Formal Foundations of Programming Languages

Lecture 01

Prof. Ralf Jung

ETH Zürich

Welcome to FFPL!

Who am I?

- Prof. Ralf Jung
- Started at ETH in November 2022

Welcome to FFPL!

Who am I?

- Prof. Ralf Jung
- Started at ETH in November 2022
- This is my first course at ETH!

Welcome to FFPL!

Who am I?

- Prof. Ralf Jung
- Started at ETH in November 2022
- This is my first course at ETH!
- Research Area: Formal verification, Concurrency, Separation logic, Type systems, Machine-checked proofs (Coq)

Welcome to FFPL!

Who am I?

- Prof. Ralf Jung
- Started at ETH in November 2022
- This is my first course at ETH!
- Research Area: Formal verification, Concurrency, Separation logic, Type systems, Machine-checked proofs (Coq)
- Favorite language: Rust

What is FFPL about?

- Haskell, Rust, ML, Java: Types are everywhere.

What is FFPL about?

- Haskell, Rust, ML, Java: Types are everywhere.
- But what do types really achieve, and how can we capture them mathematically?
- Can we prove that a type system is useful? Can we have the computer assist us?
- There is no free lunch, but is there such a thing as a free theorem?

Having fun
with types and proofs

Course plan (tentative)

Sep 20th	Coq warmup, part 1
Sep 27th	Coq warmup, part 2
Oct 04th	Coq warmup, part 3
Oct 11th	Simply-typed λ-calculus, part 1
Oct 18th	Simply-typed λ-calculus, part 2
Oct 25th	System F, part 1
Nov 1st	System F, part 2
Nov 8th	**Mid-term exam**
Nov 15th	Unsafe code
Nov 22nd	Free theorems
Nov 29th	Recursive types
Dec 5th	Mutable state
Dec 13th	Unsafe code
Dec 20th	Outlook: concurrency / ?
Examination session	**Final exam**

Course logistics

- Moodle page
  - Please use the forum!

Course logistics

- Moodle page
  - Please use the forum!
- Wed 9:15 – 12:00 (CAB G 59):
  Mix of lecture and Coq exercises

Course logistics

- Moodle page
  - Please use the forum!
- Wed 9:15 – 12:00 (CAB G 59):
  Mix of lecture and Coq exercises
- Weekly exercise sheet appears on Fri
  - Not graded
  - Sample solutions the following week

Course logistics

- Moodle page
  - Please use the forum!
- Wed 9:15 – 12:00 (CAB G 59):
  Mix of lecture and Coq exercises
- Weekly exercise sheet appears on Fri
  - Not graded
  - Sample solutions the following week
- Fri 11:15 – 12:00 (CAB G 32):
  Exercise group, discuss exercise sheet & everything else

Questions?

Ralf Jung: ralf.jung@inf.ethz.ch
Max Vistrup: max.vistrup@inf.ethz.ch

Questions?

Feedback?

Ralf Jung: ralf.jung@inf.ethz.ch
Max Vistrup: max.vistrup@inf.ethz.ch

Types: Pro and Contra

- Discuss with your neighbor:
  Are types any good?

Types: Pro and Contra

- Discuss with your neighbor:
  Are types any good?
  - One of you picks their favorite typed language
  - The other their favorite untyped language
  - Collect arguments for why (not) having types is a good idea

Types: Pro and Contra

- Discuss with your neighbor:
  Are types any good?
  - One of you picks their favorite typed language
  - The other their favorite untyped language
  - Collect arguments for why (not) having types is a good idea
- After 5min, we will collect arguments

Types: Pro and Contra

- Discuss with your neighbor:
  Are types any good?
  - One of you picks their favorite typed language
  - The other their favorite untyped language
  - Collect arguments for why (not) having types is a good idea
- After 5min, we will collect arguments
- Quick vote: „for or against“ types

Types: Pro and Contra

- Discuss with your neighbor:
  Are types any good?
  - One of you picks their favorite typed language
  - The other their favorite untyped language
  - Collect arguments for why (not) having types is a good idea
- After 5min, we will collect arguments
- Quick vote: „for or against“ types
  - We will repeat this vote at the end of the semester, and compare results

Today

- Some high-level motivation
- Coq warmup, part 1

What are types good for?

Types prevent bugs

```
fn call(x: i32, y: char) -> i32 {
    x(y)
    // ERROR: cannot call value of type ‘i32‘
}
```

Types prevent bugs

```
fn call(x: i32, y: char) -> i32 {
    x(y)
    // ERROR: cannot call value of type ‘i32‘
}
fn add(x: fn(i32) -> i32) -> i32 {
    x + 2
    // ERROR: cannot add a function and an integer
}
```

Types prevent bugs

```
fn call(x: i32, y: char) -> i32 {
    x(y)
    // ERROR: cannot call value of type `i32`
}
fn add(x: fn(i32) -> i32) -> i32 {
    x + 2
    // ERROR: cannot add a function and an integer
}
fn not_init() -> i32 {
    let x: fn() -> i32;
    x()
    // ERROR: cannot read from uninitialized variable
}
```

Type soundness:

Well-typed programs do not go wrong

Types enable refactoring

```rust
struct IntList { ints: Vec<i32> /* NOT public! */ }
impl IntList {
    /// Construct a new empty list.
    pub fn new() -> IntList {
        IntList { ints: Vec::new() }
    }
    /// Add an element to the list.
    pub fn push(&mut self, x: i32) {
        self.ints.push(x);
    }
    /// Return the sum of the elements.
    pub fn sum(&self) -> i32 {
        self.ints.iter().sum()
    }
}
```

Types enable refactoring

```rust
struct IntList { ints: Vec<i32>, sum: i32 /* NOT public! */ }
impl IntList {
    /// Construct a new empty list.
    pub fn new() -> IntList {
        IntList { ints: Vec::new(), sum: 0 }
    }
    /// Add an element to the list.
    pub fn push(&mut self, x: i32) {
        self.ints.push(x); self.sum += x;
    }
    /// Return the sum of the elements.
    pub fn sum(&self) -> i32 {
        self.sum
    }
}
```

Type systems
enable abstraction.

Abstraction should not be taken for granted

Imagine we add a new operation to Rust:

```
/// Finds a value of type ‘i32‘ reachable somewhere
/// from ‘x‘ (if one exists), and changes its value
/// arbitrarily.
fn clobber_i32<T>(x: &mut T);
```

What does this mean for our `IntList` type?

Abstraction should not be taken for granted

Imagine we add a new operation to Rust:

```rust
/// Finds a value of type ‘i32‘ reachable somewhere
/// from ‘x‘ (if one exists), and changes its value
/// arbitrarily.
fn clobber_i32<T>(x: &mut T);
```

What does this mean for our `IntList` type?

Consider reflection in Java.
What does it mean for abstraction?

Abstraction should not be taken for granted

Imagine we add a new operation to Rust:

> Abstraction is not just useful for refactoring.
>
> But first we have to talk about **unsafe code.**

Consider reflection in Java.
What does it mean for abstraction?

Unsafe code

```
fn get_mid(x: &[i32]) -> &i32 {
    // This will perform a bounds-check each time.
    x[x.len()/2]
}
```

Unsafe code

```rust
fn get_mid(x: &[i32]) -> &i32 {
    // This will perform a bounds-check each time.
    x[x.len()/2]
}
fn get_mid_fast(x: &[i32]) -> &i32 {
    // Let's avoid the bounds-check.
    unsafe { x.get_unchecked(x.len()/2) }
}
```

Unsafe code

```
fn get_mid(x: &[i32]) -> &i32 {
    // This will perform a bounds-check each time.
    x[x.len()/2]
}
fn get_mid_fast(x: &[i32]) -> &i32 {
    // Let's avoid the bounds-check.
    unsafe { x.get_unchecked(x.len()/2) }
}
fn get_mid_correct(x: &[i32]) -> &i32 {
    if x.is_empty() { panic!(); }
    unsafe { x.get_unchecked(x.len()/2) }
}
```

Unsafe code

```
fn get_mid(x: &[i32]) -> &i32 {
```

Sometimes, the compiler is **not smart enough** to understand why a piece of code is safe.

Then we can use `unsafe` to put the **safety burden** on our own shoulders.

```
    unsafe { x.get_unchecked(x.len()/2) }
}
```

Unsafe abstractions

```rust
struct IntList { ints: Vec<i32>, last: usize }
impl IntList {
    /// Construct a new one-element list.
    pub fn new(x: i32) -> IntList {
        IntList { ints: vec![x], last: 0 }
    }
    /// Add an element to the list.
    pub fn push(&mut self, x: i32) {
        self.last = self.ints.len();
        self.ints.push(x);
    }
    /// Return the last element.
    pub fn last(&self) -> i32 {
        unsafe { *self.ints.get_unchecked(self.last) }
    }
}
```

Type systems enable safe encapsulation.

In this class, you will learn…

- …how to prove type soundness
- …how to prove that a type system provides abstraction
- …how to prove safe encapsulation

In this class, you will learn…

- …how to prove type soundness
- …how to prove that a type system provides abstraction
- …how to prove safe encapsulation

This will make you…

- …a better language designer
- …a better programmer*