Lecture Notes on Law and Order

15-417/817: HOT Compilation
Frank Pfenning

Lecture 11
February 20, 2025

1 Introduction

So far, we have implicitly assumed that the order of the hypotheses in the context does not matter. From the logical perspective, this means we alway assume the rule of *exchange*:

$$\frac{\Gamma_1, B, A, \Gamma_2 \vdash C}{\Gamma_1, A, B, \Gamma_2 \vdash C}\ \mathsf{exchange}$$

What does rejecting this structural property mean from the programming language perspective? Somehow it seems to say that we must use the assumptions "in order", but actually that's not quite the case. Petersen et al. [2003] pursued the notion that order may be related to data layout. And while this proved to be the case to some extent, it didn't scale very well beyond the specific technical realization in that paper. We'll hint at later in the lecture why it didn't extend.

Once one gets over this particular interpretation, though, it seems at first that ordered logic (and the corresponding ordered types) are mostly useful for *modeling*. For example, the words in a sentence are ordered, so it makes sense that a logical approach to grammars and parsing may use an ordered logic. This was the origin of ordered logic in the form of the *Lambek Calculus* [Lambek, 1958], a remarkably prescient paper that has stood the test of time. Generalized, there are multiple applications in logical frameworks [Polakow and Pfenning, 1998, 1999, 2000, Polakow, 2000, Polakow and Yi, 2000, Polakow, 2001] and modeling of other systems such as Turing machines [Pfenning, 2023].

But what about functional programming? Perhaps somewhat surprisingly, there are interesting functions that have strong ordered properties. Consider list append or reverse, tree traversals, mapping or folding over a list, etc. And not only can we program them using ordered types, but in conjunction with parametric polymorphism, their behavior can be fully characterized by their types. We will explore the latter in the next lecture. In this one, we will introduce the ordered type system and write some well-ordered functions.

2 An Ordered Type System

This and the next lecture will follow Aberlé et al. [2025], with some suitable adjustments for the context of this course. One imagines to start with

$$\Omega \vdash e : A$$

where Ω is an ordered context. But what principles should one apply to sanity-check the rules? One is clearly *preservation*: If $\cdot \vdash e : A$ and $e \hookrightarrow v$ then $v : A$. Another is progress, but we do not have a small-step semantics so progress cannot be directly formulated. Preservation ultimately relies on the substitution property:

If $\Omega \vdash e : A$ *and* $\Omega_L\,(x : A)\,\Omega_R \vdash e' : C$ *then* $\Omega_L\,\Omega\,\Omega_R \vdash [e/x]e' : C$.

One might suspect that we could leave Ω_L or Ω_R empty, but then the substitution property will fail because function types add further variables to the right or left and we still have to be able to substitute into the expression. In the sequent calculus, a corresponding test would be *cut elimination* [Kanovich et al., 2018].

Because the context is ordered, there are in fact two forms of implication: one adding a variable on the right (written $A \twoheadrightarrow B$), and one adding a variable on the left (written $A \rightarrowtail B$). Lambek's original notation $B \,/\, A$ (B *over* A) for right implication and $A \setminus B$ (A *under* B) for left implication works well in the sequent calculus and modeling, but turns out to be often counterintuitive for functional programming.

In this lecture we do not care about bidirectional checking, so we just write ordinary typing rules. Because we want to write the usual functional programs, just classify them with precise ordere types, we use the same syntax for pairs, functions, unit, etc. as we have done so far for the linear and adjoint case.

Variables. That's not controversial, because with a single variable there is no order to check.

$$\frac{}{x : A \vdash x : A}\ \mathsf{var}$$

Ordered Pairs $A \bullet B$. For the introduction rule, we have to split the context somewhere, but keep each side in order. For a context of n variables, there are $n+1$ ways to apply this rules bottom-up.

$$\frac{\Omega_A \vdash e_1 : A \quad \Omega_B \vdash e_2 : B}{\Omega_A\,\Omega_B \vdash (e_1, e_2) : A \bullet B}\ \bullet I$$

In the elimination rule, the pair must be taken from somewhere in the middle of the context. Trying to force it to be at one end or the other means that the substitution property fails. Consequently, it is not straightforward to think of the context as a queue or a stack: we can always "access" it at any point.

$$\frac{\Omega_{AB} \vdash e : A \bullet B \quad \Omega_L\,(x : A)\,(y : B)\,\Omega_R \vdash e' : C}{\Omega_L\,\Omega_{AB}\,\Omega_R \vdash \mathbf{match}\ e\ ((x, y) \Rightarrow e') : C}\ \bullet E$$

Note that x and y are inserted in the context where Ω was.

We can already try to convince ourselves that $A \bullet B$ is non-commutative. The best approach for a formal proof would be a cut-free sequent calculus.

$$\vdots$$
$$p : A \bullet B \vdash ? : B \bullet A$$

We cannot apply $\bullet I$, because p would only be available for either B or A, but not both. So we can apply $\bullet E$, and in the first premise only the variable rule is promising.

$$\frac{\dfrac{}{p : A \bullet B \vdash p : A \bullet B}\ \mathsf{var} \qquad \begin{array}{c}\vdots\\ (x : A)\,(y : B) \vdash ? : B \bullet A\end{array}}{p : A \bullet B \vdash \mathbf{match}\ p\ ((x, y) \Rightarrow ?) : B \bullet A}\ \bullet E$$

But now applying $\bullet R$ does not work, because we need to split the context $(x : A)\,(y : B)$ moving the left part to prove B and the right part to prove A.

Left and right functions. The introductions add the new variables either to the right end or the left end of the context. There is no need to introduce any new expression, though, since ultimately they both are functions.

$$\frac{\Omega\,(x : A) \vdash e : B}{\Omega \vdash \lambda x.\, e : A \twoheadrightarrow B}\twoheadrightarrow I \qquad \frac{(x : A)\,\Omega \vdash e : B}{\Omega \vdash \lambda x.\, e : A \rightarrowtail B}\rightarrowtail I$$

We could also try to have a kind of function that inserts x arbitrarily somewhere in the middle of the context, but that would fail out substitution test (or cut elimination in a sequent calculus).

Like $A \bullet B$, the elimination rules split the the context when read bottom up, but at fixed place. Read top-down, they concatenate the contexts. We just need to make sure we have the correct form of concatenation for each form of function.

$$\frac{\Omega_{AB} \vdash e_1 : A \twoheadrightarrow B \quad \Omega_A \vdash e_2 : A}{\Omega_{AB}\,\Omega_A \vdash e_1\,e_2 : B}\twoheadrightarrow E \qquad \frac{\Omega_{AB} \vdash e_1 : A \rightarrowtail B \quad \Omega_A \vdash e_2 : A}{\Omega_A\,\Omega_{AB} \vdash e_1\,e_2 : B}\rightarrowtail E$$

Because the variable x in $A \twoheadrightarrow B$ is added on the right side of the context, that's where the argument in a function application need to be checked. The reverse is true for $A \rightarrowtail B$.

Twist. We see that the left function $A \rightarrowtail B$ crosses the two parts of the context when compared to the order of the subexpression e_1 and e_2. This suggests there may be another form of conjunction $A \circ B$ (read: A *twist* B) that behaves similarly. Indeed:

$$\frac{\Omega_A \vdash e_1 : A \quad \Omega_B \vdash e_2 : B}{\Omega_B\,\Omega_A \vdash (e_1, e_2) : A \circ B}\circ I \qquad \frac{\Omega_{AB} \vdash e : A \circ B \quad \Omega_L\,(y : B)\,(x : A)\,\Omega_R \vdash e' : C}{\Omega_L\,\Omega_{AB}\,\Omega_R \vdash \mathbf{match}\ e\ ((x, y) \Rightarrow e') : C}\circ E$$

Note that x and y are added to the context in reverse order, to mirror the fact that the contexts in the introduction form are crossed.

3 Some Examples

We consider the function to curry a function from pairs,

$$g : (A \bullet B) \twoheadrightarrow C \vdash \lambda x.\, \lambda y.\, g\,(x, y) : A \twoheadrightarrow (B \twoheadrightarrow C)$$

It may be useful to write out the typing derivation, we just note that after two steps we are at

$$(g : (A \bullet B) \twoheadrightarrow C)\,(x : A)\,(y : B) \vdash g\,(x, y) : C$$

Now the typing succeeds because x and y are to the right of g and in the correct order. Note that both implication $A \twoheadrightarrow (B \twoheadrightarrow C)$ need to be right implications.

Perhaps surprisingly, we can give exactly the exact function another type!

$$g : (A \circ B) \rightarrowtail C \vdash \lambda x.\, \lambda y.\, g\,(x, y) : A \rightarrowtail (B \rightarrowtail C)$$

After two steps we arrive at

$$(y : B)\,(x : A)\,(g : (A \circ B) \rightarrowtail C) \vdash g\,(x, y) : C$$

We see that y and x are on the correct side of g, and in the correct (twisted) order.

We can uncurry as well.

$$f : A \twoheadrightarrow (B \twoheadrightarrow C) \vdash \lambda p.\, \mathbf{match}\ p\ ((x, y) \Rightarrow f\, x\, y) : (A \bullet B) \twoheadrightarrow C$$

After two steps we arrive at

$$(f : A \twoheadrightarrow (B \twoheadrightarrow C))\, (x : A)\, (y : B) \vdash f\, x\, y : C$$

We see that x and y are to the right of f and in the correct order. Again, we can dualize this to

$$f : A \rightarrowtail (B \rightarrowtail C) \vdash \lambda p.\, \mathbf{match}\ p\ ((x, y) \Rightarrow f\, x\, y) : (A \circ B) \rightarrowtail C$$

After two steps, we arrive at

$$(y : B)\, (x : A)\, (f : A \rightarrowtail (B \rightarrowtail C)) \vdash f\, x\, y : C$$

where, again, y and x are on the correct side of f, in the correct order.

In the research compiler, we currently don't have polymorphism so we need to fake this using type definition that are not related by subtypes. The order feature is experimental, and checking proceeds by first checking linearity for ordered modes, and then order when an instance declaration **inst** with an ordered mode is given. We use the following concrete syntax:

$A \bullet B$	`A * B`
$A \circ B$	`A @ B`
$A \twoheadrightarrow B$	`A -> B`
$A \rightarrowtail B$	`A \ B`

This means that the right implication and fuse are consistent with linear functions and pairs, reading them left to right.

As explained in the last lecture, we separate declarations from definitions, so we can give the same definition multiple different types.

```
mode U structural :> O
mode O ordered

(* fake polymorphism *)
type A[m] = +{'a : 1}
type B[m] = +{'b : 1}
type C[m] = +{'c : 1}

decl fuse (x : A[m]) (y : B[m]) : A[m] * B[m]
decl fuse (x : A[m]) (y : B[m]) : A[m] @ B[m]
defn fuse x y = (x, y)

inst fuse (x : A[O]) (y : B[O]) : A[O] * B[O]
fail
inst fuse (x : A[O]) (y : B[O]) : A[O] @ B[O]

decl uncurry (f : A[m] -> B[m] -> C[m]) : (A[m] * B[m] -> C[m])
decl uncurry (f : A[m] \ B[m] \ C[m]) : (A[m] @ B[m] \ C[m])
defn uncurry f = fun p => match p with | (x, y) => f x y
```

```
inst uncurry (f : A[O] -> B[O] -> C[O]) : (A[O] * B[O] -> C[O])
inst uncurry (f : A[O] \ B[O] \ C[O]) : (A[O] @ B[O] \ C[O])

decl curry (g : A[m] * B[m] -> C[m]) : A[m] -> B[m] -> C[m]
decl curry (g : A[m] @ B[m] \ C[m]) : A[m] \ B[m] \ C[m]
defn curry g = fun x => fun y => g (x, y)

inst curry (g : A[O] * B[O] -> C[O]) : A[O] -> B[O] -> C[O]
inst curry (g : A[O] @ B[O] \ C[O]) : A[O] \ B[O] \ C[O]
```

In general we can swap left and right arrows together with fuse and twist and preserve typability, as long as we also swap the arguments of top-level definitions if there is more than one. They are typed with in order:

$$\frac{\Omega \vdash e : A}{F\,[\Omega] : A = e\ \mathit{valid}}\ \textsf{defn} \qquad \frac{F\,[\Omega_F] : A = e \quad \Omega \vdash \sigma : \Omega_F}{\Omega \vdash F[\sigma] : A}\ \textsf{call}$$

$$\frac{\Omega_L \vdash \sigma : \Omega \quad \Omega_R \vdash e : A}{\Omega_L\,\Omega_R \vdash (\sigma, x \mapsto e) : (\Omega, x : A)} \qquad \frac{}{\cdot \vdash (\cdot) : (\cdot)}$$

That is, the arguments are seen as constituting an ordered context, so as in the example of the `fuse` function above, their order is relevant.

4 Additional Connectives

Unit, sums and lazy records are somewhat boring. Because type definitions are equirecursive, they do not explicitly contribute to the rules. Even though order has to be respected, these connectives do not split into two, the way that pairs and functions do.

Unit. It looks like nothing special, although as we see in Lecture 13, it actually is tricky for type-checking purposes.

$$\frac{}{\cdot \vdash (\,) : \mathbf{1}}\ \mathbf{1}I \qquad \frac{\Omega_1 \vdash e : \mathbf{1} \quad \Omega_L\,\Omega_R \vdash e' : C}{\Omega_L\,\Omega_1\,\Omega_R \vdash \mathbf{match}\ e\ (() \Rightarrow e') : C}\ \mathbf{1}E$$

The reason this ends up being tricky is because there is no explicit marker in the $\Omega_L\,\Omega_R$ that helps us decide where Ω_1 should be inserted.

Sums. The requirements on the elimination rule are that all branches must use the same ordered context. In that way, it isn't significantly different from the linear case.

$$\frac{\Omega \vdash e : A_k}{\Omega \vdash k(e) : \oplus\{\ell : A_\ell\}_{\ell \in L}}\ \oplus I \qquad \frac{\Omega \vdash e : \oplus\{\ell : A_\ell\}_{\ell \in L} \quad (\Omega_L\,(x_\ell : A_\ell)\,\Omega_R \vdash e_\ell : C) \quad (\forall \ell \in L)}{\Omega_L\,\Omega\,\Omega_R \vdash \mathbf{match}\ e\ \{\ell(x_\ell) \Rightarrow e_\ell\}_{\ell \in L} : C}\ \oplus E$$

Lazy Records.

$$\frac{(\Omega \vdash e_\ell : A_\ell) \quad (\forall \ell \in L)}{\Omega \vdash \{\ell \Rightarrow e_\ell\}_{\ell \in L} : \&\{\ell : A_\ell\}_{\ell \in L}}\ \&I \qquad \frac{\Omega \vdash e : \&\{\ell : A_\ell\}_{\ell \in L} \quad (k \in L)}{\Omega \vdash e.k : A_k}\ \&E$$

5 Additional Examples

For lists, we define two different types. An llist A is typed left-to-right, the way we generally think about it being laid out. That is,

$$\frac{\Omega_1 \vdash v_1 : A \quad \ldots \quad \Omega_n \vdash v_n : A}{\Omega_1 \ldots \Omega_n \vdash \underline{\mathsf{cons}}(v_1, \ldots, \underline{\mathsf{cons}}(v_n, \underline{\mathsf{nil}}\,())) : \mathsf{llist}\ A}$$

We can define this type as

```
type llist[m] = +{'nil : 1, 'cons : <A[m]> * <llist[m]>}
```

where we do not distinguish the mode of the elements from the mode of the list. Ignore the shifts $\langle - \rangle$ and remember that `A * B` stands for $A \bullet B$.

With this type we can define the ordered append function.

```
decl append (l1 : llist[m]) (l2 : llist[m]) : llist[m]

defn append l1 l2 = match l1 with
| 'nil() => l2
| 'cons(<x>, <xs>) => 'cons(<x>, <append xs l2>)

inst append (l1 : llist[O]) (l2 : llist[O]) : llist[O]
```

A remarkable fact, explained in the next lecture, is that up to extensional equality the append function is the only one inhabiting the given polymorphic type!

An rlist has the same values, but is typed in reverse.

$$\frac{\Omega_n \vdash v_n : A \quad \ldots \quad \Omega_1 \vdash v_1 : A}{\Omega_n \ldots \Omega_1 \vdash \underline{\mathsf{cons}}(v_1, \ldots, \underline{\mathsf{cons}}(v_n, \underline{\mathsf{nil}}\,())) : \mathsf{rlist}\ A}$$

This type can be defined as

```
type rlist[m] = +{'nil : 1, 'cons : <A[m]> @ <rlist[m]>}
```

Recall that `A @ B` stands for $A \circ B$.

With this, we can define the ordered reverse function. Here we need to list the accumulator argument l_2 *before* the input list l_1. We have indicated the typing context and result type as a comment in the cons branch of the pattern match. Because the pair in an rlist is typed with a twist, l_2 and x must be in reverse order and before xs in order to type the recursive call.

```
decl reverse (l2 : rlist[m]) (l1 : llist[m]) : rlist[m]

defn reverse l2 l1 = match l1 with
| 'nil() => l2
| 'cons(<x>, <xs>) => (* (l2 : rlist A) (x : A) (xs : llist A) |- rlist A *)
                      reverse ('cons(<x>, <l2>)) xs

inst reverse (l2 : rlist[O]) (l1 : llist[O]) : rlist[O]
inst reverse (l2 : rlist[U]) (l1 : llist[U]) : rlist[U]
```

Again, somewhat remarkably, modulo extensional equality (and there are many ways to define list reversal) the reverse function is the only one inhabiting the first of the two given instance type. We also show the second, which means we can apply exactly the same function to unordered lists, of course, with the same results.

Remember that at the point of this writing, the polymorphism in the lists is only faked, but since the functions do not analyze the structure of the list elements, the examples could be typed polymorphically in a future extension.

At the moment, we do not know if order can be exploited inside a compiler for efficiency, perhaps in a way that is analogous to the way the lack of contraction can be exploited for in-place update. One point is that the elimination rules for positives replace the middle of the ordered context with some number of new variables, and possibly none. As such, *adjacency* is not preserved by ordered types, only the relative order of data in their layout. This is why I think the original approach to data layout by Petersen et al. [2003] doesn't scale well beyond a certain allocation strategy.

References

C. B. Aberlé, Chris Martens, and Frank Pfenning. Substructural parametricity. Submitted, February 2025. URL http://www.cs.cmu.edu/~fp/papers/ordered25.pdf.

Max Kanovich, Stepan Kuznetsov, Vivek Nigam, and Andre Scedrov. A logical framework with commutative and non-commutative subexponentials. In *International Joint Conference on Automated Reasoning (IJCAR 2018)*, pages 228–245. Springer LNAI 10900, 2018.

Joachim Lambek. The mathematics of sentence structure. *The American Mathematical Monthly*, 65 (3):154–170, 1958.

Leaf Petersen, Robert Harper, Karl Crary, and Frank Pfenning. A type theory for memory allocation and data layout. In G. Morrisett, editor, *Conference Record of the 30th Annual Symposium on Principles of Programming Languages (POPL'03)*, pages 172–184, New Orleans, Louisiana, January 2003. ACM Press. Extended version available as Technical Report CMU-CS-02-171, December 2002.

Frank Pfenning. Substructural logics, Fall 2023. Course materials including lecture notes available at https://www.cs.cmu.edu/~fp/courses/15836-f23/.

Jeff Polakow. Linear logic programming with an ordered context. In M. Gabbrielli and F. Pfenning, editors, *Conference on Principles and Practice of Declarative Programming (PPDP 2000)*, pages 68–79, Montreal, Canada, September 2000. ACM.

Jeff Polakow. *Ordered Linear Logic and Applications*. PhD thesis, Department of Computer Science, Carnegie Mellon University, August 2001.

Jeff Polakow and Frank Pfenning. Ordered linear logic programming. Technical Report CMU-CS-98-183, Department of Computer Science, Carnegie Mellon University, December 1998.

Jeff Polakow and Frank Pfenning. Natural deduction for intuitionistic non-commutative linear logic. In J.-Y. Girard, editor, *Proceedings of the 4th International Conference on Typed Lambda Calculi and Applications (TLCA'99)*, pages 295–309, L'Aquila, Italy, April 1999. Springer-Verlag LNCS 1581.

Jeff Polakow and Frank Pfenning. Properties of terms in continuation-passing style in an ordered logical framework. In Joëlle Despeyroux, editor, *2nd Workshop on Logical Frameworks and Meta-languages (LFM'00)*, Santa Barbara, California, June 2000. Proceedings available as INRIA Technical Report.

Jeff Polakow and Kwangkeun Yi. Proving syntactic properties of exceptions in an ordered logical framework. In *Proceedings of the First Asian Workshop on Programming Languages and Systems (APLAS'00)*, pages 23–32, December 2000.