual` programs like the above into programs that pass evidence explicitly. We do this by typing them in an evidence environment, which names the evidence for each predicate:

$$\Delta ::= [(x\ \pi)\ ...]$$

We can use the evidence environment to summon or construct evidence if it's available. In particular, the judgment $\Delta \Vdash e : \pi$ uses evidence environment Δ to construct e, which is evidence of predicate π. In particular, if π is $(\texttt{Eq}\ \tau)$ then e should be an equality function of type $(\rightarrow\ (\texttt{Prod}\ \tau\ \tau)\ \texttt{Int})$; if π is $(\texttt{Ord}\ \tau)$ then e should be a less-than function of type $(\rightarrow\ (\texttt{Prod}\ \tau\ \tau)\ \texttt{Int})$.

For base type `Int`, the evidence is just a primitive function performing the correct operation:

$$\frac{}{\Delta \Vdash \texttt{=/int} : (\texttt{Eq Int})}\ [\text{eq-int}]$$

$$\frac{}{\Delta \Vdash \texttt{</int} : (\texttt{Ord Int})}\ [\text{ord-int}]$$

For a product type, we summon evidence for each component type, and then construct the equality function for the product.

$$\frac{\begin{array}{c}\Delta \Vdash e_1 : (\texttt{Eq}\ \tau_1) \\ \Delta \Vdash e_2 : (\texttt{Eq}\ \tau_2) \\ e_{\text{out}} = (\lambda\ \texttt{p}\ (\texttt{if0}\ (\texttt{ap}\ e_1\ (\texttt{pair}\ (\texttt{ap fst}\ (\texttt{ap fst p}))\ (\texttt{ap fst}\ (\texttt{ap snd p}))))\\ \qquad\qquad (\texttt{ap}\ e_2\ (\texttt{pair}\ (\texttt{ap snd}\ (\texttt{ap fst p}))\ (\texttt{ap snd}\ (\texttt{ap snd p}))))\\ \qquad\qquad 1))\end{array}}{\Delta \Vdash e_{\text{out}} : (\texttt{Eq}\ (\texttt{Prod}\ \tau_1\ \tau_2))}\ [\text{eq-prod}]$$

Other types are looked up in the evidence environment:

$$\frac{}{[(x_{\text{i}}\ \pi_{\text{i}})\ \ldots\ (x\ \pi)\ (x_{\text{j}}\ \pi_{\text{j}})\ \ldots] \Vdash x : \pi}\ [\text{lookup}]$$

Note that if difference evidence appears for the same repeated predicate, then the behavior can be incoherent.

The evidence translation uses two more auxiliary judgments. The first is for applying a term that expect evidence to its expected evidence:

$$\frac{}{\Delta \Vdash [] \Rightarrow e \twoheadrightsquigarrow e}\ [\text{nil}]$$

$$\frac{\begin{array}{c}\Delta \Vdash e_{\text{ev}} : \pi \\ \Delta \Vdash [\pi_{\text{i}}\ \ldots] \Rightarrow (\texttt{ap}\ e\ e_{\text{ev}}) \twoheadrightsquigarrow e_{\text{out}}\end{array}}{\Delta \Vdash [\pi\ \pi_{\text{i}}\ \ldots] \Rightarrow e \twoheadrightsquigarrow e_{\text{out}}}\ [\text{cons}]$$

The second abstracts over the evidence expected by a term based on its context:

$$\frac{}{[] \Vdash e \twoheadrightsquigarrow [] \Rightarrow e}\ [\text{nil}]$$

$$\frac{[(x_{\text{i}}\ \pi_{\text{i}})\ \ldots] \Vdash (\lambda\ x\ e) \twoheadrightsquigarrow [\pi_{\text{out}}\ \ldots] \Rightarrow e_{\text{out}}}{[(x\ \pi)\ (x_{\text{i}}\ \pi_{\text{i}})\ \ldots] \Vdash e \twoheadrightsquigarrow [\pi\ \pi_{\text{out}}\ \ldots] \Rightarrow e_{\text{out}}}\ [\text{cons}]$$

Four rules of the typing judgment are unremarkable, simply passing the evidence environment through and translating homomorphically:

$$\frac{\Delta \mid \Gamma, x{:}\tau_1 \vdash e \rightsquigarrow e^\dagger : \tau_2}{\Delta \mid \Gamma \vdash (\lambda\ x\ e) \rightsquigarrow e^\dagger : (\to\ \tau_1\ \tau_2)}\ [\text{abs}]$$

$$\frac{\begin{array}{c}\Delta \mid \Gamma \vdash e_1 \rightsquigarrow e_1^\dagger : (\to\ \tau_2\ \tau) \\ \Delta \mid \Gamma \vdash e_2 \rightsquigarrow e_2^\dagger : \tau_2\end{array}}{\Delta \mid \Gamma \vdash (\texttt{ap}\ e_1\ e_2) \rightsquigarrow (\texttt{ap}\ e_1^\dagger\ e_2^\dagger) : \tau}\ [\text{app}]$$

$$\frac{\begin{array}{c}\Delta \mid \Gamma \vdash e_1 \rightsquigarrow e_1^\dagger : \texttt{Int} \\ \Delta \mid \Gamma \vdash e_2 \rightsquigarrow e_2^\dagger : \tau \\ \Delta \mid \Gamma \vdash e_3 \rightsquigarrow e_3^\dagger : \tau\end{array}}{\Delta \mid \Gamma \vdash (\texttt{if0}\ e_1\ e_2\ e_3) \rightsquigarrow (\texttt{if0}\ e_1^\dagger\ e_2^\dagger\ e_3^\dagger) : \tau}\ [\text{if0}]$$

$$\frac{\begin{array}{c}\Delta \mid \Gamma \vdash e_1 \rightsquigarrow e_1^\dagger : \tau_1 \\ \Delta \mid \Gamma \vdash e_2 \rightsquigarrow e_2^\dagger : \tau_2\end{array}}{\Delta \mid \Gamma \vdash (\texttt{pair}\ e_1\ e_2) \rightsquigarrow (\texttt{pair}\ e_1^\dagger\ e_2^\dagger) : (\texttt{Prod}\ \tau_1\ \tau_2)}\ [\text{pair}]$$

The rules for variables and constants take a polymorphic value and apply it to the required evidence for any predicates contained in its qualified type, using the evidence application judgment:

$$\frac{\begin{array}{c}\Gamma(x) > (\Rightarrow P\ \tau) \\ \Delta \Vdash P \Rightarrow x \rightsquigarrow e\end{array}}{\Delta \mid \Gamma \vdash x \rightsquigarrow e : \tau}\ [\text{var}]$$

$$\frac{\begin{array}{c}\text{type-of}[\![c]\!] > (\Rightarrow P\ \tau) \\ \Delta \Vdash P \Rightarrow c \rightsquigarrow e\end{array}}{\Delta \mid \Gamma \vdash c \rightsquigarrow e : \tau}\ [\text{const}]$$

The let form, as above, generalizes, by abstracting the right-hand side e_1 over evidence corresponding to its inferred evidence context:

$$\frac{\begin{array}{c}\Delta_1 \mid \Gamma \vdash e_1 \rightsquigarrow e_1^\dagger : \tau_1 \\ \Delta_1 \Vdash e_1^\dagger \rightsquigarrow P \Rightarrow e_1^\ddagger \\ \sigma = (\texttt{all}\ ((\text{ftv}(P) \cup \text{ftv}(\tau_1)) \setminus \text{ftv}(\Gamma))\ (\Rightarrow P\ \tau_1)) \\ \Delta_2 \mid \Gamma, x{:}\sigma \vdash e_2 \rightsquigarrow e_2^\dagger : \tau\end{array}}{\Delta_2 \mid \Gamma \vdash (\texttt{let}\ x\ e_1\ e_2) \rightsquigarrow (\texttt{let}\ x\ e_1^\ddagger\ e_2^\dagger) : \tau}\ [\text{let}]$$

Exercise 55. *Rust uses monomorphization to implement generics and traits. It does this by duplicating polymorphic code, specializing it at each required type. Write a relation that formalizes monomorphization for describes* λ`-qual`.

8 The Lambda Cube: λ`-cube`

The λ`-cube` provides a systematic organization of types systems that captures a range of expressiveness, from the simply-typed lambda calculus (in section 2) through the polymorphic lambda calculus (in section 4), the higher-order lambda calculus (in section 5), and up to λC, the calculus of constructions, which is the focus of this section.

8.1 Syntax

The basic idea of the structure of the λ cube is to eliminate the distinction between types and terms and then use typing judgments to control which classes of expression are allowed in type positions. To get started, we first just get rid of the distinction between types and terms using this syntax:

$$\begin{array}{rcl} e, \tau & ::= & x \\ & \mid & (\lambda\ (x\ \texttt{:}\ \tau)\ e) \\ & \mid & (\texttt{ap}\ e\ e) \\ & \mid & s \\ & \mid & (x\ \texttt{:}\ \tau \to \tau) \\ s & ::= & * \mid \square \end{array}$$

The first three expression forms are the familiar variables, lambda abstractions, and application expressions. The $*$ is the type of types, just as in section 4, and $\square$ is analgous, but one level up. That is, it represents the type of kinds or, expressions that have the type $\square$ are expressions that themselves compute kinds.

The final expression form, $\to$, represents function types, but it is dependent. In its simplest form, the type $(x\ \texttt{:}\ \tau_1 \to \tau_2)$, where x does not appear free in τ_2, represents functions from τ_1 to τ_2. In general, however, the variable x can appear free in τ_2, meaning that the type of the result of the function can depend on the argument actually supplied to the function.

This notation specializes to the earlier type systems we considered; as an example, recall the function composition operator from the beginning of section 4. Here's the original version of the function:

```
(Λ a1
  (Λ a2
    (Λ a3
      (λ x1 (→ a2 a3)
        (λ x2 (→ a1 a2)
          (λ y a1
            (ap x1 (ap x2 y))))))))
```

In the new language, the Λ and λ are not distinguished by the constructor, but a Λ is the same thing as a λ where the argument has type ∗. So, this is the composition operator in `λ-cube`:

```
(λ (a1 : ∗)
  (λ (a2 : ∗)
    (λ (a3 : ∗)
      (λ (x1 : (i2 : a2 → a3))
        (λ (x2 : (i1 : a1 → a2))
          (λ (y : a1)
            (ap x1 (ap x2 y))))))))
```

We also adjust the syntax to require an extra set of parentheses and a colon to make it a little bit easier to read expressions (because other distinctions are removed).

Another example that's worth considering is the identity function. Here it is:

```
(λ (α : ∗)
  (λ (x : α)
    x))
```

This term is what you would expect, simply replacing the capital Λ with the lowercase λ and adding a ∗. But consider its type:

$(\alpha : * \to (x : \alpha \to \alpha))$

This is a type that cannot be expressed with just the arrow. Or, in other words, this is a dependent type because the variable bound by the outer function type is used in its body. It is the same as the type `(all α (→ α α))` but we can use → for both the function type and for the `all` type.

8.2 Typing Rules

First, we just assert that ∗ is a □

$$\frac{}{\Gamma \vdash * : \Box}\ [\text{axiom}]$$

and then we have what appears to be the standard variable rule:

$$\frac{\vdash \Gamma}{\Gamma \vdash x : \Gamma(x)}\,[\text{variable}]$$

but note the premise that ensures that the environment is well-formed. In earlier type systems, that was a self-contained check that the types were well-formed. Now, because we have eliminated the distinction between types and terms, it uses the typing judgment itself:

$$\frac{}{\vdash \bullet}\,[\text{nil}] \qquad \frac{\Gamma \vdash A : s}{\vdash \Gamma, x{:}A}\,[\text{cons}]$$

The application rule handles all forms of abstraction:

$$\frac{\Gamma \vdash F : (x : A \to B) \quad \Gamma \vdash a : A}{\Gamma \vdash (\mathsf{ap}\ F\ a) : B[x{:=}a]}\,[\text{application}]$$

It looks something like a combination of the application and type application rule from λ-2. Like the normal function application rule, we make sure that the two subexpressions have appropriate types: one a function and one a matching argument type (the type in the parameter of the function type). Like the type application rule, however, we perform a substitution, computing the type of the result of the function based on the argument that was actually supplied.

Sometimes, the type A that we get on the function is different than the type A on the argument. This rule allows us to do some computation in order to make two such types match up to each other, where the $\equiv$ relation allows us to perform β substitutions in the types as needed.

$$\frac{\Gamma \vdash A : B_1 \quad B_1 \equiv B_2 \quad \Gamma \vdash B_2 : s}{\Gamma \vdash A : B_2}\,[\text{conversion}]$$

In order to type check a λ abstraction,

$$\frac{\Gamma, x{:}A \vdash b : B \quad \Gamma \vdash (x : A \to B) : s}{\Gamma \vdash (\lambda\ (x : A)\ b) : (x : A \to B)}\,[\text{abstraction}]$$

we check the body, on the assumption that the argument has the type on the λ. The second premise ensures that the type that we get for the result itself makes sense.

The final rule covers function type expressions.

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x{:}A \vdash B : s_2}{\Gamma \vdash (x : A \to B) : s_2}\,[\lambda\text{C}]$$

This version allows either $*$ or $\Box$ for both the argument and the result type. This generality allows us to capture the full Calculus of Constructions, which forms the basis for the theorem proving system Coq.

If we restrict the rule so that both s_1 and s_1 are $*$, then the resulting type system is equivalent to the simply-typed lambda calculus. Intuitively, this restriction means that our functions can accept arguments that are values, i.e., can be described by types, but not by kinds.

If we allow s_1 to be either $*$ or $\Box$, but restrict s_2 so it can be only $*$, we get the polymorphic lambda calculus, λ-2. This means that functions can now play the role of `all`, expressions, accepting types, but always returning only a type. Various other restrictions in this spirit correspond to various other type systems in the literature.