Substructural Type Systems

Lecture 17 Tuesday, April 2, 2019

1 Substructural Type Systems

The Curry-Howard isomorphism gives a connection between logic and types. This connection goes both ways: we can use insights from logic to think about programming languages, and develop new language features that correspond to logical entities; and we can use insights from programming languages influence our study and use of logic?

We consider how substructural logics give rise to substructural type systems.

1.1 Natural deduction and structural inference rules

Natural deduction is a kind of proof calculus that can be used to formalize mathematical logic. It's called "natural deduction" because it is meant to correspond to a "natural" way of reasoning about truth. In natural deduction, we write

$$A_1, \ldots, A_n \vdash B$$

to mean that whenever formulas A_1 through A_n are true, then formula B is true. For example, in a propositional calculus, we may write the judgment

$$p, \neg q \vdash q \Rightarrow (p \Rightarrow r),$$

which intuitively means that if proposition p is true, and formula $\neg q$ is true, then the formula $q \Rightarrow (p \Rightarrow r)$ is true.

Like we do in programming languages, we can write inference rules and axioms for a given natural deduction calculus, that define which judgments are true. For example, we may have an axiom

$$\frac{}{\phi, \psi \vdash \phi \wedge \psi}$$

That is, if formulas ϕ and ψ are true, then the conjunction $\phi \wedge \psi$ is true.

The inference rules that are concerned with the manipulation of the assumptions (i.e., the formulas on the left of the turnstile "$\vdash$") are known as the *structural inference rules*. The structural inference rules allow us to treat the assumptions like a set. That is, there are inference rules to re-order the assumptions, to collapse identical assumptions, and to remove unneeded assumptions. We use Γ and Δ to range over (possibly empty) sequences of formulas.

$$\textsc{Exchange}\ \frac{\Gamma, A, B, \Delta \vdash C}{\Gamma, B, A, \Delta \vdash C} \qquad \textsc{Contraction}\ \frac{\Gamma, A, A, \Delta \vdash B}{\Gamma, A, \Delta \vdash B} \qquad \textsc{Weakening}\ \frac{\Gamma, \Delta \vdash B}{\Gamma, A, \Delta \vdash B}$$

There are, of course, other inference rules, depending on the logic. Here are some of the inference rules for a propositional logic. Note that the assumptions in the conclusion of the inference rules are conserved in the premises, e.g., there is no duplication or dropping of assumptions.

$$\frac{}{A \vdash A} \qquad \frac{\Gamma, B \vdash A}{\Gamma \vdash B \Rightarrow A} \qquad \frac{\Gamma \vdash B \Rightarrow A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A} \qquad \frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma, \Delta \vdash A \wedge B} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

If we drop any of the structural inference rules from the definition of our logic, we have a *substructural logic*.

For example, if we allow Exchange but drop Weakening and Contraction we have a *linear logic*: every assumption must be used exactly once. If we allow both Exchange and Weakening, but drop Contraction, we have an *affine logic*: every assumption may be used at most once.

1.2 Substructural type systems

So, what new programming language features or designs do substructural logic give us? You may have noticed the similarity between the natural deduction judgments for logic and the type judgments we use in programming languages. A type judgment looks like

$$\Gamma \vdash e : \tau$$

where we can think of the type context Γ as being a sequence $x_1 : \tau_1, \ldots, x_n : \tau_n$. Inference rules for such a type system would need to have rules for manipulating the type context.

$$\textsc{Exchange}\ \frac{\Gamma, x:\tau_1, y:\tau_2, \Delta \vdash e:\tau}{\Gamma, y:\tau_2, x:\tau_1, \Delta \vdash e:\tau} \qquad \textsc{Contraction}\ \frac{\Gamma, x:\tau, x:\tau, \Delta \vdash e:\tau'}{\Gamma, x:\tau, \Delta \vdash e:\tau'}$$

$$\textsc{Weakening}\ \frac{\Gamma, \Delta \vdash e:\tau}{\Gamma, x:\tau', \Delta \vdash e:\tau}\ x \text{ not in } \Gamma, \Delta$$

If we drop any of these structural inference rules, we have a *substructural type system*.

- *Linear* type systems ensure that every variable is used exactly once. Linear type systems drop Contraction and Weakening (but keep Exchange).
- *Affine* type systems ensure that every variable is used at most once. Affine type systems drop Contraction (but keep Weakening and Exchange).
- *Relevant* type systems ensure that every variable is used at least once. Relevant type systems drop Weakening (but keep Contraction and Exchange).
- *Ordered* type systems ensure that every variable is exactly once, in the order in which they are introduced. Ordered type systems drop Weakening, Contraction, and Exchange.

Linear type systems So if we drop Contraction and Weakening, we have a *linear type system*, by analogy with a linear logic. So how does dropping Contraction and Weakening affect the set of programs that will be well typed? Well, variables placed into the typing context must be used exactly once along any control flow path. The rule Contraction would allow us to use a variable multiple times, and Weakening allows us to not use a variable at all. This makes linear type systems good for tracking the use of resources. For example, we can use a linear type system to track open file handles, and ensure that a client must always close a file (i.e., must use the file handle at least once), and cannot close a file multiple times (i.e., must use the file handle at most once). In a similar way, we can use a linear type system to track objects allocated on the heap, and (with some additional language support) ensure that we have always exactly one pointer to a heap object. This can be useful for reasoning about aliasing: if we maintain an invariant that each heap object has exactly one pointer, then if a function is given two pointers, it knows they must not alias.

To use linear type systems in practice, it is often necessary to allow both linear types, and non-linear types (i.e., variables that can contraction and weakening applied to them), and/or to allow linearity to be weakened locally.

Ordered type systems Think about a type system where we drop all three structural rules (Exchange, Contraction, and Weakening). This is known as an *ordered type system*. Every variable is used exactly once, in the order it was introduced. In the same way that we can use a linear type system to help us reason about heap-allocated memory, we can use an ordered type system to help us reason about stack-allocated memory: not only must we use (i.e., deallocate) every piece of memory exactly once, but we must do so in stack order: i.e., the most recently allocated memory on the stack must be deallocated first.

1.3 Linear lambda calculus

Let's consider a calculus that uses a linear type system to track use of objects.[1] The motivation for this calculus is that by ensuring that objects are used exactly once, after an object has been used, we can safely deallocate the object (i.e., reclaim the physical resources, such as memory, that are associated with the object).

The syntax for our calculus is as follows. We will consider booleans, pairs, and functions.

$$
\begin{aligned}
&q ::= \mathsf{lin} \mid \mathsf{un} \\
&e ::= x \mid q\ b \mid q\ (e_1, e_2) \mid q\ \lambda x{:}\tau.\, e \mid e_1\ e_2 \mid \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid \mathsf{split}\, e_1\, \mathsf{as}\, x, y\, \mathsf{in}\, e_2 \\
&b \in \{\mathsf{true}, \mathsf{false}\}
\end{aligned}
$$

Note that all values have a qualifier q, indicating whether the value is to be treated linearly (meaning that the value can be used exactly once, and the resources used to represent the resource can be reclaimed after the single use), or whether use is unrestricted. The other unusual construct is $\mathsf{split}\, e_1\, \mathsf{as}\, x, y\, \mathsf{in}\, e_2$, which evaluates e_1 to a pair value, and then binds variable x to the first element of the pair, and binds variable y to the second element of the pair, and proceeds with evaluation of e_2 (which can use variables x and y). This allows us to extract the elements of a pair value with just a single use of the pair. If we had the projection operations—$\#1\ e$ and $\#2\ e$—then we would need to use a pair value twice in order to extract both of its constituent values.

We will present the operational semantics for this language after presenting the type system.

1.3.1 Type system

The syntax for the types are presented below. A *pretype* π is either a product type, a function type, or the type for Booleans. A type $\tau = q\ \pi$ is a pretype and a qualifier (either lin or un). A value of type $\mathsf{lin}\ \pi$ should be treated linearly (i.e., used exactly once) whereas a value of type $\mathsf{un}\ \pi$ can be used as many times as desired. We treat type contexts as a (possibly empty) sequence of pairs $x{:}\tau$.

$$
\begin{aligned}
&\pi ::= \mathbf{bool} \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \\
&\tau ::= q\ \pi \\
&\Gamma ::= \emptyset \mid \Gamma, x{:}\tau
\end{aligned}
$$

The inference rules for the typing judgment maintains two invariants: (1) linear variables are used exactly once on each control flow path; and (2) unrestricted data structures may not contain linear data structures.

Linear variables will be substituted with linear values at run time, and thus the first invariant ensures that those linear values are used exactly once. The second invariant ensures, for example, that we cannot have a value such as $\mathsf{un}\ (\mathsf{lin}\ \mathsf{true}, \mathsf{lin}\ \mathsf{true})$. The pair value is marked as unrestricted, meaning we can use it as many times as we would like, whereas the contents of the pair data structure are marked as linear. If we could in fact use the pair value many times, then we could repeatedly extract the contents of the pair and use them. Thus, we prevent unrestricted data structures (such as pairs or functions) from containing linear values.

Before we present the typing rules, we first introduce a relation that allows us to split a context Γ into two pieces Γ_1 and Γ_2 (written $\Gamma = \Gamma_1 \circ \Gamma_2$), such that each variable with linear type in Γ appears in exactly one of Γ_1 and Γ_2, and unrestricted variables in Γ appear in both Γ_1 and Γ_2. This relation will help us ensure that each linear variable in Γ is used exactly once.

[1] This calculus is from the chapter "Substructural Type Systems," by David Walker, in *Advanced Topics in Types and Programming Languages*, editor Benjamin C. Pierce, MIT Press, 2002.

$$\frac{}{\emptyset = \emptyset \circ \emptyset} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x{:}\mathsf{un}\ \pi = (\Gamma_1, x{:}\mathsf{un}\ \pi) \circ (\Gamma_2, x{:}\mathsf{un}\ \pi)}$$

$$\frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x{:}\mathsf{lin}\ \pi = (\Gamma_1, x{:}\mathsf{lin}\ \pi) \circ \Gamma_2} \qquad \frac{\Gamma = \Gamma_1 \circ \Gamma_2}{\Gamma, x{:}\mathsf{lin}\ \pi = \Gamma_1 \circ (\Gamma_2, x{:}\mathsf{lin}\ \pi)}$$

We also define predicates on types and contexts, to help us identify whether a type τ is a linear type ($\mathsf{lin}(\tau)$) or an unrestricted type ($\mathsf{un}(\tau)$), and whether a type context Γ consists entirely of linear types ($\mathsf{lin}(\Gamma)$) or entirely of unrestricted types ($\mathsf{un}(\Gamma)$). More formally:

- $\mathsf{un}(\tau)$ if and only if $\tau = \mathsf{un}\ \pi$.
- $\mathsf{lin}(\tau)$ if and only if $\tau = \mathsf{un}\ \pi$ or $\tau = \mathsf{lin}\ \pi$.
- $q(\Gamma)$ if and only if for all $(x{:}\tau) \in \Gamma$, we have $q(\tau)$.

The typing judgment has the form $\Gamma \vdash e{:}\tau$, and the inference rules for the judgment are as follows.

$$\text{T-VAR}\ \frac{\mathsf{un}(\Gamma_1, \Gamma_2)}{\Gamma_1, x{:}\tau, \Gamma_2 \vdash x{:}\tau} \qquad \text{T-BOOL}\ \frac{\mathsf{un}(\Gamma)}{\Gamma \vdash q\ b{:}q\ \mathbf{bool}}$$

$$\text{T-IF}\ \frac{\Gamma_1 \vdash e_1{:}q\ \mathbf{bool} \qquad \Gamma_2 \vdash e_2{:}\tau \qquad \Gamma_2 \vdash e_3{:}\tau}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3{:}\tau}\ \Gamma = \Gamma_1 \circ \Gamma_2$$

$$\text{T-PAIR}\ \frac{\Gamma_1 \vdash e_1{:}\tau_1 \quad \Gamma_2 \vdash e_2{:}\tau_2 \quad q(\tau_1) \quad q(\tau_2)}{\Gamma \vdash q\ (e_1, e_2){:}q\ (\tau_1, \tau_2)}\ \Gamma = \Gamma_1 \circ \Gamma_2$$

$$\text{T-SPLIT}\ \frac{\Gamma_1 \vdash e_1{:}q\ (\tau_1 \times \tau_2) \quad \Gamma_2, x{:}\tau_1, y{:}\tau_2 \vdash e_2{:}\tau}{\Gamma \vdash \mathsf{split}\ e_1\ \mathsf{as}\ x, y\ \mathsf{in}\ e_2{:}\tau}\ \Gamma = \Gamma_1 \circ \Gamma_2$$

$$\text{T-ABS}\ \frac{q(\Gamma) \qquad \Gamma, x{:}\tau \vdash e{:}\tau'}{\Gamma \vdash q\ \lambda x{:}\tau.\, e{:}q\ \tau \to \tau'} \qquad \text{T-APP}\ \frac{\Gamma_1 \vdash e_1{:}q\ \tau \to \tau' \qquad \Gamma_2 \vdash e_2{:}\tau}{\Gamma \vdash e_1\ e_2{:}\tau'}\ \Gamma = \Gamma_1 \circ \Gamma_2$$

For inference rules for expressions that have subexpressions (e.g., application $e_1\ e_2$ which has sub-expressions e_1 and e_2) we need to split the type context in order to type check the sub-expressions. This ensures that a linear variable is used by exactly one sub-expression. However, in the rule T-IF, we split the context into Γ_1 (which is used to type check the conditional expression e_1), but use Γ_2 to type check both branches e_2 and e_3. This is because execution will either e_2 or e_3. That is, no matter which branch is taken, the linear variables in Γ_2 will be used.

Note that in rules T-VAR and T-BOOL, we check in the premises that the remaining variables in the context are all unrestricted. That is because those remaining variables are not used, which would be a problem if any of the variables had linear type.

In rule T-PAIR, the premises $q(\tau_1)$ and $q(\tau_2)$ ensure that an unrestricted pair can not contain any linear values, enforcing invariant (2) above. Similarly, in rule T-ABS, premise $\mathsf{un}(\Gamma)$ ensures that if the function value is unrestricted ($q = \mathsf{un}$), then there are no linear variables closed over in the body of the function.

Consider the following example code, that attempts to discard a linear value (i.e., the linear variable x will not used when the expression is applied).

$$\mathsf{lin}\ \lambda x{:}\mathsf{lin}\ \mathbf{bool}.\ (\mathsf{lin}\ \lambda f{:}\mathsf{un}\ (\mathsf{un}\ \mathbf{bool} \to \mathsf{lin}\ \mathbf{bool}).\ \mathsf{lin}\ \mathsf{true})\ (\mathsf{un}\ \lambda y{:}\mathsf{un}\ \mathbf{bool}.\ x)$$

This program is not well-typed. In particular, the premise $q(\Gamma)$ of T-ABS fails to hold.

The following example is also not well-typed: it tries to duplicate linear values, i.e., the linear variable x will be used twice when the expression is applied.

$$\mathsf{lin}\ \lambda x{:}\mathsf{lin}\ \mathbf{bool}.\ (\mathsf{lin}\ \lambda f{:}\mathsf{un}\ (\mathsf{un}\ \mathbf{bool} \to \mathsf{lin}\ \mathbf{bool}).\ \mathsf{lin}\ (f\ (\mathsf{un}\ \mathsf{true}), f\ (\mathsf{un}\ \mathsf{true})))\ (\mathsf{un}\ \lambda y{:}\mathsf{un}\ \mathbf{bool}.\ x)$$

Finally, note that rule T-VAR allows the variable being used, x, to appear anywhere in the type context. Because of this, we do not explicitly need an inference rule corresponding to EXCHANGE. We could, however replace rule T-VAR with the following two rules, T-EXCHANGE and T-VAR2, and have an equivalent type system.

$$\text{T-EXCHANGE}\ \frac{\Gamma_1, y{:}\tau', x{:}\tau, \Gamma_2 \vdash e{:}\tau}{\Gamma_1, x{:}\tau, y{:}\tau', \Gamma_2 \vdash e{:}\tau} \qquad \text{T-VAR2}\ \frac{\mathsf{un}(\Gamma_1)}{\Gamma, x{:}\tau \vdash x{:}\tau}$$

1.3.2 Operational semantics

We use a store-based semantics. That is, to emphasize the usefulness of the linear type system, we store values in the heap. (This makes sense for data structures, but for simplicity we inefficiently store all values, including boolean primitives, in the heap.)

A prevalue p is either a boolean constant, a function, or a pair of values. A value $v = q\ p$ is a qualified prevalue. We use a call-by-value evaluation order.

$$\begin{aligned} p &::= b \mid \lambda x{:}\tau.\, e \mid (v_1, v_2) \\ v &::= q\ p \\ E &::= [\cdot] \mid \mathsf{if}\ E\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 \mid q\ (E, e) \mid q\ (\ell, E) \mid \mathsf{split}\ E\ \mathsf{as}\ x, y\ \mathsf{in}\ e \mid E\ e \mid \ell\ E \end{aligned}$$

$$\text{CONTEXT}\ \frac{\langle e, \sigma\rangle \longrightarrow \langle e', \sigma'\rangle}{\langle E[e], \sigma\rangle \longrightarrow \langle E[e'], \sigma'\rangle} \qquad \text{VAL}\ \frac{}{\langle v, \sigma\rangle \longrightarrow \langle \ell, \sigma[\ell \mapsto v]\rangle}\ \ell \notin \mathrm{dom}(\sigma)$$

$$\text{IF-TRUE}\ \frac{\sigma(\ell) = q\ \mathsf{true} \qquad \sigma' = \begin{cases} \sigma & \text{if } q = \mathsf{un} \\ \sigma \setminus \ell & \text{if } q = \mathsf{lin} \end{cases}}{\langle \mathsf{if}\ \ell\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, \sigma\rangle \longrightarrow \langle e_1, \sigma'\rangle} \qquad \text{IF-FALSE}\ \frac{\sigma(\ell) = q\ \mathsf{false} \qquad \sigma' = \begin{cases} \sigma & \text{if } q = \mathsf{un} \\ \sigma \setminus \ell & \text{if } q = \mathsf{lin} \end{cases}}{\langle \mathsf{if}\ \ell\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2, \sigma\rangle \longrightarrow \langle e_2, \sigma'\rangle}$$

$$\text{SPLIT}\ \frac{\sigma(\ell) = q\ (\ell_1, \ell_2) \qquad \sigma' = \begin{cases} \sigma & \text{if } q = \mathsf{un} \\ \sigma \setminus \ell & \text{if } q = \mathsf{lin} \end{cases}}{\langle \mathsf{split}\ \ell\ \mathsf{as}\ x, y\ \mathsf{in}\ e, \sigma\rangle \longrightarrow \langle e\{\ell_1/x\}\{\ell_2/y\}, \sigma'\rangle} \qquad \text{APP}\ \frac{\sigma(\ell_1) = q\ \lambda x{:}\tau.\, e \qquad \sigma' = \begin{cases} \sigma & \text{if } q = \mathsf{un} \\ \sigma \setminus \ell & \text{if } q = \mathsf{lin} \end{cases}}{\langle \ell_1\ \ell_2, \sigma\rangle \longrightarrow \langle e\{\ell_2/x\}, \sigma'\rangle}$$

Rule VAL ensures that once we have evaluated an expression to a value, we allocate a new location ℓ, and put the value in the location. All the rules that use a linear value immediately free the location that they used (expressed as $\sigma' = \sigma \setminus \ell$, meaning that store σ' is the same as σ except that location ℓ has been removed). Otherwise, the semantics are standard.

Note that the use of the linear type system ensures that a location is never accessed after it is freed.

1.4 The Rust programming language

Rust is a programming language (developed by Mozilla Research) that aims to be a memory-safe, concurrent, and practical systems programming language. In order to ensure memory safety, Rust uses a sophisticated type system to reason about the lifetime of data. This ensures that memory can be efficiently freed as soon as it is no longer needed (without the need for a garbage collector), also that pointers to data are handled correctly, including not dereferencing freed memory (no use-after-free) and pointers are not used to free memory multiple times (no double-free).

Rust does not use a linear type system. However, some of the concepts involved in reasoning about the number of references to data objects are similar to the sub-structural type systems we have considered. We summarize these concepts here.

In Rust, we can create memory objects (which are allocated on the stack when possible, or in the heap otherwise), and then create references (which are similar to pointers) to objects. An *immutable reference* can not be used to change the state of an object. That is, with only an immutable reference to an object, we can only read (but not update) the object. A *mutable reference* allows us to both read and update an object. References are immutable by default.

For example, in the following code, think of `v`, `w`, and `z` as (stack allocated) 32-bit integers, `a` is an immutable reference to `v`, and `b` is a mutable reference to `w`.

```
let v : i32 = 13;
let mut w : i32 = 14;
let z : i32 = 15;

let a : &i32 = &v;
let b : &mut i32 = &mut w;

println!("The answer is {}", *a + *b + z);
*b = 7; // Modify the contents of w
*a = 5; // ERROR: a is not a mutable reference.
        // (Indeed, v is not a mutable i32!)
```

Rust allows many of type annotations to be elided (it will infer them), and also can automatically coerce references. We could write the above code equivalently as follows.

```
let v = 13;
let mut w = 14;
let z = 15;

let a = &v;
let b = &mut w;

println!("The answer is {}", *a + *b + z);
*b = 7;
*a = 5; // ERROR: ...
```

Here's a more interesting example, where we use vector objects instead of integers.

```
let v : Vec<i32> = vec![11,12,13];
let mut w : Vec<i32> = vec![13,14,15];
let z : Vec<i32> = vec![15,16,17];

let a : &Vec<i32> = &v;
let b : &mut Vec<i32> = &mut w;

println!("The answer is {}", a[2] + b[1] + z[0]);

b.push(4); // Modify the contents of vector w
a.push(5); // ERROR: a is not a mutable reference.
           // (Indeed, v is not a mutable Vector!)
```

In Rust, there is a notion of *ownership* of an object. When a variable is bound to a newly allocated memory object, the variable *owns* the object. For example, in the code above, `z` initially owns the memory object it is bound to. Note that once variable `z` goes out of scope, we could free the memory that `z` is bound

to. This is good! It means that we can automatically and efficiently recover the memory resources, without the programmer needing to worry about explicitly freeing memory.

Ownership can *move* from one variable to another, permanently or temporarily. If ownership is moved temporarily, it is called *borrowing*. In the following example, ownership of the vector object is transferred from `v` to `w`, and, once `a` goes out of scope, it is transferred back to `v`.

```
let mut v = vec![20,21,22];
{
    let a = &mut v;
    a.push(23);
    println!("The answer is {}", a[3]);
}
// ownership is now back with v
v.push(24);
```

The Rust type system enforces two invariants:

1. A reference cannot outlive its referent
2. A mutable reference cannot be aliased

To see why the first invariant is important, consider the following code snippet, where the lifetime of reference a is longer than the lifetime of the object bound to v. (Note: we mark `a` as `mut`able so that we can assign to it a reference to `v`.)

```
let mut a = &mut vec![0];
{
  let mut v = vec![20,21,22];
  let a = &mut v;
  a.push(23);
} // vector object is deallocated here

println!("The answer is {}", a[3]); // ERROR: use after free,
                                    // since a outlives v
```

The second invariant in primarily useful to ensure safety in concurrent settings (and we will consider concurrency in later lectures). However, note that it is completely fine to have multiple immutable references to the same object! In the following example, one mutable reference to the object is borrowed by multiple immutable references.

```
let mut v = vec![20,21,22];
{
    let a = &v; // immutable reference to v
    let b = &v; // immutable reference to v

    // ownership of v is now shared between a and b

    println!("The answer is {}", a[0] + b[2]);
}
// ownership is now back with v
// we can use v to update the object.
v.push(24);
```

Rust includes many more features beyond those we've had space to discuss here. More information about Rust is available at `https://www.rust-lang.org/`. Extensive pointers to resources for learning Rust are at `https://github.com/ctjhoa/rust-learning`. For the material covered in this lecture, the Chapter 4 (Ownership and Lifetimes) of *The Rustonomicon* is most relevant (see `https://doc.rust-lang.org/nomicon/ownership.html`).