University of Glasgow

Type-based Modelling and Design

Advanced Systems Programming (M)

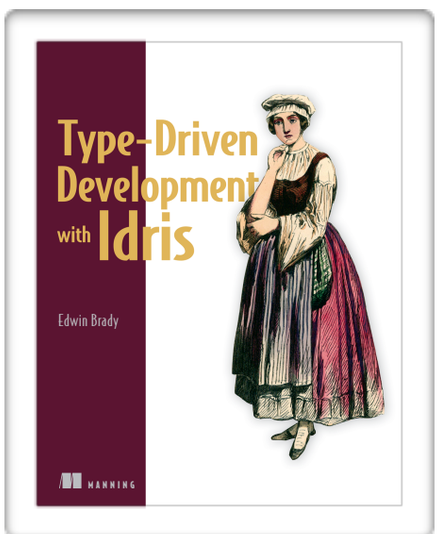
Lecture 3

Type-driven Development

- Define the types
- Write the functions
- Refine as needed

Type-driven Development

- **Define the types** first
- Using the types as a guide, **write the functions**
  - Write the input and output types
  - Write the function, using the structure of the types as a guide
- **Refine** and edit types and functions as necessary



Type-drive development approach adapted from: E. Brady, "Type-Driven Development with Idris", Manning, March 2017.

- Don't think of the types as checking the code, think of them as a plan – a model – for the solution

Define the Types (1/2)

- Define the types needed to build a domain model
  - Who is interacting? What do they interact with? What sorts of things do they exchange?

    `Employee` `Sender` `TcpSegment` `Connection`

    `Receiver` `Vehicle` `Cargo`

  - What sort of properties describe those people and things? What data is associated with each?

    `TemperatureInCelsius` `Manufacturer` `Colour`

    `SequenceNumber` `Name` `EmailAddress`

  - What states can the interaction be in?

    `Connecting` `Sent` `LoggedIn` `AuthenticationRequired`

- Types might initially be ill-defined and abstract
  - Write them down anyway – refine later

Define the Types (2/2)

- Associate properties with the types:
  - What data is associated with a thing? What properties does it have?

```
struct Sender {
  name    : Name,
  email   : EmailAddress,
  address : PostalAddress
}
```

  - What state is something in?

```
enum State {
  NotConnected,
  Connecting,
  AuthenticationRequired,
  LoggedIn,
  ...
}
```

```
struct UnauthenticatedConnection {
  socket : TcpSocket,
 ...
}
```

```
struct AuthenticatedConnection {
  socket : TcpSocket,
 ...
}
```

- Refine and extend the types as needed

Write the Functions (1/2)

- Using the types as a guide, write the **function prototypes**, leaving the concrete implementation for later

```
impl UnauthenticatedConnection {
  fn login(self, c: Credentials) -> Result<AuthenticatedConnection, LoginError> {
    ...
  }

  fn disconnect(self) {
    ...
  }
}
```

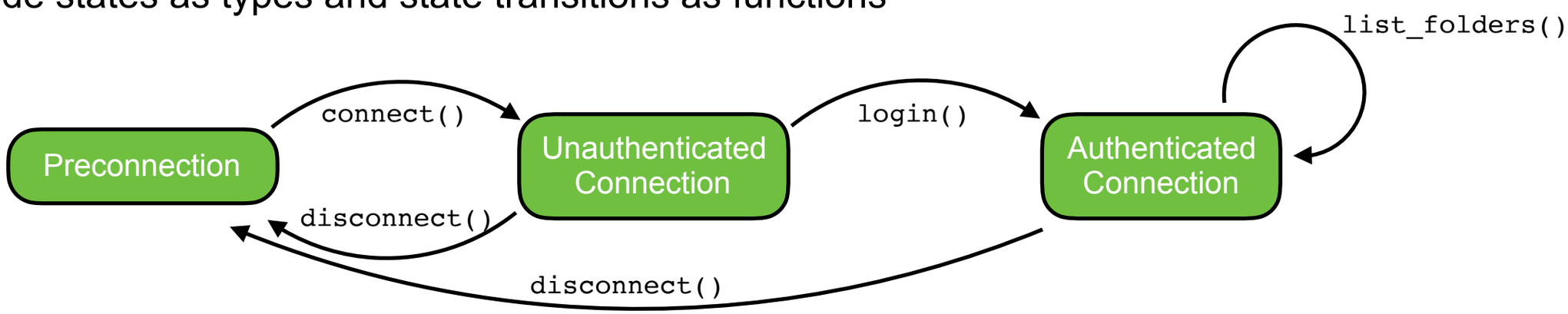
```
impl AuthenticatedConnection {
  fn list_folders(self) -> List<EmailFolder> {
    ...
  }

  fn list_messages(self, f : EmailFolder) -> List<EmailMessage> {
    ...
  }

  fn disconnect(self) {
    ...
  }
}
```

Write the Functions (2/2)

- Behaviour obvious from the types – and types constrain behaviour
  - Use specific rather than generic types
    - e.g., take a `Username` as a parameter, rather than a `String`
    - Types provide machine checkable documentation
  - Encode states as types and state transitions as functions



- Functions only implemented for the types where they make sense
- e.g., cannot `list_folders()` until after `login()`; types prevent invalid operations

Refine Types and Functions

- **Types** and **functions** provide a model of the system
- Iterate – filling in just enough details to keep it compiling
  - **Interactive design** using the compiler to check consistency
  - Gradually refine until the entire system is modelled – then add the concrete implementations, refining as needed
  - Work with the compiler to validate the design, before detailed implementation

Correct by Construction

- Use types to check the design, debugging before you run the code
- Non-sensical operations don't cause a crash – *they don't compile*

Design Patterns

- Specific numeric types
- Enumerations and string typing

Numeric Types

- Is a value really a `double` or `int`, or does it have some meaning?
  - Temperature in degrees celsius
  - Speed in miles per hour
  - UserID
  - Packet sequence number
  - ...
- Encode the meaning as a type, so the compiler checks for consistent usage
  - Operations that mix types should fail, or perform safe unit conversions if possible
  - Operations that are inappropriate for a type shouldn't be possible

Numeric Types – Strong Typing

```
fn main() {
    let c = 15.0;  // Celsius
    let f = 50.0;  // Fahrenheit

    let t = c + f;

    println!("{:?}", t); // 65.0
}
```

- Weakly typed – programmer knows `c` and `f` are different types, but the compiler does not
- Program silently calculates the wrong answer

Numeric Types – Strong Typing

- Strongly typed – `struct` with single unnamed field wraps numeric value
  - `derive` and `impl` standard operations
    https://crates.io/crates/newtype_derive has macros to auto-generate `impl` blocks
  - Resulting code won't compile since types are mismatched

```
use std::ops::Add;

#[derive(Debug,PartialEq,PartialOrd)]
struct Celsius(f32);

#[derive(Debug,PartialEq,PartialOrd)]
struct Fahrenheit(f32);

impl Add for Celsius {
    type Output = Celsius;

    fn add(self, other : Celsius) -> Self::Output {
        Celsius(self.0 + other.0)
    }
}

impl Add for Fahrenheit {
    type Output = Fahrenheit;

    fn add(self, other : Fahrenheit) -> Self::Output {
        Fahrenheit(self.0 + other.0)
    }
}
fn main() {
    let c = Celsius(15.0);
    let f = Fahrenheit(50.0);

    let t = c + f;

    println!("{:?}", t);
}
```

```
> cargo build
   Compiling foo v0.1.0 (/Users/csp/tmp/foo)
error[E0308]: mismatched types
  --> src/main.rs:28:17
   |
28 |     let t = c + f;
   |                 ^ expected struct `Celsius`, found struct `Fahrenheit`
   |
   = note: expected type `Celsius`
              found type `Fahrenheit`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308`.
error: Could not compile `foo`.

To learn more, run the command again with --verbose.
>
```

Numeric Types – Conversion

- We can add implementations that perform unit conversion

```
impl Add<Fahrenheit> for Celsius {
    type Output = Celsius;

    fn add(self, other: Fahrenheit) -> Self::Output {
        Celsius(self.0 + ((other.0 - 32.0) * 5.0 / 9.0))
    }
}

fn main() {
    let c = Celsius(15.0);
    let f = Fahrenheit(50.0);

    let t = c + f;
    println!("{:?}", t);   // Celsius(25.0)
}
```

Numeric Types – Operations

- Do all the standard operations make sense for the type?
  - It's reasonable to compare `Celsius` values:

```
fn is_freezing(temp: Celsius) -> bool {
    temp < Celsius(0.0)
}
```

  so you'd implement the `Ord` trait that provides these operations

- But this might not make sense for a `UserID` type
  - You likely want to be able to compare two `UserID` values for equality (the `Eq` trait), but adding two `UserID` values or comparing to see which is largest might not be meaningful
  - Not all standard operations need to be implemented for a type

Numeric Types – No Runtime Cost

- Wrapping value inside `struct` adds zero runtime overhead in Rust
  - Programmer must implement standard operations – some extra code, but no runtime cost
    - https://crates.io/crates/newtype_derive provides macros for the common cases
  - Why no runtime cost?
    - No information added to the `struct`, so same size
    - Passed in the same way – not automatically boxed on the heap
    - Optimiser will recognise that the code collapses down to operations on primitive types, and generate the code to do so
    - All the additions are a compile-time model of the ways the data can be used, they don't affect the compiled code
    - *(Equivalent C++ code has the same properties)*

Alternative Types: `enum`

- Encode alternatives and options as types:
  - Optional values: `Option<T>`
  - Results: `Result<T, E>`
  - Features and response codes
- Use of `enum` types and pattern matching allows for rich modelling of alternatives and options

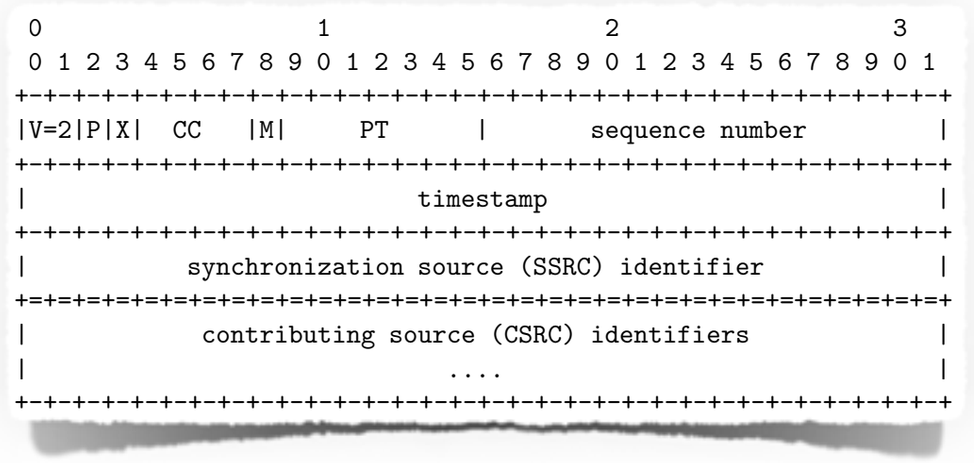
Optional Values: `Option<T>`

- If a value might not exist, use `Option<T>`
  - As function return value, pattern matching on result:

```
fn lookup(self, user : Username) -> Option<User>
```

  - In `struct` definition:

```
struct RtpHeader {
  v    : Version,
  pt   : PayloadType,
  seq  : SequenceNumber,
  ts   : Timestamp,
  ssrc : SourceId,
  csrc : Vec<SourceId>,
  extn : Option<HeaderExtension>,
  payload : RtpPayload
}
```



- Compiler enforces that both variants of `Option<T>` are handled
  - `Some(T)`, `None`
  - Can't accidentally write code that assumes the value is present and crashes otherwise

Results: `Result<T, E>`

- The `Result<T, E>` type represents results that can fail
- Used as a result type for a function:

```
fn load_document() -> Result<Document, DatabaseError> {
    let db = open_database()?;
    db.load("document_1")?
}

...
match load_document() {
  Ok(doc) => println!(doc), // success
  Err(e)  => ...            // failed
}
```

Use of ? operator for early return on error

Custom error types can be defined: https://doc.rust-lang.org/rust-by-example/error/multiple_error_types/define_error_type.html

- C code frequently uses a signal value to indicate errors
  - e.g., `socket()` returns -1 on error, file descriptor >0 on success
- Easy to forget to check error codes – such code won't compile in Rust

Features and Response Codes

- Anti-pattern: “string typing”
  - Method parameters that are strings, rather than some more appropriate type
  - Strings returned from network function (e.g., HTTP response codes) directly used, rather than converted to appropriate type

- Use `enum` to represent values that can be one of several alternatives
  - Exhaustiveness checking – catches bugs if new codes introduced
  - Ease of refactoring – decouples code from external representation
  - Make nonsensical values unrepresentable
  - Types are machine checkable documentation – strings are not

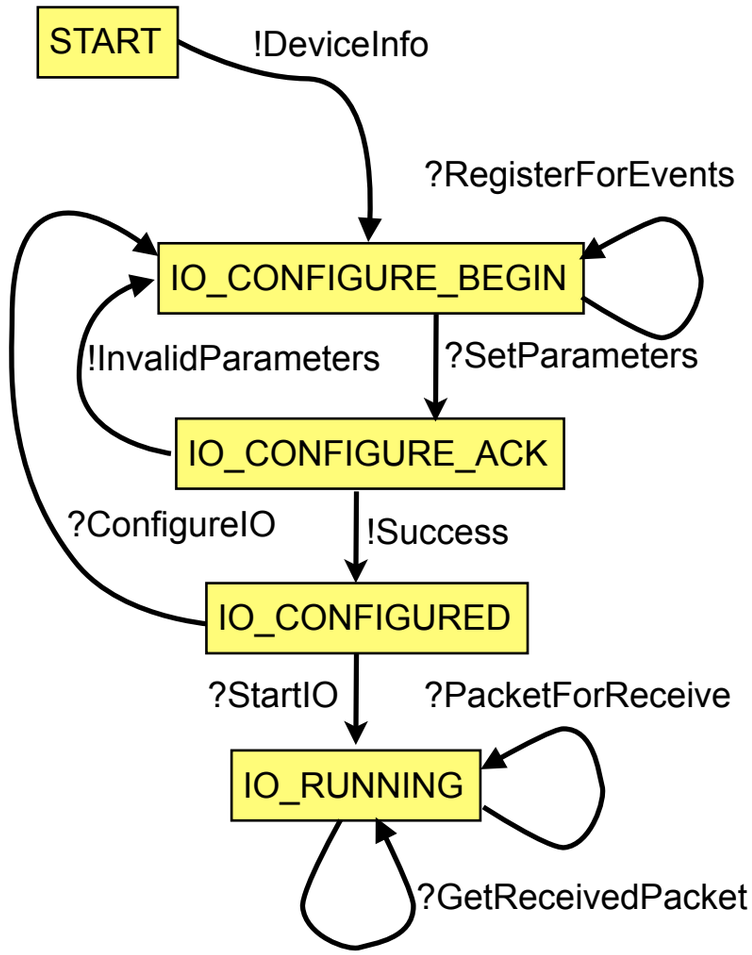
State Machines

- What is a state machine?
- Implementation using `enum` types
- Implementation using `struct` types

State Machines

- State machines common in systems code
  - Network protocols
  - File systems
  - Device drivers

- System behaviour modelled as a finite state machine comprising:
  - **States** that reflect the status of the system
  - **Events** that trigger transitions between states
  - **State** variables that hold system configuration
- Clean high-level model of a system
  - Captures the essence of the behaviour
  - Easy to reason about and prove properties such as termination, absence of deadlocks, reachability, etc.



Implementing State Machines

- Hard to cleanly model state machine in code
  - Structure of code tends not to match structure of state machine; not easy to visualise transitions
  - Difficult to validate code against specification

- Approaches to modelling state machines in strongly-typed functional languages:
  - Encode states and events as enumerations, pattern match on (state, event) tuples
  - Encode states as types and transitions as functions
  - Add first-class state machine support to language
    - Microsoft Singularity research operating system
    - `async`/`await` asynchronous code → lecture 7

```
contract NicDevice {
  out message DeviceInfo(...);
  in  message RegisterForEvents(NicEvents.Exp:READY
  c);
  in  message SetParameters(...);
  out message InvalidParameters(...);
  out message Success();
  in  message StartIO();
  in  message ConfigureIO();
  in  message PacketForReceive(byte[] in ExHeap p);
  out message BadPacketSize(byte[] in ExHeap p, int
  m);
  in  message GetReceivedPacket();
  out message ReceivedPacket(Packet * in ExHeap p);
  out message NoPacket();

  state START: one {
    DeviceInfo! → IO_CONFIGURE_BEGIN;
  }
  state IO_CONFIGURE_BEGIN: one {
    RegisterForEvents? →
      SetParameters? → IO_CONFIGURE_ACK;
  }
  state IO_CONFIGURE_ACK: one {
    InvalidParameters! → IO_CONFIGURE_BEGIN;
    Success! → IO_CONFIGURED;
  }
  state IO_CONFIGURED: one {
    StartIO? → IO_RUNNING;
    ConfigureIO? → IO_CONFIGURE_BEGIN;
  }
  state IO_RUNNING: one {
    PacketForReceive? → (Success! or BadPacketSize!)
      → IO_RUNNING;
    GetReceivedPacket? → (ReceivedPacket! or
    NoPacket!)
      → IO_RUNNING;
    ...
  }
}
```

Listing 1. Contract to access a network device driver.

G. Hunt and J. Larus. "Singularity: Rethinking the software stack", ACM SIGOPS OS Review, 41(2), April 2007. DOI:10.1145/1243418.1243424

Enumerations for modelling state machines

- Possible state machine representation:
  - An enumerated type (`enum`) models alternatives
    - Define an `enum` to represent the states
    - Define an `enum` to represent the events
  - Functions represent transitions and actions:
    - Define a function to map from (state, events) tuples to next state
    - Define a function to perform the actions associated with each state
- Builds on the intuition that `enum` types express alternatives, and a state machine comprises a list of alternative states

Using `enum` to Model State Machines: Example (1/3)

```
enum ApcState {
    Initialize,
    WaitForConnect,
    Accept(TcpStream),
    StartTransfer(TcpStream),
    Waiting(TcpStream),
    ReceiveMsg(TcpStream, Vec<u8>),
    SendNop(TcpStream),
    Closed,
    Finish,
    Failure(String),
}
```

```
enum ApcEvent {
    TcpConnected(TcpStream),
    ResponseValid(bool),
    IncomingTcpClosed,
    AspMsgIn(Vec<u8>),
    NopTimeout,
    Finished,
    Uct,
}
```

- Define an `enum` to represent the states and another the events
- State variables specific to a state encoded as `enum` parameters

Example adapted from comment on https://hoverbear.org/2016/10/12/rust-state-machine-pattern/

Using `enum` to Model State Machines: Example (2/3)

```
impl ApcState {
    pub fn next(self, event: ApcEvent) -> Self {
        use self::ApcState::*;
        use self::ApcEvent::*;

        println!("NEXT with {:?}/{:?}", self, event);

        match (self, event) {
            (Initialize,          TcpConnected(tcp)   ) => Accept(tcp),
            (Initialize,          Finished            ) => Finish,
            (Accept(tcp),         ResponseValid(true) ) => StartTransfer(tcp),
            (Accept(_),           ResponseValid(false)) => Closed,
            (StartTransfer(tcp), Uct                  ) => Waiting(tcp),
            (Waiting(_),          IncomingTcpClosed   ) => Closed,
            (Waiting(_),          Finished            ) => Finish,
            (Waiting(tcp),        AspMsgIn(msg)       ) => ReceiveMsg(tcp, msg),
            (Waiting(tcp),        NopTimeout          ) => SendNop(tcp),
            (ReceiveMsg(tcp, _), Uct                  ) => Waiting(tcp),
            (SendNop(tcp),        Uct                 ) => Waiting(tcp),
            (s, e) => Failure(format!("Invalid State/Event combination: {:?}/{:?}", s, e)),
        }
    }
}
```

- Gives clean representation of state-transition table
- Straight-forward to validate against specification

Using `enum` to Model State Machines: Example (3/3)

```
pub struct ApcStateMachine {
    pub state: ApcState,
    addr: SocketAddr,
    timeout: u64,
}

impl ApcStateMachine {
    fn new() -> ApcStateMachine {
        ...
    }

    fn run_once(&self) -> ApcEvent {
        match self.state {
            Initialize          => ...
            WaitForConnect      => ...
            Accept(tcp)         => ...
            StartTransfer(_)    => ...
            Waiting(tcp)        => ...
            ReceiveMsg(_, msg)  => ...
            SendNop(_)          => ...
            Closed              => ...
            Finish              => ...
        }
    }
}
```

```
fn run_state_machine() {
    let mut sm = ApcStateMachine::new();
    loop {
        let event = sm.run_once();
        sm.state = sm.state.next(event);
        if sm.state == ApcState::Finish {
            break;
        }
    }
}
```

- `match` loop dispatches to functions
  - Performs actions each state, returns next event to process → determine next state
  - Parameterised `enum` with state variables makes it easy to pass parameters
- Pattern matching on `enum` gives a clear implementation
  - Compiler checks all alternates covered
  - Easy to pass state variables

Structures for Modelling State Machines

- Alternative state machine representation:
  - Define a `struct` representing each state
  - Model an event as a method call on a `struct`
  - Model state transitions by returning a `struct` representing the new state
- Builds on the intuition that states hold concrete *state*, and events are things that happen in states

Using `struct` to Model State Machines: Example

```
struct UnauthenticatedConnection {
  socket : TcpSocket,
 ...
}
```

```
struct AuthenticatedConnection {
  socket : TcpSocket,
 ...
}
```

- Define a `struct` representing each state
  - State variables are fields within the `struct`
  - Methods implemented on the `struct` encode state transitions and event handlers
    - Return `Self` if state is unchanged
    - Return `struct` representing new state if state changes

```
impl UnauthenticatedConnection {
  fn login(self, c: Credentials) -> Result<AuthenticatedConnection, LoginError> {
    ...
  }

  fn disconnect(self) -> NotConnected {
    ...
  }
}
```

- Encodes states and state transitions in types
  - `enum`-based approach codes states and events as types, and transitions as a table

Approaches to Representing State Machines

- `enum`-based approach is compact, makes states and events clear in the types, and has clear state transition table
  - Relies on expressive `enum` types for implementation – harder to express in languages with weaker `enum`
- `struct`-based approach encodes states and state transitions in the types, events as methods on those types
  - State transition table is less obviously explicit in the code
  - State transitions map to Rust ownership rules – enforce transitions

Ownership

- Ownership of data in Rust
- Enforcing state transitions

Ownership

- Systems programs care about ownership of resources
  - To control memory management, close files, etc. → lecture 4
  - To model state machines

- Programmer maintains a mental model of what part of the code owns each resource
  - What function is responsible for calling `free()`, `close()`, etc.
  - Garbage collected languages still require understanding of ownership – but make `free()` call automatic when lifetime ends
  - C++ and Python tie resource ownership to scoping:

```
with open(filename) as file:
    data = file.read()
    ...
```

    gives automatic resource clean-up at end of scope

State Machines and Ownership

- State machines manage resources
  - A network protocol manages connections, and the data sent over them
  - A device driver manages hardware resource
  - ...
- State transitions indicate when resources created/go out-of-scope
  - Transition consumes the old state, returns a new state

Ownership in Rust

- Rust tracks ownership of data – enforces that every value has a single owner
- Function calls explicitly manage ownership of values
  - Take explicit ownership of a value

```
fn consume(r : Resource) {
  ...
}
```

Function *takes ownership* of parameter passed by value
No longer accessible to caller; freed at end of function

  - Borrow a value

```
fn borrow(r : &Resource) {
  ...
}
```

Function *borrows* the parameter passed via reference
Ownership remains with caller

  - Return ownership of a value

```
fn generate() -> Resource {
  ...
}
```

Function *passes ownership* of return value to caller

Ownership in Rust

```
struct Resource {
  value : u32
}
```

```
fn consume(r : Resource) {
  println!("consumed");
}
```

```
fn main() {
  let r = Resource{value: 42};
  consume(r);
  println!("{}", r.value);
}
```

Function *takes ownership* of parameter passed by value
No longer accessible to caller; freed at end of function

The **`consume()`** function takes ownership of the resource – doesn't return it to the caller

Above code won't compile: **`println!()`** cannot access **`r.value`**, since **`main()`** no longer has access to **`r`** because it gave ownership to **`consume()`**

Ownership and state machines (1/2)

- `struct`-based approach to state machines uses ownership rules to enforce state transitions
  - Methods that change state take ownership of `self`, return new `struct`:

```
impl UnauthenticatedConnection {
  fn login(self, c: Credentials) -> Result<AuthenticatedConnection, LoginError> {
    ...
  }

  fn disconnect(self) -> NotConnected {
    ...
  }
}
```

- e.g., the **`login()`** function consumes its **`UnauthenticatedConnection`** and returns a new **`AuthenticatedConnection`** on success
  - The compiler enforces this – the **`UnauthenticatedConnection`** is not accessible after this call; all resources it owned are reclaimed
  - Except any the **`login()`** method explicitly copies to the **`AuthenticatedConnection`**

Ownership and state machines (2/2)

- `struct`-based approach to state machines uses ownership rules to enforce state transitions
  - Guarantees resource cleanup on state transition
  - Better for ensuring resources are cleaned-up after use
- `enum`-based approach to state machines makes the state transition diagram clearer, but relies on programmer discipline to clean-up
- Better for ensuring complex state machines correctly reflected in code

Type-driven Development – Recap

- **Define the types** first
  - Define concrete numeric types, identifiers
  - Define `enum` types to represent alternatives
  - Indicate optional values, results, error types
- Using the types as a guide, **write the functions**
  - Write the input and output types
  - Write the function, using the structure of the types as a guide
  - Make state machines explicit
  - Consider ownership of data
- **Refine** and edit types and functions as necessary
  - Use the compiler as a tool to help you debug your design

- Don't think of the types as checking the code, think of them as a plan, a model, for the solution – and as machine checkable documentation

Summary

- Type-drive development
- Design patterns
- State machines
- Ownership