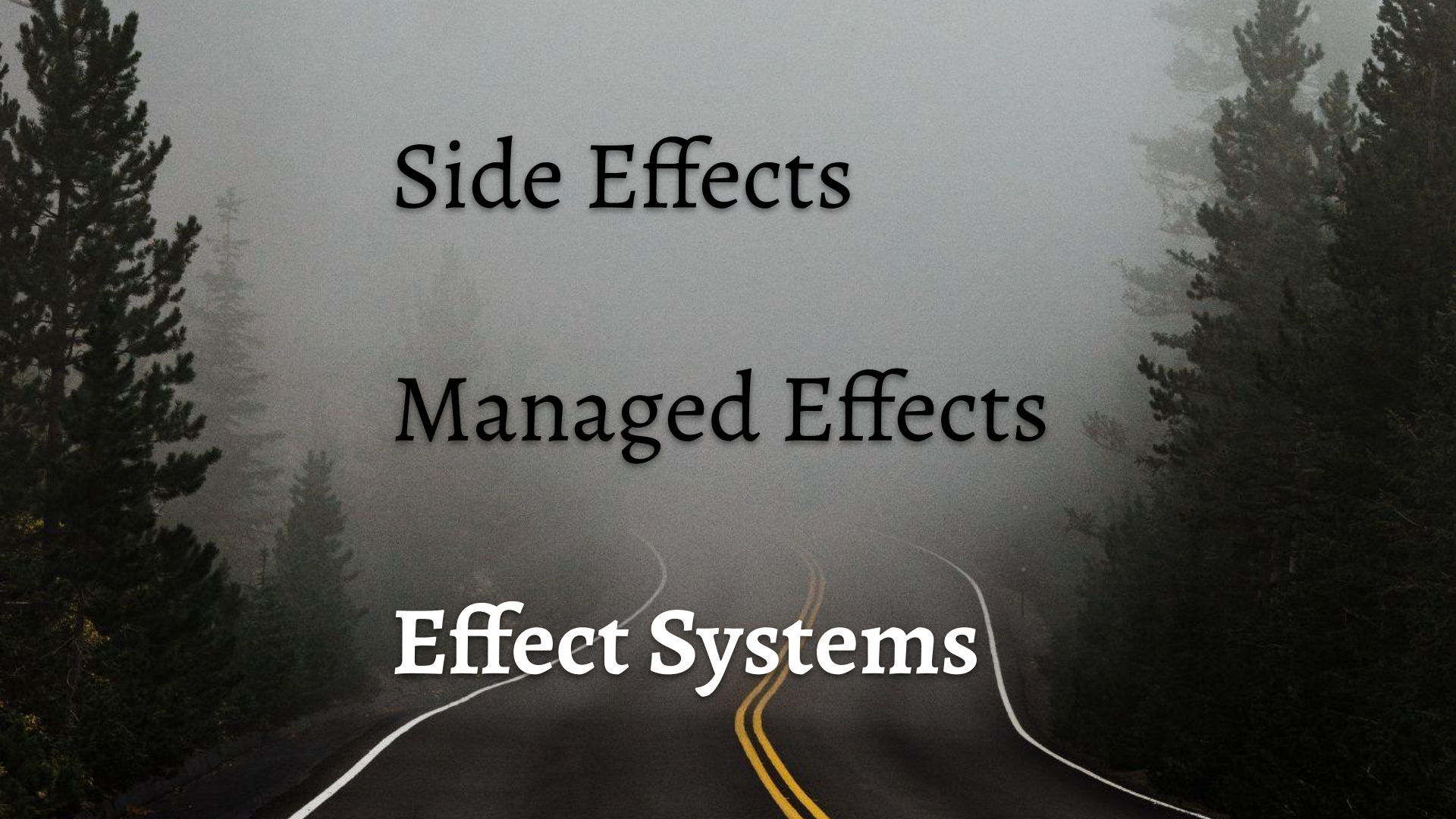
Simple Functional Effects with Tag Unions

@rtfeldman


Side Effects
Managed Effects
Effect Systems

Outline

Motivation

Tag Unions

The System

Comparisons


1. Motivation
Testing
Handling Errors
Logging

Package Downloader

`https://example.com/a9fdb2.tar.gz`

1. Download compressed tarball from URL
2. Verify contents against hash in URL
3. Decompress into a local directory

```
fn download_tarball(url: String)
```

```
fn download_tarball(url: String)
    -> Result<Hash, io::Error>
```

```
fn download_tarball(url: String)
    -> Result<Hash, io::Error> {
    let resp = https::get(url);



    return hash(resp);
}
```

```
fn download_tarball(url: String)
    -> Result<Hash, io::Error> {
    let resp = https::get(url);
    let enc = get_encoding(…);

    return hash(resp);
}
```

```
fn download_tarball(url: String)
    -> Result<Hash, io::Error> {
    let resp = https::get(url);
    let enc = get_encoding(…);
    extract_tarball(enc, resp);
    return hash(resp);
}
```

```
fn download_tarball(url: String)
    -> Result<Hash, io::Error> {
    let resp = https::get(url)?;
    let enc = get_encoding(…);
    extract_tarball(enc, resp);
    return hash(resp);
}
```

if Err,
return Err

```
fn download_tarball(url: String)
    -> Result<Hash, io::Error> {
    let resp = https::get(url)?;
    let enc = get_encoding(…);
    extract_tarball(enc, resp)?;
    return hash(resp);
}
```

```
fn download_tarball(url: String)
    -> Result<Hash, io::Error> {
    let resp = https::get(url)?;
    let enc = get_encoding(…);
    extract_tarball(enc, resp)?;
    return hash(resp);
}
```

```
fn download_tarball(url: String)
    -> Result<Hash, io::Error> {
    let resp = https::get(url)?;
    let enc = get_encoding(…)?;
    extract_tarball(enc, resp)?;
    return hash(resp);
}
```

doesn't return io::Error

```
enum ContentEncoding {
    Gzip, Brotli, Uncompressed
}

enum Problem {
    Io(io::Error),
    Enc(EncError),
}
```

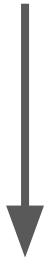
```
https::get(url)?;  Result<Response, io::Error>
get_encoding(…)?;  Result<Encoding, EncError>


enum Problem {
    Io(io::Error),
    Enc(EncError),
}
```

```
https::get(url).map_err(Io)?;    Problem
get_encoding(…).map_err(Enc)?;   Problem


enum Problem {
    Io(io::Error),
    Enc(EncError),
}
```

```
fn download_tarball(url: String)
    -> Result<Hash, io::Error>
```



```
fn download_tarball(url: String)
    -> Result<Hash, Problem>
```

What I like about this

Errors are visible in the type

I can't accidentally forget to handle errors

`map_err` lets me tag errors with my own info

the `?` operator lets me short-circuit easily

What I dislike about this

Use a second error type? `.map_err` everywhere!

Promotes overbroad errors (`AddrInUse` for files!)

Easy to miss early returns from the `?` operator

Testing

```
fn get_encoding(url: …, header: …)
    -> ContentEncoding {
    // if header missing, look at URL
}
```

pure function - call it and check return value!

Testing

```
fn download_tarball(url: String)
    -> Result<Hash, io::Error> {
    // do lots of side effects
}
```

side-effecting function - calling runs effects

Logging in Web Servers

```
fn download_tarball(url: String) … {
    let response = https::get(url);
    extract_tarball(response);
    return hash(response);
}
```

What I'd really like

Different **errors accumulate automatically**

Testing is as easy as **testing pure functions**

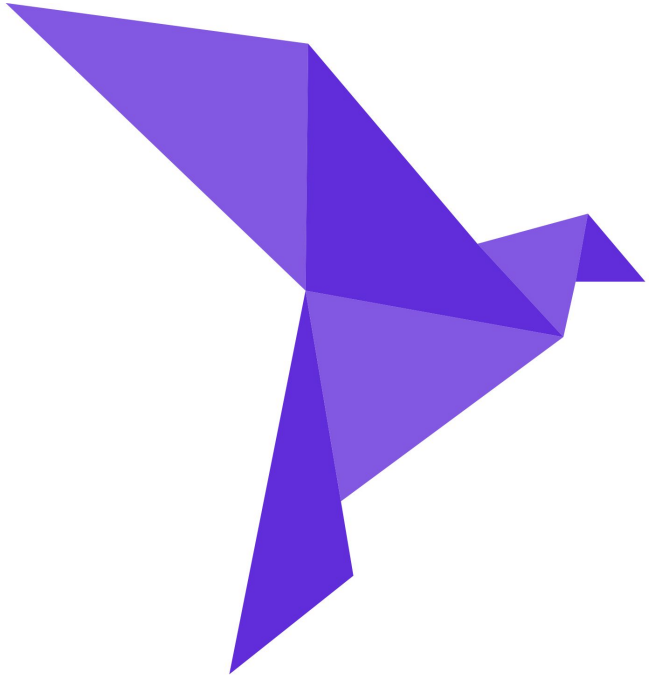
Automatic, **centralized logging** of all I/O

2. Tag Unions

Anonymous Sum Types

Accumulating Tags

Pattern Matching



roc-lang.org

work in progress!

purely functional language
compiles to **machine code**
(or to **web assembly**)

type system includes
Tag Unions

Tags

```
color = Green

color = Red

color = Gold
```

Tags

```
color : [Red, Green, Gold]
color = Green

color : [Red, Green, Gold]
color = Red

color : [Red, Green, Gold]
color = Gold
```

Tags

```
color : [Red, Green, Gold]
color = Green

color : [Red, Green, Gold]
color = Red

color : [Red, Green, Gold]
color = Gold

color : [Red, Green, Gold]
color = Blue
```

Tag Unions

```
color : [Green, Gold]
color =
    if x > 0 then
        Green
    else
        Gold
```

Tag Unions

```
toStr : [Red, Green, Gold] -> Str
toStr = \color ->
    when color is
        Red -> "red"
        Green -> "green"
        Gold -> "gold"
```

Tag Unions

```
toStr : [Red, Green, Gold] -> Str
toStr = \color ->
    when color is
        Red -> "red"
        Green -> "green"
```

Tag Unions

```
── UNSAFE PATTERN ──────────────────────────────────────────── tags.roc ─

This when does not cover all the possibilities:

11 >      when color is
12 >          Red -> "red"
13 >          Green -> "green"

Other possibilities include:

    Gold

I would have to crash if I saw one of those! Add branches for them!

─────────────────────────────────────────────────────────────────────────
```

Tag Unions

```
toStr : [Red, Green, Other Str] -> Str
toStr = \color ->
    when color is
        Red -> "red"
        Green -> "green"
        Other str -> "Other: \(str)"


toStr (Other "purple")
```

Tag Unions, Summarized

Anonymous **sum types**

Tags can have **payloads**

Exhaustive **pattern matching**

Tags **accumulate** across conditional branches

3. The System

I/O Example

Error Handling

Internal Representation

```
Http.getBytes : Url -> Task Bytes HttpErr

File.exists : Path -> Task Bool MetadataErr

File.writeBytes : Path, Bytes -> Task {} WriteErr
```

```
download = \filename, url ->
    exists <- File.exists filename |> Task.await

    if exists then
        Task.succeed {}
    else
        tarball <- Http.getBytes url |> Task.await
        File.writeBytes filename tarball
```

short-circuits on error,
like Rust's ? operator

```
result <- download filename url |> Task.attempt

when result is
    HttpErr url problem -> …
    FileWriteErr path problem -> …
    FileMetadataErr path problem -> …
```

exhaustiveness checking

```
tarball <- Http.getBytes url |> Task.await
```

```
when result is
    HttpErr url problem -> …
    FileWriteErr path problem -> …
    FileMetadataErr path problem -> …
```

```
tarball <-
    Http.getBytes url
    |> Task.await
```

```
when result is
    HttpErr url problem -> …
    FileWriteErr path problem -> …
    FileMetadataErr path problem -> …
```

```
tarball <-
    Http.getBytes url
    |> Task.mapErr DownloadTarball
    |> Task.await
```

---

io::Error

```
when result is
    DownloadTarball (HttpErr url problem) -> …
    FileWriteErr path problem -> …
    FileMetadataErr path problem -> …
```

Quick Shout-Out!

William Brandon

twitter.com/exists_forall

```
Http.getBytes : Url -> Task Bytes HttpErr

File.exists : Path -> Task Bool MetadataErr

File.writeBytes : Path, Bytes -> Task {} WriteErr
```

```
Operation : [



]
```

```
Operation : [
    # Http.getBytes : Url -> Task Bytes HttpErr
    HttpGetBytes Url
        ([Ok Bytes, Err Http.Err] -> Operation),

    # File.exists : Path -> Task Bool MetadataErr
    FileExists Path
        ([Ok Bool, Err File.MetaErr] -> Operation),
    …
]
```

```
Operation : [
    # Http.getBytes : Url -> Task Bytes HttpErr
    HttpGetBytes Url
        ([Ok Bytes, Err Http.Err] -> Operation),

    # File.exists : Path -> Task Bool MetadataErr
    FileExists Path
        ([Ok Bool, Err File.MetaErr] -> Operation),
    …
]
```

```
when operation is
    HttpGetBytes url getNextOperation ->

    FileExists path getNextOperation ->

    FileWriteBytes path bytes getNextOperation ->
```

Simulatable!

Loggable!

```
Task ok err : ([Ok ok, Err err] -> Op) -> Op

succeed : ok -> Task ok *
succeed = \ok -> \continue -> continue (Ok ok)

await : Task a err, (a -> Task b err) -> Task b err
```

Still Simulatable/Loggable!

```
Http.getBytes : [
    Task Bytes [HttpErr Http.Err]

File.exists : Path ->
    Task Bool [MetaErr File.MetaErr]

File.writeBytes : Path, Bytes ->
    Task {} [WriteErr File.WriteErr]
```

```
Http.getBytes : [
    Task Bytes [HttpErr Http.Err] [Network]

File.exists : Path ->
    Task Bool [MetaErr File.MetaErr] [FileRead]

File.writeBytes : Path, Bytes ->
    Task {} [WriteErr File.WriteErr] [FileWrite]
```

Runtime Representation

Tag Unions & functions

```
Operation : [
    HttpGetBytes Url (… -> Operation),
    FileExists Path (… -> Operation),
    FileWriteBytes Path Bytes (… -> Operation),
]

Task ok err : (… -> Operation) -> Operation
```

Runtime Representation

Tag Unions & functions

Roc tag unions are C “tagged unions”

No heap allocations by default

Roc closures are implemented as tag unions

`Operation` is like Rust’s async state machine

4. Comparisons

Capabilities

Ergonomics

Performance

There are a **lot** of effect systems out there!

Stdlib systems

Third-party systems

Algebraic Effects

Capabilities

- ✅ Simulation Testing
- ✅ Errors accumulate automatically
- ✅ Can't forget to handle errors
- ✅ Can track which effects a `Task` may perform
- ✅ Can use `mapErr` to tag custom error types

Non-Capabilities

❌ Composing `Task` with non-`Task` effects

❌ Calling effectful functions with same syntax

❌ “Colorless” effectful functions (no type change to do effects)

Ergonomics

Very simple, gentle learning curve

Similar verbosity to asnyc/await

Error accumulation Just Works

Performance: Tag Unions

Same as any other sum types (enums/ADTs/etc.)

In Roc's case, same performance as Rust enums

(Could be done with union types too, e.g. in TS)

Performance: State Machine

Depends on how the language represents closures

Roc's are not heap-allocated (very unusual!)

`Task` wrapper performance depends on inlining

Performance: Effects

Effects in Roc can be written in systems languages

Languages with C FFI could do something similar

Wrappers around stdlib I/O also a fine option

SUMMARY


Motivation
Testing
Handling Errors
Logging

Tag Unions

Anonymous **sum types**

Tags can have **payloads**

Exhaustive **pattern matching**

Tags **accumulate** across conditional branches

```
download = \filename, url ->
    exists <- File.exists filename |> Task.await

    if exists then
        Task.succeed {}
    else
        tarball <- Http.getBytes url |> Task.await
        File.writeBytes filename tarball
```

Tag Unions

```
── UNSAFE PATTERN ──────────────────────────────────────────── tags.roc ─

This when does not cover all the possibilities:

11 >      when color is
12 >          Red -> "red"
13 >          Green -> "green"

Other possibilities include:

    Gold

I would have to crash if I saw one of those! Add branches for them!

─────────────────────────────────────────────────────────────────────────
```

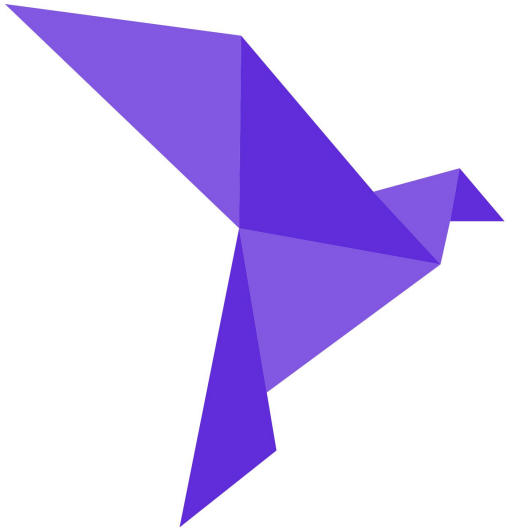
```
when operation is
    HttpGetBytes url getNextOperation ->

    FileExists path getNextOperation ->

    FileWriteBytes path bytes getNextOperation ->
```

Simulatable! 🚀 Loggable!

Simple Functional Effects with Tag Unions



roc-lang.org

I host a podcast!

software-unscripted.com