Lecture Notes on Adjoint Types

15-417/817: HOT Compilation
Frank Pfenning

Lecture 10
February 13, 2025

1 Introduction

At this point we have an expressive functional language with higher-order types, but all definitions are purely linear. We have been able to get by using explicit functions to copy or drop values of purely positive type. However, both from a programming and a efficiency perspective, copying data is not desirable or sustainable. Furthermore, values of negative type such as functions cannot be copied (because their structure is not observable) and therefore higher-order functions like map and fold cannot be written.

The solution is to parameterize types by *modes*, where different modes have different structural properties. The term "modes" comes from *modal logic*, where we might distinguish different modes of truth, such as validity, knowledge, or truth at a particular time or in a particular world.

What are these structural properties? We have *contraction*, which computationally means that we can use a variable more than once, and *weakening*, which means that we do not need to use a variable. The origin of these terms lies in the sequent calculus [Gentzen, 1935] with the rules

$$\frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ \textsf{contraction} \qquad \frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ \textsf{weakening}$$

If we read them top-down, the name makes sense: contraction identifies two copies of A and weakening goes from a stronger to a weaker property by adding an unused antecedent.

If we read these rules bottom up, we see that contraction allows us to use a variable more than once (or create an alias for an existing variable), while weakening allows us **not** to use a variable.

From a programmer's perspective using explicit rules in this form is unnecessarily tedious. It would be much better if we could simply use a variable more than once (contraction) or not at all (weakening). We will takes this path and develop an system of typing rules in which this will be the case, leading to *adjoint natural deduction* [Jang et al., 2024], which in turn is based on the *adjoint sequent calculus* [Reed, 2009, Pruiksma et al., 2018]. The ancestor of these formulations is Benton's mixed linear/non-linear logic [1994]. It departs from Girard's formulation of linear logic [1987] not only by being intuitionistic [Barber, 1996] but also by decomposing the exponential modality $!A$.

The present lecture material is adapted from Jang et al. [2024]. Specifically, we simplify the system by omitting empty sums and lazy records.

2 A Preorder of Modes

The programmer specifies a set of modes $m, n, k, \ldots$ subject to a preorder $m \geq k$. Each type has an intrinsic mode, indicated by writing A_m. The preorder specifies the allowed dependencies: when we write

$$\Gamma \vdash e \iff A_k$$

we presuppose that for every $x : A_m$ in Γ we have $m \geq k$.

In addition, the programmer specifies a set of structural properties allowed for each mode, $\sigma(m) \subseteq \{\mathsf{W}, \mathsf{C}\}$. The preorder must be monotonic with respect to the structural properties, that is, if $m \geq k$ then $\sigma(m) \supseteq \sigma(k)$. This, together with independence, is paramount to guarantee that the structural properties associated with each mode are suitably enforced.

For example, consider $A_m \otimes B_m$ where m is an affine mode, that is $\sigma(m) = \{\mathsf{W}\}$. Then in $\mathbf{match}\ e\ ((x, y) \Rightarrow e')$ we have $x : A_m$ and $y : A_m$ in the typing of e'. Because m admits weakening, neither x nor y need to be used. But if there were *linear* variables in e, they might ultimately not be used because the components of the pair they construct might not be used. Analogous remarks apply to modes admitting contraction. At the technical level, for the sequent calculus, the all-important property of *cut elimination* would fail [Benton, 1994, Pruiksma et al., 2018].

We use the following names to refer to a mode m with particular properties:

- *Linear* if $\sigma(m) = \{\,\}$ ($x : A_m$ must be used exactly once)
- *Affine* if $\sigma(m) = \{\mathsf{W}\}$ ($x : A_m$ may be used at most once)
- *Strict* if $\sigma(m) = \{\mathsf{C}\}$ ($x : A_m$ must be used at least once)
- *Structural* (or *unrestricted*) if $\sigma(m) = \{\mathsf{W}, \mathsf{C}\}$. ($x : A_m$ may be used arbitrarily many times)

For the purpose of the specification and this course, we proceed as if the dynamics for each mode is exactly the same. In other words, $e \hookrightarrow v$ for $\cdot \vdash e : A_m$ applies regardless of the mode m. However, the system is set up such that different modes can have different interpretations, for example, sequential, parallel with shared memory, message-passing, or even more exotic effects. In each case we'd have to make sure the intended semantics is compatible with properties of the mode. As a small example of this we observe that the *reuse optimization* from Lecture 6 applies only to linear or affine modes, while the cut/identity optimization applies to all modes.

3 Shifts

All type constructors we have introduced so far have components at the same mode as the type itself. This means that, for example, if they whole program is linear than everything can written down *exactly* as we have written it so far. Or if the whole program is to be interpreted *structurally*, again, we just work with a single mode across the whole program and no changes are required.

But there are many examples where we would like to mix modes in the functions or the data structures. For example, we might want to have a *linear list* whose elements are *unrestricted* (for example, share structure). Or in a binary search tree, we have to compare the key with elements many times while traversing the tree, but at the same time the tree itself might be used linearly, as an ephemeral data structure. Or consider linear lists with a map function:

```
type bin = +{'b0 : bin, 'b1 : bin, 'e : 1}
type list = +{'nil : 1, 'cons : bin * list}
```

```
defn map (f : bin -> bin) (l : list) : list =
match l with
| 'nil() => 'nil()
| 'cons(hd, tl) => 'cons(f hd, map f tl)
end
```

We see that the function f is not used linearly: it is not used at all in the case of `'nil()`, and it is used twice in the case of `'cons(ht, tl)`. On the other hand, it should be possible for the argument and result to be linear lists, since the list itself is actually used linearly.

The solution is to use explicit *shift* modalities that switch between modes. It turns out there are two: a positive downshift $\downarrow_m^k A_k$ that goes from $k \geq m$ down to m, and a negative upshift $\uparrow_n^m A_n$ that goes from n to $m \geq n$. With that, we can define the language of types indexed by modes.

$$\begin{array}{lrcll} \text{Positive types} & A_m, B_m & ::= & A_m \otimes B_m \mid \mathbf{1} \mid \oplus\{\ell : A_m^\ell\}_{\ell \in L} \mid \downarrow_m^k A_k & (k \geq m) \\ \text{Negative types} & & \mid & A_m \to B_m \mid \&\{\ell : A_m^\ell\}_{\ell \in L} \mid \uparrow_n^m A_n & (m \geq n) \end{array}$$

The polarity of the shifts derives from their proof-theoretic properties in the sequent calculus [Pruiksma et al., 2018]: The downshift is invertible on the left while the upshift is invertible on the right. This also implies that the downshift is an eager constructor while the upshift is lazy. We will discuss the dynamics when we come to these operators.

Our syntax of types is mostly derived from intuitionistic linear logic, but for some modes there would be a corresponding, say, nonlinear notation that we don't use. For example, for an unrestricted mode U, one might write $A_\mathsf{U} \times B_\mathsf{U}$ instead of $A_\mathsf{U} \otimes B_\mathsf{U}$. For function types, the reverse is the case and one would typically write $A_\mathsf{L} \multimap B_\mathsf{L}$ for a linear mode L.

4 Adjoint Typing for ND

We have already set up the rules for linear ND such they will generalize to the adjoint case. It turns out, surprisingly few adjustments need to be made. For each rule and operation, we need to carefully track (a) independence, and (b) the structural properties. We will also see that except for the shifts, the language of expressions doesn't need to change at all. We now go through the connectives one by one, determining the changes to be made.

In general, for the judgment

$$\Gamma \vdash e \Longleftrightarrow A_m \mathbin{/} \Xi$$

we have that Γ tracks all variables that are lexically in scope, while Ξ contains the variables actually used in the typing of e. We note that independence does **not** apply to Γ, but to Ξ! That's because we want e to only depend on $y : A_k$ for $k \geq m$ for the variables actually used in e, not the all variables that are lexically in scope.

So our guiding principles will be to enforce:

> If $\Gamma \vdash e \Longleftrightarrow A_m \mathbin{/} \Xi$ then $\Xi \geq m$ and variables in Ξ must be used in e in accordance with their permitted structural properties.

Eager pairs, $A \otimes B$. First, the constructor.

$$\frac{\Gamma \vdash e_1 \Longleftarrow A_m \mathbin{/} \Xi_1 \quad \Gamma \vdash e_2 \Longleftarrow B_m \mathbin{/} \Xi_2}{\Gamma \vdash (e_1, e_2) \Longleftarrow A_1 \otimes A_2 \mathbin{/} \Xi_1 \mathbin{;} \Xi_2} \otimes I$$

By our invariant, we will have $\Xi_1 \geq m$ and $\Xi_2 \geq m$. As a result, $\Xi_1 \mathbin{;} \Xi_2 \geq m$ should also hold and independence comes for free.

Secondly, we need to make sure that only variables whose mode admits contraction are used in both Ξ_1 and Ξ_2. So we refine the definition of the merge operator. A variable **can** in fact appear on both sides, but only if its mode admits contraction.

$$\begin{array}{lcll}
(\Xi_1, x : A_m) & ; & (\Xi_2, x : A_m) & = & (\Xi_1 \,;\, \Xi_2), x : A_m & \text{provided } \mathsf{C} \in \sigma(m) \\
(\Xi_1, x : A_m) & ; & \Xi_2 & = & (\Xi_1 \,;\, \Xi_2), x : A_m & \text{provided } x \notin \Xi_2 \\
\Xi_1 & ; & (\Xi_2, x : A_m) & = & (\Xi_1 \,;\, \Xi_2), x : A_m & \text{provided } x \notin \Xi_1 \\
(\cdot) & ; & (\cdot) & = & (\cdot)
\end{array}$$

One point to note is that if all modes in the program admit contraction (for example, if we check a program that is entirely unrestricted) then the merge operation can never fail. Another note is that the prior definition is a special case, because for every linear mode m with have $\sigma(m) = \{\,\}$.

Let's review the prior elimination rule, writing in modes, including r for the mode of the succedent.

$$\frac{\Gamma \vdash e \Longrightarrow A_m \otimes B_m \;/\; \Xi_1 \quad \Gamma, x : A_m, y : B_m \vdash e' \Longleftarrow C_r \;/\; \Xi_2}{\Gamma \vdash (\mathbf{match}\ e\ \mathbf{with}\ (x, y) \Rightarrow e') \Longleftarrow C_r \;/\; \Xi_1 \,;\, ((\Xi_2 \setminus x) \setminus y)}\ \otimes E?$$

First we note that if the mode m allows weakening, then $(\Xi_2 \setminus x) \setminus y$ should succeed even if x and y do **not** occur in Ξ_2. So we refine the previous definition.

$$\begin{array}{lcll}
(\Xi, x : A_m) & \setminus & x_m & = & \Xi & \\
(\Xi, y : A_k) & \setminus & x_m & = & (\Xi \setminus x_m), y : A_k & \text{for } x \neq y \\
(\cdot) & \setminus & x_m & = & (\cdot) & \text{provided } \mathsf{W} \in \sigma(m)
\end{array}$$

Again, if all modes admit weakening this cannot fail, and if all modes are linear than it specializes to the previous definition.

But we are not quite home free yet: we also need to consider independence. From our invariant we conclude $\Xi_1 \geq m$ and $\Xi_2 \geq r$. What we need is $\Xi_1 \,;((\Xi_2 \setminus x_m) \setminus y_m) \geq r$. Because of transitivity of the preorder between modes, this only requires $m \geq r$. Rewriting the rule with all of this in mind:

$$\frac{\Gamma \vdash e \Longrightarrow A_m \otimes B_m \;/\; \Xi_1 \quad (m \geq r) \quad \Gamma, x : A_m, y : B_m \vdash e' \Longleftarrow C_r \;/\; \Xi_2}{\Gamma \vdash (\mathbf{match}\ e\ \mathbf{with}\ (x, y) \Rightarrow e') \Longleftarrow C_r \;/\; \Xi_1 \,;\, ((\Xi_2 \setminus x_m) \setminus y_m)}\ \otimes E$$

Our previous update for the join operator makes sure that only variables with a mode that permits contraction can be used in both premises.

Variables. The rule for variables is straightforward because both independence and any substructural requirements are automatically satisfied.

$$\frac{}{\Gamma \vdash x \Longrightarrow A_m \;/\; x : A_m}\ \mathsf{var}$$

Subtyping. Recall that the only use of subtyping appears in the transition from synthesis to checking.

$$\frac{\Gamma \vdash e \Longrightarrow A_m \;/\; \Xi \quad A_m \leq B_m}{\Gamma \vdash e \Longleftarrow B_m \;/\; \Xi}\ \Rightarrow/\Leftarrow$$

Here we just need to make sure both types have the same mode, and extend the coinductive rules for subtyping to account for the shifts covariantly.

Functions $A_m \to B_m$**.** Usually, negative types require quite different consideration from positive ones, but not in this case. The main operations are context join and removal, both of which we already generalized to account for multiple modes.

$$\frac{\Gamma, x : A_m \vdash e \Longleftarrow B_m \;/\; \Xi}{\Gamma \vdash \lambda x.\, e \Longleftarrow A_m \to B_m \;/\; (\Xi \setminus x_m)} \to I \qquad \frac{\Gamma \vdash e_1 \Longrightarrow A_m \to B_m \;/\; \Xi_1 \quad \Delta \vdash e_2 \Longleftarrow A \;/\; \Xi_2}{\Gamma\,;\Delta \vdash e_1\, e_2 \Longrightarrow B_m \;/\; \Xi_1\,;\Xi_2} \to E$$

Definitions $F[\Delta] : A_m = e$**.** Since we just considered functions, let's consider top-level definitions. To check $F[\Delta] : A_m = e$ we check

$$\frac{\Delta \geq m \quad \Delta \vdash e \Longleftarrow A_m \;/\; \Xi \quad \Xi \setminus \Delta \; \textit{defined}}{F[\Delta] : A_m = e \; \textit{valid}} \;\textsf{defn}$$

The first condition checks independence, which is a kind of well-formedness required for the type of F which abstracts over a whole context. The removal $\Xi \setminus \Delta$ removes each variable in Δ in turn, checking that variables that do not permit weakening are indeed used. At the call site we have

$$\frac{F[\Delta] : A_m = e \quad \Gamma \vdash \sigma \Longleftarrow \Delta \;/\; \Xi}{\Gamma \vdash F[\sigma] \Longrightarrow A_m \;/\; \Xi} \;\textsf{call}$$

$$\frac{\Gamma \vdash \sigma \Longleftarrow \Delta \;/\; \Xi_1 \quad \Gamma \vdash e \Longleftarrow A_m \;/\; \Xi_2}{\Gamma \vdash (\sigma, x \mapsto e) \Longleftarrow (\Delta, x : A_m) \;/\; \Xi_1\,;\Xi_2} \qquad \frac{}{\Gamma \vdash (\cdot) \Longleftarrow (\cdot) \;/\; (\cdot)}$$

Here, there is additional global requirement for Ξ since different types in the context Δ may have different modes. All of these modes are above m, so all of the Ξ_i and therefore Ξ are also above m,

Lazy Records $\&\{\ell : A^\ell_m\}_{\ell \in L}$**.** This is one of the most pervasive changes. Let's annotate the previous rule from Lecture 7 with modes to start. Because we use the mode m as a subscript, we write the label index ℓ as a superscript.

$$\frac{(\Gamma \vdash e^\ell \Longleftarrow A^\ell_m \;/\; \Xi) \quad (\forall \ell \in L)}{\Gamma \vdash \{\ell \Rightarrow e^\ell\}_{\ell \in L} \Longleftarrow \&\{\ell : A^\ell_m\}_{\ell \in L} \;/\; \Xi} \;\&I?$$

In the linear case, there will be one projection from the record, so all branches need to use the same Ξ. Now, due to weakening, there is the possibility that some branches do not use some of the variables. For this purpose we need a new operation, $\Xi_1 \sqcup \Xi_2$. It is defined as follows

$$\begin{array}{lcll}
(\Xi_1, x : A_m) & \sqcup & (\Xi_2, x : A_m) & = (\Xi_1 \sqcup \Xi_2), x : A_m \\
(\Xi_1, x : A_m) & \sqcup & \Xi_2 & = (\Xi_1 \sqcup \Xi_2), x : A_m \quad \text{provided } \mathsf{W} \in \sigma(m) \\
\Xi_1 & \sqcup & (\Xi_2, x : A_m) & = (\Xi_1 \sqcup \Xi_2), x : A_m \quad \text{provided } \mathsf{W} \in \sigma(m) \\
(\cdot) & \sqcup & (\cdot) & = (\cdot)
\end{array}$$

In the case that all modes allow weakening, this is always defined. If all modes are linear, then it specializes to the case where Ξ_1 and Ξ_2 need to be equal. We then define $\bigsqcup_{\ell \in L} \Xi^\ell$ as the iterated binary least upper bound operation. We assumed that lazy records are never empty ($L \neq \{\}$) precisely so that this is always well-defined. If empty records are allowed, we require something called *provisional bindings* [Jang et al., 2024], a complication we wish to avoid here since the payoff is small.

We can then rewrite the rule correctly as

$$\frac{(\Gamma \vdash e^\ell \Longleftarrow A^\ell_m \mathbin{/} \Xi^\ell) \quad (\forall \ell \in L) \quad \Xi = \bigsqcup_{\ell \in L} \Xi^\ell}{\Gamma \vdash \{\ell \Rightarrow e^\ell\}_{\ell \in L} \Longleftarrow \&\{\ell : A^\ell_m\}_{\ell \in L} \mathbin{/} \Xi}\ \&I$$

In contrast, the elimination rule is straightforward.

$$\frac{\Gamma \vdash e \Longrightarrow \&\{\ell : A^\ell_m\}_{\ell \in L} \mathbin{/} \Xi \quad (k \in L)}{\Gamma \vdash e.k \Longrightarrow A^k_m \mathbin{/} \Xi}\ \&E$$

Sums $\oplus\{\ell : A^\ell_m\}_{\ell \in L}$. We have already seen the key idea in lazy records, so we present the rules without further comment.

$$\frac{(k \in L) \quad \Gamma \vdash e \Longleftarrow A^k_m \mathbin{/} \Xi}{\Gamma \vdash k(e) \Longleftarrow \oplus\{\ell : A^\ell_m\}_{\ell \in L} \mathbin{/} \Xi}\ \oplus I$$

$$\frac{\Gamma \vdash e \Longrightarrow \oplus\{\ell : A^\ell_m\}_{\ell \in L} \mathbin{/} \Xi_1 \quad (m \geq r) \quad (\Gamma, x^\ell : A^\ell_m \vdash e^\ell \Longleftarrow C_r \mathbin{/} \Xi^\ell) \quad (\forall \ell \in L) \quad \Xi_2 = \bigsqcup_{\ell \in L} (\Xi^\ell \setminus x^\ell)}{\Gamma \vdash \mathbf{match}\ e\ \{\ell(x^\ell) \Rightarrow e^\ell\}_{\ell \in L} \Longleftarrow C_r \mathbin{/} \Xi_1 \,;\, \Xi_2}\ \oplus E$$

5 Shifts

We have not yet discussed the new operators and their corresponding constructors and destructors. The constructor for the downshift is $\langle e \rangle : \downarrow^k_m A_k$. Because it is positive, it represents just a wrapper around a value at a different mode. So:

$$\begin{array}{rrcll} \text{Large values} & V & ::= & (V_1, V_2) \mid ( ) \mid k(V) \mid \langle V \rangle & \text{(positive)} \\ & & \mid & \lambda x.\, e \mid \{\ell \Rightarrow e^\ell\}_{\ell \in L} \mid \mathbf{susp}\ e & \text{(negative)} \end{array}$$

Here we have anticipated the new values for the upshift, $\mathbf{susp}\ e$. Because the downshift is positive, its elimination is a **match** construct. Dynamically, we have

$$\frac{e \hookrightarrow v}{\langle e \rangle \hookrightarrow \langle v \rangle} \qquad \frac{e \hookrightarrow \langle v \rangle \quad e'(v) \hookrightarrow v'}{\mathbf{match}\ e\ (\langle x \rangle \Rightarrow e'(x)) \hookrightarrow v'}$$

From the typing perspective we need to check independence as well as structural properties.

$$\frac{\Gamma \vdash e \Longleftarrow A_k \mathbin{/} \Xi}{\Gamma \vdash \langle e \rangle \Longleftarrow \downarrow^k_m A_k \mathbin{/} \Xi}\ \downarrow I$$

By our independence invariant, we know $\Xi \geq k$ from the premise. Because $k \geq m$ by presupposition on the shift, we also have $\Xi \geq m$ and no further checks are needed.

$$\frac{\Gamma \vdash e \Longrightarrow \downarrow^k_m A_k \mathbin{/} \Xi_1 \quad (m \geq r) \quad \Gamma, x : A_k \vdash e' \Longleftarrow C_r \mathbin{/} \Xi_2}{\Gamma \vdash \mathbf{match}\ e\ (\langle x \rangle \Rightarrow e') \Longleftarrow C_r \mathbin{/} \Xi_1 \,;\, (\Xi_2 \setminus x_k)}\ \downarrow E$$

By our invariant $\Xi_1 \geq m$. Together with $m \geq r$ that is sufficient to guarantee $\Xi_1 \geq r$. Also by invariant $\Xi_2 \geq r$, so independence is preserved for the conclusion.

The upshift is negative and therefore lazy. We write the elimination form in postfix notation as $e.\mathbf{force}$, which is consistent with function application and record projection. Dynamically, we have

$$\frac{}{\mathbf{susp}\ e \hookrightarrow \mathbf{susp}\ e} \qquad \frac{e \hookrightarrow \mathbf{susp}\ e' \quad e' \hookrightarrow v'}{e.\mathbf{force} \hookrightarrow v'}$$

In the typing rules we just need to be careful to respect independence.

$$\frac{\Gamma \vdash e \Longleftarrow A_n \mathbin{/} \Xi}{\Gamma \vdash \mathbf{susp}\ e \Longleftarrow \uparrow_n^m A_n \mathbin{/} \Xi}\ \uparrow I?$$

By invariant we know $\Xi \geq n$ and by presupposition $m \geq n$. That's not enough to know that $\Xi \geq m$, so we need to enforce that with a new condition.

$$\frac{\Gamma \vdash e \Longleftarrow A_n \mathbin{/} \Xi \quad (\Xi \geq m)}{\Gamma \vdash \mathbf{susp}\ e \Longleftarrow \uparrow_n^m A_n \mathbin{/} \Xi}\ \uparrow I$$

Note that no condition on weakening would help, since Ξ already contains only the variables actually used in checking e. No such condition is needed in the elimination rule because $\Xi \geq m$ implies $\Xi \geq n$ by presupposition on the shifts.

$$\frac{\Gamma \vdash e \Longrightarrow \uparrow_n^m A_n \mathbin{/} \Xi}{\Gamma \vdash e.\mathbf{force} \Longrightarrow A_n \mathbin{/} \Xi}\ \uparrow E$$

This completes the set of rules for adjoint natural deduction and auxiliary operations on context.

6 Adjoint Types for Sax

We did not discuss adjoint types for Sax and compilation from ND to sax in this lecture in any detail. So we postpone this to a future lecture.

7 Some Examples

We show the examples with the research compiler we have developed, because the compiler for Lab 4 has not been written yet. The syntax may have some slight differences. For example, a match expression does not have an **end** marker the way it will have in the course.

We start off by declaring two modes and the preorder between them. The first U is unrestricted and the second L is linear.

```
mode U structural :> L
mode L linear
```

When defining a type, we write

```
type name[m k1 ... kn] = A
```

where m is the mode of A, and $k_1, \ldots, k_n$ are additional modes that might be used in the definition of A. For example, a list of mode m with elements of mode k could be written as

```
type nat[k] = +{'zero : 1, 'succ : <nat[k]>}
type list[m k] = +{'nil : 1, 'cons : <nat[k]> * <list[m k]>}
```

Here, m and k are *mode variables* that can be instantiated at concrete use sites. Also note the notation `<A>` which stands for $\downarrow_m^k A$, where k and m are determined from context. In the type of natural numbers, `<nat[k]>` stands for $\downarrow_k^k \mathsf{nat}_k$. This shift is technically redundant for Sax, but in a future extension it is used to specify data layout, so any recursive type in this research compiler needs to be guarded by a shift or negative type constructor.

In the type `list[m k]`, m is the mode of the list, and k the mode of the elements. So `<nat[k]>` is $\downarrow_m^k \mathsf{nat}_k$. The second shift `<list[m k]>` stands for $\downarrow_m^m \mathsf{list}_m$.

In this research compiler, we separate type declarations for metavariables from their actual definitions so we can assign it multiple types. Here are four different types for the append function.

```
decl append (l1 : list[L U]) (l2 : list[L U]) : list[L U]
decl append (l1 : list[L L]) (l2 : list[L L]) : list[L L]
decl append (l1 : list[U U]) (l2 : list[U U]) : list[U U]
decl append (l1 : list[U U]) (l2 : list[L U]) : list[L U]

defn append l1 l2 = match l1 with
| 'nil() => l2
| 'cons(<hd>, <tl>) => 'cons(<hd>, <append tl l2>)
(* end *)
```

Perhaps most surprising is the last one. It shows that we can make a linear copy of an unrestricted list by appending the empty list to it. Also note that a type such as `list[U L]` would be ill-formed, because $\mathsf{L} \not\geq \mathsf{U}$, violating the presupposition on the downshift. Also illegal would be

```
decl append (l1 : list[L U]) (l2 : list[L U]) : list[U U]  % invalid
```

because by independence we cannot have an unrestricted result depend on a linear parameter.

Functions often have many different modes, so the research compiler has *mode inference* to determine the most general mode. It does so by collecting constraints on mode variables. We will come back to this in a future lecture.

As an example of an upshift, we consider the map function. Again, multiple modes can be assigned. However, the function itself is not used linearly, so it should be upshifted if the list elements are linear.

```
decl map (f : [U] up[L] (nat[L] -> nat[L])) (l : list[L L]) : list[L L]
decl map (f : [U] up[U] (nat[U] -> nat[U])) (l : list[L U]) : list[L U]
decl map (f : [U] up[U] (nat[U] -> nat[U])) (l : list[U U]) : list[L U]

defn map f l = match l with
| 'nil() => 'nil()
| 'cons(<x>, <xs>) => 'cons(<f.force x>, <map f xs>)
(* end *)
```

Here `[U] up[L]` stands for $\uparrow_{\mathsf{L}}^{\mathsf{U}}$. This first type shows that we can map a linear function over a linear list, as long as the function's type is upshifted. This is manifest in the code by applying f.**force** x rather than just $f\ x$. Again, the most interesting may be the last type which says that even if the input list is unrestricted, we can construct a linear list as a result.

References

Andrew Barber. Dual intuitionistic linear logic. Technical Report ECS-LFCS-96-347, Department of Computer Science, University of Edinburgh, September 1996.

P. N. Benton. A mixed linear and non-linear logic: Proofs, terms and models. In Leszek Pacholski and Jerzy Tiuryn, editors, *Selected Papers from the 8th International Workshop on Computer Science Logic (CSL'94)*, pages 121–135, Kazimierz, Poland, September 1994. Springer LNCS 933. An extended version appears as Technical Report UCAM-CL-TR-352, University of Cambridge.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

Junyoung Jang, Sophia Roshal, Frank Pfenning, and Brigitte Pientka. Adjoint natural deduction. In Jakob Rehof, editor, *9th International Conference on Formal Structures for Computation and Deduction (FSCD 2024)*, pages 15:1–15:23, Tallinn, Estonia, July 2024. LIPIcs 299. Extended version available as https://arxiv.org/abs/2402.01428.

Klaas Pruiksma, William Chargin, Frank Pfenning, and Jason Reed. Adjoint logic. Unpublished manuscript, April 2018. URL http://www.cs.cmu.edu/~fp/papers/adjoint18b.pdf.

Jason Reed. A judgmental deconstruction of modal logic. Unpublished manuscript, May 2009. URL http://www.cs.cmu.edu/~jcreed/papers/jdml2.pdf.