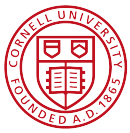
CS 4110 – Programming Languages and Logics
Lecture #29: Linear Types



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 language.

1 Linear Logic

In ordinary logic, we think of $\phi \vdash \psi$ 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 $A \to B$, $A \to C$, and A, you can prove both B and C by "reusing" the A premise. In other words, it's possible to derive:

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

Linear logic modifies this view to restrict the way that premises are "used" to create conclusions. 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 $A \to B$ 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:

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

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:

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

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.)

Here are the core proof rules for this logic:

$$\frac{}{\phi \vdash \phi}\ \textsc{axiom} \qquad \frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \multimap \psi}\ \multimap\textsc{-intro} \qquad \frac{\Gamma \vdash \phi \multimap \psi \qquad \Delta \vdash \phi}{\Gamma, \Delta \vdash \psi}\ \multimap\textsc{-elim}$$

$$\frac{\Gamma \vdash \phi \qquad \Delta \vdash \psi}{\Gamma, \Delta \vdash \phi \otimes \psi}\ \otimes\textsc{-intro} \qquad \frac{\Gamma \vdash \phi \otimes \psi \qquad \Delta, \phi, \psi \vdash \chi}{\Gamma, \Delta \vdash \chi}\ \otimes\textsc{-elim}$$

$$\frac{\Gamma \vdash \phi}{\Gamma \vdash \phi \oplus \psi}\ \oplus\textsc{-intro-l} \qquad \frac{\Gamma \vdash \psi}{\Gamma \vdash \phi \oplus \psi}\ \oplus\textsc{-intro-r} \qquad \frac{\Gamma \vdash \phi \oplus \psi \qquad \Delta, \phi \vdash \chi \qquad \Delta, \psi \vdash \chi}{\Gamma, \Delta \vdash \chi}\ \oplus\textsc{-elim}$$

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 $\wedge$:

$$\frac{\Gamma \vdash \phi \qquad \Gamma \vdash \psi}{\Gamma \vdash \phi \wedge \psi}\ \wedge\textsc{-intro}$$

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 $\Gamma \vdash \phi$. 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 $\Gamma, \Delta \vdash \phi \otimes \psi$. Atoms cannot just disappear, so it is not possible to derive that $\Gamma, \Delta \vdash \phi$ (which would be allowed in an intuitionistic world). The rule also prevents you from using the raw materials in Γ to create two water molecules, $\phi \otimes \phi$: you would need double the reagents to produce that, i.e., $\Gamma, \Gamma \vdash \phi \otimes \phi$.

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

$$\frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi}\ \wedge\textsc{-elim-l} \qquad \frac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi}\ \wedge\textsc{-elim-r}$$

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 $\Gamma, \phi, \psi \vdash \chi$.

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:

$$\frac{\Gamma \vdash \phi}{\Gamma, \psi \vdash \phi}\ \textsc{weakening} \qquad \frac{\Gamma, \Delta \vdash \phi}{\Delta, \Gamma \vdash \phi}\ \textsc{exchange} \qquad \frac{\Gamma, \psi, \psi \vdash \phi}{\Gamma, \psi \vdash \phi}\ \textsc{contraction}$$

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 $\lambda^{\to}$. 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. The grammar for expressions is the same:

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

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:

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

For example, instead of the old $\lambda^{\to}$ 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 $\lambda x : \mathbf{int}.\, x$, which has the type $\mathbf{int} \multimap \mathbf{int}$. However, this larger program:

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

is not well-typed, even though its plain $\lambda^{\to}$ equivalent is perfectly fine, because it reuses f twice. Even the seemingly mundane $\lambda x : \mathbf{int}.\, x + x$ 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, 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.

To realize this GC-free, memory-safe vision, we will design a language with linearly-typed pointers. The main problem we'll encounter is that we'll want to "use" a pointer multiple times, in the sense that "using" consists of loading and storing through the pointer, but free it exactly once.

3.1 Unrestricted Pointers

Imagine adding 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:

$$\begin{array}{l}\mathsf{let}\ p\ =\ \mathsf{malloc}\ 4\ \mathsf{in}\\ \mathsf{store}\ p\ ((\mathsf{load}\ p) + 38);\\ \mathsf{free}\ p\end{array}$$

We'll use expressions named store and load instead of := and ! as in our old λ-calculus extension for references, but the meaning is the same. The types for this extension would look like this:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{malloc}\ e : \tau\ \mathbf{ptr}} \qquad \frac{\Gamma \vdash e : \tau\ \mathbf{ptr}}{\Gamma \vdash \mathsf{free}\ e : \mathbf{unit}} \qquad \frac{\Gamma \vdash e : \tau\ \mathbf{ptr}}{\Gamma \vdash \mathsf{load}\ e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau\ \mathbf{ptr} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathsf{store}\ e_1\ e_2 : \mathbf{unit}}$$

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. We can also write:

$$\begin{array}{l}\mathsf{let}\ p\ =\ \mathsf{malloc}\ 4\ \mathsf{in}\\ \mathsf{free}\ p;\\ \mathsf{free}\ p\end{array}$$

which will crash (i.e., get stuck) when it tries to free the memory twice. To rule out both of these problems, we want to require the program to call free exactly once for every malloc'd pointer.

3.2 Linear Pointers

Now imagine the same type extension in a linearly-typed λ-calculus. When a pointer p is in a context Γ, an expression e is well-typed in Γ if and only if it refers to p exactly once. That means that our double-free example above is now illegal, and so is any program that mallocs a pointer and then drops it without freeing it.

But we've created a new problem: the first example above, which innocuously loads and stores a pointer p before freeing it, is also illegal because it uses p three times. A naïve linear type system would prevent you from doing anything useful with a pointer—all this language can do is allocate and free memory.

To fix this, the idea is to have *safe* uses of linear values—ones that neither create nor destroy the value—produce a *new copy* of the pointer to indicate that it wasn't consumed. You can think of a linear pointer value as carrying an obligation to free the pointer's memory. When you call a function, for example, that function has two choices: it can either free the memory itself, or it return the pointer back to you to delegate the responsibility for freeing the memory.

In particular, we might model load and store as linear functions:

$$\mathsf{store} : \forall\alpha.\ \alpha\ \mathbf{ptr} \times \alpha \multimap \alpha\ \mathbf{ptr} \qquad\qquad \mathsf{load} : \forall\alpha.\ \alpha\ \mathbf{ptr} \multimap (\alpha\ \mathbf{ptr} \times \alpha)$$

Both functions now "return" a pointer, and load's result is now a pair consisting of the value from the heap *and* another copy of the input pointer. Our first example is now a bit more inconvenient to write, but it typechecks:

$$\begin{array}{l}\mathsf{let}\ p\ =\ \mathsf{malloc}\ 4\ \mathsf{in}\\ \mathsf{let}\ (p', n) = \mathsf{load}\ p\ \mathsf{in}\\ \mathsf{let}\ p'' = \mathsf{store}\ p\ (n + 38)\ \mathsf{in}\\ \mathsf{free}\ p''\end{array}$$

This works, but there are many more details to work out to turn this sketch into a usable programming language. We'd like to hide the boilerplate involved in "threading through" copies of p, for example, so we don't need to come up with an ever-expanding set of names like p' and p''. We also need a way to integrate linear types with nonlinear types—for example, there is probably no good reason to give **int** the same linear restrictions as **int ptr**, for example.