University of Glasgow

Type-based Modelling and Design

Advanced Systems Programming (H/M)

Lecture 8

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow | This work is licensed under the Creative Commons Attribution-NoDerivatives 4.0 International License.
To view a copy of this license, visit http://creativecommons.org/licenses/by-nd/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

Type-driven Development

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

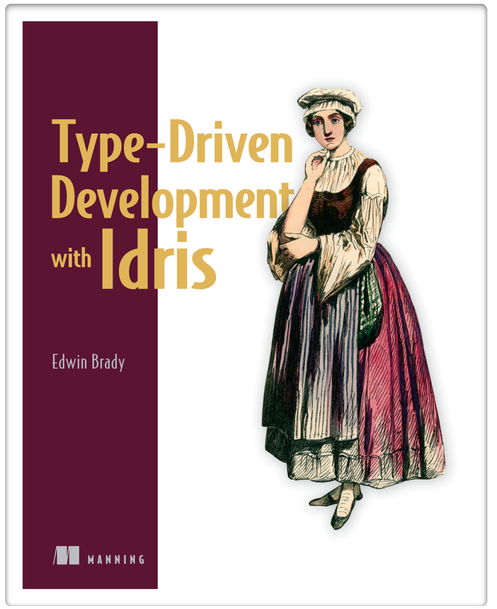
Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Type-driven Development

- With expressive, strongly typed, languages – such as Rust, Swift, OCaml – can use the type system to help ensure correctness

- **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

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



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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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` `Connection` `Cargo`

    `Receiver` `TcpSegment` `Vehicle`

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

    `TemperatureInCelsius` `Name` `SequenceNumber`

    `Manufacturer` `EmailAddress` `Colour`

  - What states can the interaction be in?

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

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

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

- Refine and extend the types as needed

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Write the Functions (1/2)

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

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

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

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

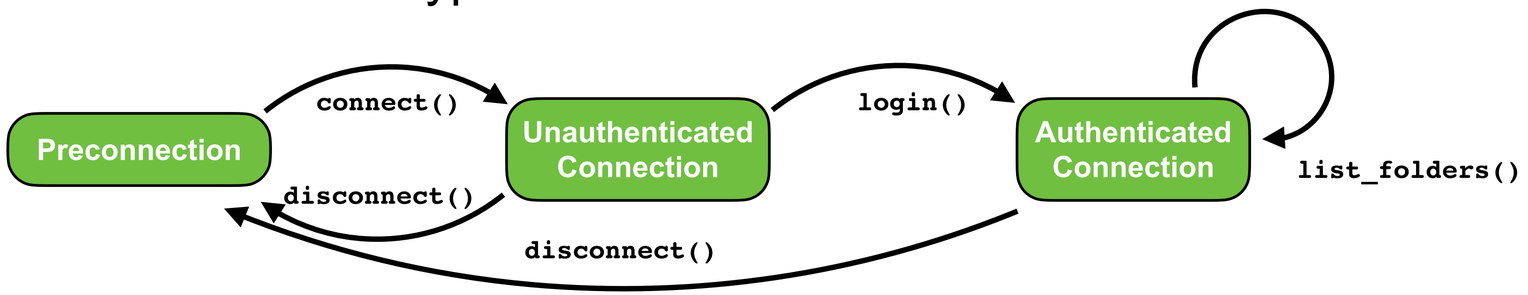
```
impl UnauthenticatedConnection {
  fn login(self, c: Credentials) -> Result<AuthenticatedConnection, LoginError> {
    ...
  }

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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



    - State transitions return new types: **`login()`** consumes an **`UnathenticatedConnection`**, returns an **`AuthenticatedConnection`** object
    - Functions only implemented for the types where they make sense
      - e.g., **`list_folders()`** only implemented on **`UnathenticatedConnection`**, so can't be called until after successful **`login()`**

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

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

- Change of perspective: the compiler is a model checking tool that can help validate your design

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow

Type-driven Development

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

Colin Perkins | https://csperkins.org/ | Copyright © 2020 University of Glasgow