# Lecture 7: Types and Typechecking

Programming Languages (H)

**Simon Fowler** & Michele Sevegnani

Semester 2, 2024/2025

sli.do 3400724

sli.do 3400724

## Overview

- **Last time**
  - Variables, scope, and let-binding
- **This lecture:** Types and Typechecking
  - Types + Typing rules
  - How to write a simple typechecker
  - Static vs. dynamic typing

sli.do 3400724

## Where are we up to?

- Up to now, we've seen how to implement a programming language that includes binary operations, conditionals, functions, and recursion
- We've also seen how to formally specify the **syntax** and **semantics**, and shown how the formal specifications can be implemented in practice
- However, we've not really looked at the **correctness** of programs: how can we avoid common programming mistakes?

sli.do 3400724

# Recap: $L_{Rec}$ Abstract Syntax

**$L_{Lam}$ Abstract Syntax**

```
Integers n
Variables x, y, z
Operators ⊙  ::= + | - | *  | /
Values V, W   ::= n | λx . M
Terms L, M, N ::= x | n
                | λx . M
                | rec f(x)
                | M N
                | L ⊙ M
```

- Last time we looked at $L_{Rec}$, a core language with anonymous functions and anonymous recursive functions
- We also saw how let / let-fun / let-rec could be encoded

sli.do 3400724

## Question from Week 2:

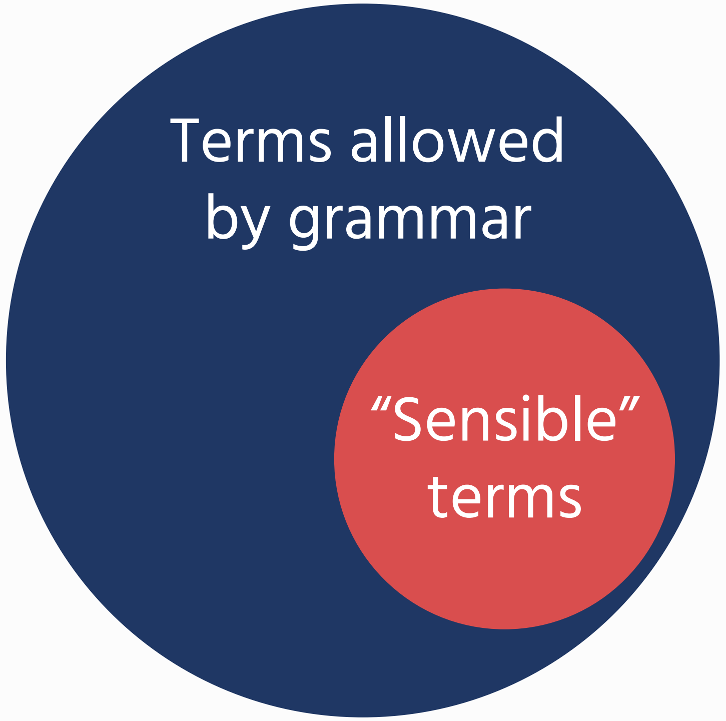
"Do you need an extra operational semantics rule if you're trying to evaluate `if 5 then true else false`?"

sli.do 3400724

## Let's try out some examples…

sli.do 3400724

## Our grammar permits nonsensical terms!



$$\text{true } 5$$

$$(\lambda x.\, x + 10)\ \text{false}$$

$$((\lambda x.\, x)\ \text{false})\ \text{true}$$

$$(10 + 15) + \text{true}$$

We want a **lightweight** way of **only talking about sensible terms**

**Example**: "We can only add integers together, and the result is an integer"

sli.do 3400724

## Some languages try their best anyway…

- A big design mistake historically has been to try and "make up" a semantics for nonsensical terms
  - This gives rise to things like "truthy" values being OK in if-expressions, or implicit type conversions
- In general this isn't a good idea: if a semantics isn't obvious then it's likely that there will be weird edge cases and behaviours
  - For example: `"hello" + 1 = "hello1"` in JavaScript, but `"hello" - 1 = NaN`
  - These edge cases then need to be supported for backwards compatibility

sli.do 3400724

## Typing Rules, Informally

- Let's consider $L_{Arith}$, where we have integer literals *n* and binary operators $M \odot N$, and also add Boolean literals true and false
- We want to rule out any "nonsensical" terms, for example adding an integer to a Boolean
- This needs to work for nested expressions, too, so we need some way of **classifying** expressions

"Integer literals have type Int"

"Boolean literals have type Bool"

"If $M$ has type Int and $N$ has type Int, then $M + N$ has type Int"

sli.do 3400724

## We need formal descriptions!

- Again, informal descriptions run the risk of ambiguity: they often introduce annoying edge cases that aren't clear from the prose
- We'll also want to prove properties about our type system (for example, that well typed programs don't introduce runtime errors)
- Additionally, for our operational semantics there was a fairly direct mapping between the formal rules and the implementations
- Can we describe our type systems in a similar way?

sli.do 3400724

## Types

Types classify terms. Let's begin by considering **base types**: Booleans and Integers

$$\text{Types} \quad A, B \; ::= \text{Int} \mid \text{Bool}$$

$\boxed{\vdash M : A}$ "Term M has type A"

$$\frac{}{\vdash n : \text{Int}}$$

"Integer literals have type Int"

$$\frac{}{\vdash b : \text{Bool}}$$

"Boolean literals have type Bool"

$$\frac{\vdash M : \text{Int} \qquad \vdash N : \text{Int}}{\vdash M + N : \text{Int}}$$

"If $M$ has type Int and $N$ has type Int, then $M + N$ has type Int"

sli.do 3400724

## Example: Addition (Well-typed)

$$\frac{}{\vdash n : \mathrm{Int}} \qquad \frac{}{\vdash b : \mathrm{Bool}} \qquad \frac{\vdash M : \mathrm{Int} \qquad \vdash N : \mathrm{Int}}{\vdash M + N : \mathrm{Int}}$$

$$\frac{}{\vdash (10 + 15) + 20 : ?}$$

sli.do 3400724

## Example: Addition (Well-typed)

$$\frac{}{\vdash n : \mathrm{Int}} \qquad \frac{}{\vdash b : \mathrm{Bool}} \qquad \frac{\vdash M : \mathrm{Int} \qquad \vdash N : \mathrm{Int}}{\vdash M + N : \mathrm{Int}}$$

$$\frac{\dfrac{}{\vdash 10 + 15 : ?} \qquad \dfrac{}{\vdash 20 : \mathrm{Int}}}{\vdash (10 + 15) + 20 : ?}$$

sli.do 3400724

## Example: Addition (Well-typed)

$$\frac{}{\vdash n : \mathrm{Int}} \quad \frac{}{\vdash b : \mathrm{Bool}} \quad \frac{\vdash M : \mathrm{Int} \quad \vdash N : \mathrm{Int}}{\vdash M + N : \mathrm{Int}}$$

$$\frac{\dfrac{\dfrac{}{\vdash 10 : \mathrm{Int}} \quad \dfrac{}{\vdash 15 : \mathrm{Int}}}{\vdash 10 + 15 : ?} \quad \dfrac{}{\vdash 20 : \mathrm{Int}}}{\vdash (10 + 15) + 20 : ?}$$

sli.do 3400724

## Example: Addition (Well-typed)

$$\frac{}{\vdash n : \text{Int}} \qquad \frac{}{\vdash b : \text{Bool}} \qquad \frac{\vdash M : \text{Int} \qquad \vdash N : \text{Int}}{\vdash M + N : \text{Int}}$$

$$\dfrac{\dfrac{\dfrac{}{\vdash 10 : \text{Int}} \qquad \dfrac{}{\vdash 15 : \text{Int}}}{\vdash 10 + 15 : \text{Int}} \qquad \dfrac{}{\vdash 20 : \text{Int}}}{\vdash (10 + 15) + 20 : ?}$$

sli.do 3400724

## Example: Addition (Well-typed)

$$\frac{}{\vdash n : \mathrm{Int}} \qquad \frac{}{\vdash b : \mathrm{Bool}} \qquad \frac{\vdash M : \mathrm{Int} \qquad \vdash N : \mathrm{Int}}{\vdash M + N : \mathrm{Int}}$$

$$\dfrac{\dfrac{\dfrac{}{\vdash 10 : \mathrm{Int}} \qquad \dfrac{}{\vdash 15 : \mathrm{Int}}}{\vdash 10 + 15 : \mathrm{Int}} \qquad \dfrac{}{\vdash 20 : \mathrm{Int}}}{\vdash (10 + 15) + 20 : \mathrm{Int}}$$

sli.do 3400724

## Example: Addition (Type error)

$$\frac{}{\vdash n : \mathrm{Int}} \quad \frac{}{\vdash b : \mathrm{Bool}} \quad \frac{\vdash M : \mathrm{Int} \qquad \vdash N : \mathrm{Int}}{\vdash M + N : \mathrm{Int}}$$

$$\frac{}{\vdash (10 + 15) + \mathrm{true} : ?}$$

sli.do 3400724

## Example: Addition (Ill-typed)

$$\frac{}{\vdash n : \mathrm{Int}} \quad \frac{}{\vdash b : \mathrm{Bool}} \quad \frac{\vdash M : \mathrm{Int} \qquad \vdash N : \mathrm{Int}}{\vdash M + N : \mathrm{Int}}$$

$$\frac{\dfrac{}{\vdash 10 + 15 : ?} \qquad \dfrac{}{\vdash \mathrm{true} : \mathrm{Bool}}}{\vdash (10 + 15) + \mathrm{true} : ?}$$

sli.do 3400724

## Example: Addition (Ill-typed)

$$\frac{}{\vdash n : \mathrm{Int}} \qquad \frac{}{\vdash b : \mathrm{Bool}} \qquad \frac{\vdash M : \mathrm{Int} \qquad \vdash N : \mathrm{Int}}{\vdash M + N : \mathrm{Int}}$$

$$\frac{\dfrac{\dfrac{}{\vdash 10 : \mathrm{Int}} \qquad \dfrac{}{\vdash 15 : \mathrm{Int}}}{\vdash 10 + 15 : ?} \qquad \dfrac{}{\vdash \mathrm{true} : \mathrm{Bool}}}{\vdash (10 + 15) + \mathrm{true} : ?}$$

sli.do 3400724

## Example: Addition (Ill-typed)

$$\frac{}{\vdash n : \mathrm{Int}} \quad \frac{}{\vdash b : \mathrm{Bool}} \quad \frac{\vdash M : \mathrm{Int} \qquad \vdash N : \mathrm{Int}}{\vdash M + N : \mathrm{Int}}$$

$$\frac{\dfrac{\dfrac{}{\vdash 10 : \mathrm{Int}} \qquad \dfrac{}{\vdash 15 : \mathrm{Int}}}{\vdash 10 + 15 : \mathrm{Int}} \qquad \dfrac{}{\vdash \mathrm{true} : \mathrm{Bool}}}{\vdash (10 + 15) + \mathrm{true} : ?}$$

sli.do 3400724

## Example: Addition (Ill-typed)

$$\frac{}{\vdash n : \mathrm{Int}} \quad \frac{}{\vdash b : \mathrm{Bool}} \quad \frac{\vdash M : \mathrm{Int} \qquad \vdash N : \mathrm{Int}}{\vdash M + N : \mathrm{Int}}$$

$$\frac{\dfrac{\dfrac{}{\vdash 10 : \mathrm{Int}} \qquad \dfrac{}{\vdash 15 : \mathrm{Int}}}{\vdash 10 + 15 : \mathrm{Int}} \qquad \dfrac{}{\vdash \mathrm{true} : \mathrm{Bool}}}{\nvdash (10 + 15) + \mathrm{true} : ?}$$

sli.do 3400724

# Generalising to other binary operators

- So far we've seen only the rule for addition: what about other operators?

$$\mathrm{ty}(\odot) = A \to B \to C$$

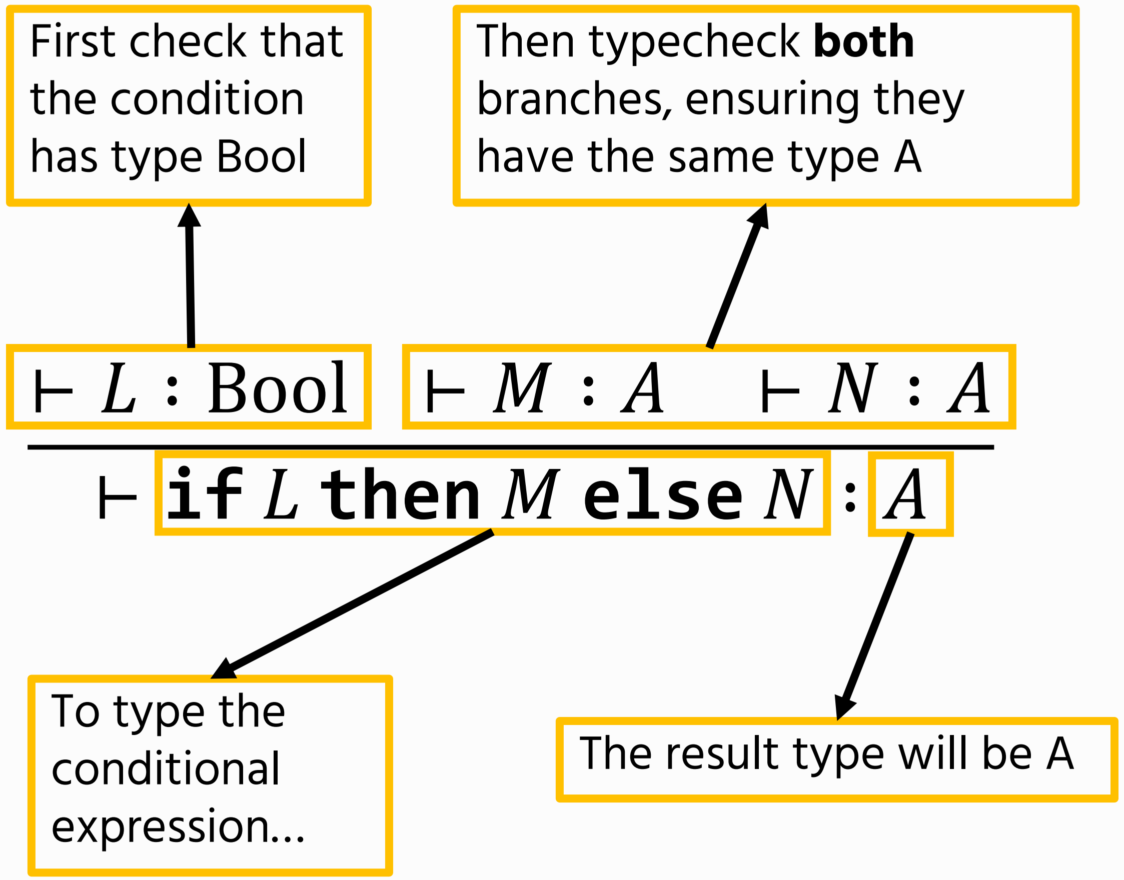
$$\frac{\vdash M : A \quad \vdash N : B}{\vdash M \odot N : \mathrm{Int}}$$

```
ty(+) = Int -> Int -> Int
ty(-) = Int -> Int -> Int
ty(*) = Int -> Int -> Int
ty(/) = Int -> Int -> Int
ty(&&) = Bool -> Bool -> Bool
ty(||) = Bool -> Bool -> Bool
ty(<) = Int -> Int -> Bool
ty(>) = Int -> Int -> Bool
ty(==) = A -> A -> Bool
  (if A = Int or A = Bool)
```

- ty(-) is an example of a **meta-level function**: a function we're defining when describing our language (rather than a function in the language itself)

sli.do 3400724

# Typing Conditionals


First check that the condition has type Bool
Then typecheck **both** branches, ensuring they have the same type A
To type the conditional expression…
The result type will be A

$$\frac{\vdash L : \mathrm{Bool} \qquad \vdash M : A \qquad \vdash N : A}{\vdash \texttt{if}\ L\ \texttt{then}\ M\ \texttt{else}\ N : A}$$

- Note that both the then and the else branches must have the same type
- This is because if is an **expression** and must evaluate to a result, and we need to give the overall expression a type
- Some type systems (e.g. the one used for TypeScript) *do* allow different types in each branch, but this requires heavier machinery (set-theoretic types)

sli.do 3400724

# Functions: Additional Types and Syntax

First, we need a **function type**, expressing a function from type A to type B

$$A, B ::= \boxed{A \to B} \mid \mathrm{Int} \mid \mathrm{Bool}$$

$$L, M, N ::= x \mid \boxed{\lambda x^{A}.M} \mid M\ N \mid \cdots$$

Next, we need a **type annotation** on the parameter in the function expressions

sli.do 3400724

## Typechecking functions: First attempt

$$\frac{}{\vdash \lambda x^{\mathrm{Int}}.\, x + 1 : ?}$$

sli.do 3400724

## Typechecking functions: First attempt

$$\dfrac{\dfrac{}{\vdash x + 1 :?}}{\vdash \lambda x^{\mathrm{Int}}. x + 1 : ?}$$

sli.do 3400724

## Typechecking functions

How do we determine the type of a variable?

$$\dfrac{\dfrac{\boxed{\dfrac{}{\vdash x : ?}} \qquad \dfrac{}{\vdash 1 : \mathrm{Int}}}{\vdash x + 1 : ?}}{\vdash \lambda x^{\mathrm{Int}}.\, x + 1 : ?}$$

sli.do 3400724

## Type Environments

- To handle variables, we need to store their types in a **type environment** (sometimes also called a **typing context**)
- A type environment (normally written as the Greek letter Γ), is a mapping from variables to types

$$\boxed{\Gamma ::= \cdot \mid \Gamma, x : A}$$

- Whenever we encounter a binding occurrence of a variable, we add it to the type environment so that we can look up the type later
- The other rules (e.g. binary operators and conditionals) also need to be updated to use environments

sli.do 3400724

## Typechecking functions

$$\frac{}{\cdot \vdash \lambda x^{\mathrm{Int}}.\, x + 1 : ?}$$

sli.do 3400724

## Typechecking functions

$$\frac{\overline{x\colon \mathrm{Int} \vdash x + 1 \; :?}}{\cdot \vdash \lambda x^{\mathrm{Int}}.\, x + 1 \colon ?}$$

sli.do 3400724

## Typechecking functions

$$\dfrac{\dfrac{\dfrac{}{x\colon \mathrm{Int} \vdash x : \mathrm{Int}} \qquad \dfrac{}{x\colon \mathrm{Int} \vdash 1 : \mathrm{Int}}}{x\colon \mathrm{Int} \vdash x + 1 : \mathrm{Int}}}{\cdot \vdash \lambda x^{\mathrm{Int}}.\, x + 1 \colon \mathrm{Int}}$$

sli.do 3400724

## Typechecking functions

$$\dfrac{\dfrac{\dfrac{}{x{:}\,\mathrm{Int} \vdash x : \mathrm{Int}} \qquad \dfrac{}{x{:}\,\mathrm{Int} \vdash 1 : \mathrm{Int}}}{x{:}\,\mathrm{Int} \vdash x + 1 : \mathrm{Int}}}{\cdot \vdash \lambda x^{\mathrm{Int}}.\, x + 1 : \mathrm{Int} \to \mathrm{Int}}$$

sli.do 3400724

# Typing Rules for Variables & Functions

$$\boxed{\Gamma \vdash M : A}$$

"Under typing environment $\Gamma$, term $M$ has type $A$."

$$\Gamma ::= \cdot \mid \Gamma, x : A$$

| Typing environment knows type x has type A | Extend the type environment with x of type A | Under extended environment, body has type B | Function has type $A \to B$ | Argument has matching type $A$ |
|---|---|---|---|---|

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x^A . M : A \to B}$$

$$\frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash M\,N : B}$$

Then, we can say that x has type A

Then, function has type $A \to B$

Then applied function has result type B

sli.do 3400724

## Recursive Functions

Extend environment Γ with:

- $f : A \to B$
- $x : A$

and check that M has type B

$$\frac{\boxed{\Gamma, f : A \to B, x : A \vdash M : B}}{\boxed{\Gamma \vdash \mathbf{rec}\ f(x{:}\,A){:}\,B\,.\,M} : \boxed{A \to B}}$$

To typecheck a recursive function f with argument type A and return type B…

The resulting type is the function type $A \to B$

sli.do 3400724

## Implementing a Typechecker

- The goal of typechecking is to determine whether an expression is **typable** (i.e., accepted by the typing rules of the system)
- Essentially this boils down to implementing the following function (as you will do in the lab)

```
public Type(Environment<Type>, Expr e)
  throws TypeErrorException
```

- You may also hear the term "contextual analysis" in some of the course materials on the Fun language (to be discussed next week), but this term is no longer in common use

sli.do 3400724

## Implementing Type Environments

- A type environment is essentially a map from variables (strings) to types
- Maps in Java are **mutable**: extending the map changes the underlying data, which is is fiddly with scope
- Instead, we'll use an **immutable** map: extending an environment returns a **copy** that includes the new mapping
- The map is **generic** in the type of value each variable refers to, but we'll consider `Environment<Type>`

```
public class Environment<T> {
    // Internal (mutable) map
    private Map<String, T> env;

    // Create new environment
    public Environment() { ... }

    // Variable lookup
    public T lookup(String var)
      throws TypeErrorException { ... }

    // Return new environment containing
    // new mapping
    public Environment<T>
       extend(String var, T t) { ... }
}
```

sli.do 3400724

# Example: Typechecking Lambdas

$$\frac{\boxed{\boxed{\Gamma, x : A} \vdash M : B}}{\boxed{\Gamma \vdash \lambda x^{A}. M} : \boxed{A \rightarrow B}}$$

```
public Type typecheck(
    Environment<Type> env, Expr e)
    throws TypeErrorException {
    …
  } else if (e instanceof ELambda elam) {
    Environment<Type> bodyEnv =
      env.extend(elam.getBinder(),
                 elam.getAnnotation());
    Type bodyType =
      typecheck(bodyEnv, elam.getBody());
    return new TyFun(elam.getAnnotation(),
                     bodyType);
  }
```

**Key idea:** Use annotation to extend type environment before recursively checking body.

Result type is a function from the annotated parameter type to the body type.

sli.do 3400724

# Example: Typechecking Conditionals

$$\frac{\Gamma \vdash L : \mathrm{Bool} \qquad \Gamma \vdash M : A \qquad \Gamma \vdash N : A}{\Gamma \vdash \textbf{if}\ L\ \textbf{then}\ M\ \textbf{else}\ N : A}$$

```
public Type typecheck(Environment<Type> env, Expr e)
    throws TypeErrorException {
   …
   } else if (e instanceof ECond eif) {
      Type predType =
        typecheck(env, eif.getPredicate());
      if (!predType.equals(TyBool.type())) {
        throw new TypeErrorException("Predicate must have
type Bool");
      }
      Type thenType =
        typecheck(env, eif.getThenBranch());
      Type elseType =
        typecheck(env, eif.getElseBranch());

      if (!thenType.equals(elseType)) {
        throw new TypeErrorException("Both branches of an
if-expression must have the same type");
      }
      return thenType;
}
```

**Key idea:** Check predicate is a Bool. After that, typecheck both branches and check to see if they have the same type. Return that type as the result.

sli.do 3400724

## Static vs. Dynamic Typechecking

- So far, we've considered **static** typechecking, where all typechecking happens **before** a program is run
- Alternatively, **dynamically-checked** languages (e.g. Python) tag values with their types, and perform typechecking dynamically to avoid crashes
- The main benefits of static typechecking are early error detection, the ability to check all code paths without running them, and the lower memory / runtime due to absence of dynamic checks
- The main benefit of dynamic typechecking is flexibility, allowing e.g. branching control flow where branches have different types.

sli.do 3400724

## Type Inference (non-examinable)

- Our typechecker requires **type annotations** on function parameters
  - Many languages (e.g., Java / Scala) require this by default, too
- It is also possible to design and implement a type system that does **not** need these annotations, and works by **inferring** the types
- Type inference is widely used in the ML family of languages, as well as languages like Haskell
- Roughly speaking, type inference works by:
  - Introducing **type variables** that stand for an unresolved type
  - Generating **constraints** between types
  - Solving the sets of constraints through a technique called **unification**

sli.do 3400724

## More interesting type systems (non-examinable)

We have looked at a type system with **simple types**. Designing and implementing type systems is an exciting field (the one I work in) and there are many interesting extensions:

### Polymorphism

$$id : \forall X. X \to X$$

$$id[\mathrm{Int}] : \mathrm{Int} \to \mathrm{Int}$$

$$id[\mathrm{Bool}] : \mathrm{Bool} \to \mathrm{Bool}$$

**Polymorphism** allows us to write functions that work given many different types

### Dependent types

```
append:
   Vect n a ->
   Vect m a ->
   Vect (n + m) a
```

**Dependent types** allow types to contain values, so we can write much more specific types

### Substructural types

$$x : A \vdash x : A$$

**Substructural type systems** constrain how variables can be used (e.g. only once) and are good for encoding resource usage constraints

sli.do 3400724

## Today's lab

- In today's lab you'll implement your very own typechecker!
- The lab sheet contains the full set of rules you will need to implement
    - For practice, I've also given the explicit rules for let, let-fun, and let-rec rather than relying on desugaring
- Please come to the lab! We're also happy to help with the W2 and W3 labs if you haven't had a chance to look at those yet

sli.do 3400724

## Conclusion

- **This lecture:**
  - Types
  - Formal description of typing rules
  - How to write a type checker
  - Static vs. dynamic typing, type inference, and more exotic type systems
- **In 10 minutes or so:**
  - Type soundness: what does a type system allow us to guarantee?
  - Small-step operational semantics