Welcome to Hype for Types!

Welcome to Hype for Types!

- Introductions
  - Main instructors: Cam (cjwong), Jacob (jacobneu), Thejas (tkadur)

Welcome to Hype for Types!

- Introductions
  - Main instructors: Cam (cjwong), Jacob (jacobneu), Thejas (tkadur)
  - Guest lecturers: Aditi, Ariel, Avery, Harrison, Matthew

Welcome to Hype for Types!

- Introductions
  - Main instructors: Cam (cjwong), Jacob (jacobneu), Thejas (tkadur)
  - Guest lecturers: Aditi, Ariel, Avery, Harrison, Matthew
- Attendance
  - You must attend in order to pass!
  - Let us know if you can't make it

Welcome to Hype for Types!

- Introductions
  - Main instructors: Cam (cjwong), Jacob (jacobneu), Thejas (tkadur)
  - Guest lecturers: Aditi, Ariel, Avery, Harrison, Matthew
- Attendance
  - You must attend in order to pass!
  - Let us know if you can't make it
- Homework
  - Some lectures will have associated homeworks
  - Graded for effort, not completeness
  - Shouldn't take more than an hour, but let us know if you need more time

Welcome to Hype for Types!

- Introductions
  - Main instructors: Cam (cjwong), Jacob (jacobneu), Thejas (tkadur)
  - Guest lecturers: Aditi, Ariel, Avery, Harrison, Matthew
- Attendance
  - You must attend in order to pass!
  - Let us know if you can't make it
- Homework
  - Some lectures will have associated homeworks
  - Graded for effort, not completeness
  - Shouldn't take more than an hour, but let us know if you need more time
- Etc.
  - Functional Programming & Type Theory prerequisite knowledge
  - You don't have to understand every detail: some of these lectures could fill a semester's worth of content.
  - Let us know if you're having any issues. This course should *not* be a source of stress
  - The more feedback you give us (and the more questions you ask), the better!

Intro to Type Theory and Lambda Calculus

Hype for Types

Jacob Neumann

14 January 2020

Table of Contents

Section 1

Functional Programming

Basic Idea of Functional Programming

- **Programming**: Feeding a computer strings of symbols to tell it to do stuff

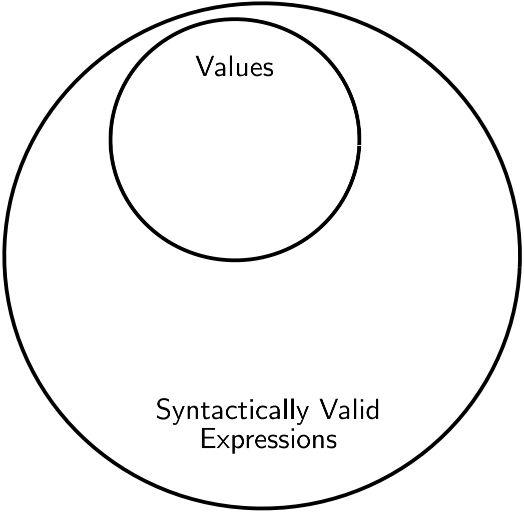
Basic Idea of Functional Programming

- **Programming**: Feeding a computer strings of symbols to tell it to do stuff
- **Imperative programming**: The strings represent *instructions* which are *executed* for their *effect* on the computer's state

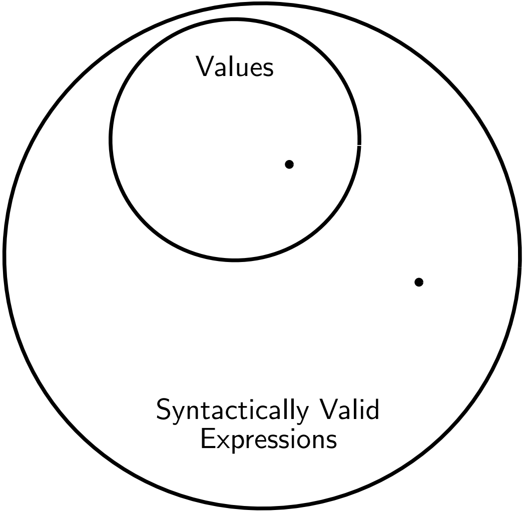
Basic Idea of Functional Programming

- **Programming**: Feeding a computer strings of symbols to tell it to do stuff
- **Imperative programming**: The strings represent *instructions* which are *executed* for their *effect* on the computer's state
- **Functional programming**: The strings are *expressions* which are *evaluated* to obtain *values*

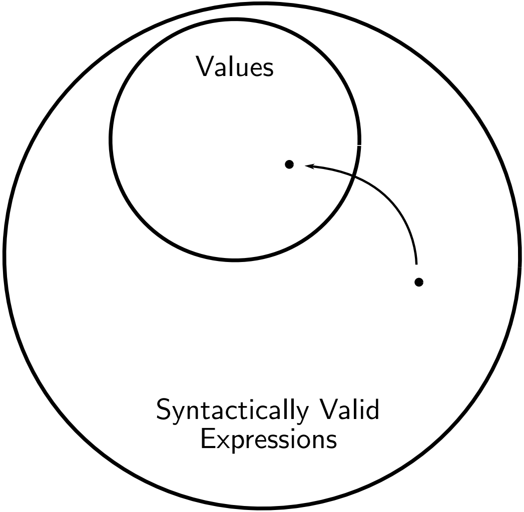
Basic Idea of Functional Programming



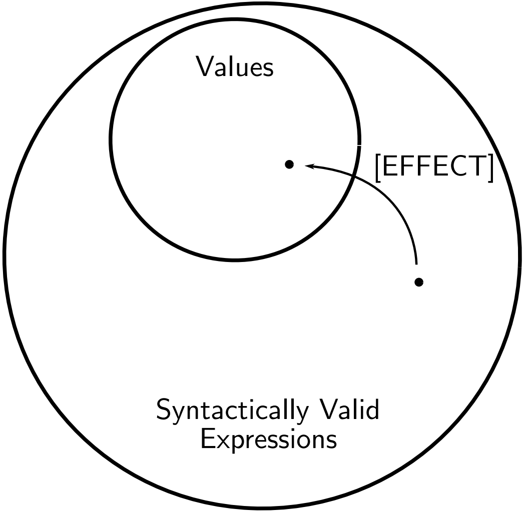
Basic Idea of Functional Programming



Basic Idea of Functional Programming



Basic Idea of Functional Programming



Examples of Evaluation

Examples of Evaluation

- $0 \implies 0$

Examples of Evaluation

- $\texttt{0} \implies \texttt{0}$
- $\texttt{1+1} \implies \texttt{2}$

Examples of Evaluation

- `0` $\Longrightarrow$ `0`
- `1+1` $\Longrightarrow$ `2`
- `if true then 6 else 2` $\Longrightarrow$ `6`

Examples of Evaluation

- `0` $\Longrightarrow$ `0`
- `1+1` $\Longrightarrow$ `2`
- `if true then 6 else 2` $\Longrightarrow$ `6`
- `(fn y => y::[])(NONE)` $\Longrightarrow$ `NONE::[]`

Examples of Evaluation

- `0` $\Longrightarrow$ `0`
- `1+1` $\Longrightarrow$ `2`
- `if true then 6 else 2` $\Longrightarrow$ `6`
- `(fn y => y::[])(NONE)` $\Longrightarrow$ `NONE::[]`
- `map (print o Int.toString) [2,2]` $\Longrightarrow$

Examples of Evaluation

- `0` $\Longrightarrow$ `0`
- `1+1` $\Longrightarrow$ `2`
- `if true then 6 else 2` $\Longrightarrow$ `6`
- `(fn y => y::[])(NONE)` $\Longrightarrow$ `NONE::[]`
- `map (print o Int.toString) [2,2]` $\Longrightarrow$ `[(),()]`

Examples of Evaluation

- `0` $\Longrightarrow$ `0`
- `1+1` $\Longrightarrow$ `2`
- `if true then 6 else 2` $\Longrightarrow$ `6`
- `(fn y => y::[])(NONE)` $\Longrightarrow$ `NONE::[]`
- `map (print o Int.toString) [2,2]` $\Longrightarrow$ `[(),()]`
- `(fn x => x+2+3)` $\Longrightarrow$

Examples of Evaluation

- `0` $\Longrightarrow$ `0`
- `1+1` $\Longrightarrow$ `2`
- `if true then 6 else 2` $\Longrightarrow$ `6`
- `(fn y => y::[])(NONE)` $\Longrightarrow$ `NONE::[]`
- `map (print o Int.toString) [2,2]` $\Longrightarrow$ `[(),()]`
- `(fn x => x+2+3)` $\Longrightarrow$ `(fn x => x+2+3)`

Correctness

But a lot of things can go wrong:

Correctness

But a lot of things can go wrong:

- Type Errors: `0 + false`

Correctness

But a lot of things can go wrong:

- Type Errors: `0 + false`
- Exceptions:
  - `1 div 0`
  - `List.hd []`

Correctness

But a lot of things can go wrong:

- Type Errors: `0 + false`
- Exceptions:
  - `1 div 0`
  - `List.hd []`
- Nontermination: `fact` $\sim$1

Correctness

But a lot of things can go wrong:

- Type Errors: `0 + false`
- Exceptions:
  - `1 div 0`
  - `List.hd []`
- Nontermination: `fact ~1`
- Incorrect return values: `fact 3` $\Longrightarrow$ `41297`

Correctness

But a lot of things can go wrong:

- Type Errors: `0 + false`
- Exceptions:
  - `1 div 0`
  - `List.hd []`
- Nontermination: `fact ~1`
- Incorrect return values: `fact 3` $\Longrightarrow$ `41297`
- Bad effects: Injection attacks because of lack of input sanitization

Section 2

Ensuring Correctness

Solution 1: Runtime checks

Solution 1: Runtime checks

- Idea: Put in pieces of code which make sure everything's going fine, and crash/do something if not

Solution 1: Runtime checks

- Idea: Put in pieces of code which make sure everything's going fine, and crash/do something if not
- Example:

```
if ![ -f $FILE]; then
  exit 1;
fi
```

Solution 1: Runtime checks

- Idea: Put in pieces of code which make sure everything's going fine, and crash/do something if not
- Example:

```
if ![ -f $FILE]; then
  exit 1;
fi
```

- Example: `@requires` and `@ensures` from 122, try/catch, exception handling

Solution 1: Runtime checks

- Idea: Put in pieces of code which make sure everything's going fine, and crash/do something if not
- Example:

```
if ![ -f $FILE]; then
   exit 1;
fi
```

- Example: `@requires` and `@ensures` from 122, try/catch, exception handling
- Advantages: Easy to implement, flexible

Solution 1: Runtime checks

- Idea: Put in pieces of code which make sure everything's going fine, and crash/do something if not
- Example:

```
if ![ -f $FILE]; then
   exit 1;
fi
```

- Example: `@requires` and `@ensures` from 122, try/catch, exception handling
- Advantages: Easy to implement, flexible
- Disadvantages: It's difficult to anticipate every problem (so you can't say for sure there's no problem),

Solution 1: Runtime checks

- Idea: Put in pieces of code which make sure everything's going fine, and crash/do something if not
- Example:

```
if ![ -f $FILE]; then
   exit 1;
fi
```

- Example: `@requires` and `@ensures` from 122, try/catch, exception handling
- Advantages: Easy to implement, flexible
- Disadvantages: It's difficult to anticipate every problem (so you can't say for sure there's no problem), can be disruptive for users,

Solution 1: Runtime checks

- Idea: Put in pieces of code which make sure everything's going fine, and crash/do something if not
- Example:

```
if ![ -f $FILE]; then
   exit 1;
fi
```

- Example: `@requires` and `@ensures` from 122, try/catch, exception handling
- Advantages: Easy to implement, flexible
- Disadvantages: It's difficult to anticipate every problem (so you can't say for sure there's no problem), can be disruptive for users, runtime is too late to catch bugs

Solution 2: Formal Reasoning

Solution 2: Formal Reasoning

- Idea: Mathematically prove that the code works before running it

Solution 2: Formal Reasoning

- Idea: Mathematically prove that the code works before running it
- Example:

```
(* divmod(n,d) == (q,r) such that n=qd+r *)
```

Solution 2: Formal Reasoning

- Idea: Mathematically prove that the code works before running it
- Example:

```
(* divmod(n,d) == (q,r) such that n=qd+r *)
```

- Advantages: Mathematical certainty that the code works,

Solution 2: Formal Reasoning

- Idea: Mathematically prove that the code works before running it
- Example:

```
(* divmod(n,d) == (q,r) such that n=qd+r *)
```

- Advantages: Mathematical certainty that the code works, doesn't require any extra machinery to implement

Solution 2: Formal Reasoning

- Idea: Mathematically prove that the code works before running it
- Example:

```
(* divmod(n,d) == (q,r) such that n=qd+r *)
```

- Advantages: Mathematical certainty that the code works, doesn't require any extra machinery to implement
- Disadvantages: Requires lots of effort and original thought,

Solution 2: Formal Reasoning

- Idea: Mathematically prove that the code works before running it
- Example:

```
(* divmod(n,d) == (q,r) such that n=qd+r *)
```

- Advantages: Mathematical certainty that the code works, doesn't require any extra machinery to implement
- Disadvantages: Requires lots of effort and original thought, requires everyone be fluent in (nontrivial) mathematics,

Solution 2: Formal Reasoning

- Idea: Mathematically prove that the code works before running it
- Example:

```
(* divmod(n,d) == (q,r) such that n=qd+r *)
```

- Advantages: Mathematical certainty that the code works, doesn't require any extra machinery to implement
- Disadvantages: Requires lots of effort and original thought, requires everyone be fluent in (nontrivial) mathematics, most code is too complex to be proven correct

What do we want? Correctness checks!
When do we want it? At compile-time!

We want checks which are

What do we want? Correctness checks! When do we want it? At compile-time!

We want checks which are

- Automated (the programmer doesn't have to sit down and come up with a proof)

What do we want? Correctness checks! When do we want it? At compile-time!

We want checks which are

- Automated (the programmer doesn't have to sit down and come up with a proof)
- Guaranteed (if it passes the check, then the code should be free of whatever bugs we're checking for)

What do we want? Correctness checks! When do we want it? At compile-time!

We want checks which are

- Automated (the programmer doesn't have to sit down and come up with a proof)
- Guaranteed (if it passes the check, then the code should be free of whatever bugs we're checking for)
- Pre-runtime (the checks are completed before the code is run)

What do we want? Correctness checks! When do we want it? At compile-time!

We want checks which are

- Automated (the programmer doesn't have to sit down and come up with a proof)
- Guaranteed (if it passes the check, then the code should be free of whatever bugs we're checking for)
- Pre-runtime (the checks are completed before the code is run)

To do this, we want to implement a system of *compile-time* checks: we annotate the code with information telling the compiler how it works. Then, when we compile the code, the compiler verifies that the code will work as intended.

What do we want? Correctness checks! When do we want it? At compile-time!

We want checks which are

- Automated (the programmer doesn't have to sit down and come up with a proof)
- Guaranteed (if it passes the check, then the code should be free of whatever bugs we're checking for)
- Pre-runtime (the checks are completed before the code is run)

To do this, we want to implement a system of *compile-time* checks: we annotate the code with information telling the compiler how it works. Then, when we compile the code, the compiler verifies that the code will work as intended.

"If it compiles, it works"

The Central Dogma of Hype for Types

Push it to compile time!

Section 3

Type Theory

```
val x : int = 2
val y : int = 3
val b : bool = (x = y)
val s : string = if b
                 then "good"
                 else "bad"
```

```
val x : int = 2
val y : int = 3
val b : bool = (x = y)
val s : string = if b
                   then "good"
                   else "bad"
```

```
val x : int = 2
val y : int = 3
val b : bool = (x = y)
val s : string = if b
                 then "good"
                 else "bad"
```

RULE

if n is an integer, then
the expression n
is of type int

```
val x : int = 2
val y : int = 3
val b : bool =((x:int)= (y:int))
val s : string = if b
                 then "good"
                 else "bad"
```

RULE

if e1:t and e2:t, the expression e1=e2 has type bool

```
val x : int = 2
val y : int = 3
val b : bool =((x:int)= (y:int))
val s : string = if (b:bool)
                 then ("good":string)
                 else ("bad":string)
```

RULE

if e:bool and e1:t and e2:t, then the expression if e then e1 else e2 has type t

Type Systems Have Two Faces

So, we specify a type system for our functional language by giving rules to determine the type of each expression.

Type Systems Have Two Faces

So, we specify a type system for our functional language by giving rules to determine the type of each expression.

This means there's two "sides" to the type system:

- The "practical" side: how the types guarantee features about how the code will evaluate (and how to design compilers that perform this typechecking)
- The "theoretical" side: the logical properties of the type system itself, and what the rules tell us about the relationships between types and expressions

The Theory Side

As an example of the theory side, let's look at the simplest typed functional programming language, the *simply-typed lambda calculus*.

The Theory Side

As an example of the theory side, let's look at the simplest typed functional programming language, the *simply-typed lambda calculus*. The simply-typed lambda calculus includes four things:

The Theory Side

As an example of the theory side, let's look at the simplest typed functional programming language, the *simply-typed lambda calculus*. The simply-typed lambda calculus includes four things:

- Some basic types and expressions of those types
- A unit type
- Product types
- Function types

Section 4

The Simply-Typed Lambda Calculus

A Simple Type System

The simply-typed lambda calculus – true to its name – has a very simple type system. We define it inductively, with two base cases and two binary inductive type constructors.

A Simple Type System

The simply-typed lambda calculus – true to its name – has a very simple type system. We define it inductively, with two base cases and two binary inductive type constructors.

- We assume there are some "basic types" A, B, C, ...

A Simple Type System

The simply-typed lambda calculus – true to its name – has a very simple type system. We define it inductively, with two base cases and two binary inductive type constructors.

- We assume there are some "basic types" $A, B, C, \ldots$
- There is a special type called **unit**

A Simple Type System

The simply-typed lambda calculus – true to its name – has a very simple type system. We define it inductively, with two base cases and two binary inductive type constructors.

- We assume there are some "basic types" A, B, C, ...
- There is a special type called **unit**
- If σ and τ are types, $\sigma * \tau$ is a type

A Simple Type System

The simply-typed lambda calculus – true to its name – has a very simple type system. We define it inductively, with two base cases and two binary inductive type constructors.

- We assume there are some "basic types" A, B, C, $\ldots$
- There is a special type called **unit**
- If σ and τ are types, $\sigma * \tau$ is a type
- If σ and τ are types, $\sigma \to \tau$ is a type

Grammars

To state this concisely, we can give it as a *grammar*:

$$
\begin{aligned}
\sigma, \tau ::= \;& A && \text{(basic types)} \\
\mid\;& \textbf{unit} && \text{(unit)} \\
\mid\;& \sigma * \tau && \text{(product types)} \\
\mid\;& \sigma \to \tau && \text{(arrow types)}
\end{aligned}
$$

Contexts

In functional programming languages, the value (and the type) of expressions depends on the *context*. For instance, consider the following code.

Contexts

In functional programming languages, the value (and the type) of expressions depends on the *context*. For instance, consider the following code.

```
val x :  int = 2
val y = x + x
```

Contexts

In functional programming languages, the value (and the type) of expressions depends on the *context*. For instance, consider the following code.

```
val x :  int = 2
val y = x + x
```

As written, the variable `y` here gets bound to `4:int`.

Contexts

In functional programming languages, the value (and the type) of expressions depends on the *context*. For instance, consider the following code.

```
val x : int = 2
val y = x + x
```

As written, the variable `y` here gets bound to `4:int`.But if we replaced the first line with

```
val x : real = 3.0
```

then `y` would get bound to `6.0:real`. This is what we mean when we say that the type and value of an expression depend on the context.

Judgements and Contexts

For some syntactically-valid expression $\mathtt{x}$ and some type τ, we call this string of symbols

$$\mathtt{x} : \tau$$

a "**typing judgement**". We read that as "$\mathtt{x}$ is of type τ".

Judgements and Contexts

For some syntactically-valid expression $\mathtt{x}$ and some type τ, we call this string of symbols

$$\mathtt{x} : \tau$$

a "**typing judgement**". We read that as "$\mathtt{x}$ is of type τ".
A **context** Γ is just a finite list of typing judgements: the variables we've bound so far.

$$\Gamma = x_1 : \tau_1, \ldots, x_n : \tau_n$$

Terms-in-context

With this, we can specify how the terms of the lambda calculus are formed. Consider the following example:

```
val y = (x,x)
```

Terms-in-context

With this, we can specify how the terms of the lambda calculus are formed. Consider the following example:

```
val y = (x,x)
```

This variable binding needs (at least) one variable in its context, `x`. If `x` : τ, then `y` : $\tau * \tau$.

Terms-in-context

With this, we can specify how the terms of the lambda calculus are formed. Consider the following example:

```
val y = (x,x)
```

This variable binding needs (at least) one variable in its context, $\mathtt{x}$. If $\mathtt{x} : \tau$, then $\mathtt{y} : \tau * \tau$. We indicate this more formally by writing:

$$\mathtt{x} : \tau \vdash \mathtt{(x,x)} : \tau * \tau$$

Terms-in-context

With this, we can specify how the terms of the lambda calculus are formed. Consider the following example:

```
val y = (x,x)
```

This variable binding needs (at least) one variable in its context, $\mathtt{x}$. If $\mathtt{x} : \tau$, then $\mathtt{y} : \tau * \tau$. We indicate this more formally by writing:

$$\mathtt{x} : \tau \vdash \mathtt{(x,x)} : \tau * \tau$$

We say that $\mathtt{(x,x)}$ is a term of type $\tau * \tau$, in context $\mathtt{x} : \tau$. The lambda calculus is specified using these *terms-in-context*.

Example: Pairing

The following is a term-formation rule of the lambda calculus:

- If Γ is some context such that $\mathtt{x} : \sigma$ in context Γ and $\mathtt{y} : \tau$ in context Γ, then $\mathtt{(x,y)} : \sigma * \tau$ in context Γ.

Example: Pairing

The following is a term-formation rule of the lambda calculus:

- If Γ is some context such that $\mathtt{x} : \sigma$ in context Γ and $\mathtt{y} : \tau$ in context Γ, then $(\mathtt{x},\mathtt{y}) : \sigma * \tau$ in context Γ.

We indicate this more compactly using an **inference rule**.

$$\frac{\Gamma \vdash \mathtt{x} : \sigma \qquad \Gamma \vdash \mathtt{y} : \tau}{\Gamma \vdash (\mathtt{x},\mathtt{y}) : \sigma * \tau}$$

Example: Pairing

The following is a term-formation rule of the lambda calculus:

- If Γ is some context such that $\mathtt{x} : \sigma$ in context Γ and $\mathtt{y} : \tau$ in context Γ, then $(\mathtt{x},\mathtt{y}) : \sigma * \tau$ in context Γ.

We indicate this more compactly using an **inference rule**.

$$\frac{\Gamma \vdash \mathtt{x} : \sigma \qquad \Gamma \vdash \mathtt{y} : \tau}{\Gamma \vdash (\mathtt{x},\mathtt{y}) : \sigma * \tau}$$

This gives an indication of how we'd recursively implement a lambda calculus typechecker: in order to verify that the expression $(\mathtt{x},\mathtt{y})$ is indeed of type $\sigma * \tau$, we just need to check that $\mathtt{x}$ is of type σ and $\mathtt{y}$ is of type τ.

Some rules of the lambda calculus

There are a lot of rules specifying how the lambda calculus works. Here are some highlights.

Some rules of the lambda calculus

There are a lot of rules specifying how the lambda calculus works. Here are some highlights.

$$\frac{}{\Gamma \vdash () : \textbf{unit}}$$

Some rules of the lambda calculus

There are a lot of rules specifying how the lambda calculus works. Here are some highlights.

$$\overline{\Gamma \vdash () : \textbf{unit}}$$

This says that, with no assumptions, in any context, () is of type **unit**.

Some rules of the lambda calculus

There are a lot of rules specifying how the lambda calculus works. Here are some highlights.

$$\frac{}{\Gamma \vdash () : \textbf{unit}}$$

This says that, with no assumptions, in any context, () is of type **unit**.

$$\frac{\Gamma \vdash p : \sigma * \tau}{\Gamma \vdash \mathsf{fst}(p) : \sigma} \qquad \frac{\Gamma \vdash p : \sigma * \tau}{\Gamma \vdash \mathsf{snd}(p) : \tau}$$

Some rules of the lambda calculus

$$\frac{\Gamma, x : \sigma \vdash e(x) : \tau}{\Gamma \vdash (\lambda x.e(x)) : \sigma \to \tau}$$

Some rules of the lambda calculus

$$\frac{\Gamma, x : \sigma \vdash e(x) : \tau}{\Gamma \vdash (\lambda x. e(x)) : \sigma \to \tau}$$

$$\frac{\Gamma \vdash f : \sigma \to \tau \qquad \Gamma \vdash t : \sigma}{\Gamma \vdash (ft) : \tau}$$

Type Practice

Type Practice

- What if we had more than just arrows, unit and products? (Algebraic Datatypes)

Type Practice

- What if we had more than just arrows, unit and products? (Algebraic Datatypes)
- What if we had a type-theoretic way of distinguishing sanitized input from unsantized input, and other similar distinctions? What if we had a way to encode *in the types* that `List.hd` cannot be called on `[]`? (Phantom Types and Generalized Algebraic Datatypes)

Type Practice

- What if we had more than just arrows, unit and products? (Algebraic Datatypes)
- What if we had a type-theoretic way of distinguishing sanitized input from unsantized input, and other similar distinctions? What if we had a way to encode *in the types* that `List.hd` cannot be called on `[]`? (Phantom Types and Generalized Algebraic Datatypes)
- What if we had a way of proving (in a way that could be verified by the typechecker) that our code must meet a certain spec? (Interactive Theorem Proving)

The theory of types is also interesting in its own right!

Type Theory

The theory of types is also interesting in its own right!

- Type theory shares an intimate connection to intuitionistic logic (Curry-Howard)

Type Theory

The theory of types is also interesting in its own right!

- Type theory shares an intimate connection to intuitionistic logic (Curry-Howard)
- We can give a fascinating mathematical treatment of typed functional programming (Category Theory)

Type Theory

The theory of types is also interesting in its own right!

- Type theory shares an intimate connection to intuitionistic logic (Curry-Howard)
- We can give a fascinating mathematical treatment of typed functional programming (Category Theory)
- We can give a fascinating type-theoretic treatment of mathematics (Homotopy Type Theory)

Type Theory

The theory of types is also interesting in its own right!

- Type theory shares an intimate connection to intuitionistic logic (Curry-Howard)
- We can give a fascinating mathematical treatment of typed functional programming (Category Theory)
- We can give a fascinating type-theoretic treatment of mathematics (Homotopy Type Theory)

So stay tuned!

Thank you!