Lecture Notes on Linear Functional Programming

15-814: Types and Programming Languages
Frank Pfenning

Lecture 19
Tue Nov 18, 2025

1 Introduction

The idea that linear functional programs can avoid garbage collection goes back to Girard and Lafont [1987]. In various forms, it has now made its way into mainstream programming languages like Rust [Klabnik et al., 2026]. On the functional side, a significant discovery has been that linear and affine type systems can support update-in-place even for purely functional programs [Hofmann, 2000, Lorenzen et al., 2023]. Roughly: instead of simply deallocating a (linear or affine) memory cell when we read it, we reuse immediately.

In this lecture we can only scratch the surface of these developments. In practice, the difficulty is that some part of programs can be linear of affine, while others can not be, due to the nature of the algorithm that a program implements. This means we have to look for ways to combine linear and nonlinear types into a coherent language. Girard's [1987] solution of the exponential modality turns out to be quite verbose in practice. A more usable alternative is Benton's LNL [1994] in which we can seamlessly switch back and forth between linear and nonlinear programs. A generalization to allow a richer integration of sublanguages has been proposed by Jang et al. [2024] and implemented in the SNAX language [Snax]. Since LAMBDA does not support substructural types, we will roughly follow the latter paper and use the Snax implementation for our illustrative code.

2 Linear Typing

We already have the rules of linear logic, so linear typing is just a matter of annotating them with proof terms. Plus, we should add recursive types and fixed points for recursive definitions and see how they fit in. So our judgment is just $\Gamma \vdash e : \tau$ where we need to keep in mind that the hypotheses Γ are *linear*, that is, each should be used exactly once. We will see in Section 3 precisely what this means.

First the rule for hypotheses which is entirely straightforward. Recall that x must be the only variable in the context.

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

2.1 Multiplicatives

The first multiplicative is the eager pair. We recall the logical rules for comparison.

$$\frac{\Gamma \vdash A \quad \Delta \vdash B}{\Gamma, \Delta \vdash A \otimes B}\ \otimes I \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Delta \vdash e_2 : \tau_2}{\Gamma, \Delta \vdash \langle e_1, e_2 \rangle : \tau_1 \otimes \tau_2}\ \mathsf{tp/pair}$$

$$\frac{\Gamma \vdash A \otimes B \quad \Delta, x : A, y : B \vdash C}{\Gamma, \Delta \vdash C}\ \otimes E^{x,y} \qquad \frac{\Gamma \vdash e : \tau_1 \otimes \tau_2 \quad \Delta, x : \tau_1, y : \tau_2 \vdash e' : \tau'}{\Gamma, \Delta \vdash \mathsf{case}\ e\ (\langle x, y \rangle \Rightarrow e') : \tau'}\ \mathsf{tp/case/pair}$$

Observe that the typing rules are almost the same as for LAMBDA, the only difference being that we have to split the hypotheses between the premises. This is a critical property that allows for an effective combination of a mixed linear/nonlinear language. The same principle still applies for functions.

$$\frac{\Gamma, x : A \vdash B}{\Gamma \vdash A \multimap B}\ \multimap I \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.\, e : A \multimap B}\ \mathsf{tp/lam}$$

$$\frac{\Gamma \vdash A \multimap B \quad \Delta \vdash A}{\Gamma, \Delta \vdash B}\ \multimap E \qquad \frac{\Gamma \vdash e_1 : \tau_2 \multimap \tau_1 \quad \Delta \vdash e_2 : \tau_2}{\Gamma, \Delta \vdash e_1\, e_2 : \tau_1}\ \mathsf{tp/app}$$

At this point we can already write a simple function and see that it typechecks in the linear system.

$$\dfrac{\dfrac{\dfrac{}{p : \tau_1 \otimes \tau_2 \vdash p : \tau_1 \otimes \tau_2}\ \mathsf{var} \quad \dfrac{\dfrac{}{y : \tau_2 \vdash y : \tau_2}\ \mathsf{var} \quad \dfrac{}{x : \tau_1 \vdash x : \tau_1}\ \mathsf{var}}{x : \tau_1, y : \tau_2 \vdash \langle y, x \rangle : \tau_2 \otimes \tau_1}\ \mathsf{tp/pair}}{p : \tau_1 \otimes \tau_2 \vdash \mathsf{case}\ p\ (\langle x, y \rangle \Rightarrow \langle y, x \rangle) : \tau_2 \otimes \tau_1}\ \mathsf{tp/case/pair}}{\cdot \vdash \lambda p.\, \mathsf{case}\ p\ (\langle x, y \rangle \Rightarrow \langle y, x \rangle) : \tau_1 \otimes \tau_2 \multimap \tau_2 \otimes \tau_1}\ \mathsf{tp/lam}$$

This example should remind us that the context is unordered, so in the application of the tp/pair rule we can carry y to the first premise and x to the second. We can also see that the exact same function also has type $\tau_1 \times \tau_2 \to \tau_2 \otimes \tau_1$. In general, expressions are the same but the typing is stricter. Here is an example that doesn't type-check.

$$\dfrac{\dfrac{\mathsf{XXX}}{x : \tau, y : \sigma \vdash x : \tau}\ \mathsf{tp/lam}}{x : \tau \vdash \lambda y.\, x : \sigma \multimap \tau}\ \mathsf{tp/lam}}{\cdot \vdash \lambda x.\, \lambda y.\, x : \tau \multimap \sigma \multimap \tau}$$

The typing fails because we have an extraneous hypotheses which is not allowed in a linear type system. If our type system where *affine* we could either explicitly or silently drop y and the function would check.

To complete the multiplicative, we show the rules for the unit.

$$\frac{}{\cdot \vdash 1}\ 1I \qquad \frac{}{\cdot \vdash \langle\rangle : 1}\ \mathsf{tp/unit}$$

$$\frac{\Gamma \vdash 1 \quad \Delta \vdash C}{\Gamma, \Delta \vdash C}\ 1E \qquad \frac{\Gamma \vdash e : 1 \quad \Delta \vdash e' : \tau'}{\Gamma, \Delta \vdash \mathsf{case}\ e\ (\langle\rangle \Rightarrow e') : \tau'}\ \mathsf{tp/case/unit}$$

2.2 Additives

As we have seen, conjunction comes in two forms, the second one here being for lazy pairs. Except for the precise ways that the contexts are propagated these aren't very interesting at this point.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \,\&\, B}\ \&I \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \,\&\, \tau_2}\ \mathsf{tp/lpair}$$

$$\frac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash A}\ \&E_1 \qquad \frac{\Gamma \vdash A \,\&\, B}{\Gamma \vdash B}\ \&E_2 \qquad \frac{\Gamma \vdash e : \tau_1 \,\&\, \tau_2}{\Gamma \vdash e.\pi_1 : \tau_1}\ \mathsf{tp/proj}_1 \qquad \frac{\Gamma \vdash e : \tau_1 \,\&\, \tau_2}{\Gamma \vdash e.\pi_2 : \tau_2}\ \mathsf{tp/proj}_2$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \oplus B}\ \oplus I_1 \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \oplus B}\ \oplus I_2 \qquad \frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathbf{l} \cdot e : \tau_1 \oplus \tau_2}\ \mathsf{tp/inj}_1 \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathbf{r} \cdot e : \tau_1 \oplus \tau_2}\ \mathsf{tp/inj}_2$$

$$\frac{\Gamma \vdash A \oplus B \quad \Delta, x : A \vdash C \quad \Delta, y : B \vdash C}{\Gamma, \Delta \vdash C}\ \oplus E$$

$$\frac{\Gamma \vdash e : \tau_1 \oplus \tau_2 \quad \Delta, x : \tau_1 \vdash e_1 : \sigma \quad \Delta, y : \tau_2 \vdash e_2 : \sigma}{\Gamma, \Delta \vdash \mathsf{case}\ e\ (\mathbf{l} \cdot x \Rightarrow e_1 \mid \mathbf{r} \cdot y \Rightarrow e_2) : \sigma}\ \mathsf{tp/case/plus}$$

The nullary cases of these constructors also exist. $\top$ lacks an elimination rule, and 0 lacks an introduction rule.

$$\frac{}{\Gamma \vdash \top}\ \top I \qquad \frac{}{\Gamma \vdash \langle\,\rangle : \top}\ \mathsf{tp/lunit}$$

$$\frac{\Gamma \vdash 0}{\Gamma, \Delta \vdash C}\ 0E \qquad \frac{\Gamma \vdash e : 0}{\Gamma, \Delta \vdash \mathsf{case}\ e\ (\,) : \sigma}\ \mathsf{tp/case/zero}$$

We forego the exponentials, since the reintroduction of the structural rules is handled differently in SNAX.

3 Dynamics with a Global Environment

Let's recall simple, standard rule of reduction.

$$\frac{v\ \mathit{value}}{(\lambda x.\, e)\, v \mapsto [v/x]e}\ \mathsf{beta}$$

This rule still works exactly like this even in the linear case, because we have a linear substitution property. Since $(\lambda x.\, e)$ and v are both closed, we just need that $\cdot \vdash v : \tau$ and $x : \tau \vdash e : \sigma$ implies that $\cdot \vdash [v/x]e : \tau$. A slightly more general form is the following:

Property 1 (Linear Substitution) *If* $\Gamma \vdash e_1 : \tau$ *and* $\Delta, x : \tau \vdash e_2 : \sigma$ *then* $\Gamma, \Delta \vdash [e_1/x]e_2 : \sigma$.

The "problem" is that the original form of dynamics, even though it still applies, obscures the interesting properties of linearity. Specifically, whether the program is typed nonlinearly (as in LAMBDA) or linearly, the computation of the program is exactly the same. This means we cannot even talk about garbage collection, or show that none is required!

We therefore need a semantics that makes some form of storage explicit. For this purpose we use a *global environment* that maps variables to values. It will turn out that the values don't necessarily have to be closed, which is another departure from the small step substitution semantics we have used so far.

The state of computation is described by a pair $\eta \mid e$ where η is an environment and e is an expression. Because it is a bit different from the usual form of substitution, we write $\eta = (x_1 = v_1, \ldots, x_n = v_n)$. We start with a rule that *changes* the global environment.

$$\frac{v \ \textit{value} \quad x \notin \mathrm{dom}(\eta)}{\eta \mid (\lambda x.\, e)\, v \ \mapsto \ (\eta, x = v) \mid e}\ \mathsf{beta}$$

We assume that we can silently rename x so that the binding is to a fresh variable, avoiding ambiguity. We also assume that the value judgment does not presuppose that v is closed (even if variables by themselves still don't count as values). For example, we have $\lambda x.\, y$ *value* but **not** $\langle (\lambda x.\, x), y \rangle$ *value*.

We now also have to change the congruence rules to account for a possible change to the global environment.

$$\frac{\eta \mid e_1 \ \mapsto \ \eta' \mid e_1'}{\eta \mid e_1\, e_2 \ \mapsto \ \eta' \mid e_1'\, e_2}\ \mathsf{step/app}_1$$

$$\frac{v_1 \ \textit{value} \quad \eta \mid e_2 \ \mapsto \ \eta' \mid e_2'}{\eta \mid v_1\, e_2 \ \mapsto \ \eta' \mid v_1\, e_2'}\ \mathsf{step/app}_2$$

All the other rules are updated in a similar way. We show just one.

$$\frac{v_1 \ \textit{value} \quad v_2 \ \textit{value} \quad x, y \notin \mathrm{dom}(\eta)}{\eta \mid \mathsf{case}\ \langle v_1, v_2 \rangle\ (\langle x, y \rangle \Rightarrow e_3) \ \mapsto \ (\eta, x = v_1, y = v_2) \mid e_3}\ \mathsf{step/case/pair}$$

Finally, we come to a rule that *uses* the environment. There is only one such rule, in which variables are looked up.

$$\frac{}{(\eta, x = v) \mid x \ \mapsto \ \eta \mid v}\ \mathsf{step/var(L)} \qquad \left[\ \frac{x = v \in \eta}{\eta \mid x \ \mapsto \ \eta \mid v}\ \mathsf{step/var(U)}\ \right]$$

The key point is that we explicitly remove the binding for x when lookup up a linear variable. Nonlinear variables instead would keep the binding for x. That is, the rule on the right would work in a dynamics for LAMBDA while the one of the left works for a purely linear language.

Here are some sample computations for the purely linear case.

$$\begin{array}{lll} & \cdot & \mid\ (\lambda y.\, y)\, ((\lambda x.\, x)\, 3) \\ \mapsto & x = 3 & \mid\ (\lambda y.\, y)\, x \\ \mapsto & \cdot & \mid\ (\lambda y.\, y)\, 3 \\ \mapsto & y = 3 & \mid\ y \\ \mapsto & \cdot & \mid\ 3 \end{array}$$

When evaluating an expression of non-observable type like a function or lazy pair, then the environment may not be entirely empty at the end. Here is an example:

$$\begin{array}{lll} & \cdot & \mid\ (\lambda f.\, f)\, ((\lambda x.\, \lambda y.\, y\, x)\, 3) \\ \mapsto & x = 3 & \mid\ (\lambda f.\, f)\, (\lambda y.\, y\, x) \\ \mapsto & x = 3, f = \lambda y.\, y\, x & \mid\ f \\ \mapsto & x = 3 & \mid\ \lambda y.\, y\, x \end{array}$$

For fully observable types τ^+ (built out of eager pairs and unit, sums, and recursive types) we get the following theorem.

Theorem 2 (Freedom from Garbage) *If* $\cdot \vdash e : \tau^+$ *and* $\cdot \mid e \mapsto^* \eta \mid v$ *then* $\eta = (\cdot)$.

By repeating the usual progress and preservation theorems we also know that $\cdot \vdash v : \tau^+$.

We could force the global environment to be empty, and also be closer to an implementation if we constructed *closures* for values that are λ-abstractions or lazy pairs. An example rule would might read:

$$\frac{\mathrm{dom}(\eta_2) = \mathrm{FV}(\lambda x.\, e)}{\eta_1, \eta_2 \mid \lambda x.\, e \quad \mapsto \quad \eta_1 \mid (\eta_2, \lambda x.\, e)}$$

where $\mathrm{FV}(e)$ computes the free variables of e. This, however, requires further cascading changes since certain values are now closures which have to be unpacked when functions are applied, etc.

4 Linear Type Checking

As usual, we can turn the typing rules into bidirectional rules. We just show those for functions and sums.

$$\frac{}{x \Rightarrow \tau \vdash x \Leftarrow \tau}\ \mathsf{var/linear} \qquad \frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash e \Leftarrow \tau}\ \Rightarrow/\Leftarrow$$

$$\frac{\Gamma, x \Rightarrow \tau_1 \vdash e \Leftarrow \tau_2}{\Gamma \vdash \lambda x.\, e \Leftarrow \tau_1 \multimap \tau_2}\ \mathsf{tp/lam} \qquad \frac{\Gamma \vdash e_1 \Rightarrow \tau_2 \multimap \tau_1 \quad \Delta \vdash e_2 \Leftarrow \tau_2}{\Gamma, \Delta \vdash e_1\, e_2 \Rightarrow \tau_1}\ \mathsf{tp/app}$$

$$\frac{\Gamma \vdash e \Leftarrow \tau_1}{\Gamma \vdash \mathbf{l} \cdot e \Leftarrow \tau_1 \oplus \tau_2}\ \mathsf{tp/inj}_1 \qquad \frac{\Gamma \vdash e \Leftarrow \tau_2}{\Gamma \vdash \mathbf{r} \cdot e \Leftarrow \tau_1 \oplus \tau_2}\ \mathsf{tp/inj}_2$$

$$\frac{\Gamma \vdash e \Rightarrow \tau_1 \oplus \tau_2 \quad \Delta, x \Rightarrow \tau_1 \vdash e_1 \Leftarrow \sigma \quad \Delta, y \Rightarrow \tau_2 \vdash e_2 \Leftarrow \sigma}{\Gamma, \Delta \vdash \mathsf{case}\ e\ (\mathbf{l} \cdot x \Rightarrow e_1 \mid \mathbf{r} \cdot y \Rightarrow e_2) \Leftarrow \sigma}\ \mathsf{tp/case/plus}$$

We would expect the following modes:

$\Gamma^+ \vdash e^+ \Leftarrow \tau^+$
$\Gamma^+ \vdash e^+ \Rightarrow \tau^-$

On quick check we see that these are not correct! The difficulty is that even if we know Γ, Δ we know neither Γ nor Δ! There are essentially two approaches to generalize these two judgments for the purpose of type-checking.

The *additive* approach passes all variables that are in scope to all premises, as we previously did in the nonlinear typing rules. In addition, we generate a context with the variables that are *actually* used as an output of the judgment. In rules such as tp/app when then have to combine the outputs.

The *subtractive* approach passes all variables that may still be used into the first premise, but then generates, as an output, a context with the variables that have *not* be used.

It turns out in this situation the additive approach is bit simpler. So we have two judgments

$\Gamma^+ \vdash e^+ \Leftarrow \tau^+ \dashv \Delta^-$
$\Gamma^+ \vdash e^+ \Leftarrow \tau^- \dashv \Delta^-$

Here are the theorems relating these to the judgment detailed above, where we combine a couple of statements writing "$\Leftrightarrow$" for either checking or synthesis.

Theorem 3 (Additing Typing)

Soundness *If* $\Gamma \vdash e \Leftrightarrow \tau \dashv \Delta$ *then* $\Gamma \supseteq \Delta$ *and* $\Delta \vdash e \Leftrightarrow \tau$.

Completeness *If* $\Delta \vdash e \Leftrightarrow \tau$ *then for every* $\Gamma \supseteq \Delta$, $\Gamma \vdash e \Leftrightarrow \tau \dashv \Delta$.

We show a few rules, which can be generalized to all constructs except for the additive units 0 and $\top$ (empty sums and lazy records). You can find the difficulties they represent for type-checking and the techniques to handle them in Jang et al. [2024].

For variables, we only need to pick out the needed hypothesis. We show the new output context in red. Because Γ is determined just by scoping, it may contain many variables, but only one is actually used. Nothing interesting happens at the transition between synthesis and checking.

$$\frac{x \Rightarrow \tau \in \Gamma}{\Gamma \vdash x \Leftarrow \tau \dashv x \Rightarrow \tau}\ \mathsf{var/linear} \qquad \frac{\Gamma \vdash e \Rightarrow \tau \dashv \Delta}{\Gamma \vdash e \Leftarrow \tau \dashv \Delta}\ \Rightarrow/\Leftarrow$$

λ-abstractions require us to check that the bound variable was actually used. And if so, we have to remove it. Otherwise, this rule will fail to apply.

$$\frac{\Gamma, x \Rightarrow \tau_1 \vdash e \Leftarrow \tau_2 \dashv \Delta, x \Rightarrow \tau_1}{\Gamma \vdash \lambda x.\, e \Leftarrow \tau_1 \multimap \tau_2 \dashv \Delta}\ \mathsf{tp/lam}$$

For applications, we have to combine the contexts going downward. It is implicit in the notation Δ_1, Δ_2 that they share no variables. Otherwise, this rule will fail as it should.

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_2 \multimap \tau_1 \dashv \Delta_1 \quad \Gamma \vdash e_2 \Leftarrow \tau_2 \dashv \Delta_2}{\Gamma \vdash e_1\, e_2 \Rightarrow \tau_1 \dashv \Delta_1, \Delta_2}\ \mathsf{tp/app}$$

The constructor rules for sums are straightforward.

$$\frac{\Gamma \vdash e \Leftarrow \tau_1 \dashv \Delta}{\Gamma \vdash \mathbf{l} \cdot e \Leftarrow \tau_1 \oplus \tau_2 \dashv \Delta}\ \mathsf{tp/inj}_1 \qquad \frac{\Gamma \vdash e \Leftarrow \tau_2 \dashv \Delta}{\Gamma \vdash \mathbf{r} \cdot e \Leftarrow \tau_1 \oplus \tau_2 \dashv \Delta}\ \mathsf{tp/inj}_2$$

The case rule for sums requires more care. In the two branches (a) the bound variable must be used, and (b) the other used variables must coincide exactly. Because they are outputs in the algorithmic interpretation of the judgment, this requires some explicit equality test. They are then combined with the disjoint set of variables used in the first premise.

$$\frac{\Gamma \vdash e \Rightarrow \tau_1 \oplus \tau_2 \dashv \Delta \quad \Gamma, x \Rightarrow \tau_1 \vdash e_1 \Leftarrow \sigma \dashv \Delta', x \Rightarrow \tau_1 \quad \Gamma, y \Rightarrow \tau_2 \vdash e_2 \Leftarrow \sigma \dashv \Delta', y \Rightarrow \tau_2}{\Gamma \vdash \mathsf{case}\ e\ (\mathbf{l} \cdot x \Rightarrow e_1 \mid \mathbf{r} \cdot y \Rightarrow e_2) \Leftarrow \sigma \dashv \Delta, \Delta'}\ \mathsf{tp/case/plus}$$

If we were interested in only enforcing *affine types*, that is, each variable has to be use at most once, the modification is relatively straightforward. For example, instead of forcing x to occur in the output of the premise in tp/lam, we would just remove it in case it is there. Also, the two branches of a case expression we would just take the non-disjoint(!) union of the output contexts in the two branches.

5 Some Sample Functions

Even though it uses slightly different syntax and has equirecursive types (instead of isorecursive ones like LAMBDA), we can check that `append` is a linear function in both arguments. Functions (including recursive ones) that are defined at the top level are considered unrestricted and can be used freely. You can find this code in lec19.adj1.

```
type nat = +{'zero : 1, 'succ : nat}

type list = +{'nil : 1, 'cons : nat * list}

(* linear, check with snax -m L *)
decl append (l1 : list) (l2 : list) : list
defn append l1 l2 = match l1 with
     | 'nil () => l2
     | 'cons (x, xs) => 'cons (x, append xs l2)
```

You can explicitly check the linearity of the function by computing contexts, or you can track each variable separately:

1. l_1 is linear because we match against it.
2. l_2 is linear because it is used in both branches.
3. x is linear because it used once on its scope.
4. xs is linear for the same reason.

A function that is not linear is `length` because we drop the elements on the list. Perhaps worse, taking the length of a list consumes it, so we could not use it again after determining its length.

```
decl length (l : list) : nat
defn length l = match l with
     | 'nil () => 'zero ()
     | 'cons (x, xs) => 'succ (length xs)
```

As defined, we can see it should be *affine* because the variable x is not used in its scope. Everything else is linear and therefore also automatically affine.

Finally, the functional programming staple `map`.

```
decl map (f : nat -> nat) (l : list) : list
defn map f l = match l with
     | 'nil () => 'nil ()
     | 'cons (x, xs) => 'cons (f x, map f xs)
```

While this looks to be linear in l, x, and xs, it is not linear in f: f is dropped in the `'nil` branch and used twice in the `'cons` branch.

In SNAX we have the ability to combine functions and data with different modes. See Jang et al. [2024] and the Snax implementation for the details. We just show how the `map` function would look, using the modes `U` for "unrestricted" (can be used arbitrarily often) and `L` for "linear" (must be used exactly once). The upshift type constructor is lazy requires `.force` in order to obtain the underlying function from linear natural numbers to linear natural numbers. You can find this code and a couple of additional functions in lec19.adj.

```
type nat[k] = +{'zero : 1, 'succ : <nat[k]>}

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

mode U structural :> L
mode L linear

decl map (f : [U] up[L] (nat[L] -> nat[L])) (l : list[L L]) : list[L L]
defn map f l = match l with
     | 'nil () => 'nil ()
     | 'cons (<x>, <xs>) => 'cons (<f.force x>, <map f xs>)
```

References

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.

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

Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proceedings of the International Joint Conference on Theory and Practice of Software Development*, volume 2, pages 52–66, Pisa, Italy, March 1987. Springer-Verlag LNCS 250.

Martin Hofmann. A type system for bounded space and functional in-place update. In G. Smolka, editor, *Proceedings of the European Symposium on Programming (ESOP 2000)*, pages 165–179, Berlin, Germany, March 2000. Springer LNCS 1782.

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.

Steve Klabnik, Carol Nichols, and Chris Krycho. *The Rust Programming Language*. No Starch Press, 3rd edition edition, February 2026.

Anton Lorenzen, Daan Leijen, and Wouter Swierstra. FP^2: Fully in-place functional programming. In *Proceedings of the ACM on Programming Languages*, volume 7 (ICFP), pages 275–304, 2023.

Snax. The Snax adjoint functional language. Available at https://bitbucket.org/fpfenning/snax/, June 2024.