Substructural Type Systems

Password: $f(a+b) = f(a) + f(b)$

Arrays vs Lists

Arrays

Arrays

```
val A1 = [|"9", "8", "3", "1", "7"|]
val () = Array.update A1 (4, "2")
val "2" = Array.nth A1 4
```

Arrays

Arrays

Pros:

- $O(1)$ access
- $O(1)$ update

Arrays

Pros:

- $O(1)$ access
- $O(1)$ update

Cons:

- Reliant on mutation

Lists

Lists

```
val L1 = ["9", "8", "3", "1", "7"]
val L2 = List.update L1 (4, "2")
val "2" = List.nth L2 4
```

Lists

Lists

Pros:

- Purely functional

Lists

Pros:

- Purely functional

Cons:

- $O(n)$ access
- $O(n)$ update

We want a purely functional data structure with $O(1)$ access and update.

(Array)Sequences?

(Array)Sequences?

```
val S1 = <"9", "8", "3", "1", "7">
val S2 = Seq.update S1 (4, "2")
val "2" = Seq.nth S2 4
```

(Array)Sequences?

(Array)Sequences?

- Purely functional (interface)

(Array)Sequences?

- Purely functional (interface)
- $O(1)$ access

(Array)Sequences?

- Purely functional (interface)
- $O(1)$ access
- $O(n)$ update...

Why is update $O(n)$?

Why is update $O(n)$?

```
val S1 = <"9", "8", "3", "1", "7">

(* Makes a copy of S1 *)
val S2 = Seq.update S1 (4, "2")
```

Why does `update` perform a copy?

Why does `update` perform a copy?

```
val S1 = <"9", "8", "3", "1", "7">

(* Makes a copy of S1 *)
val S2 = Seq.update S1 (4, "2")

(* Expects to see S1 unmodified *)
val "7" = Seq.nth S1 4
```

We *need* mutability for $O(1)$ update…

We *need* mutability for $O(1)$ update...

but we want purely functional code.

Where does mutability go wrong?

```
val S1 = <"9", "8", "3", "1", "7">

val S2 = Seq.update S1 (4, "2")

(* Expects to see S1 unmodified *)
val "7" = Seq.nth S1 4
```

“Obvious” rules

“Obvious” rules

1. Variables can be used multiple times

“Obvious” rules

1. ~~Variables can be used multiple times~~

Affine Type System

Affine Type System

Variables can be used at most once.

Affine types

```
val S1 = <"9", "8", "3", "1", "7">

val S2 = Seq.update S1 (4, "2")

(* Compiler error *)
val "7" = Seq.nth S1 4
```

Using types to improve performance

Questions?

Theory break

Theory break

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

Theory break

$$\frac{}{\Gamma, x : \tau \vdash x : \tau}$$

Recall: Γ is a context mapping variables to their types.

We will treat Γ as a (possibly empty) unordered list of the form $x_1 : \tau_1, \ldots, x_n : \tau_n$.

Theory break

Think of elements of Γ as being "used up" whenever they are referenced.

“Obvious” rules

“Obvious” rules

“Variables can be used multiple times”

“Obvious” rules

“Variables can be used multiple times”

$$\frac{\Gamma, x : \tau, x : \tau \vdash x : \tau}{\Gamma, x : \tau \vdash x : \tau}\ \textsc{Contraction}$$

Questions?

The success story

The success story

Similar code in Rust

```rust
let V1 = vec!["9", "8", "3", "1", "7"];
let V2 = update(V1, 4, "2");
let _ = V1[4];
```

Error message

```
error[E0382]: borrow of moved value: `V1`
  --> test.rs:11:13
   |
9  |     let V1 = vec!["9", "8", "3", "1", "7"];
   |         -- move occurs because `V1` has type `std::vec::Vec<&str>`, which does not implement the `Copy` trait
10 |     let V2 = update(V1, 4, "2");
   |                     -- value moved here
11 |     let _ = V1[4];
   |             ^^ value borrowed here after move
```

Why does Rust have affine types?

Why does Rust have affine types?

Consider this program without affine types:

```
fun f x = []
```

Why does Rust have affine types?

Consider this program without affine types:

```
fun f x = []
```

Q: Can `x` be garbage collected at the end of `f`?

Why does Rust have affine types?

Consider this program without affine types:

```
fun f x = []
```

Q: Can `x` be garbage collected at the end of `f`?

A: Not necessarily - `f`'s caller may continue to reference `x`.

Why does Rust have affine types?

What about with affine types?

```
fun f x = []
```

Why does Rust have affine types?

What about with affine types?

```
fun f x = []
```

Q: Can `x` be garbage collected at the end of `f`?

Why does Rust have affine types?

What about with affine types?

```
fun f x = []
```

Q: Can `x` be garbage collected at the end of `f`?

A: Yes! `f`'s caller can no longer refer to `x` after passing it to `f`.

Rust has no global garbage collector at runtime - it needs to statically know when to dispose of values.

Using types to improve performance predictability

Concurrency

Concurrency

```
val t = create_thread ()
val x = ref 0

(* Send x to another thread *)
val () = send t x

(* Possible race *)
val () = x := 1
```

Concurrency with affine types

Concurrency with affine types

```
val t = create_thread ()
val x = ref 0

(* Send x to another thread *)
val () = send t x

(* Compiler error *)
val () = x := 1
```

Concurrency with affine types

```
val t = create_thread ()
val x = ref 0

val () = send t x

val x = recv t
val () = x := 1
```

Using types to improve correctness

Questions?

Resources

Files

Files

```
val openFile: path -> file
val closeFile: file -> unit
```

What could go wrong?

What could go wrong?

```
val f = openFile "free_uc_stones.gif"
val () = closeFile f
val () = closeFile f
```

What could go wrong?

```
val f = openFile "free_uc_stones.gif"
val () = closeFile f
val () = closeFile f
```

Affine types save us here.

What else could go wrong?

```
val f = openFile "free_uc_stones.gif"
```

What else could go wrong?

```
val f = openFile "free_uc_stones.gif"
```

Affine types won't help us here.

“Obvious” rules

1. ~~Variables can be used multiple times~~

“Obvious” rules

1. ~~Variables can be used multiple times~~
2. Variables can be used not at all

“Obvious” rules

1. ~~Variables can be used multiple times~~
2. ~~Variables can be used not at all~~

Linear Type System

Linear Type System

Variables must be used exactly once.

malloc/free

malloc/free

```c
int *x = malloc(sizeof(int));

// Don't forget to free
free(x);

// Don't double free
// free(x);
```

“Obvious” rules

“Obvious” rules

“Variables can be used not at all”

“Obvious” rules

“Variables can be used not at all”

$$\frac{\Gamma \vdash e : \tau}{\Gamma, x : \sigma \vdash e : \tau} \textsc{Weakening}$$

Questions?

“Obvious” rules

1. ~~Variables can be used multiple times~~
2. ~~Variables can be used not at all~~

“Obvious” rules

1. ~~Variables can be used multiple times~~
2. ~~Variables can be used not at all~~
3. Variables can be used in any order

Using variables out of order

```
val x = 1
val y = 2

val _ = f y
val _ = f x
```

“Obvious” rules

1. ~~Variables can be used multiple times~~
2. ~~Variables can be used not at all~~
3. ~~Variables can be used in any order~~

Ordered type system

Ordered type system

Variables must be used exactly once, in the order they were introduced.

Theory break

Theory break

We will treat Γ as a (possibly empty) *ordered* list of the form $x_1 : \tau_1, \ldots, x_n : \tau_n$.

“Obvious” rules

$$\frac{\Gamma, x : \tau, x : \tau, \Delta \vdash x : \tau}{\Gamma, x : \tau, \Delta \vdash x : \tau} \textsc{Contraction}$$

$$\frac{\Gamma, \Delta \vdash e : \tau}{\Gamma, x : \sigma, \Delta \vdash e : \tau} \textsc{Weakening}$$

“Obvious” rules

“Obvious” rules

“Variables can be used in any order”

$$\frac{\Gamma, x : \sigma, y : \sigma', \Delta \vdash e : \tau}{\Gamma, y : \sigma', x : \sigma, \Delta \vdash e : \tau} \textsc{Exchange}$$

Substructural Type System

Structural rules

$$\frac{\Gamma, x : \tau, x : \tau, \Delta \vdash x : \tau}{\Gamma, x : \tau, \Delta \vdash x : \tau} \textsc{Contraction}$$

$$\frac{\Gamma, \Delta \vdash e : \tau}{\Gamma, x : \sigma, \Delta \vdash e : \tau} \textsc{Weakening}$$

$$\frac{\Gamma, x : \sigma, y : \sigma', \Delta \vdash e : \tau}{\Gamma, y : \sigma', x : \sigma, \Delta \vdash e : \tau} \textsc{Exchange}$$

A substructural type system is one which omits one or more of contraction, weakening, and exchange.

	Exchange	Weakening	Contraction
Normal	Y	Y	Y
Relevant	Y	-	Y
Affine	Y	Y	-
Linear	Y	-	-
Ordered	-	-	-

Questions?