

This lecture explores one of the many ways that logic has influenced programming languages through the propositions-as-types principle. We'll show a logical system called *linear logic* can be transposed into a *linear type system* with interesting implications on the programming model. Linear types (and the general area of *substructural type systems*, of which linear types are one example) have recently influenced a cadre of safe programming languages with manual memory management: most prominently, Rust, which was in turn influenced by Cornell's own Cyclone [3] language.

1 Linear Logic

In ordinary logic, we think of $\varphi \vdash \psi$ as meaning *given that you know φ is true, you can also conclude that ψ is true*. Once you have proven that ψ is true, that has no bearing on the truth of φ : that premise is still just as true as it ever was. For example, if you have the three premises $A \rightarrow B$, $A \rightarrow C$, and A , you can prove both B and C by “reusing” the A premise. In other words, it's possible to derive:

$$A \rightarrow B, A \rightarrow C, A \vdash B \wedge C$$

Linear logic [5] modifies this view to restrict the way that premises are “used” to create conclusions.¹ The right intuition is closer to reasoning about physical materials—for example, chemical reactions. In linear logic, you have to “use” every premise *exactly once* to construct the conclusion. Following the chemical intuition, where the premises are the reagents (raw materials) and the consequent is the product, no raw materials are allowed to disappear, and no input molecule is allowed to show up twice in the product—both would violate the conservation of matter.

In notation, linear logic uses \multimap to denote its version of “matter-conserving” implication. So whereas you might read a standard implication $A \rightarrow B$ as *if A is true, then B is true*, you can instead read $A \multimap B$ as *given exactly one A , you can consume it to produce exactly one B* . Similarly, we introduce a new \wedge -like operator, \otimes , so $A \otimes B$ means you have *both* of the “chemicals” A and B . In linear logic, therefore, this version of the above is *not* derivable:

$$A \multimap B, A \multimap C, A \not\vdash B \otimes C$$

You would have to use two copies of A to conclude both B and C :

$$A \multimap B, A \multimap C, A, A \vdash B \otimes C$$

To complete the language, we'll add one more operator, \oplus , that is like a linear version of \vee . Here's a grammar for the linear-logic propositions we've been discussing:

$$\varphi ::= A \mid \varphi \multimap \psi \mid \varphi \otimes \psi \mid \varphi \oplus \psi$$

where A is a metavariable that ranges over constants. (This is a simple version of linear logic—other, more complete versions often have operators, written $!$ and $\&$, to let the linear style “coexist” with plain intuitionistic logic.)

¹These notes use the syntactic convention from Wadler [5]. The original formulation of linear logic, by Girard [2], has more forms that we won't discuss here.

Here are the core proof rules for this logic:

$$\begin{array}{c}
\frac{}{\varphi \vdash \varphi} \text{AXIOM} \qquad \frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \multimap \psi} \multimap\text{-INTRO} \qquad \frac{\Gamma \vdash \varphi \multimap \psi \quad \Delta \vdash \varphi}{\Gamma, \Delta \vdash \psi} \multimap\text{-ELIM} \\
\\
\frac{\Gamma \vdash \varphi \quad \Delta \vdash \psi}{\Gamma, \Delta \vdash \varphi \otimes \psi} \otimes\text{-INTRO} \qquad \frac{\Gamma \vdash \varphi \otimes \psi \quad \Delta, \varphi, \psi \vdash \chi}{\Gamma, \Delta \vdash \chi} \otimes\text{-ELIM} \\
\\
\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \oplus \psi} \oplus\text{-INTRO-L} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \varphi \oplus \psi} \oplus\text{-INTRO-R} \qquad \frac{\Gamma \vdash \varphi \oplus \psi \quad \Delta, \varphi \vdash \chi \quad \Delta, \psi \vdash \chi}{\Gamma, \Delta \vdash \chi} \oplus\text{-ELIM}
\end{array}$$

Let's carefully compare the rules for plain intuitionistic logic with their corresponding rules in linear logic. Take the rule for introducing \otimes expressions, for example, and compare it to the intuitionistic rule for introducing \wedge :

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} \wedge\text{-INTRO}$$

The old \wedge rule says that the implication and the assumption are proved in the same environment, Γ , as the consequent. In the linear version for \otimes , the two contexts from the premises are *combined* in the consequent.

This reflects the idea that all our “reagents” must be used to create our “products”: they are not allowed to disappear, and they are not allowed to be used twice. Using our chemistry analogy, let $\Gamma = H, H, O$, meaning that it contains two hydrogen atoms and one oxygen atom, and let φ denote a water molecule, so we have $\Gamma \vdash \varphi$. If $\Delta = C, O, O$ and ψ is carbon dioxide, then we also have $\Delta \vdash \psi$. To produce *both* water and carbon dioxide, however, we need *both* sets of reagents, so we write $\Gamma, \Delta \vdash \varphi \otimes \psi$. Atoms cannot just disappear, so it is not possible to derive that $\Gamma, \Delta \vdash \varphi$ (which would be required in an intuitionistic world). The rule also prevents you from using the raw materials in Γ to create two water molecules, $\varphi \oplus \varphi$: you would need double the reagents to produce that, i.e., $\Gamma, \Gamma \vdash \varphi \oplus \varphi$.

The \otimes elimination rule also differs from \wedge , where there are two rules to obtain either side of the conjunction:

$$\frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi} \wedge\text{-ELIM-L} \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \psi} \wedge\text{-ELIM-R}$$

In a linear world, these rules would let you “destroy” matter. Instead, \otimes elimination forces you to “consume” both sides of the conjunction to produce a single result χ by proving $\Gamma, \varphi, \psi \vdash \chi$.

The rules for \multimap and \oplus are more similar to their intuitionistic counterparts, except for how they duplicate and combine contexts.

To complete the system, we need one more rule to perform some bookkeeping on those contexts. Consider these three rules from intuitionistic logic:

$$\frac{\Gamma \vdash \varphi}{\Gamma, \psi \vdash \varphi} \text{WEAKENING} \qquad \frac{\Gamma, \Delta \vdash \varphi}{\Delta, \Gamma \vdash \varphi} \text{EXCHANGE} \qquad \frac{\Gamma, \psi, \psi \vdash \varphi}{\Gamma, \psi \vdash \varphi} \text{CONTRACTION}$$

These bookkeeping rules, called the *structural* rules, are so boring that we left them off of our presentation of intuitionistic logic from the previous lecture. They say that you are allowed to ignore unnecessary assumptions in your proofs (weakening), that the order of assumptions doesn't matter (exchange), and that assuming the same thing twice doesn't buy you anything (contraction). These rules are important for making the formal system work out. Notice, for example, that all the other rules presume that the assumption we need is always at the “rightmost” position in the context. We need the exchange rule to be able to reorder premises to put the salient one into the right place.

The key difference in linear logic is that it *eliminates the weakening and contraction rules* but keeps the exchange rule. Intuitively, this means that you're no longer allowed to ignore assumptions, and you're not allowed to use them twice. Together, the result is that every assumption is used exactly once in the proof.

2 Linear Types

A linear type system is to a linear logic as a “normal” type system, like the simply-typed λ -calculus, is to an intuitionistic logic. The overall effect is that every *variable* must be used exactly once. In fact, linear type systems are part of a larger family of *substructural* type systems that remove the typey equivalents of the structural logic rules above. Using different combinations of the structural rules leads to different constraints on the programming language:

- *Linear* type systems have exchange only, so every variable must be used exactly once.
- *Affine* type systems have exchange and weakening, so every variable can be used *at most* once.
- *Relevant* type systems have exchange and contraction, so every variable must be used *at least* once.
- *Ordered* type systems don’t use any of the three structural rules, so you have to use every variable exactly once in the order of creation.

But we’ll focus on linear types here. The core idea is that, in order for the typing judgment $\Gamma \vdash e : \tau$ to hold, the set of variables in Γ must be exactly the same as the set of variables in e .

Let’s define a linearly typed version of λ^\rightarrow . This language will be very restrictive—the usual thing to do is to add nonlinear types back into the language too so you can choose which one you need [4]. The grammar for expressions is the same:

$$e ::= x \mid \lambda x. e : \tau \mid e_1 e_2$$

And the grammar for types includes base types and linear functions:

$$\tau ::= b \mid \tau_1 \multimap \tau_2$$

The typing rules follow the proof rules for linear logic:

$$\frac{}{x : \tau \vdash x : \tau} \quad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1. e : \tau_1 \multimap \tau_2} \quad \frac{\Gamma \vdash e_1 : \tau_1 \multimap \tau_2 \quad \Delta \vdash e_2 : \tau_1}{\Gamma, \Delta \vdash e_1 e_2 : \tau_2}$$

For example, instead of the old λ^\rightarrow variable rule that says $\Gamma \vdash x : \tau$ as long as $\Gamma(x) = \tau$, the new linear variable rule requires x to be the *only* variable in the context.

This language, if were to extend it with integers, would still allow $\lambda x : \text{int}. x$, which has the type $\text{int} \multimap \text{int}$. However, this larger program:

$$\lambda f : \text{int} \multimap \text{int}. \lambda x : \text{int}. f (f x)$$

is not well-typed, even though its plain λ^\rightarrow equivalent is perfectly fine, because it reuses f twice. Even the seemingly mundane $\lambda x : \text{int}. x + x$ does not have a type.

3 Safe Manual Memory Management

This restriction may seem a little silly when dealing with integers: who cares how many times you use an integer? But imagine a language where there are types that represent consumable computational resources: allocated memory, open file handles, socket connections, or files in the filesystem.

If you’ve ever written a console program that uses Unix’s standard input and standard output streams, for example, you know that you have to be very careful about how you consume `stdin`. If your program tries to read the entirety of `stdin` (`sys.stdin.read()` in Python) more than once, for example, it will hang. If you forget to read from it at all sometimes, then other programs that pipe large amounts of data into your process will

hang when they fill up the OS buffer. It might be nice to have a compile-time guarantee that your program, no matter how it executes, will read `stdin` exactly once. That’s the idea with practical linear type systems: they don’t use them for *every* value in the program; just the ones where resource consumption really matters.

One particular resource has recently been shown to be a good match for linear (and affine) type systems: memory. Specifically, programming languages can use linear types to get the “best of both worlds” between memory-safe, garbage-collected languages like Java and “memory-unsafe” languages like C and C++ that use manual memory management. People like the fact that Java rules out use-after-free and double-free bugs, for example, but garbage collection comes at a cost. Video games, for example, often can’t afford to have the garbage collector take control at seemingly random times and delay a frame from being rendered, leading to unpleasant jitter. So is it possible to have manual, deterministic memory management with `malloc` and `free` but still be sure that you never dereference a pointer after freeing it?

That’s the goal of a modern genre of memory-safe systems languages. The most prominent research language that uses this strategy is Cyclone [3], which was developed at Cornell in the early 2000s. More recently, Rust has adopted many of the ideas from Cyclone in the context of a full-fledged, industrial language. Rust takes it a step further and hides the `malloc` and `free` calls from you altogether.

In these languages, the idea is not to make pointers themselves linearly typed. You are allowed to dereference (“use”) a pointer multiple times, fortunately, and you can even ignore a pointer altogether without dereferencing it even once. Instead, the languages associate every pointer with a hidden *capability* value that represents the ability to call `free`. In Cyclone, this capability is associated with a region of memory holding multiple variables so they can all be deallocated at the same time [1]; in Rust, the capability is coupled with the *lifetime* concept.

To capture the intuition, let’s add `malloc` and `free` to the λ -calculus. First, let’s write a C-like version without memory safety. We’ll make `malloc` behave sort of like the `ref` expression from our references extension, so we can write programs like this:

```
let p = malloc 4 in
  p := 2;
  free p
```

The types for this extension would look like this:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{malloc } e : \tau \text{ ptr}} \qquad \frac{\Gamma \vdash e : \tau \text{ ptr}}{\Gamma \vdash \text{free } e : \text{unit}}$$

The problem, of course, is that we can now write `malloc 4` and *never* free the pointer, which leaks memory if we don’t have a garbage collector; and we can write:

```
let p = malloc 4 in
  free p;
  free p
```

which will crash when it tries to free the memory twice. To rule out both of these problems, we want to require the program to call `malloc` exactly once.

To use linear types, we’ll modify `malloc` to produce a pair of the pointer and a *capability* that controls access to it. We’ll add a new base type to our language, `cap`, and make the “return type” of `malloc` into a product like $\tau \text{ ptr} \times \text{cap}$. Then, we’ll define the rule for `free` to it takes a capability in addition to the pointer. This way, the only way use can “use” a capability value is to `free` it. By applying the linear typing rules to capability values, we can guarantee that the program will free the pointer exactly once! Programs can even pass the capability into other functions that *don’t* call `free`—as long as they pass it back to the caller, who will then receive the obligation for managing the memory.

There are a few more details to work out, including how to create an association between each pointer value and its corresponding capability value so that you can only dereference a pointer when you have a valid

capability for it. For more on linear typing in Cyclone, and a stepping stone to Rust’s lifetime tracking, I recommend the paper “Linear Regions are All You Need” by Fluet et al. [1].

References

- [1] Matthew Fluet, Greg Morrisett, and Amal J. Ahmed. Linear regions are all you need. In *European Symposium on Programming (ESOP)*, 2006.
- [2] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, January 1987.
- [3] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference (ATC)*, 2002.
- [4] Philip Wadler. Linear types can change the world. In *IFIP TC 2 Working Conference on Programming Concepts and Methods*, 1990.
- [5] Philip Wadler. A taste of linear logic. In *Mathematical Foundations of Computer Science (MFCS)*, 1993.