Linear Logic and Linear Type Systems

Hype for Types

September 28, 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:

```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:

```
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

```
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

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

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

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

Identity

Constructive Logic

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

Linear Logic

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

Identity

Constructive Logic

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

Linear Logic

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

Intuition

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

Conjunction

Constructive Logic

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

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{fst}(e) : A_1} \ (\wedge\mathrm{E1}) \qquad \frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{snd}(e) : A_2} \ (\wedge\mathrm{E2})$$

Conjunction

Constructive Logic

$$\frac{\Gamma \vdash e_1 : A_1 \qquad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \land A_2} \ (\land\mathrm{I})$$

$$\frac{\Gamma \vdash e : A_1 \land A_2}{\Gamma \vdash \mathbf{fst}(e) : A_1} \ (\land\mathrm{E1}) \qquad \frac{\Gamma \vdash e : A_1 \land A_2}{\Gamma \vdash \mathbf{snd}(e) : A_2} \ (\land\mathrm{E2})$$

Linear Logic

Conjunction

Constructive Logic

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

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{fst}(e) : A_1} \; (\wedge\mathrm{E1}) \qquad \frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{snd}(e) : A_2} \; (\wedge\mathrm{E2})$$

Linear Logic

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

Conjunction

Constructive Logic

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

$$\frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{fst}(e) : A_1} \ (\wedge\mathrm{E1}) \qquad \frac{\Gamma \vdash e : A_1 \wedge A_2}{\Gamma \vdash \mathbf{snd}(e) : A_2} \ (\wedge\mathrm{E2})$$

Linear Logic

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

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

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash e : A_1}{\Gamma \vdash \textbf{Left}\ e : A_1 \vee A_2}\ (\vee I_1) \qquad \frac{\Gamma \vdash e : A_2}{\Gamma \vdash \textbf{Right}\ e : A_1 \vee A_2}\ (\vee I_2)$$

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

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash e : A_1}{\Gamma \vdash \textbf{Left } e : A_1 \vee A_2} \ (\vee I_1) \qquad \frac{\Gamma \vdash e : A_2}{\Gamma \vdash \textbf{Right } e : A_1 \vee A_2} \ (\vee I_2)$$

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

Linear Logic

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash e : A_1}{\Gamma \vdash \textbf{Left } e : A_1 \vee A_2}\ (\vee I_1) \qquad \frac{\Gamma \vdash e : A_2}{\Gamma \vdash \textbf{Right } e : A_1 \vee A_2}\ (\vee I_2)$$

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

Linear Logic

$$\frac{\Delta \vdash e : A_1}{\Delta \vdash \textbf{Left } e : A_1 \oplus A_2}\ (\oplus \mathrm{I1})$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash e : A_1}{\Gamma \vdash \textbf{Left } e : A_1 \vee A_2}\ (\vee I_1) \qquad \frac{\Gamma \vdash e : A_2}{\Gamma \vdash \textbf{Right } e : A_1 \vee A_2}\ (\vee I_2)$$

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

Linear Logic

$$\frac{\Delta \vdash e : A_1}{\Delta \vdash \textbf{Left } e : A_1 \oplus A_2}\ (\oplus \mathrm{I1}) \qquad \frac{\Delta \vdash e : A_2}{\Delta \vdash \textbf{Right } e : A_1 \oplus A_2}\ (\oplus \mathrm{I2})$$

Disjunction

Constructive Logic

$$\frac{\Gamma \vdash e : A_1}{\Gamma \vdash \textbf{Left}\ e : A_1 \vee A_2}\ (\vee I_1) \qquad \frac{\Gamma \vdash e : A_2}{\Gamma \vdash \textbf{Right}\ e : A_1 \vee A_2}\ (\vee I_2)$$

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

Linear Logic

$$\frac{\Delta \vdash e : A_1}{\Delta \vdash \textbf{Left}\ e : A_1 \oplus A_2}\ (\oplus \mathrm{I1}) \qquad \frac{\Delta \vdash e : A_2}{\Delta \vdash \textbf{Right}\ e : A_1 \oplus A_2}\ (\oplus \mathrm{I2})$$

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

Towards a Linear C[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., +, -).

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

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

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

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`).

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

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

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

Resource Splitting: Null Checks

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

Resource Splitting: Null Checks

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

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

Resource Splitting: Null Checks

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

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

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

- 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