Substructural Logic

(Linear Logic and Linear Type Systems)

Hype for Types

October 3, 2023

What We'll Talk About

- What it means for a logic to be "substructural"

What We'll Talk About

- What it means for a logic to be "substructural"
- A case study of a particular substructural logic (linear logic)

What We'll Talk About

- What it means for a logic to be "substructural"
- A case study of a particular substructural logic (linear logic)
- How to make `malloc` and `free` safe

What We'll Talk About

- What it means for a logic to be "substructural"
- A case study of a particular substructural logic (linear logic)
- How to make `malloc` and `free` safe
- What it looks like to code in a language with resource-aware types

Substructural Logic

She Sub on my Structure

The constructive logic we have been working with so far has the following admissible rules, which we call "structural properties" of the logic:

$$\frac{\Gamma \vdash C}{\Gamma, A \vdash C}\ (\textsc{Weak}) \qquad \frac{\Gamma, A, A \vdash C}{\Gamma, A \vdash C}\ (\textsc{Cntr})$$

$$\frac{\Gamma, A, B \vdash C}{\Gamma, B, A \vdash C}\ (\textsc{Exch})$$

She Sub on my Structure

What happens if you remove some of these structural properties?

She Sub on my Structure

What happens if you remove some of these structural properties? You would get a new logical system!

She Sub on my Structure

What happens if you remove some of these structural properties? You would get a new logical system!

- Affine Logic: no contraction

She Sub on my Structure

What happens if you remove some of these structural properties? You would get a new logical system!

- Affine Logic: no contraction
- Linear Logic: no weakening or contraction

She Sub on my Structure

What happens if you remove some of these structural properties? You would get a new logical system!

- Affine Logic: no contraction
- Linear Logic: no weakening or contraction
- Ordered Logic: no weakening, contraction, or exchange

She Sub on my Structure

What happens if you remove some of these structural properties? You would get a new logical system!

- Affine Logic: no contraction
- Linear Logic: no weakening or contraction
- Ordered Logic: no weakening, contraction, or exchange

Question

What are the consequences of not having these structural properties?

She Sub on my Structure

What happens if you remove some of these structural properties? You would get a new logical system!

- Affine Logic: no contraction
- Linear Logic: no weakening or contraction
- Ordered Logic: no weakening, contraction, or exchange

Question

What are the consequences of not having these structural properties?

Today, we'll be focusing on *linear logic*, and how we can use it to implement a memory-safe version of C0.

Linear Logic

Is Logic Logical?

The moon is made of green cheese. Therefore, you come to hype for types today.

Question

Is this logical?

Different Interpretation of Implication

Constructive logic interprets $A \Rightarrow B$ as "If you give me A is true, then I give you B is true".
But what it really says is "If you give me as many copy of A as I need, then I give you B is true".

Different Interpretation of Implication

Constructive logic interprets $A \Rightarrow B$ as "If you give me A is true, then I give you B is true".
But what it really says is "If you give me as many copy of A as I need, then I give you B is true".

Idea

The problem of previous example is that to prove the conclusion we only need zero copies of the assumption, hence lacking "relevance".

Different Interpretation of Implication

Constructive logic interprets $A \Rightarrow B$ as "If you give me A is true, then I give you B is true".
But what it really says is "If you give me as many copy of A as I need, then I give you B is true".

Idea

The problem of previous example is that to prove the conclusion we only need zero copies of the assumption, hence lacking "relevance".

Idea

We need a logic that forces relevance.

Malloc is Scary...

Consider the following C code:

```c
int main () {
  char *str;
  str = (char *) malloc(13);
  strcpy(str, "hypefortypes");
  free(str);
  return(0);
}
```

In C, we have to make sure we allocate and deallocate every memory cell exactly once.

Malloc is Scary...

Consider the following C code:

```c
int main () {
  char *str;
  str = (char *) malloc(13);
  strcpy(str, "hypefortypes");
  free(str);
  return(0);
}
```

In C, we have to make sure we allocate and deallocate every memory cell exactly once.

Question

Is there a way to make our *types* guarantee correctness?

The Problem With Constructive Logic

In “normal” constructive logic, we have no concept of *state*.

The Problem With Constructive Logic

In "normal" constructive logic, we have no concept of *state*.

Big Idea

Proofs should no longer be *persistent*, but rather *ephemeral*.

The Problem With Constructive Logic

In "normal" constructive logic, we have no concept of *state*.

Big Idea

Proofs should no longer be *persistent*, but rather *ephemeral*.

Persistence is due to implicit **structural rules**: weakening and contraction.

Weakening

```c
int main() {
  int *x = (int *) malloc(sizeof(int));
  *x = 3;
  return 0;
}
```

Weakening

```c
int main() {
  int *x = (int *) malloc(sizeof(int));
  *x = 3;
  return 0;
}
```

Weakening: we can "drop" assumptions

$$\frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau} \;(\textsc{Weak})$$

Contraction

```c
void f(int *x) {
  free(x);
}

int main() {
  int *x = (int *) malloc(sizeof(int));
  *x = 3;
  f(x);
  f(x);
  return 0;
}
```

Contraction

```
void f(int *x) {
  free(x);
}

int main() {
  int *x = (int *) malloc(sizeof(int));
  *x = 3;
  f(x);
  f(x);
  return 0;
}
```

Contraction: we can "duplicate" assumptions

$$\frac{\Gamma, x_1 : \tau, x_2 : \tau \vdash e : \tau'}{\Gamma, x : \tau \vdash [x, x / x_1, x_2]e : \tau'} \text{ (CNTR)}$$

Introduction to Linear Logic

In **linear logic**, we have neither weakening nor contraction.

- Requirement that we use each piece of data *exactly* once - no duplication, no dropping
- Comes with an inherent idea of "resources" that are used up
- Allows us to write safe, stateful (imperative!) programs

The Linear Rules

Identity

Constructive Logic

$$\frac{A \in \Gamma}{\Gamma \vdash A} \ (\textsc{Hyp})$$

Identity

Constructive Logic

$$\frac{A \in \Gamma}{\Gamma \vdash A} \text{ (Hyp)}$$

Linear Logic

$$\frac{}{A \vdash A} \text{ (Hyp)}$$

Identity

Constructive Logic

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ (\textsc{Hyp})$$

Linear Logic

$$\frac{}{A \vdash A}\ (\textsc{Hyp})$$

Intuition

"Given A and nothing else, we can use up A"

Conjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \wedge A_2}\ (\wedge\mathrm{I})$$

$$\frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_1}\ (\wedge\mathrm{E1}) \qquad \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_2}\ (\wedge\mathrm{E2})$$

Conjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \wedge A_2}\ (\wedge\mathrm{I})$$

$$\frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_1}\ (\wedge\mathrm{E1}) \qquad \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_2}\ (\wedge\mathrm{E2})$$

Linear Logic

Conjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \wedge A_2}\ (\wedge\mathrm{I})$$

$$\frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_1}\ (\wedge\mathrm{E1}) \qquad\qquad \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_2}\ (\wedge\mathrm{E2})$$

Linear Logic

$$\frac{\Delta_1 \vdash A_1 \qquad \Delta_2 \vdash A_2}{\Delta_1, \Delta_2 \vdash A_1 \otimes A_2}\ (\otimes\mathrm{I})$$

Conjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1 \qquad \Gamma \vdash A_2}{\Gamma \vdash A_1 \wedge A_2}\ (\wedge\mathrm{I})$$

$$\frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_1}\ (\wedge\mathrm{E1}) \qquad\qquad \frac{\Gamma \vdash A_1 \wedge A_2}{\Gamma \vdash A_2}\ (\wedge\mathrm{E2})$$

Linear Logic

$$\frac{\Delta_1 \vdash A_1 \qquad \Delta_2 \vdash A_2}{\Delta_1, \Delta_2 \vdash A_1 \otimes A_2}\ (\otimes\mathrm{I})$$

$$\frac{\Delta \vdash A_1 \otimes A_2 \qquad \Delta', A_1, A_2 \vdash C}{\Delta, \Delta' \vdash C}\ (\otimes\mathrm{E})$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_1) \qquad \frac{\Gamma \vdash A_2}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_2)$$

$$\frac{\Gamma \vdash A_1 \vee A_2 \qquad \Gamma, A_1 \vdash B \qquad \Gamma, A_2 \vdash B}{\Gamma \vdash B}\ (\vee E)$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_1) \qquad \frac{\Gamma \vdash A_2}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_2)$$

$$\frac{\Gamma \vdash A_1 \vee A_2 \qquad \Gamma, A_1 \vdash B \qquad \Gamma, A_2 \vdash B}{\Gamma \vdash B}\ (\vee E)$$

Linear Logic

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_1) \qquad \frac{\Gamma \vdash A_2}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_2)$$

$$\frac{\Gamma \vdash A_1 \vee A_2 \qquad \Gamma, A_1 \vdash B \qquad \Gamma, A_2 \vdash B}{\Gamma \vdash B}\ (\vee E)$$

Linear Logic

$$\frac{\Delta \vdash A_1}{\Delta \vdash A_1 \oplus A_2}\ (\oplus \mathrm{I1})$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_1) \qquad \frac{\Gamma \vdash A_2}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_2)$$

$$\frac{\Gamma \vdash A_1 \vee A_2 \qquad \Gamma, A_1 \vdash B \qquad \Gamma, A_2 \vdash B}{\Gamma \vdash B}\ (\vee E)$$

Linear Logic

$$\frac{\Delta \vdash A_1}{\Delta \vdash A_1 \oplus A_2}\ (\oplus\mathrm{I1}) \qquad \frac{\Delta \vdash A_2}{\Delta \vdash A_1 \oplus A_2}\ (\oplus\mathrm{I2})$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash A_1}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_1) \qquad \frac{\Gamma \vdash A_2}{\Gamma \vdash A_1 \vee A_2}\ (\vee I_2)$$

$$\frac{\Gamma \vdash A_1 \vee A_2 \qquad \Gamma, A_1 \vdash B \qquad \Gamma, A_2 \vdash B}{\Gamma \vdash B}\ (\vee E)$$

Linear Logic

$$\frac{\Delta \vdash A_1}{\Delta \vdash A_1 \oplus A_2}\ (\oplus\mathrm{I1}) \qquad \frac{\Delta \vdash A_2}{\Delta \vdash A_1 \oplus A_2}\ (\oplus\mathrm{I2})$$

$$\frac{\Delta \vdash A_1 \oplus A_2 \qquad \Delta', A_1 \vdash B \qquad \Delta', A_2 \vdash B}{\Delta, \Delta' \vdash B}\ (\oplus\mathrm{E})$$

Implication

Constructive Logic

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \supset A_2}\ (\supset I) \qquad\qquad \frac{\Gamma \vdash A_1 \supset A_2 \quad \Gamma \vdash A_1}{\Gamma \vdash A_2}\ (\supset E)$$

Implication

Constructive Logic

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \supset A_2} \; (\supset I) \qquad\qquad \frac{\Gamma \vdash A_1 \supset A_2 \quad \Gamma \vdash A_1}{\Gamma \vdash A_2} \; (\supset E)$$

Linear Logic

Implication

Constructive Logic

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \supset A_2} \; (\supset I) \qquad\qquad \frac{\Gamma \vdash A_1 \supset A_2 \quad \Gamma \vdash A_1}{\Gamma \vdash A_2} \; (\supset E)$$

Linear Logic

$$\frac{\Delta, A_1 \vdash A_2}{\Delta \vdash A_1 \multimap A_2} \; (\multimap I)$$

Implication

Constructive Logic

$$\frac{\Gamma, A_1 \vdash A_2}{\Gamma \vdash A_1 \supset A_2} \; (\supset I) \qquad \frac{\Gamma \vdash A_1 \supset A_2 \quad \Gamma \vdash A_1}{\Gamma \vdash A_2} \; (\supset E)$$

Linear Logic

$$\frac{\Delta, A_1 \vdash A_2}{\Delta \vdash A_1 \multimap A_2} \; (\multimap I) \qquad \frac{\Delta \vdash A_1 \multimap A_2 \quad \Delta' \vdash A_1}{\Delta, \Delta' \vdash A_2} \; (\multimap E)$$

Model Real Worlds Using Linear Logic

5 dollars can buy one coffee and one donut.

$$\$5 \multimap \texttt{coffee} \otimes \texttt{donut}$$

Model Real Worlds Using Linear Logic

5 dollars can buy one coffee and one donut.

$$\$5 \multimap \texttt{coffee} \otimes \texttt{donut}$$

Buffet entrance is 10 dollars. Once you enter, you can eat some beef, and with 2 more dollars you can eat some chicken.

$$\$10 \multimap \texttt{beef} \otimes (\$2 \multimap \texttt{chicken})$$

Towards a Linear C0[0]

[0]Fine, C0.

What are resources in C0?

- `int`? `string`? `int*`?

What are resources in C0?

- `int`? `string`? `int*`?
- We'll just treat pointers as linear
- Use a **reusable context**, Γ, to represent reusable variables and a **linear context**, Δ, for linear variables

What are resources in C0?

- `int`? `string`? `int*`?
- We'll just treat pointers as linear
- Use a **reusable context**, Γ, to represent reusable variables and a **linear context**, Δ, for linear variables

$$\frac{}{\Gamma, x : \tau; \cdot \vdash x : \tau} \ (\textsc{Var-Reusable})$$

What are resources in C0?

- `int`? `string`? `int*`?
- We'll just treat pointers as linear
- Use a **reusable context**, Γ, to represent reusable variables and a **linear context**, Δ, for linear variables

$$\frac{}{\Gamma, x : \tau; \cdot \vdash x : \tau} \ (\textsc{Var-Reusable}) \qquad \frac{}{\Gamma; x : \tau \vdash x : \tau} \ (\textsc{Var-Linear})$$

Resource Splitting: Operators

In C0, we have built-in operators (e.g., +, -).

Resource Splitting: Operators

In C0, we have built-in operators (e.g., +, -).

$$\frac{}{+ : (\textbf{int}, \textbf{int}) \to \textbf{int}} \qquad \frac{}{- : (\textbf{int}, \textbf{int}) \to \textbf{int}} \qquad \frac{}{== : (\textbf{int}, \textbf{int}) \to \textbf{bool}}$$

Resource Splitting: Operators

In C0, we have built-in operators (e.g., +, -).

$$\frac{}{+ : (\textbf{int}, \textbf{int}) \to \textbf{int}} \qquad \frac{}{- : (\textbf{int}, \textbf{int}) \to \textbf{int}} \qquad \frac{}{== : (\textbf{int}, \textbf{int}) \to \textbf{bool}}$$

$$\frac{\odot : (\tau_1, \tau_2) \to \tau \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \odot e_2 : \tau} \text{ (SML BinOp)}$$

Resource Splitting: Operators

In C0, we have built-in operators (e.g., +, -).

$$\frac{}{+ : (\textbf{int}, \textbf{int}) \to \textbf{int}} \qquad \frac{}{- : (\textbf{int}, \textbf{int}) \to \textbf{int}} \qquad \frac{}{== : (\textbf{int}, \textbf{int}) \to \textbf{bool}}$$

$$\frac{\odot : (\tau_1, \tau_2) \to \tau \qquad \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \odot e_2 : \tau} \text{(SML BinOp)}$$

$$\frac{\odot : (\tau_1, \tau_2) \to \tau \qquad \Gamma; \Delta_1 \vdash e_1 : \tau_1 \qquad \Gamma; \Delta_2 \vdash e_2 : \tau_2}{\Gamma; \Delta_1, \Delta_2 \vdash e_1 \odot e_2 : \tau} \text{(C0 BinOp)}$$

Resource Splitting: Function Application

We also have user-defined top-level functions (e.g. `foo`, `reverse_list`).

Resource Splitting: Function Application

We also have user-defined top-level functions (e.g. `foo`, `reverse_list`).

$$\frac{(\tau_1, \ldots, \tau_n) \to \tau \qquad \Gamma; \mathbf{?} \vdash e_i : \tau_i \quad (\forall i)}{\Gamma; \mathbf{?} \vdash f(e_1, \ldots, e_n) : \tau} \text{ (C0 APPLICATION)}$$

Resource Splitting: Function Application

We also have user-defined top-level functions (e.g. `foo`, `reverse_list`).

$$\frac{(\tau_1, \ldots, \tau_n) \to \tau \qquad \Gamma; \Delta_i \vdash e_i : \tau_i \quad (\forall i)}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash f(e_1, \ldots, e_n) : \tau} \text{ (C0 APPLICATION)}$$

Resource Splitting: Function Application

We also have user-defined top-level functions (e.g. `foo`, `reverse_list`).

$$\frac{(\tau_1, \ldots, \tau_n) \to \tau \qquad \Gamma; \Delta_i \vdash e_i : \tau_i \quad (\forall i)}{\Gamma; \Delta_1, \ldots, \Delta_n \vdash f(e_1, \ldots, e_n) : \tau} \text{ (C0 APPLICATION)}$$

```
int* foo(int* a, int* b) {
  free(a); return b;
}

int main() {
  int* x = alloc(int);
  int* y = foo(x, x);  // now a type error!
  free(y);
  return 0;
}
```

Resource Splitting: Null Checks

In general, pointer equality won't make sense in our language, since all pointers should be distinct.

However, in C, we need a way to check if pointers are `NULL`! Introducing:

```
int* create() /* ... */

int main() {
  int* x = create();

  if (x is NULL) {
    return 0;
  } else {
    int y = *x;  // still have x here!
    return y;
  }
}
```

Resource Splitting: Null Checks

$$\frac{}{\Gamma; \textcolor{red}{\textbf{?}} \vdash \textbf{NULL} : \tau^*} \text{ (N}\textsc{ull}\text{)}$$

Resource Splitting: Null Checks

$$\frac{}{\Gamma; \cdot \vdash \textbf{NULL} : \tau^*} \ (\textsc{Null})$$

Resource Splitting: Null Checks

$$\frac{}{\Gamma; \cdot \vdash \textbf{NULL} : \tau^*}\ (\textsc{Null})$$

$$\frac{\Gamma; \textcolor{red}{\textbf{?}} \vdash e_1 : \tau_2 \qquad \Gamma; \textcolor{red}{\textbf{?}} \vdash e_2 : \tau_2}{\Gamma; \Delta, x : \tau_1^* \vdash \textbf{ifnull}(x; e_1; e_2)}\ (\textsc{IfNull})$$

Resource Splitting: Null Checks

$$\frac{}{\Gamma; \cdot \vdash \textbf{NULL} : \tau^*} \text{(NULL)}$$

$$\frac{\Gamma; \Delta \vdash e_1 : \tau_2 \qquad \Gamma; \textbf{?} \vdash e_2 : \tau_2}{\Gamma; \Delta, x : \tau_1^* \vdash \textbf{ifnull}(x; e_1; e_2)} \text{(IFNULL)}$$

Resource Splitting: Null Checks

$$\frac{}{\Gamma; \cdot \vdash \textbf{NULL} : \tau^*} \ (\textsc{Null})$$

$$\frac{\Gamma; \Delta \vdash e_1 : \tau_2 \qquad \Gamma; \Delta, x : \tau_1^* \vdash e_2 : \tau_2}{\Gamma; \Delta, x : \tau_1^* \vdash \textbf{ifnull}(x; e_1; e_2)} \ (\textsc{IfNull})$$

Resource Tracking: Struct Introduction

Just like standard C0, we can allocate structs:

```
struct list {
  int head;
  struct list* tail;
};

struct list* nil() {
  return NULL;
}

struct list* cons(int x, struct list* xs) {
  struct list* node = alloc(struct list);
  node->head = x;
  node->tail = xs;
  return node;
}
```

Resource Tracking: Struct Elimination

Problem

We can't eliminate structs like we used to. How will we know that each field is used exactly once?

Resource Tracking: Struct Elimination

Problem

We can't eliminate structs like we used to. How will we know that each field is used exactly once?

Structs are just like products - so, pattern match!

Resource Tracking: Struct Elimination

Problem

We can't eliminate structs like we used to. How will we know that each field is used exactly once?

Structs are just like products - so, pattern match!

```
struct list {
  int head;
  struct list* tail;
};

int list_sum(struct list* l) {
  if (l is NULL)
    return 0;

  let { head = x; tail = xs; } = l; // new syntax
  return x + list_sum(xs);
}
```

Live Coding

Conclusion

Things We Talked About

Conclusion

Things We Talked About

- The idea of a logic being substructural

Conclusion

Things We Talked About

- The idea of a logic being substructural
- Linearity as a way of representing state

Conclusion

Things We Talked About

- The idea of a logic being substructural
- Linearity as a way of representing state
- Linear propositions in terms of resources

Conclusion

Things We Talked About

- The idea of a logic being substructural
- Linearity as a way of representing state
- Linear propositions in terms of resources
- A practical example of linear logic for memory safety

Conclusion

Things We Talked About

- The idea of a logic being substructural
- Linearity as a way of representing state
- Linear propositions in terms of resources
- A practical example of linear logic for memory safety

Things We Didn't Cover

Conclusion

Things We Talked About

- The idea of a logic being substructural
- Linearity as a way of representing state
- Linear propositions in terms of resources
- A practical example of linear logic for memory safety

Things We Didn't Cover

- Linear logic is actually all about processes and messages
  - Concurrency!

Conclusion

Things We Talked About

- The idea of a logic being substructural
- Linearity as a way of representing state
- Linear propositions in terms of resources
- A practical example of linear logic for memory safety

Things We Didn't Cover

- Linear logic is actually all about processes and messages
  - Concurrency!
- Resource tracking (identify the cost of different programs)

Conclusion

Things We Talked About

- The idea of a logic being substructural
- Linearity as a way of representing state
- Linear propositions in terms of resources
- A practical example of linear logic for memory safety

Things We Didn't Cover

- Linear logic is actually all about processes and messages
  - Concurrency!
- Resource tracking (identify the cost of different programs)
- Rust (which utilizes an affine logic system)

Conclusion

Things We Talked About

- The idea of a logic being substructural
- Linearity as a way of representing state
- Linear propositions in terms of resources
- A practical example of linear logic for memory safety

Things We Didn't Cover

- Linear logic is actually all about processes and messages
  - Concurrency!
- Resource tracking (identify the cost of different programs)
- Rust (which utilizes an affine logic system)
- More constrained substructural logics, such as ordered logic