Practical Introduction to Substructural Type Systems through Linear Haskell and Rust

`github.com/ryanorendorff/lc-2018-linear-types`

Ryan Orendorff, Daniel Hensley

June 5th, 2018

Outline

Motivation: Properly handling a resource

Substructural Type Systems

Overview of Linear Types through Linear Haskell

Motivation part 2: A (hopefully) better way

Rust

Examples: Substructural Types and Medical Imaging

What can substructural types do for us?

In this talk we will be using substructurally typed languages, which can be used for

1. guaranteed use of a value (ex: guaranteeding a list is only permuted),
2. safe channel communication, and
3. safe handling of a resource.

Opening a file is pretty easy in any programming language

Let's say we have the following API for accessing a resource (files).

$$\textbf{data}\ File$$
$$\textbf{data}\ FilePath$$

We can open and close files,

$$openF :: FilePath \to IO\ File$$
$$closeF :: File \to IO\ ()$$

we can append to files,

$$appendF :: File \to Text \to IO\ ()$$

and we can get the current time as a string.

$$now :: IO\ Text$$

Simple IO program for accessing files

Let's write a simple program for adding the current date to the end of the file.

$$
\begin{aligned}
&appendTimeToFile :: FilePath \to IO\ () \\
&appendTimeToFile\ path = \mathbf{do} \\
&\quad f \leftarrow openF\ path \\
&\quad n \leftarrow now \\
&\quad appendF\ f\ n \\
&\quad closeF\ f
\end{aligned}
$$

What if we close the file on accident

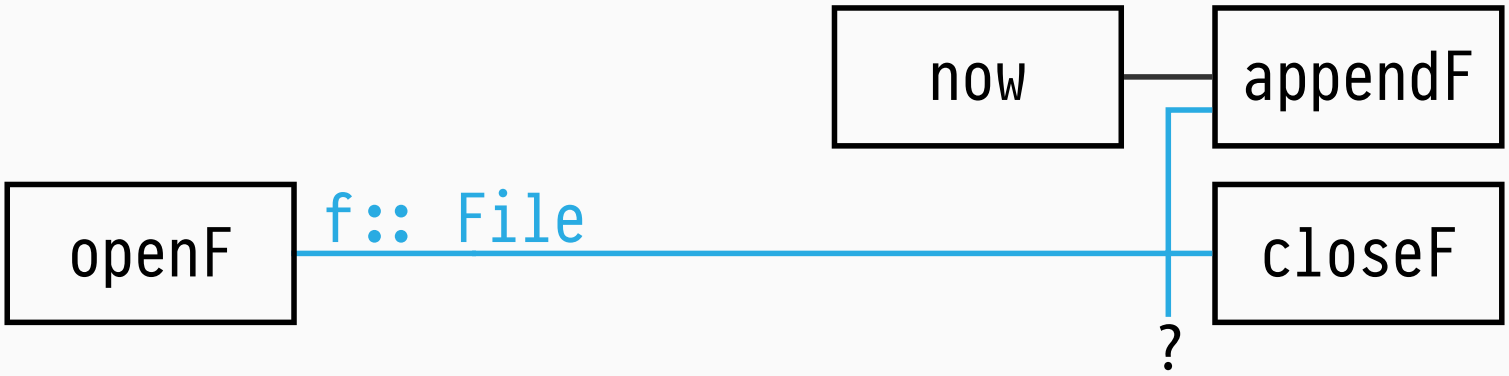
What if we made a mistake and closed the file. Does the result still typecheck?

$$
\begin{aligned}
&appendTimeToFile' :: FilePath \to IO\ () \\
&appendTimeToFile'\ path = \mathbf{do} \\
&\quad f \leftarrow openF\ path \\
&\quad n \leftarrow now \\
&\quad closeF\ f \\
&\quad appendF\ f\ n \quad \text{-- Oops, we closed the file}
\end{aligned}
$$

The developer is responsible for a property that the compiler does not check for (but ideally it should!).

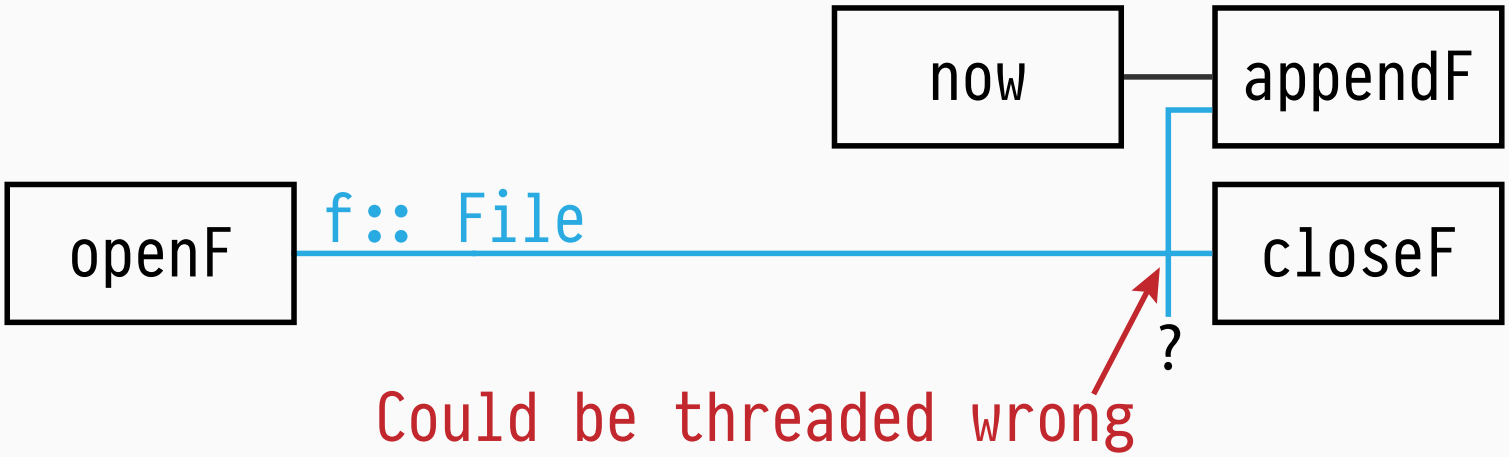
Block diagram of the api demonstrates the issue

If we look at the API interconnectivity, we can see the challenge.



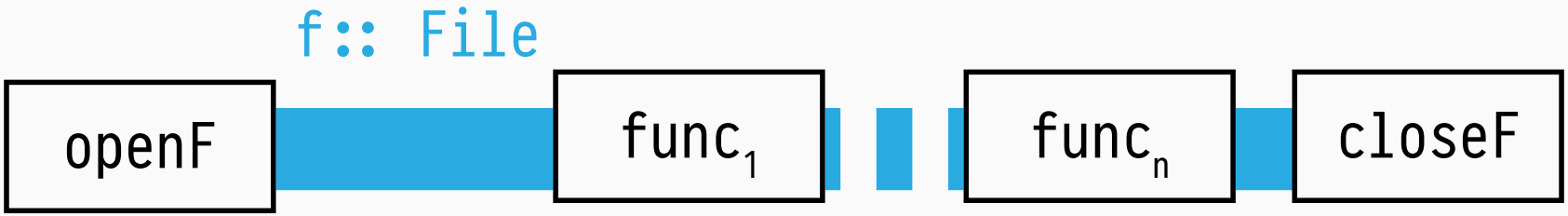
Block diagram of the api demonstrates the issue

If we look at the API interconnectivity, we can see the challenge.



Second attempt: compiler enforced resource closing

If we instead had an API where the file handle *must* be closed, then we could assure that any open resource must be closed.



Threading the resource

Linear types and Linear Haskell can help us guarantee a value is used once.

$$
\begin{aligned}
&appendTimeToFile' :: FilePath \to IO\ () \\
&appendTimeToFile'\ path = \mathbf{do} \\
&\quad f \leftarrow openF\ path \\
&\quad n \leftarrow now \\
&\quad f_1 \leftarrow appendF\ f\ n \\
&\quad closeF\ f_1
\end{aligned}
$$

To get to this compiler-checked form, let's look a bit at how linear and substructural types work.

Table of Contents

Motivation: Properly handling a resource

Substructural Type Systems

Overview of Linear Types through Linear Haskell

Motivation part 2: A (hopefully) better way

Rust

Examples: Substructural Types and Medical Imaging

Unrestricted type systems have three structural rules

In most type systems, three structural properties that allow unrestricted use of a variable; unrestricted meaning variables can be dropped, duplicated, and reordered.[1]

The substructural rules are

- Exchange,
- Contraction, and
- Weakening.

[1] Walker 2005.

Exchange: we can use type proofs in any order

The exchange rule allows us to type check in any desired order when multiple terms have to be checked at the same time.

For example, when type checking the following function

$$exchange = \textbf{let}\ x = (4 :: Int)\ \textbf{in let}\ y = (5 :: Int)\ \textbf{in}\ (x, y)$$

We can check that x or y are Ints in either order.

Restricting this property means that we have to type check terms *in a stack order* (a FILO order).

Contraction: we can make duplicates of type proofs

The contraction rule lets us use a variable twice in a type safe way.

For example, when type checking the following function

$$dup :: a \to (a, a)$$
$$dup\ x = (x, x)$$

We can use the proof that $x :: a$ twice.

Restricting this property means we *can't use a term more than once*.

Weakening: we can discard unused type proofs

The weakening rule means we can discard unnecessary type proofs.

For example, when type checking the following function

$$kill :: a \to ()$$
$$kill\ x = ()$$

We do not need to use the fact that $x :: a$ while type checking the right hand side.

Restricting this property means we *must use a term at least once*.

Substructural type systems remove 1 or more structural rule

Substructural type systems remove or replace one or more of the structural rules.

Let's look at the most useful substructural systems.

Unrestricted type systems are most common

You can use a variable as many times as you like, including zero.[2]

This is the type system for Haskell, etc.

Unrestricted
Exchange, Weakening, Contraction

[2] Walker 2005.

Relevant type systems: every variable used at least once

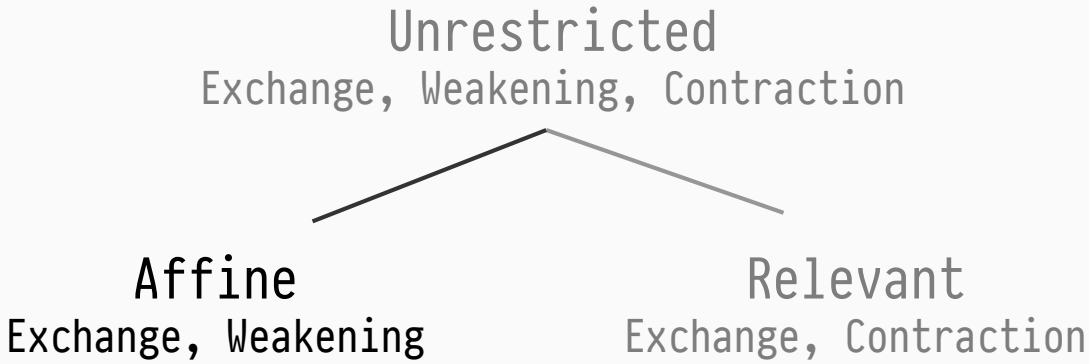
In relevant typing systems, a variable *must be used*.

Drops the weakening rule.

Affine type systems: every variable used at most once

A variable can be used zero or one times. Drops the contraction rule.

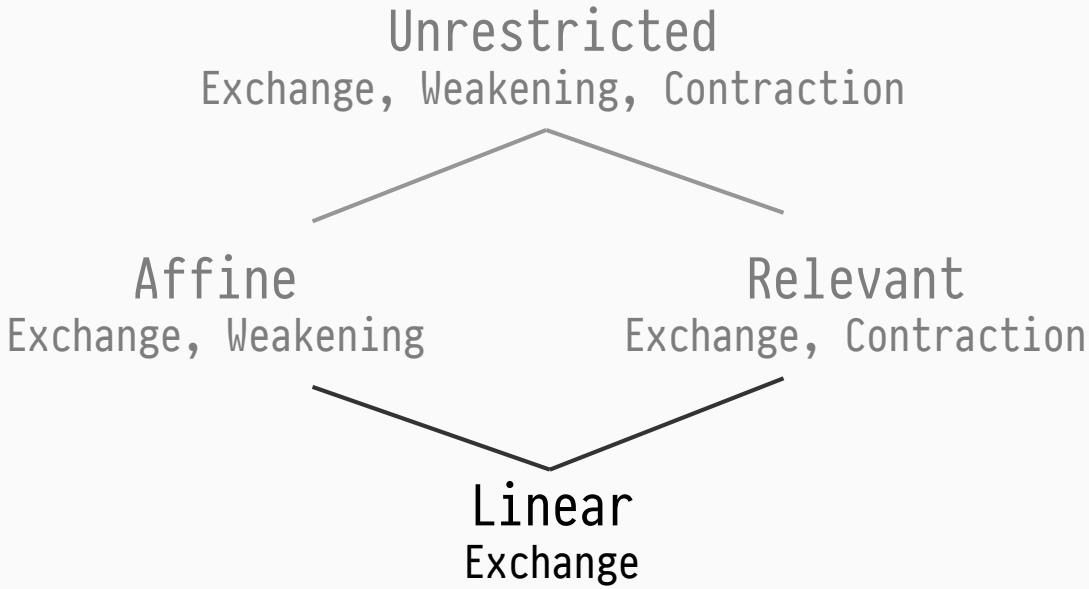
Example languages include Alms and Rust.



Linear type systems: every variable used *exactly* once

A variable must be used *exactly* once. Drops both the weakening and contraction rules.

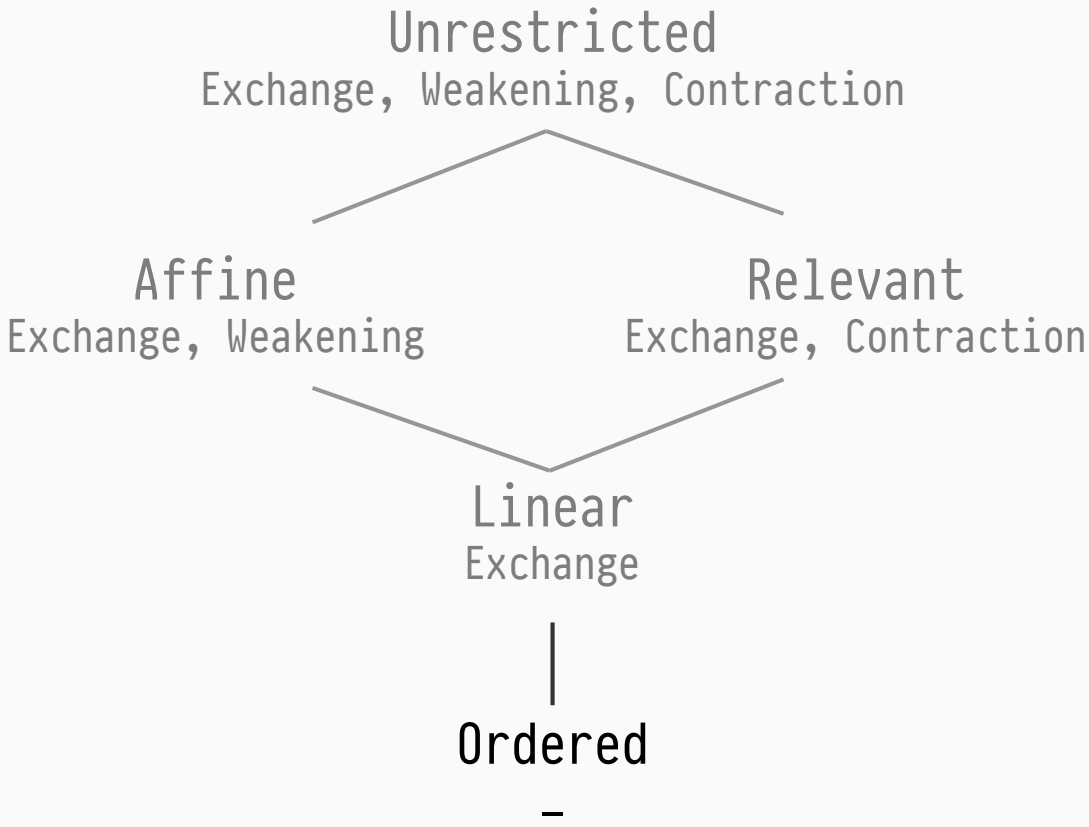
This type system is implemented in the Linear Haskell extension.



Ordered type systems: every variable must be used in order

All variables must be used and must be used in FILO order.

Ordered type systems have none of the structural rules.



Substructural type system relation diagram

This leads to a convenient diagram describing how the substructural systems relate to each other.

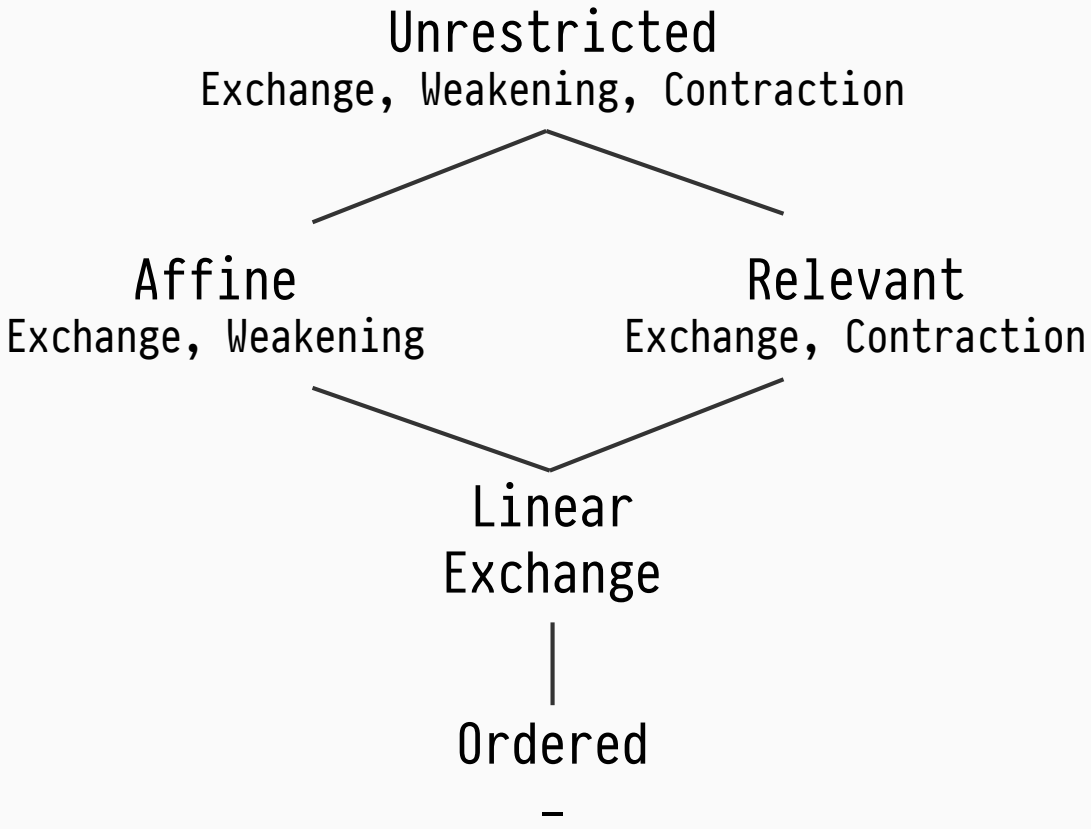


Table of Contents

Motivation: Properly handling a resource

Substructural Type Systems

Overview of Linear Types through Linear Haskell

Motivation part 2: A (hopefully) better way

Rust

Examples: Substructural Types and Medical Imaging

The sum function can be written linearly

Say we want to take the sum of a list. How do we check that we consumed every element exactly once?[3]

$$
\begin{aligned}
sum_L &:: [Int] \multimap Int \\
sum_L\ [\,] &= 0 \\
sum_L\ (x : xs) &= x +_L sum_L\ xs
\end{aligned}
$$

[3] Bernardy et al. 2017.
"$\multimap$" is represented as "`->.`" in source code

When a linear sum uses an element twice

What if we accidentally use an element twice?

$$
\begin{aligned}
&sum_L :: [Int] \multimap Int \\
&sum_L\ [\,] &&= 0 \\
&sum_L\ (x : xs) &&= x +_L sum_L\ xs +_L x
\end{aligned}
$$

The type checker will helpfully tell us we violated linearity.

```
LinearTalk.lhs:447:10: error:
```

- `Couldn't match expected weight '1' of variable 'x' with actual weight '`ω`'`

When a linear sum forgets an element?

What if we forgot to use the elements of our list?

$$
\begin{aligned}
sum_L &:: [Int] \multimap Int \\
sum_L\ [\,] &= 0 \\
sum_L\ (x : xs) &= 1 +_L sum_L\ xs
\end{aligned}
$$

The type checker will helpfully tell us we violated linearity.

```
LinearTalk.lhs:447:10: error:
```

- `Couldn't match expected weight '1' of variable 'x' with actual weight '0'`

What does it mean to "consume" a variable

To consume a variable exactly once, we use the following rules

- An atomic base type (Int, $Bool$, etc): evaluate the value once.
- A function: Pass in one argument and consume the result exactly once.
  - This is a bit tricky. It means, for $(f :: A \multimap B)$, if $f\,x$ is consumed exactly once, *then* x is consumed exactly once.
- For any algebraic data type, pattern match and consume all components exactly once.
  - For a pair, this means pattern match and consume each component exactly once.

Gotchas with the function arrow

The linear arrow says how the function *uses its argument*; it does not restrict what is passed to the function.

$$
\begin{aligned}
&f :: s \multimap t \\
&g :: s \to (t, s) \\
&g\ x = (f\ x, x)
\end{aligned}
$$

g does not specify how its argument is used; it could be aliased.

All datatypes are linear by default

In Linear Haskell, all data constructors use linear arrows by default.

$$\begin{aligned}&\mathbf{data}\ (a, b)\ \mathbf{where}\\ &\quad (\cdot, \cdot) :: a \multimap b \multimap (a, b)\end{aligned}$$

So we can define the normal fst function and have it work as expected.

$$\begin{aligned}&fst :: (a, b) \to a\\ &fst\ (a, b) = a\end{aligned}$$

But a linear fst is not possible, as it requires a linear use of b.

$$\begin{aligned}&fst_{\text{☹}} :: (a, b) \multimap a\\ &fst_{\text{☹}}\ (a, b) = a \quad \text{-- 'b' has weight 0 instead of 1}\end{aligned}$$

Linear datatypes work in a non-linear context; we do not need special data constructors for linear versus nonlinear data.

Unrestricted data

You can define a term to have unlimited use by using the following data type.

$$\begin{aligned}&\mathbf{data}\ \mathit{Unrestricted}\ a\ \mathbf{where}\\&\quad \mathit{Unrestricted} :: a \to \mathit{Unrestricted}\ a\\&\quad \text{--} \qquad\qquad\qquad \uparrow \text{note the normal arrow!}\end{aligned}$$

This allows you to define functions like so

$$\begin{aligned}&\mathit{snd}_L :: (\mathit{Unrestricted}\ a, b) \multimap b\\&\mathit{snd}_L\ ((\mathit{Unrestricted}\ a), b) = b\end{aligned}$$

Table of Contents

Motivation: Properly handling a resource

Substructural Type Systems

Overview of Linear Types through Linear Haskell

Motivation part 2: A (hopefully) better way

Rust

Examples: Substructural Types and Medical Imaging

Linear File api

Let's say we have the following API for accessing a resource (files).

$$\mathbf{data}\ File$$
$$\mathbf{data}\ FilePath$$

We can open and close files,

$$openF_L :: FilePath \to IO_L\ File$$
$$closeF_L :: File \multimap IO_L\ (Unrestricted\ ())$$

we can append to files,

$$appendF_L :: File \multimap Text \to IO_L\ File$$

And we can get the current time as a string.

$$now :: IO\ Text$$

We will also need a Linear IO Monad

To glue this all together, we need a linear IO monad.

$return$ is nearly identical, but uses a linear arrow $\multimap$.

$$return_L :: a \multimap IO_L\ a$$

And similarly $bind$ is defined using linear arrows.

$$\gg\!=_L :: IO_L\ a \multimap (a \multimap IO_L\ b) \multimap IO_L\ b$$

The linear bind forces us to use the *value linearly*, instead of relying on the linear arrow for linearity.

Appending time to a file part 2

We can now take a crack at our file example again.

$$
\begin{aligned}
&appendTimeToFile_L :: FilePath \to IO\ () \\
&appendTimeToFile_L\ path = now \gg\!= (\lambda n \to run_L\ \$\ \mathbf{do}_L \\
&\quad f \leftarrow openF_L\ path \\
&\quad f_1 \leftarrow appendF_L\ f\ n \\
&\quad closeF_L\ f_1)
\end{aligned}
$$

Appending time to a file part 2

If we forget to close the file, the compiler tells us about this error.

$$
\begin{aligned}
&appendTimeToFile_{\text{☹}} :: FilePath \to IO\ () \\
&appendTimeToFile_{\text{☹}}\ path = now \gg\!= (\lambda n \to run_L\ \$\ \mathbf{do}_L \\
&\quad f \leftarrow openF_L\ path \\
&\quad f_1 \leftarrow appendF_L\ f\ n \\
&\quad return_L\ (Unrestricted\ ()))
\end{aligned}
$$

```
LinearTalk.lhs:846:7: error:
```

- `Couldn't match expected weight '1' of variable '`f_1`' with actual weight '0'`

Table of Contents

Motivation: Properly handling a resource

Substructural Type Systems

Overview of Linear Types through Linear Haskell

Motivation part 2: A (hopefully) better way

Rust

Examples: Substructural Types and Medical Imaging

Rust

Rust is a “mainstream” language that ships with a substructural type system.[4]

[4] https://www.rust-lang.org

Using a File in Rust

```
fn append_time(p: &Path, n: String) -> io::Result<()>
{
    let mut f = File::open(p)?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    s.push_str(&n);
    drop(f); // close the file
    Ok(())
}
```

What if We Forgot to Close it?

```
fn append_time(p: &Path, n: String) -> io::Result<()>
{
    let mut f = File::open(p)?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    s.push_str(&n);
    // <- no `drop`
    Ok(())
}
```

‘drop‘ in Rust Ownership Type System

The Rust compiler establishes and tracks **ownership**.[5,6]

- The compiler automatically inserts calls to `drop` when an owned type with `move` semantics goes out of scope.
- This provides automatic memory safety without GC.
- It also means that you cannot forget to “finalize” these resources (*e.g*, files, sockets, data locks, etc.).

[5] Clarke, Potter, and Noble 1998.
[6] Reed 2015.

Cannot Forget! The Compiler Inserts ‘drop‘

```
fn append_time(p: &Path, n: String) -> io::Result<()>
{
    let mut f = File::open(p)?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    s.push_str(&n);
    Ok(())
} // compiler `drop`s all resources of scope here
```

What if We Close or ‘move‘ the File Early?

```
fn append_time(p: &Path, n: String) -> io::Result<()>
{
    let mut f = File::open(p)?;
    let mut s = String::new();
    drop(f); // early drop
    f.read_to_string(&mut s)?; // invalid second use
    s.push_str(&n);
    Ok(())
}
```

Here we accidentally close the file too early.

```
error[E0382]: use of moved value: `f`
  --> src/main.rs:11:5
   |
10 |     drop(f);
   |          - value moved here
11 |     f.read_to_string(&mut s);
   |     ^ value used here after move
   |
   = note: move occurs because `f` has type `std::fs::File`, which does not implement the `Copy` trait
```

What if We Close or ‘move‘ the File Early?

```
fn append_time(p: &Path, n: String) -> io::Result<()>
{
    let mut f = File::open(p)?;
    let mut s = String::new();
    send_file_to_other_function(f); // whoops!
    f.read_to_string(&mut s)?; // compiler error here
    s.push_str(&n);
    Ok(())
}
```

Here we **move** the file early.

```
error[E0382]: use of moved value: `f`
  --> src/main.rs:13:5
   |
12 |     send_file_to_other_function(f);
   |                                 - value moved here
13 |     f.read_to_string(&mut s);
   |     ^ value used here after move
   |
   = note: move occurs because `f` has type `std::fs::File`, which does not implement the `Copy` trait
```

Linearly Threading (‘move‘) the File

```
fn process(f: File) -> File { /* do stuff */ f }

fn append_time(p: &Path, n: String) -> io::Result<()>
{
    let f = File::open(p)?;
    let mut s = String::new();
    let mut f_moved_back = process(f); // `f` moved back
    f_moved_back.read_to_string(&mut s)?;
    s.push_str(&n);
    Ok(())
}
```

Rust and Substructural Types

Like Linear Haskell, the Rust type system supports both restricted types (affine[7]) and unrestricted types.

- Linear Haskell provides flexible opt-in linearity *on the function arrow*.
- Rust's system is a pervasive *ownership* type system that also includes *borrow* types.[8,9]

[7] linear types partially supported with compiler-inserted `drop`s

[8] Clarke, Potter, and Noble 1998.

[9] Reed 2015.

Factors Influencing Rust's Implementation

To understand Rust's substructural type system implementation, it is helpful to understand some of the goals of the Rust language.

- Enable low-level systems programming.
- Automatic memory management without GC.
- Statically verified memory and thread safety ("fearless concurrency").

Flavor of Rust's Substructural Types

For Rust owned types, *unrestricted* types obey **copy semantics** and the *affine* (all non-Copy) types obey **move semantics**.[10]

- Move types are `moved` on use and associated with heap-allocated data or data you want to be affine.
- Copy types are bit-copied on use and associated with primitive and stack-allocated data.
- Custom types are `move` by default; you must opt in to unrestricted types.

[10] Jung et al. 2017.

Affine types with ‘move‘ Semantics

```rust
// Custom types are affine by default.
struct AffineInt(i32); // Rust `newtype` idiom.
fn take<T>(n: T) { }

let n = AffineInt(1);
take(n); // `n` is `move`d; ownership is transferred.
println!("Number: {}", n.0);
```

```
error[E0382]: use of moved value: `n.0`
 --> src/main.rs:7:28
  |
6 |     take(n);
  |          - value moved here
7 |     println!("Number: {}", n.0);
  |                            ^^^ value used here after move
  |
  = note: move occurs because `n` has type `AffineInt`, which does not implement the `Copy` trait
```

Unrestricted types with ‘copy‘ Semantics

```rust
// We can opt in to unrestricted types by implementing
// the Copy trait (here, automatically derived).
#[derive(Copy, Clone)]
struct UnrestrictedInt(i32);

let n = UnrestrictedInt(1);
take(n); // data is bit-copied and sent to `take`.
println!("Number: {}", n.0);
// OK! `n` was not `move`d.
```

Rust Owned Types

Unrestricted Types

Owned types with `copy` semantics

Arc/Rc

Affine Types

Owned types with `move` semantics

References in Rust – Borrow Types

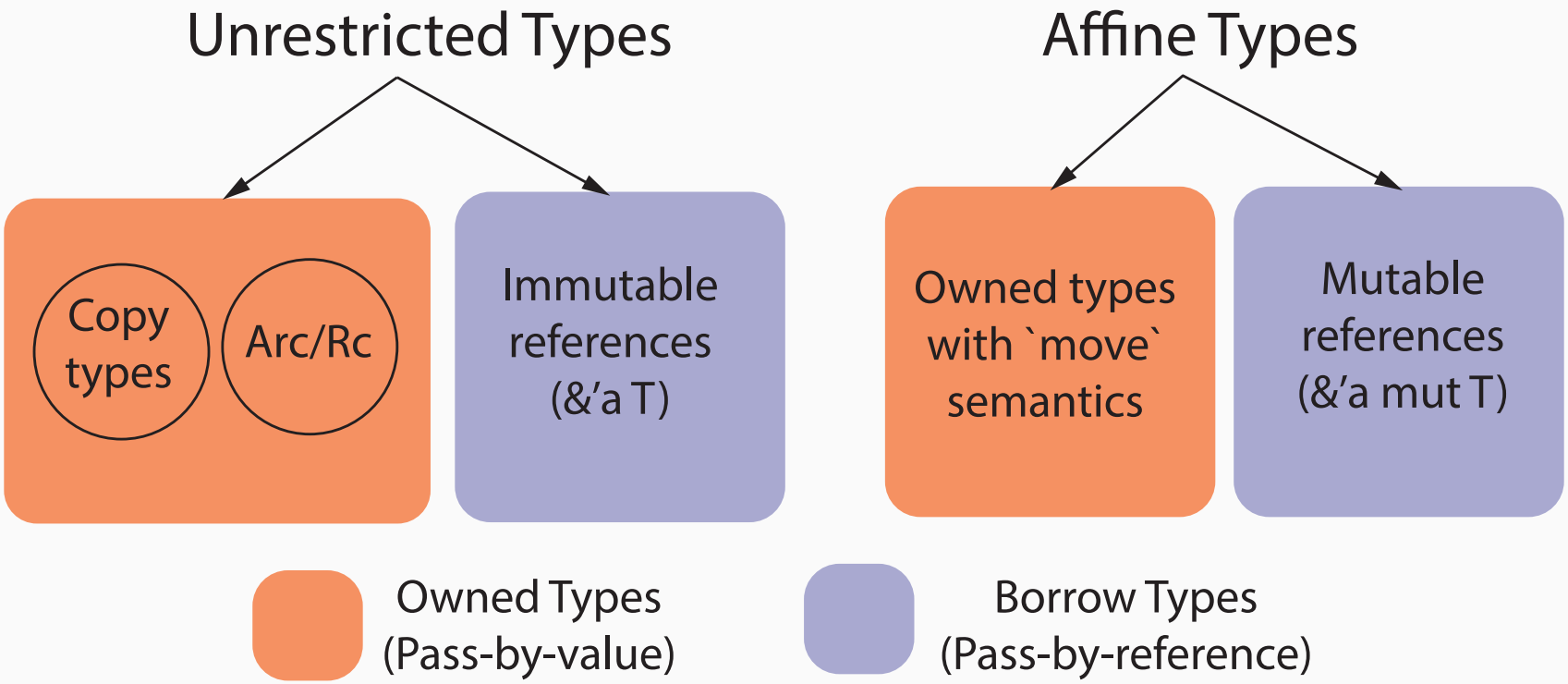
As a low-level systems language, Rust provides pass-by-reference types termed **borrow** types.[11,12]

```
&'a T // Immutable borrow to T with lifetime `a`.
&'a mut T // Mutable borrow to T with lifetime `a`.
```

- Borrow types are part of the ownership system and must enforce the same invariants.
- Aliasing and mutability mutually excluded (static guarantee).
- Borrow types also have unrestricted and affine variants.

[11] Jung et al. 2017.
[12] Fluet, Morrisett, and Ahmed 2006.

Rust Substructural Type Diagram: The Whole Story



Borrow types mediated by `borrowck` and lifetime ascription

Table of Contents

Motivation: Properly handling a resource

Substructural Type Systems

Overview of Linear Types through Linear Haskell

Motivation part 2: A (hopefully) better way

Rust

Examples: Substructural Types and Medical Imaging

Medical Imaging Software

Compile-time guarantees are very attractive for software that controls medical imaging scanners:

- Software controls complex and dangerous hardware.
- Critical communication layers.
- Monitor and respond to hardware sensors.
- We must ensure patient safety and hardware integrity.

Examples of how We Might Leverage Linear Types

We believe many of the safety-critical aspects of our system can benefit from substructural typing constructs (as well as more FP).

We'll briefly discuss how linear types are attractive for these sensitive tasks.

- **session types** (require linear threading).
- Encoding our scanner as a linear resource.

Session Types

Session types extend the notion of types from describing *data* to describing *protocols*.[13,14]

- A session type formalizes sequencing and order of a protocol in the type system.
- Often discussed in the context of channel-based communication.
- Convert developer responsibility to compiler responsibility.
- Little or no runtime cost.

[13] Pucella and Tov 2008.

[14] Jespersen, Munksgaard, and Larsen 2015.

Session Types

You can implement session types with:[15]

- types and type classes/traits (duality, sequencing)
- linear threading (linear types, indexed monads).

Haskell Session Types with (Almost) No Class

Riccardo Pucella Jesse A. Tov
Northeastern University
{riccardo,tov}@ccs.neu.edu

A major barrier to implementing session types in existing languages is aliasing.

• • •

A common way to deal with this aliasing problem is to use a substructural type system.

[15] Pucella and Tov 2008.

Session Types

You can implement session types with:[16]

- types and type classes/traits (duality, sequencing)
- linear threading (linear types, indexed monads).

Session Types for Rust

Thomas Bracht Laumann Jespersen Philip Munksgaard Ken Friis Larsen
Department of Computer Science, University of Copenhagen, Denmark
ntl316@alumni.ku.dk pmunksgaard@gmail.com kflarsen@diku.dk

[16] Jespersen, Munksgaard, and Larsen 2015.

Session Types

Some of the basic building block types used in th 2008 Haskell paper:[17]

$$\mathbf{data}\ a\ :!:\ r \quad \text{-- send 'a' then continue with 'r'}$$
$$\mathbf{data}\ a\ :?:\ r \quad \text{-- receive 'a' then continue with 'r'}$$
$$\mathbf{data}\ r\ :+:\ s \quad \text{-- Choose 'r' or 's'}$$
$$\mathbf{data}\ r\ :\&:\ s \quad \text{-- Offer 'r' or 's'}$$
$$\mathbf{data}\ Eps \quad \text{-- Protocol is depleted}$$

[17] Pucella and Tov 2008.

Session Types

Communicating channels can be implemented as dual session types:

$$s :: Int \ :!: \ Bool \ :?: \ Eps \quad \text{-- Channel 1}$$
$$\overline{s} :: Int \ :?: \ Bool \ :!: \ Eps \quad \text{-- Channel 2}$$

Session types could benefit from some linearity

Why session types require linearity:

$$\begin{aligned}
&\mathbf{data}\ Channel\ a \\
&h :: Channel\ (Int\ :!:\ Eps) \to Channel\ (Int\ :!:\ Eps) \to (Int, Int) \\
&p :: Channel\ (Int\ :!:\ Eps) \to (Int, Int) \\
&p\ c = h\ c\ c
\end{aligned}$$

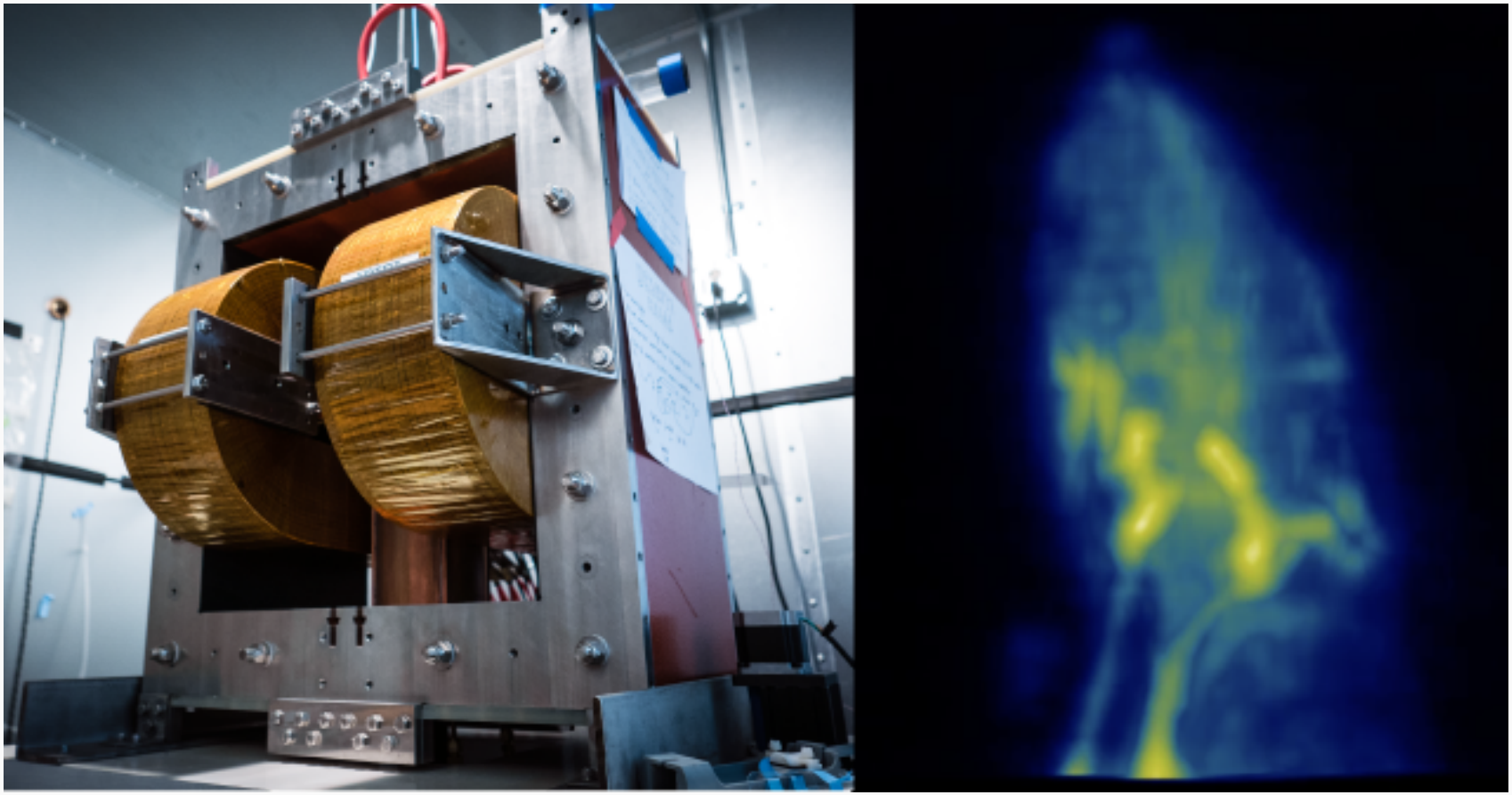
-- ‘h‘ sends Int down both channels

Two Ints will be sent down the same channel, violating protocol.

$$\begin{aligned}
&p_L :: Channel\ (Int\ :!:\ Eps) \multimap (Int, Int) \\
&p_L\ c = h\ c\ c
\end{aligned}$$

-- c has weight ω, oops!

We would like *users* to control the 30 kW scanner...



We would like *users* to control the 30 kW scanner...

Safe, *user programmable* scanner api

-- Definition of scanner

$$\mathbf{data}\ State = Setup \mid Scanning$$

$$\mathbf{data}\ Scanner\ (s :: State)$$

$$\mathbf{data}\ Parameters$$

$$scanner :: IO_L\ (Scanner\ Setup)$$

$$setParameters\ ::\ Scanner\ Setup \multimap [Parameters] \multimap IO_L\ (Scanner\ Setup)$$

$$verifyAndStart :: Scanner\ Setup \multimap IO_L\ (Scanner\ Scanning)$$

$$getData :: Scanner\ Scanning \multimap IO_L\ (Scanner\ Scanning, Unrestricted\ [Float])$$

$$scannerOff :: \forall s.Scanner\ s \multimap IO_L\ ()$$

Possible Applications for Us

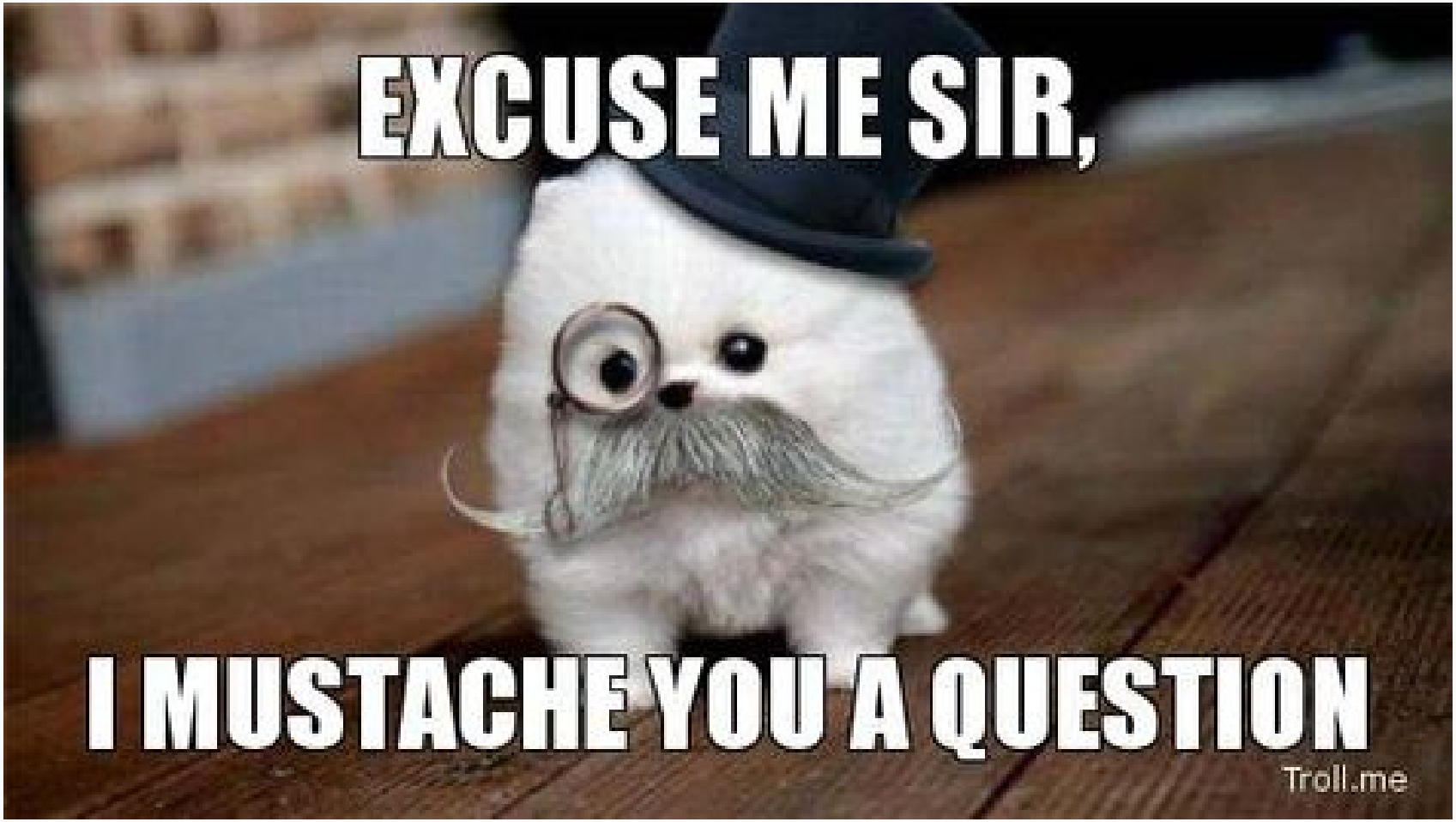
- Encoding of proper hardware boot up/shutdown sequences.
- Communication protocol between our real-time system and control computer.
- User-defined protocols for safe, dynamic scanner state modification based on real-time feedback.

Conclusion

Substructural types allow us to make stronger guarantees about

1. correct resource usage,
2. correct interfaces between two systems, and
3. potentially has medical applications.

Thank you for listening!



Scanner resource example implementation

For the scanner, we can write a program that allows users to take a scan.

$$
\begin{aligned}
&userDefinedScan :: [Parameters] \rightarrow IO\ () \\
&userDefinedScan\ ps = withLinearIO\ \$\ \mathbf{do}_L \\
&\quad s \leftarrow scanner \\
&\quad s \leftarrow setParameters\ s\ ps \\
&\quad s \leftarrow verifyAndStart\ s \\
&\quad (s, Unrestricted\ rawData) \leftarrow getData\ s \\
&\quad scannerOff\ s \\
&\quad return_L\ (Unrestricted\ ())
\end{aligned}
$$

Scanner resource example forgets to verify

If we try to write new parameters, these is an error in using the pre-scan state more than once.

$$
\begin{aligned}
&userDefinedScan_{\text{☹}} :: [Parameters] \to IO\ () \\
&userDefinedScan_{\text{☹}}\ ps = withLinearIO\ \$\ \mathbf{do}_L \\
&\quad s \leftarrow scanner \\
&\quad s_p \leftarrow setParameters\ s\ ps \\
&\quad s \leftarrow verifyAndStart\ s_p \\
&\quad (s, Unrestricted\ rawData) \leftarrow getData\ s \\
&\quad s \leftarrow setParameters\ s_p\ ps \\
&\quad scannerOff\ s \\
&\quad return_L\ (Unrestricted\ ())
\end{aligned}
$$