# Type Inference

## Concepts of Programming Languages
## Lecture 21

CAS CS 320

# Outline

- Discuss **type inference** with eye towards **Hindley–Milner typing**
- Look at a set of typing rules for **constraint-based inference**
- Walk through some **examples**

# Recap

# Explicit Typing

```
let add (x : int) (y : int) : int = x + y
let k (x : int) (y : bool) : int = x
let _ : unit = assert(add 2 3 = k 5 false)
```

# Explicit Typing

```
let add (x : int) (y : int) : int = x + y
let k (x : int) (y : bool) : int = x
let _ : unit = assert(add 2 3 = k 5 false)
```

In mini-project 2, we're implementing a PL with **explicit typing**

# Explicit Typing

```
let add (x : int) (y : int) : int = x + y
let k (x : int) (y : bool) : int = x
let _ : unit = assert(add 2 3 = k 5 false)
```

In mini-project 2, we're implementing a PL with **explicit typing**

Every function argument and let-expression is annotated with typing information

# Explicit Typing

```
let add (x : int) (y : int) : int = x + y
let k (x : int) (y : bool) : int = x
let _ : unit = assert(add 2 3 = k 5 false)
```

In mini-project 2, we're implementing a PL with **explicit typing**

Every function argument and let-expression is annotated with typing information

This is closer to what is done in a PL like **Java**

# Implicit Types

```
let add x y = x + y
let k x y = x
let _ = assert(add 2 3 = k 5 false)
```

# Implicit Types

```
let add x y = x + y
let k x y = x
let _ = assert(add 2 3 = k 5 false)
```

We rarely have to specify types in OCaml

# Implicit Types

```
let add x y = x + y
let k x y = x
let _ = assert(add 2 3 = k 5 false)
```

We rarely have to specify types in OCaml

Type inference, or type *reconstruction* is the process of determining what type we *could* have annotated our program with

# Implicit Types

'a 'b. 'a -> 'b -> 'a

```
let add x y = x + y
let k x y = x
let _ = assert(add 2 3 = k 5 false)
```

int bool

int -> bool -> int

We rarely have to specify types in OCaml

Type inference, or type *reconstruction* is the process of determining what type we *could* have annotated our program with

*But what type should we give* ***k****?*

# Recall: Parametric Polymorphism

```
let rec rev = function
  | [] -> []
  | x :: xs -> rev xs @ [x]
```

# Recall: Parametric Polymorphism

```
let rec rev = function
  | [] -> []
  | x :: xs -> rev xs @ [x]
```

**Parametric polymorphism** allows for functions which are agnostic to the types of its inputs

# Recall: Parametric Polymorphism

```
let rec rev = function
  | [] -> []
  | x :: xs -> rev xs @ [x]
```

**Parametric polymorphism** allows for functions which are agnostic to the types of its inputs

*For example, we can write a single reverse function and use it in multiple contexts*

# Recall: Type Variables

```
let id : 'a -> 'a = fun x -> x
```

# Recall: Type Variables

```
let id : 'a -> 'a = fun x -> x
```

The "parametric" part is the fact that types have *variables*

# Recall: Type Variables

```
let id : 'a -> 'a = fun x -> x
```

The "parametric" part is the fact that types have *variables*

**Type variables** are instantiated at particular types according to the context

# Recall: Type Variables

```
let id : 'a -> 'a = fun x -> x
```

The "parametric" part is the fact that types have *variables*

**Type variables** are instantiated at particular types according to the context

They are very similar to expression variables, e.g., we need to define *type-level substitution*

# Recall: Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

# Recall: Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

`In reality, types variables in OCaml are` **`quantified`**

# Recall: Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

In reality, types variables in OCaml are **quantified**

Just like with expression variables, we don't like *unbound* type variables

# Recall: Quantification

```
let id : 'a . 'a -> 'a = fun x -> x
```

$\forall \alpha.\ \alpha \to \alpha$

In reality, types variables in OCaml are **quantified**

Just like with expression variables, we don't like *unbound* type variables

We read this "**id** has type **t -> t** for any type **t**"

# Recall: System F

```
let id_int : int -> int = fun (x : int) -> x
let id : 'a . 'a -> 'a = fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

# Recall: System F

```
let id_int : int -> int = fun (x : int) -> x
let id : 'a . 'a -> 'a = fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

**System F** (**second-order lambda calculus**) was introduced by Jean-Yves Girard and John C. Reynolds in the 1970s

# Recall: System F

```
let id_int : int -> int = fun (x : int) -> x
let id : 'a . 'a -> 'a = fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

**System F (second-order lambda calculus)** was introduced by Jean-Yves Girard and John C. Reynolds in the 1970s

**The basic idea:** Introduce types into the language itself so we can *pass them as arguments to functions*

# Recall: System F

```
let id_int : int -> int = fun (x : int) -> x
let id : 'a . 'a -> 'a = fun 'a -> fun (x : 'a) -> x

let test1 = id_int 2
let test2 = id int 2
let test3 = id string "two"
```

**System F (second-order lambda calculus)** was introduced by Jean-Yves Girard and John C. Reynolds in the 1970s

**The basic idea:** Introduce types into the language itself so we can *pass them as arguments to functions*

**The big problem: Without type annotations type checking is undecidable**

# Interlude:
# Compact Derivations

# The Problem

Derivations take up a lot of horizontal space

We've been careful to choose expressions with short derivations in lecture

*We won't be able to do this moving forward*

# The Problem

$$\dfrac{\dfrac{}{\{\} \vdash \texttt{2} : \texttt{int}}\,(\text{intLit}) \qquad \dfrac{\dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\,(\text{var}) \qquad \dfrac{}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} : \texttt{int}}\,(\text{var})}{\{\texttt{y} : \texttt{int}\} \vdash \texttt{y} + \texttt{y} : \texttt{int}}\,(\text{intAdd})}{\{\} \vdash \texttt{let y = 2 in } \texttt{y} + \texttt{y} : \texttt{int}}\,(\text{let})$$

Derivations take up a lot of horizontal space

We've been careful to choose expressions with short derivations in lecture

*We won't be able to do this moving forward*

# Visualizing Trees

```
.
├── bin
│   ├── dune
│   └── main.ml
├── dune-project
├── interp2.opam
├── lib
│   ├── dune
│   ├── interp2.ml
│   ├── lexer.mll
│   ├── parser.mly
│   └── utils.ml
├── spec.pdf
└── test
    ├── dune
    └── test_interp2.ml
```

There are many ways of drawing trees. Finding a "good" visualization of trees is an art

Moving forward we'll use the *file-tree format* for writing derivations (this is what is done in the textbook)

*It's more horizontally space-efficient*

# Example

# Example

$$\dfrac{\dfrac{}{\{\} \vdash 2 : \mathtt{int}}\,(\text{intLit}) \qquad \dfrac{\dfrac{}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} : \mathtt{int}}\,(\text{var}) \quad \dfrac{}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} : \mathtt{int}}\,(\text{var})}{\{\mathtt{y} : \mathtt{int}\} \vdash \mathtt{y} + \mathtt{y} : \mathtt{int}}\,(\text{intAdd})}{\{\} \vdash \mathtt{let\ y\ =\ 2\ in\ y} + \mathtt{y} : \mathtt{int}}\,(\text{let})$$

$\{\} \vdash$ let $y = 2$ in $y + y$ : int (let)

- $\{\} \vdash 2$ : int (intLit)
- $\{y : \text{int}\} \vdash y + y$ : int (intAdd)
  - $\{y : \text{int}\} \vdash y$ : int (var)
  - $\{y : \text{int}\} \vdash y$int (var)

# Practice Problem

$\cdot \vdash$ `fun f -> fun x -> f (x + 1) : (int -> int) -> int -> int`

*Give a typing derivation in compact form of the above judgment using 320Caml typing rules*

# Answer

$\cdot \vdash$ `fun f -> fun x -> f (x + 1) : (int -> int) -> int -> int`

- $\{f : \text{int} \to \text{int}\} \vdash \text{fun } x \to f\ (x+1) : \text{int} \to \text{int}$
  - $\overbrace{\{f : \text{int} \to \text{int},\ x : \text{int}\}}^{\Gamma} \vdash f\ (x+1) : \text{int}$
    - $\Gamma \vdash f : \text{int} \to \text{int}$
    - $\Gamma \vdash x + 1 : \text{int}$
      - $\Gamma \vdash x : \text{int}$
      - $\Gamma \vdash 1 : \text{int}$

# Hindley-Milner

# High Level

# High Level

**Hindley–Milner type systems** are typed $\lambda$–calculi with parametric polymorphism

# High Level

**Hindley-Milner type systems** are typed $\lambda$-calculi with parametric polymorphism

They underlie nearly all functional PLs currently in use (e.g., OCaml, Haskell, Elm)

# High Level

**Hindley–Milner type systems** are typed $\lambda$–calculi with parametric polymorphism

They underlie nearly all functional PLs currently in use (e.g., OCaml, Haskell, Elm)

They allow for a *restricted* form of type quantification, in which quantifiers always appear in the "outermost" position

# High Level

**Hindley-Milner type systems** are typed $\lambda$-calculi with parametric polymorphism

They underlie nearly all functional PLs currently in use (e.g., OCaml, Haskell, Elm)

They allow for a *restricted* form of type quantification, in which quantifiers always appear in the "outermost" position

*Type inference is decidable and (fairly) efficient*

# Type Inference (High Level)

$$\Gamma \vdash e : \tau \dashv \mathscr{C}$$

# Type Inference (High Level)

$$\Gamma \vdash e : \tau \dashv \mathscr{C}$$

`The type inference process follows the rough procedure:`

# Type Inference (High Level)

$$\Gamma \vdash e : \tau \dashv \mathscr{C}$$

The type inference process follows the rough procedure:

1. Derive $\Gamma \vdash e : \tau$ *relative to some constraints* $\mathscr{C}$

# Type Inference (High Level)

$$\Gamma \vdash e : \tau \dashv \mathscr{C}$$

The type inference process follows the rough procedure:

1. Derive $\Gamma \vdash e : \tau$ *relative to some constraints* $\mathscr{C}$
2. Use the constraints $\mathscr{C}$ to determine the "actual" type of $e$ in $\Gamma$

# Type Inference (High Level)

$$\Gamma \vdash e : \tau \dashv \mathcal{C}$$

The type inference process follows the rough procedure:

today

1. Derive $\Gamma \vdash e : \tau$ *relative to some constraints* $\mathcal{C}$
2. Use the constraints $\mathcal{C}$ to determine the "actual" type of $e$ in $\Gamma$

# Example (by Intuition)

$(\text{int} \to ?_5) \to \text{int} \to ?_5$

$\text{int} \to ?_5$

$\text{int} \to ?_5$

**fun f -> fun x -> f (x + 1)**

$\cdot \vdash \text{fun } f \to \text{fun } x \to f\ (x+1) : ?_2$

$\quad \{f : ?_1\} \vdash \text{fun } x \to f\ (x+1) : ?_3$

$\quad\quad \Gamma$

$\quad\quad \{f : ?_1,\ x : ?_4\ \text{int}\} \vdash f\ (x+1) : ?_5$

$\quad\quad\quad \Gamma \vdash f : ?_1$

$\quad\quad\quad \Gamma \vdash x + 1 : \text{int}$

$\quad\quad\quad\quad \Gamma \vdash x : ?_4\ \text{int}$

$\quad\quad\quad\quad \Gamma \vdash 1 : \text{int}$

constraints

$?_2 = ?_1 \to ?_3$

$?_3 = ?_4\ \text{int} \to ?_5$

$?_4 = \text{int}$

$?_1 = \text{int} \to ?_5$

# Hindley-Milner Light (Syntax)

```
<expr> ::= fun <var> -> <expr> | <expr> <expr>
         | let <var> = <expr> in <expr>
         | if <expr> then <expr> else <expr>
         | <expr> + <expr> | <expr> = <expr>
         | <int> | <var>
```

type vars.

```
<mty> ::= int | bool | <tyvar> | <mty> -> <mty>
 <ty> ::= <tyvar>.<ty> | <mty>
```

no quantification

quantification.

The syntax of HM$^-$ is the same as that of system F except:

» we've added a couple things to make our examples more interesting

» type quantification is *restricted*

# Hindley-Milner Light (Mathematical)

$$
\begin{aligned}
e ::= {} & \lambda x \,.\, e \mid e e \\
& \mid \text{let } x = e \text{ in } e \\
& \mid \text{if } e \text{ then } e \text{ else } e \\
& \mid e + e \mid e = e \\
& \mid n \mid x \\
\sigma ::= {} & \text{int} \mid \text{bool} \mid \alpha \mid \sigma \to \sigma \\
\tau ::= {} & \sigma \mid \forall \alpha \,.\, \tau
\end{aligned}
$$

As usual, we'll often use concise mathematical notation for writing down inference rules and derivations

# Type Variables and Type Schemes

$$\sigma ::= \mathsf{int} \mid \mathsf{bool} \mid \alpha \mid \sigma \to \sigma$$

$$\tau ::= \sigma \mid \forall \alpha \, . \, \tau$$

# Type Variables and Type Schemes

$$\sigma ::= \mathsf{int} \mid \mathsf{bool} \mid \alpha \mid \sigma \to \sigma$$

$$\tau ::= \sigma \mid \forall \alpha \,.\, \tau$$

$\sigma$ represents **monotypes**, types with *no quantification*. A type is **monomorphic** if it is a monotype with $\underbrace{\text{no type variables}}_{\text{closed}}$

# Type Variables and Type Schemes

$$\sigma ::= \mathsf{int} \mid \mathsf{bool} \mid \alpha \mid \sigma \to \sigma$$

$$\tau ::= \sigma \mid \forall \alpha \,.\, \tau$$

$\sigma$ represents **monotypes**, types with *no quantification*. A type is **monomorphic** if it is a monotype with no type variables

$\tau$ represents **type schemes**, which are types with some number of quantified type variables

# Type Variables and Type Schemes

$\forall\alpha.\alpha \to \beta$ : type scheme

$\forall\alpha.\forall\beta.\alpha \to \beta$ : polymorphic

$\alpha$ : monotype

int : monomorphic type

$$\sigma ::= \mathsf{int} \mid \mathsf{bool} \mid \alpha \mid \sigma \to \sigma$$

$$\tau ::= \sigma \mid \forall\alpha.\tau$$

NOT POSSIBLE

$\boxed{\forall\alpha.\ \boxed{\alpha \to \forall\beta.\ \beta \to \alpha}}$

not mono.

$\sigma$ represents **monotypes**, types with *no quantification*. A type is **monomorphic** if it is a monotype with no type variables

$\tau$ represents **type schemes**, which are types with some number of quantified type variables

We say a type is **polymorphic** if it is a *closed* type scheme

# Free Variables (Monotypes)

$FV(\alpha \to (\text{int} \to \beta)) =$

$FV(\alpha) \cup FV(\text{int} \to \beta) =$

$\{\alpha\} \cup FV(\text{int}) \cup FV(\beta) =$

$\{\alpha\} \cup \emptyset \cup \{\beta\} =$

$\{\alpha, \beta\}$

$$FV(\text{int}) = \varnothing$$
$$FV(\text{bool}) = \varnothing$$
$$FV(\alpha) = \{\alpha\}$$
$$FV(\tau_1 \to \tau_2) = FV(\tau_1) \cup FV(\tau_2)$$

Once we introduce variables, we have to again talk about free and bound variables

Unlike in System F, we will only need to consider free variables of **monotypes** so there is *no issue with variable capture*

# Understanding Check

*Define substitution* $[\tau_1/\alpha]\tau_2$ *for monotypes*

$$[\tau/\alpha]\,\text{int} = \text{int}$$

$$[\tau/\alpha]\,\text{bool} = \text{bool}$$

$$[\tau/\alpha]\,\beta = \begin{cases} \tau & \alpha = \beta \\ \beta & \text{ow} \end{cases}$$

$$[\tau/\alpha](\tau_1 \to \tau_2) = [\tau/\alpha]\tau_1 \to [\tau/\alpha]\tau_2$$

# Constraint-Based Inference

$$\Gamma \vdash e : \tau \dashv \mathscr{C}$$

# Constraint-Based Inference

$$\Gamma \vdash e : \tau \dashv \mathscr{C}$$

Our typing rules well need to keep track of a set of **constraints**, which tell use what must hold for $e$ to be well–typed

# Constraint-Based Inference

type schemes

$$\Gamma \vdash e : \tau \dashv \mathscr{C}$$

Our typing rules well need to keep track of a set of **constraints**, which tell use what must hold for $e$ to be well-typed

Contexts are are collections of variable declaration, i.e., mapping of variables to **type schemes**

# Constraint-Based Inference

$$\Gamma \vdash e : \tau \dashv \mathcal{C}$$

Our typing rules well need to keep track of a set of **constraints**, which tell use what must hold for $e$ to be well-typed

Contexts are are collections of variable declaration, i.e., mapping of variables to **type schemes**

The idea: We're formalizing the idea of "collecting together" our constraints, as in our intuitive example

# What is a constraint?

$$\tau_1 \doteq \tau_2$$

In general, a **type constraint** is a predicate on types. The only kind we will consider:

"$\tau_1$ should be the same as $\tau_2$"

Enforcing a constraint like this is called **unifying** $\tau_1$ and $\tau_2$

# Constraint-Based Inference

$$\Gamma \vdash e : \tau \dashv \mathscr{C}$$

# Constraint-Based Inference

$$\Gamma \vdash e : \tau \dashv \mathscr{C}$$

<u>The idea:</u> For each rule, we need to determine:

# Constraint-Based Inference

$$\Gamma \vdash e : \tau \dashv \mathscr{C}$$

<u>The idea:</u> For each rule, we need to determine:

» What is the *most general* type $\tau$ we could give $e$?

# Constraint-Based Inference

$$\Gamma \vdash e : \tau \dashv \mathscr{C}$$

<u>The idea:</u> For each rule, we need to determine:

» What is the *most general* type $\tau$ we could give $e$?

» What must be true of $\tau$, i.e., what *constrains* $\tau$?

# Constraint-Based Inference

$$\Gamma \vdash e : \tau \dashv \mathscr{C}$$

The idea: For each rule, we need to determine:

- What is the *most general* type $\tau$ we could give $e$?
- What must be true of $\tau$, i.e., what *constrains* $\tau$?

If we don't know what type something should be, *we create a fresh type variable for it*

Let's see some typing rules...

# HM$^-$ (Typing Literals)

$$\frac{n \texttt{ is an integer}}{\Gamma \vdash n : \mathsf{int} \dashv \varnothing}\ (\texttt{int})$$

$$\frac{}{\varnothing \vdash 5 : \mathsf{int} \dashv \varnothing}$$

`Literals have their expected types` *`without any constraints`*

# HM$^-$ (Typing Operators)

$\alpha \doteq \beta,\ \beta \doteq \text{bool}$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathscr{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathscr{C}_2}{\Gamma \vdash e_1 + e_2 : \text{int} \dashv \tau_1 \doteq \text{int}, \tau_2 \doteq \text{int}, \mathscr{C}_1, \mathscr{C}_2} \ (\text{add})$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathscr{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathscr{C}_2}{\Gamma \vdash e_1 = e_2 : \text{bool} \dashv \tau_1 \doteq \tau_2, \mathscr{C}_1, \mathscr{C}_2} \ (\text{eq})$$

$e_1 + e_2$ is an **int** if the types of $e_1$ and $e_2$ can be *unified* to **int**

We don't require that $\tau_i$ is *exactly* **int**, e.g., it may be a type variable!

# HM$^-$ (Typing If-Expressions)

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathcal{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathcal{C}_2 \qquad \Gamma \vdash e_3 : \tau_3 \dashv \mathcal{C}_3}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau_3 \dashv \tau_1 \doteq \text{bool}, \tau_2 \doteq \tau_3, \mathcal{C}_1, \mathcal{C}_2, \mathcal{C}_3} \ (\texttt{if})$$

An if-expression has the same type as its else-case when:

» the type of the condition can be *unified* with **bool**

» the types of the then-case and else-case can be *unified to each other*

# Example $\{x : \alpha, y : \beta\} \vdash \text{if } x \text{ then } x \text{ else } y : \tau \dashv \mathscr{C}$

$\Gamma$

$\{x:\alpha, y:\beta\} \vdash \text{if } x \text{ then } x \text{ else } y : \beta : \underbrace{\boxed{\alpha \doteq \text{bool}, \alpha \doteq \beta}}_{\mathscr{C}}$

$$\frac{x : \sigma \in \Gamma \quad \sigma \text{ is mono.}}{\Gamma \vdash x : \sigma}$$

$\Gamma \vdash x : \alpha \dashv \emptyset$

$\Gamma \vdash x : \alpha \dashv \emptyset$

$\Gamma \vdash y : \beta \dashv \emptyset$

bool

# HM$^-$ (Typing Functions)

$$\frac{\alpha \text{ is fresh} \qquad \Gamma, x : \alpha \vdash e : \tau \dashv \mathscr{C}}{\Gamma \vdash \lambda x \,.\, e : \alpha \to \tau \dashv \mathscr{C}} \; (\texttt{fun})$$

The input type of a function is some type $\alpha$ and it's output type is the type of the body

We don't know the input type, so we give it the most general form, i.e., a fresh type variable with no constraints

# HM$^-$ (Typing Application)

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv \mathscr{C}_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv \mathscr{C}_2 \qquad \alpha \text{ is fresh}}{\Gamma \vdash e_1\, e_2 : \alpha \dashv \tau_1 \doteq \tau_2 \to \alpha, \mathscr{C}_1, \mathscr{C}_2} \ (\texttt{app})$$

The type of an application is some type $\alpha$, such that the type of the function unifies to a function type with output type $\alpha$, and the input type matches the type of the argument (wordy...)

# $\mathbf{HM}^{-}$ (Typing Variables)

$$\frac{(x : \forall \alpha_1 . \forall \alpha_2 \ldots \forall \alpha_k . \tau) \in \Gamma \qquad \beta_1, \ldots, \beta_k \texttt{ are fresh}}{\Gamma \vdash x : [\beta_1 / \alpha_1] \ldots [\beta_k / \alpha_k] \tau \dashv \varnothing} \texttt{ (var)}$$

# HM$^-$ (Typing Variables)

monotype

$$\frac{(x : \forall \alpha_1 . \forall \alpha_2 \ldots \forall \alpha_k . \tau) \in \Gamma \qquad \beta_1, \ldots, \beta_k \text{ are fresh}}{\Gamma \vdash x : [\beta_1/\alpha_1] \ldots [\beta_k/\alpha_k]\tau \dashv \varnothing} \ (\text{var})$$

If $x$ is declared in $\Gamma$, then $x$ can be given the type $\tau$ *with all free variables replaced by* ***fresh variables***

# HM$^-$ (Typing Variables)

$$\frac{(x : \forall\alpha_1 . \forall\alpha_2 \ldots \forall\alpha_k . \tau) \in \Gamma \qquad \beta_1, \ldots, \beta_k \text{ are fresh}}{\Gamma \vdash x : [\beta_1/\alpha_1]\ldots[\beta_k/\alpha_k]\tau \dashv \varnothing} \text{ (var)}$$

If $x$ is declared in $\Gamma$, then $x$ can be given the type $\tau$ *with all free variables replaced by* ***fresh variables***

*This is where the polymorphism magic happens*

# HM$^-$ (Typing Variables)

$$\frac{(x : \forall \alpha_1 . \forall \alpha_2 \ldots \forall \alpha_k . \tau) \in \Gamma \qquad \beta_1, \ldots, \beta_k \text{ are fresh}}{\Gamma \vdash x : [\beta_1/\alpha_1]\ldots[\beta_k/\alpha_k]\tau \dashv \varnothing} \text{ (var)}$$

If $x$ is declared in $\Gamma$, then $x$ can be given the type $\tau$ *with all free variables replaced by* ***fresh variables***

*This is where the polymorphism magic happens*

**fresh variables can be unified with anything**

# Example $\{f : \forall\alpha\,.\,\alpha \to \alpha\} \vdash f\,(f\,2 = 2) : ? \dashv ?$

$$\overbrace{\{f : \forall\alpha.\alpha\to\alpha\}}^{\Gamma} \vdash f\,(f\,2 = 2) : \gamma \dashv \underbrace{\boxed{\beta\to\beta \doteq \mathrm{bool}\to\gamma,\ \varepsilon \doteq \mathrm{int},\ \eta\to\eta \doteq \mathrm{int}\to\varepsilon}}_{C}$$

- $\Gamma \vdash f : \beta \to \beta \dashv \emptyset$
- $\Gamma \vdash f\,2 = 2 : \mathrm{bool} \dashv \varepsilon \doteq \mathrm{int},\ \eta\to\eta \doteq \mathrm{int}\to\varepsilon$
  - $\Gamma \vdash f\,2 : \varepsilon \dashv \eta\to\eta \doteq \mathrm{int}\to\varepsilon$
    - $\Gamma \vdash f : \eta \to \eta \dashv \emptyset$
    - $\Gamma \vdash 2 : \mathrm{int} \dashv \emptyset$

# Example

```
fun f -> fun x -> f (x + 1)
```

# Up Next

We still need to:

- introduce a **unification algorithm** to determine the "actual" type given a collection of constraints
- Discuss **let-expressions** (and top-level let expressions)
- introduce **type annotations**

We *wont*:

- deal with **type errors** (tricker with unification-based inference)