Static analysis for API error detection in IoT devices

Tapioca: a library for reasoning about web requests at compile-time

Oliver Ford

28 June 2017

Imperial College London

Contents

Motivation

Errors in device communication

nest

Who will notice?

- Many IoT devices have no user-facing display

Who will notice?

- Many IoT devices have no user-facing display
  - and may not report an error anyway

Who will notice?

- Many IoT devices have no user-facing display
  - and may not report an error anyway
- Server operator might notice

Who will notice?

- Many IoT devices have no user-facing display
  - and may not report an error anyway
- Server operator might notice
  - but that may not be the device manufacturer

Who will notice?

- Many IoT devices have no user-facing display
  - and may not report an error anyway
- Server operator might notice
  - but that may not be the device manufacturer

Examples

Who will notice?

- Many IoT devices have no user-facing display
  - and may not report an error anyway
- Server operator might notice
  - but that may not be the device manufacturer

Examples

- Nest thermostat looking up local weather on third-party API

Who will notice?

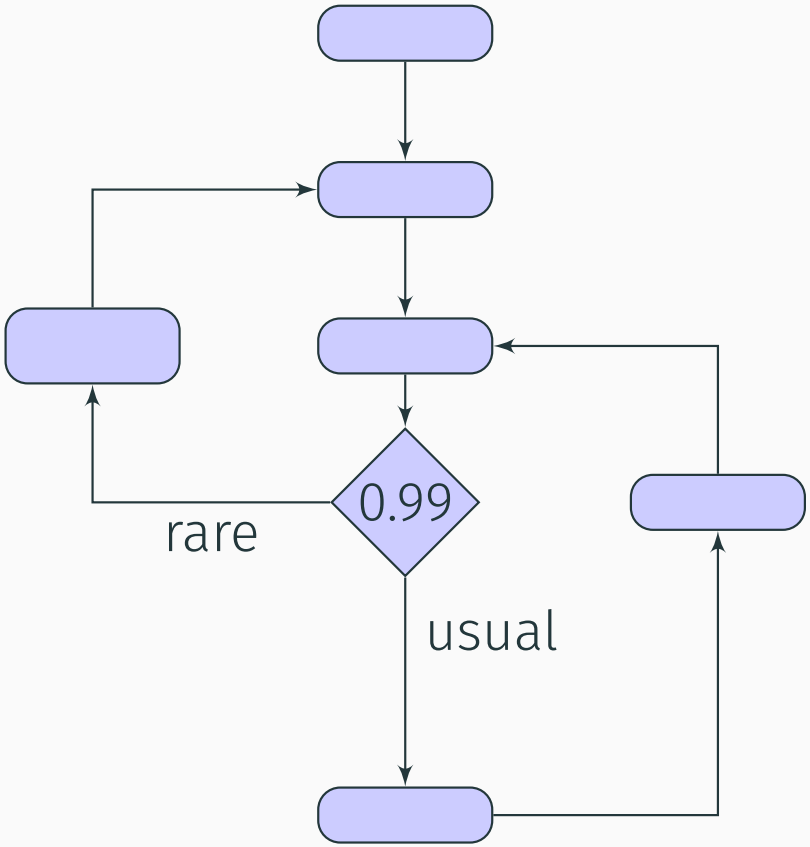
- Many IoT devices have no user-facing display
  - and may not report an error anyway
- Server operator might notice
  - but that may not be the device manufacturer

Examples

- Nest thermostat looking up local weather on third-party API
- Communication with a cross-vendor IoT device ‘bridge’

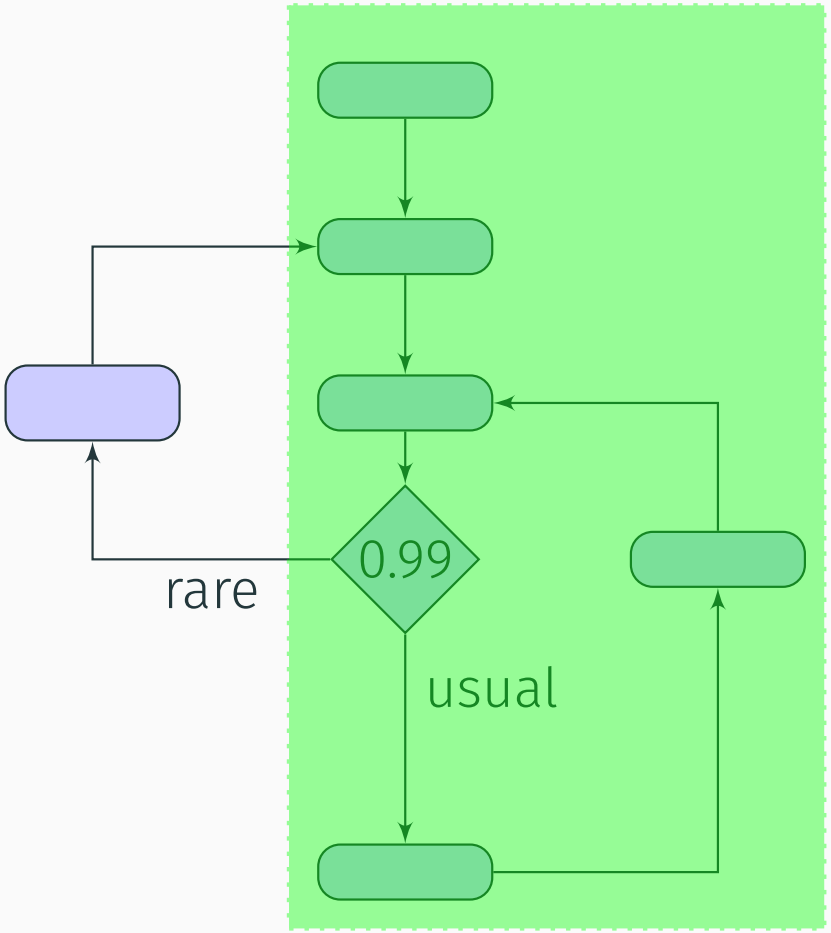
Test coverage?

- So? Just test it, right?



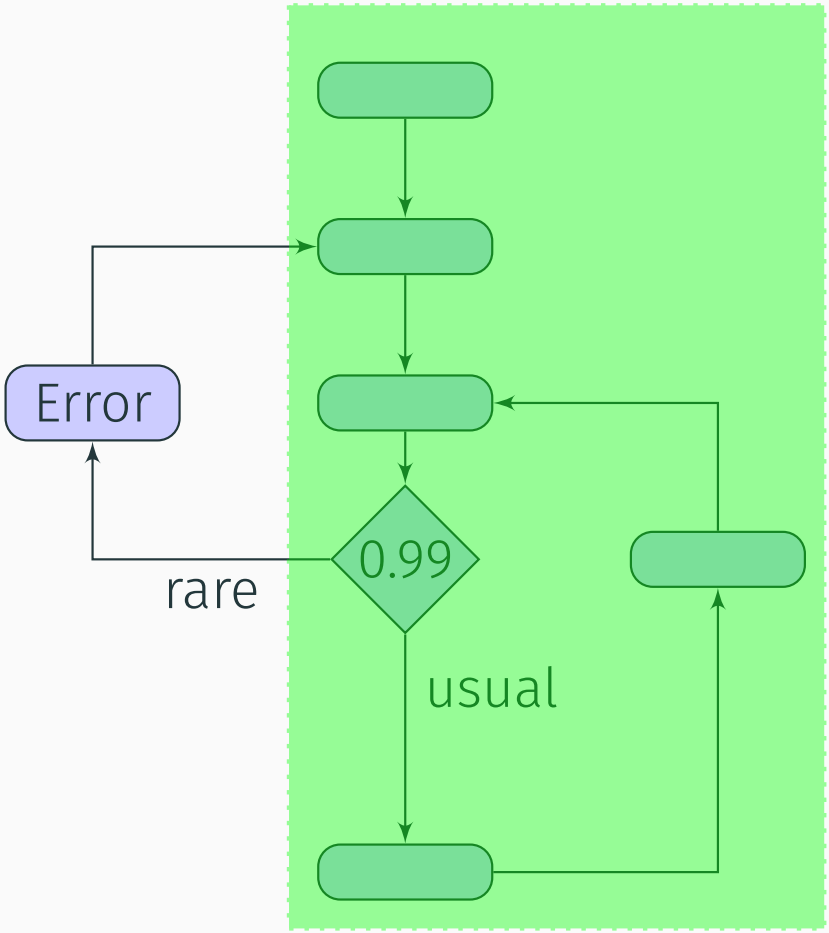
Test coverage?

- So? Just test it, right?
- $>$ 99% coverage, great!



Test coverage?

- So? Just test it, right?
- $>$ 99% coverage, great!
- Oh...



Server logging?

- Low signal to noise

Server logging?

- Low signal to noise
- Owner/operator may be distinct from device software provider

Server logging?

- Low signal to noise
- Owner/operator may be distinct from device software provider
- Not preventative

Formal verification?

- Ideal: total prevention of incorrect code

Formal verification?

- Ideal: total prevention of incorrect code
- Hard, expensive, specialised

Formal verification?

- Ideal: total prevention of incorrect code
- Hard, expensive, specialised
- Some research into extending to web services/REST

Formal verification?

- Ideal: total prevention of incorrect code
- Hard, expensive, specialised
- Some research into extending to web services/REST
- But nothing available to the application developer today

Need to do better

- it *should* be tested...

Need to do better

- it *should* be tested...
- *maybe* we learn of the error if it's not...

Need to do better

- it *should* be tested...
- *maybe* we learn of the error if it's not...
- *hopefully* we then fix it...

Need to do better

- it *should* be tested…
- *maybe* we learn of the error if it's not…
- *hopefully* we then fix it…

Not very promising: *surely* this can be avoided?

REpresentational State Transfer

REST: a stateless ’app’ model for the web

Classical REST (Fielding’s thesis):

- Stateless applications: ‘state’ contained in request/response

REST: a stateless 'app' model for the web

Classical REST (Fielding's thesis):

- Stateless applications: 'state' contained in request/response
- Resource available at its own URI

REST: a stateless 'app' model for the web

Classical REST (Fielding's thesis):

- Stateless applications: 'state' contained in request/response
- Resource available at its own URI
- Resource representation may be any hypermedia (HTML, JPEG, ...)

REST: a stateless 'app' model for the web

Classical REST (Fielding's thesis):

- Stateless applications: 'state' contained in request/response
- Resource available at its own URI
- Resource representation may be any hypermedia (HTML, JPEG, ...)
- HTTP methods have different semantics, idempotency

REST: RPC-ish over JSON

Typical modern ‘REST’ (REST API):

- Resource URIs contain human-readable ‘breadcrumb’ hierarchy

REST: RPC-ish over JSON

Typical modern ‘REST’ (REST API):

- Resource URIs contain human-readable ‘breadcrumb’ hierarchy
  - `/foobars` is a *collection* resource
  - `/foobars/42` is a single *entity* resource

REST: RPC-ish over JSON

Typical modern ‘REST’ (REST API):

- Resource URIs contain human-readable ‘breadcrumb’ hierarchy
  - `/foobars` is a *collection* resource
  - `/foobars/42` is a single *entity* resource
- JSON resource representation: `{"foobars": [{"id": 42}]}`

REST: RPC-ish over JSON

Typical modern ‘REST’ (REST API):

- Resource URIs contain human-readable ‘breadcrumb’ hierarchy
  - `/foobars` is a *collection* resource
  - `/foobars/42` is a single *entity* resource
- JSON resource representation: `{"foobars": [{"id": 42}]}`
- HTTP verbs have specialised semantics for collection/entity

The Problem

```
let response = http_client.get("api.com/foobar");
```

The Problem

```
let response = http_client.get("api.com/foobar");

// oops, wasn't it plural on the last slide? ^
```

The Problem

```
let response = http_client.get("api.com/foobar");

// oops, wasn't it plural on the last slide? ^

let json = deserialise_http_response(response.body());
let foobar_list = json["fopbars"];
do_something_with(foobar_list);
```

The Problem

```
let response = http_client.get("api.com/foobar");

// oops, wasn't it plural on the last slide? ^

let json = deserialise_http_response(response.body());
let foobar_list = json["fopbars"];
do_something_with(foobar_list);

// easy typos to make;   ^^ harder to spot
```

The Problem

```
let response = http_client.get("api.com/foobar");

// oops, wasn't it plural on the last slide? ^

let json = deserialise_http_response(response.body());
let foobar_list = json["fopbars"];
do_something_with(foobar_list);

// easy typos to make;   ^^ harder to spot
```

‘Stringly-typed’: these errors won’t fail until run-time.

The Problem

```
let response = http_client.get("api.com/foobar");

// oops, wasn't it plural on the last slide? ^

let json = deserialise_http_response(response.body());
let foobar_list = json["fopbars"];
do_something_with(foobar_list);

// easy typos to make;   ^^ harder to spot
```

‘Stringly-typed’: these errors won’t fail until run-time.

Serialisation of request bodies, query/path params is similarly problematic.

Idea: bring REST semantics ‘into’ the (client) language

‘the language’

Rust fits the bill

‘the language’

Rust fits the bill:

- Can target embedded devices

‘the language’

Rust fits the bill:

- Can target embedded devices
- Type-checking enables analysis of bodies, parameters, et al.

‘the language’

Rust fits the bill:

- Can target embedded devices
- Type-checking enables analysis of bodies, parameters, et al.
- Borrow-checking enables (some) analysis of state

‘the language’

Rust fits the bill:

- Can target embedded devices
- Type-checking enables analysis of bodies, parameters, et al.
- Borrow-checking enables (some) analysis of state

Despite their names, both checkers are really features of Rust’s type system.

Strong and static (no, that's not the PM's new slogan)

- Static: types inferred and verified at compile-time

Strong and static (no, that's not the PM's new slogan)

- Static: types inferred and verified at compile-time
  - idea: types for components of requests, responses

Strong and static (no, that's not the PM's new slogan)

- Static: types inferred and verified at compile-time
  - idea: types for components of requests, responses
- Strong: invoked methods must be implemented for that type

Strong and static (no, that's not the PM's new slogan)

- Static: types inferred and verified at compile-time
  - idea: types for components of requests, responses
- Strong: invoked methods must be implemented for that type
  - resp. fields on that `struct`, etc.

Strong and static (no, that's not the PM's new slogan)

- Static: types inferred and verified at compile-time
  - idea: types for components of requests, responses
- Strong: invoked methods must be implemented for that type
  - resp. fields on that `struct`, etc.
  - idea: `enum` variant for each response status code

Affine (theory)

A whirlwind introduction

Affine (theory)

A whirlwind introduction:

- Structural logic: its proof system consists of inference rules

Affine (theory)

A whirlwind introduction:

- Structural logic: its proof system consists of inference rules
- Contraction rule:

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}$$

Affine (theory)

A whirlwind introduction:

- Structural logic: its proof system consists of inference rules
- Contraction rule:

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}$$

- Affine logic: a *sub*-structural logic, removing contraction

Affine (theory)

A whirlwind introduction:

- Structural logic: its proof system consists of inference rules
- Contraction rule:

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}$$

- Affine logic: a *sub*-structural logic, removing contraction
- Affine type system: variables used at most once

Affine (theory)

A whirlwind introduction:

- Structural logic: its proof system consists of inference rules
- Contraction rule:

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}$$

- Affine logic: a *sub*-structural logic, removing contraction
- Affine type system: variables used at most once
  - useful for modelling external resources: I/O, locks, ...

Affine (theory)

A whirlwind introduction:

- Structural logic: its proof system consists of inference rules
- Contraction rule:

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}$$

- Affine logic: a *sub*-structural logic, removing contraction
- Affine type system: variables used at most once
  - useful for modelling external resources: I/O, locks, ...
  - idea: REST resources too?

Affine (practice)

Memory management in Rust

Affine (practice)

Memory management in Rust:

- No garbage collection (GC)

Affine (practice)

Memory management in Rust:

- No garbage collection (GC)
- No use after ‘move’ (passing by value)

Affine (practice)

Memory management in Rust:

- No garbage collection (GC)
- No use after ‘move’ (passing by value)
- No ‘borrowing’ (a reference) that’s already mutably borrowed

Affine (practice)

Memory management in Rust:

- No garbage collection (GC)
- No use after ‘move’ (passing by value)
- No ‘borrowing’ (a reference) that’s already mutably borrowed
- No borrowing for (a ‘lifetime’) longer than scope of owner

Affine (practice)

Memory management in Rust:

- No garbage collection (GC)
- No use after ‘move’ (passing by value)
- No ‘borrowing’ (a reference) that’s already mutably borrowed
- No borrowing for (a ‘lifetime’) longer than scope of owner
- Automatic ‘drop’s (deallocation) at end of scope

Affine (practice)

Memory management in Rust:

- No garbage collection (GC)
- No use after ‘move’ (passing by value)
- No ‘borrowing’ (a reference) that’s already mutably borrowed
- No borrowing for (a ‘lifetime’) longer than scope of owner
- Automatic ‘drop’s (deallocation) at end of scope

Result: no dangling pointers, double-frees, uses-after-free; memory leaks avoided; all without expensive GC and at compile-time.

Idea: ‘use-after-free’ = ‘use after DELETE’

With Rust’s ‘move’ semantics, our DELETE function can consume a resource identifier

Idea: ‘use-after-free’ = ‘use after DELETE’

With Rust’s ‘move’ semantics, our DELETE function can consume a resource identifier:

```
fn get(&ResourceId) -> Response;
fn delete(ResourceId) -> Response;
```

Idea: ‘use-after-free’ = ‘use after DELETE’

With Rust’s ‘move’ semantics, our `DELETE` function can consume a resource identifier:

```
fn get(&ResourceId) -> Response;
fn delete(ResourceId) -> Response;
```

Affine type system $\implies$ a `ResourceId` moved into `delete` cannot be reused.

Idea: ‘use-after-free’ = ‘use after DELETE’

With Rust’s ‘move’ semantics, our `DELETE` function can consume a resource identifier:

```
fn get(&ResourceId) -> Response;
fn delete(ResourceId) -> Response;
```

Affine type system $\implies$ a `ResourceId` moved into `delete` cannot be reused.

With this and similar models, we ‘tell’ the compiler how to check for invalid API use.

Okay… but what's invalid use?

OpenAPI Initiative's specification (OAS) allows YAML schema for REST API description

Okay… but what's invalid use?

OpenAPI Initiative's specification (OAS) allows YAML schema for REST API description:

```yaml
/addresses:
  summary: Address book endpoint
  get:
    description: List addresses in user's address book
    responses:
      200:
        description: List of addresses
        content:
          application/json:
            schema:
              type: array
              items:
                $ref: '#/definitions/Address'
  post:
    description: Add a new address
    requestBody:
      $ref: '#/definitions/Address'
    responses:
      405:
        description: Not implemented
```

Translating types

Map REST schema types to Rust types

Translating types

Map REST schema types to Rust types:

```rust
struct Address {
    house_no: i32,
    street: String,
    postcode: String,
    country: String,
}

enum OkBody {
    Status200(Vec<Address>),
    // ...
    UnspecifiedCode(String),
    MalformedJson(String),
}

// ...

type ResponseBody = Result<OkBody, ErrBody>;
```

Translating lifetimes

Tie REST resource lifetimes to Rust variable (reference) lifetimes

Translating lifetimes

Tie REST resource lifetimes to Rust variable (reference) lifetimes:

```
let provisioned_id = ResourceId_name::from_static("...");

// or:
let response = addresses::post(&address, &auth);
let discovered_id = extract_address_id(&response);
```

Translating lifetimes

Tie REST resource lifetimes to Rust variable (reference) lifetimes:

```
let provisioned_id = ResourceId_name::from_static("...");

// or:
let response = addresses::post(&address, &auth);
let discovered_id = extract_address_id(&response);

// then:
addresses__name_::get(&discovered_id, &auth);
addresses__name_::delete(discovered_id, &auth);
                        // ^ moved!
```

Translating lifetimes

Tie REST resource lifetimes to Rust variable (reference) lifetimes:

```rust
let provisioned_id = ResourceId_name::from_static("...");

// or:
let response = addresses::post(&address, &auth);
let discovered_id = extract_address_id(&response);

// then:
addresses__name_::get(&discovered_id, &auth);
addresses__name_::delete(discovered_id, &auth);
                        // ^ moved!

// compile error - use after move:
addresses__name_::get(&discovered_id, &auth);
```

Tapioca

Code generation

```
infer_api!(name, "http://api.io/schema.yml")
```

Code generation

- `infer_api!(name, "http://api.io/schema.yml")`
- Expanded in-place at compile-time

Code generation

- `infer_api!(name, "http://api.io/schema.yml")`
- Expanded in-place at compile-time
- A ‘typed HTTP client’ generated under the module `name`

Code generation

- `infer_api!(name, "http://api.io/schema.yml")`
- Expanded in-place at compile-time
- A ‘typed HTTP client’ generated under the module `name`
- Types correspond to definitions in the provided OAS schema

User code

```
#[macro_use]
extern crate tapioca;

infer_api!(httpbin,
    "https://raw.githubusercontent.com/OJFord/ ... /httpbin.yml"
);
use httpbin::ip;

fn main() {
    let auth = httpbin::ServerAuth::new();

    match ip::get(auth) {
        Ok(response) => match response.body() {
            ip::get::OkBody::Status200(body)
                => println!("Your IP is {}", body.origin),
            _ => printn!("httpbin.org did something unexpected"),
        },
        Err(response)
            => println!("httpbin.org error: {}", response.body()),
    }
}
```

Feedback

- Interest on /r/Rust community forum

Feedback

- Interest on /r/Rust community forum
  - ‘upvotes’ as appeal metric: 98% like the concept ($N \approx 1.1\text{k}$)

Feedback

- Interest on /r/Rust community forum
  - ‘upvotes’ as appeal metric: 98% like the concept ($N \approx 1.1\text{k}$)
- ‘Pinch of salt’ ($N = 4$) survey results:

Feedback

- Interest on /r/Rust community forum
  - ‘upvotes’ as appeal metric: 98% like the concept ($N \approx 1.1\text{k}$)
- ‘Pinch of salt’ ($N = 4$) survey results:
  - 3/4 respondents find it appealing

Feedback

- Interest on /r/Rust community forum
  - 'upvotes' as appeal metric: 98% like the concept ($N \approx 1.1\text{k}$)
- 'Pinch of salt' ($N = 4$) survey results:
  - 3/4 respondents find it appealing
  - 3/4 think it would make them more productive

Feedback

- Interest on /r/Rust community forum
  - 'upvotes' as appeal metric: 98% like the concept ($N \approx 1.1\text{k}$)
- 'Pinch of salt' ($N = 4$) survey results:
  - 3/4 respondents find it appealing
  - 3/4 think it would make them more productive
  - 1/4 would actually use it

Feedback

- Interest on /r/Rust community forum
  - 'upvotes' as appeal metric: 98% like the concept ($N \approx 1.1\text{k}$)
- 'Pinch of salt' ($N = 4$) survey results:
  - 3/4 respondents find it appealing
  - 3/4 think it would make them more productive
  - 1/4 would actually use it
    - 2/4 would prefer a schema-less client
    - 1/4 would prefer an RPC framework

Feedback

- Interest on /r/Rust community forum
  - 'upvotes' as appeal metric: 98% like the concept ($N \approx 1.1\text{k}$)
- 'Pinch of salt' ($N = 4$) survey results:
  - 3/4 respondents find it appealing
  - 3/4 think it would make them more productive
  - 1/4 would actually use it
    - 2/4 would prefer a schema-less client
    - 1/4 would prefer an RPC framework

Conclusion

- Prevent some HTTP `4xx` (client) errors at compile-time

Conclusion

- Prevent some HTTP `4xx` (client) errors at compile-time
- Force handling of errors when they do occur

Conclusion

- Prevent some HTTP `4xx` (client) errors at compile-time
- Force handling of errors when they do occur
- Boost developer productivity

Thank you!