

Linear Logic and Linear Type Systems

Hype for Types

March 9, 2021

What We'll Talk About

- A style of logic which treats variables differently than “standard” logic

What We'll Talk About

- A style of logic which treats variables differently than “standard” logic
- How to make `malloc` and `free` safe

What We'll Talk About

- A style of logic which treats variables differently than “standard” logic
- How to make `malloc` and `free` safe
- What it looks like to code in a language with resource-aware types

Linear Logic

Malloc is Scary...

Consider the following C code:

```
1 int main () {  
2     char *str;  
3     str = (char *) malloc(13);  
4     strcpy(str, "hypefortypes");  
5     free(str);  
6     return(0);  
7 }
```

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

Malloc is Scary...

Consider the following C code:

```
1 int main () {  
2     char *str;  
3     str = (char *) malloc(13);  
4     strcpy(str, "hypefortypes");  
5     free(str);  
6     return(0);  
7 }
```

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

```
1 int main() {  
2     int *x = (int *) malloc(sizeof(int));  
3     *x = 3;  
4     return 0;  
5 }
```

Weakening

```
1 int main() {  
2     int *x = (int *) malloc(sizeof(int));  
3     *x = 3;  
4     return 0;  
5 }
```

Weakening: we can “drop” assumptions

$$\frac{\Gamma \vdash e : \tau}{\Gamma, x : \tau' \vdash e : \tau} \text{ (WEAK)}$$

Contraction

```
1 void f(int *x) {  
2     free(x);  
3 }  
4  
5 int main() {  
6     int *x = (int *) malloc(sizeof(int));  
7     *x = 3;  
8     f(x);  
9     f(x);  
10    return 0;  
11 }
```

Contraction

```
1 void f(int *x) {  
2     free(x);  
3 }  
4  
5 int main() {  
6     int *x = (int *) malloc(sizeof(int));  
7     *x = 3;  
8     f(x);  
9     f(x);  
10    return 0;  
11 }
```

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

Constructive Logic

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{ (HYP)}$$

Constructive Logic

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{ (HYP)}$$

Linear Logic

$$\frac{}{x : A \vdash x : A} \text{ (HYP)}$$

Identity

Constructive Logic

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \text{ (HYP)}$$

Linear Logic

$$\frac{}{x : A \vdash x : A} \text{ (HYP)}$$

Intuition

“Given A and nothing else, we can use up A ”

Conjunction

Constructive Logic

$$\frac{\Gamma \vdash e_1 : A_1 \quad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \wedge A_2} (\wedge I)$$

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{fst}(e) : A_1} (\wedge E1)$$

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{snd}(e) : A_2} (\wedge E2)$$

Conjunction

Constructive Logic

$$\frac{\Gamma \vdash e_1 : A_1 \quad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \wedge A_2} (\wedge I)$$

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{fst}(e) : A_1} (\wedge E1)$$

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{snd}(e) : A_2} (\wedge E2)$$

Linear Logic

Conjunction

Constructive Logic

$$\frac{\Gamma \vdash e_1 : A_1 \quad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \wedge A_2} (\wedge I)$$

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{fst}(e) : A_1} (\wedge E1)$$

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{snd}(e) : A_2} (\wedge E2)$$

Linear Logic

$$\frac{\Delta_1 \vdash e_1 : A_1 \quad \Delta_2 \vdash e_2 : A_2}{\Delta_1, \Delta_2 \vdash \langle e_1, e_2 \rangle : A_1 \otimes A_2} (\otimes I)$$

Conjunction

Constructive Logic

$$\frac{\Gamma \vdash e_1 : A_1 \quad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \wedge A_2} (\wedge I)$$

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{fst}(e) : A_1} (\wedge E1)$$

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{snd}(e) : A_2} (\wedge E2)$$

Linear Logic

$$\frac{\Delta_1 \vdash e_1 : A_1 \quad \Delta_2 \vdash e_2 : A_2}{\Delta_1, \Delta_2 \vdash \langle e_1, e_2 \rangle : A_1 \otimes A_2} (\otimes I)$$

$$\frac{\Delta \vdash e_1 : A_1 \otimes A_2 \quad \Delta', x_1 : A_1, x_2 : A_2 \vdash e_2 : C}{\Delta, \Delta' \vdash \mathbf{let} \langle x_1, x_2 \rangle = e_1 \mathbf{ in } e_2 : C} (\otimes E)$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash e : A_1}{\Gamma \vdash \mathbf{Left} \ e : A_1 \vee A_2} (\vee I_1)$$

$$\frac{\Gamma \vdash e : A_2}{\Gamma \vdash \mathbf{Right} \ e : A_1 \vee A_2} (\vee I_2)$$

$$\frac{\Gamma \vdash e : A_1 \vee A_2 \quad \Gamma, x_1 : A_1 \vdash e_1 : B \quad \Gamma, x_2 : A_2 \vdash e_2 : B}{\Gamma \vdash \mathbf{case} \ e \ \mathbf{of} \ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 : B} (\vee E)$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash e : A_1}{\Gamma \vdash \mathbf{Left} \ e : A_1 \vee A_2} (\vee I_1)$$

$$\frac{\Gamma \vdash e : A_2}{\Gamma \vdash \mathbf{Right} \ e : A_1 \vee A_2} (\vee I_2)$$

$$\frac{\Gamma \vdash e : A_1 \vee A_2 \quad \Gamma, x_1 : A_1 \vdash e_1 : B \quad \Gamma, x_2 : A_2 \vdash e_2 : B}{\Gamma \vdash \mathbf{case} \ e \ \mathbf{of} \ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 : B} (\vee E)$$

Linear Logic

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash e : A_1}{\Gamma \vdash \mathbf{Left} \ e : A_1 \vee A_2} (\vee I_1)$$

$$\frac{\Gamma \vdash e : A_2}{\Gamma \vdash \mathbf{Right} \ e : A_1 \vee A_2} (\vee I_2)$$

$$\frac{\Gamma \vdash e : A_1 \vee A_2 \quad \Gamma, x_1 : A_1 \vdash e_1 : B \quad \Gamma, x_2 : A_2 \vdash e_2 : B}{\Gamma \vdash \mathbf{case} \ e \ \mathbf{of} \ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 : B} (\vee E)$$

Linear Logic

$$\frac{\Delta \vdash e : A_1}{\Delta \vdash \mathbf{Left} \ e : A_1 \oplus A_2} (\oplus I_1)$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash e : A_1}{\Gamma \vdash \mathbf{Left} \ e : A_1 \vee A_2} (\vee I_1)$$

$$\frac{\Gamma \vdash e : A_2}{\Gamma \vdash \mathbf{Right} \ e : A_1 \vee A_2} (\vee I_2)$$

$$\frac{\Gamma \vdash e : A_1 \vee A_2 \quad \Gamma, x_1 : A_1 \vdash e_1 : B \quad \Gamma, x_2 : A_2 \vdash e_2 : B}{\Gamma \vdash \mathbf{case} \ e \ \mathbf{of} \ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 : B} (\vee E)$$

Linear Logic

$$\frac{\Delta \vdash e : A_1}{\Delta \vdash \mathbf{Left} \ e : A_1 \oplus A_2} (\oplus I_1)$$

$$\frac{\Delta \vdash e : A_2}{\Delta \vdash \mathbf{Right} \ e : A_1 \oplus A_2} (\oplus I_2)$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash e : A_1}{\Gamma \vdash \mathbf{Left} \ e : A_1 \vee A_2} (\vee I_1)$$

$$\frac{\Gamma \vdash e : A_2}{\Gamma \vdash \mathbf{Right} \ e : A_1 \vee A_2} (\vee I_2)$$

$$\frac{\Gamma \vdash e : A_1 \vee A_2 \quad \Gamma, x_1 : A_1 \vdash e_1 : B \quad \Gamma, x_2 : A_2 \vdash e_2 : B}{\Gamma \vdash \mathbf{case} \ e \ \mathbf{of} \ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 : B} (\vee E)$$

Linear Logic

$$\frac{\Delta \vdash e : A_1}{\Delta \vdash \mathbf{Left} \ e : A_1 \oplus A_2} (\oplus I_1)$$

$$\frac{\Delta \vdash e : A_2}{\Delta \vdash \mathbf{Right} \ e : A_1 \oplus A_2} (\oplus I_2)$$

$$\frac{\Delta \vdash e : A_1 \oplus A_2 \quad \Delta', x_1 : A_1 \vdash e_1 : B \quad \Delta', x_2 : A_2 \vdash e_2 : B}{\Delta, \Delta' \vdash \mathbf{case} \ e \ \mathbf{of} \ x_1 \Rightarrow e_1 \mid x_2 \Rightarrow e_2 : B} (\oplus E)$$

Towards a Linear C^0

⁰Fine, C^0 .

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

$$\frac{}{\Gamma, x : \tau; \cdot \vdash x : \tau} \text{ (VAR-REUSABLE)}$$

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

$$\frac{}{\Gamma, x : \tau; \cdot \vdash x : \tau} \text{ (VAR-REUSABLE)}$$

$$\frac{}{\Gamma; x : \tau \vdash x : \tau} \text{ (VAR-LINEAR)}$$

Resource Splitting: Operators

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

Resource Splitting: Operators

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

$$\frac{}{+ : (\mathbf{int}, \mathbf{int}) \rightarrow \mathbf{int}}$$
$$\frac{}{- : (\mathbf{int}, \mathbf{int}) \rightarrow \mathbf{int}}$$
$$\frac{}{== : (\mathbf{int}, \mathbf{int}) \rightarrow \mathbf{bool}}$$

Resource Splitting: Operators

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

$$\frac{}{+ : (\mathbf{int}, \mathbf{int}) \rightarrow \mathbf{int}}$$

$$\frac{}{- : (\mathbf{int}, \mathbf{int}) \rightarrow \mathbf{int}}$$

$$\frac{}{== : (\mathbf{int}, \mathbf{int}) \rightarrow \mathbf{bool}}$$

$$\frac{\odot : (\tau_1, \tau_2) \rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \odot e_2 : \tau} \text{ (SML BINOP)}$$

Resource Splitting: Operators

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

$$\frac{}{+ : (\mathbf{int}, \mathbf{int}) \rightarrow \mathbf{int}}$$

$$\frac{}{- : (\mathbf{int}, \mathbf{int}) \rightarrow \mathbf{int}}$$

$$\frac{}{== : (\mathbf{int}, \mathbf{int}) \rightarrow \mathbf{bool}}$$

$$\frac{\odot : (\tau_1, \tau_2) \rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \odot e_2 : \tau} \text{ (SML BINOP)}$$

$$\frac{\odot : (\tau_1, \tau_2) \rightarrow \tau \quad \Gamma; \Delta_1 \vdash e_1 : \tau_1 \quad \Gamma; \Delta_2 \vdash e_2 : \tau_2}{\Gamma; \Delta_1, \Delta_2 \vdash e_1 \odot e_2 : \tau} \text{ (C0 BINOP)}$$

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`).

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

Resource Splitting: Function Application

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

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

Resource Splitting: Function Application

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

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

```
1 int* foo(int* a, int* b) {  
2   free(a); return b;  
3 }  
4  
5 int main() {  
6   int* x = alloc(int);  
7   int* y = foo(x, x); // now a type error!  
8   free(y);  
9   return 0;  
10 }
```

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:

```
1 int* create() /* ... */
2
3 int main() {
4     int* x = create();
5
6     if (x is NULL) {
7         return 0;
8     } else {
9         int y = *x; // still have x here!
10        return y;
11    }
12 }
```

Resource Splitting: Null Checks

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

Resource Splitting: Null Checks

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

Resource Splitting: Null Checks

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

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

Resource Splitting: Null Checks

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

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

Resource Splitting: Null Checks

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

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

Resource Tracking: Struct Introduction

Just like standard C0, we can allocate structs:

```
1 struct list {  
2     int head;  
3     struct list* tail;  
4 };  
5  
6 struct list* nil() {  
7     return NULL;  
8 }  
9  
10 struct list* cons(int x, struct list* xs) {  
11     struct list* node = alloc(struct list);  
12     node->head = x;  
13     node->tail = xs;  
14     return node;  
15 }
```


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!

```
1 struct list {  
2     int head;  
3     struct list* tail;  
4 };  
5  
6 int list_sum(struct list* l) {  
7     if (l is NULL)  
8         return 0;  
9  
10    let { head = x; tail = xs; } = l; // new syntax  
11    return x + list_sum(xs);  
12 }
```

Live Coding

Conclusion

Things We Talked About

Conclusion

Things We Talked About

- Linearity as a way of representing state

Conclusion

Things We Talked About

- Linearity as a way of representing state
- Linear propositions in terms of resources

Conclusion

Things We Talked About

- 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

- 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

- 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

- 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