# Type Inference

## Concepts of Programming Languages

CAS CS 320

# Outline

» Discuss **polymorphism** in general

» Discuss **type inference** with eye towards **Hindley–Milner typing**

» Look at a set of typing rules for **constraint-based inference**

» Walk through some **examples**

# Practice Problem

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

```
let rec f x = f (f (x + 1)) in f
```

*What are the types of the above OCaml expressions?*

# Answer

int → ?    int    ?

① `fun f -> fun x -> f (x + 1)`

: int → ?

② `let rec f x = f (f (x + 1)) in f`

int

?

① $(int \to \alpha) \to int \to \alpha$

② $int \to int$

# Polymorphism

# 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

```
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

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

# High Level

```
let rec rev_int (l : int list) : int list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let rec rev_string (l : string list) : string list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let _ = assert (rev_int [1;2;3] = [3;2;1])
let _ = assert (rev_string ["1";"2";"3"] = ["3";"2";"1"])
```

# High Level

```
let rec rev_int (l : int list) : int list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let rec rev_string (l : string list) : string list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let _ = assert (rev_int [1;2;3] = [3;2;1])
let _ = assert (rev_string ["1";"2";"3"] = ["3";"2";"1"])
```

Copy/pasting code is *time consuming* and *error prone*

# High Level

```
let rec rev_int (l : int list) : int list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let rec rev_string (l : string list) : string list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

let _ = assert (rev_int [1;2;3] = [3;2;1])
let _ = assert (rev_string ["1";"2";"3"] = ["3";"2";"1"])
```

Copy/pasting code is *time consuming* and *error prone*

**Polymorphism** allows for better code reuse. The *same* function can be applied in multiple contexts

# Basic Example

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

# Basic Example

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

We want to be able to define functions that can be used in multiple contexts *and* that we can type check

# Basic Example

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

We want to be able to define functions that can be used in multiple contexts *and* that we can type check

Important: We can evaluate this if we *don't* type check

# Basic Example

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

We want to be able to define functions that can be used in multiple contexts *and* that we can type check

Important: We can evaluate this if we *don't* type check

*But if we type-check, what should be the type of **id**?*

# Polymorphism

# Polymorphism

There are two common kinds of polymorphism

# Polymorphism

There are two common kinds of polymorphism

1. **Ad Hoc Polymorphism:** The ability to overload function names so that different types can share interfaces

# Polymorphism

There are two common kinds of polymorphism

1. **Ad Hoc Polymorphism:** The ability to overload function names so that different types can share interfaces

2. **Parametric polymorphism:** The ability to define functions that are *agnostic* to (parts of) the types, giving it more reusability

# Polymorphism

There are two common kinds of polymorphism

1. **Ad Hoc Polymorphism:** The ability to overload function names so that different types can share interfaces

our focus

2. **Parametric polymorphism:** The ability to define functions that are *agnostic* to (parts of) the types, giving it more reusability

# Aside: Ad Hoc Polymorphism

```
let add (x : float) (y : float) = x +. y
let add (x : string) (y : string) = x ^ y
(* This doesn't work in OCaml... *)
```

# Aside: Ad Hoc Polymorphism

```
let add (x : float) (y : float) = x +. y
let add (x : string) (y : string) = x ^ y
(* This doesn't work in OCaml... *)
```

Ad hoc polymorphism is essentially **function overloading**

# Aside: Ad Hoc Polymorphism

```
let add (x : float) (y : float) = x +. y
let add (x : string) (y : string) = x ^ y
(* This doesn't work in OCaml... *)
```

Ad hoc polymorphism is essentially **function overloading**

Functions can be defined and used for different types of inputs

# Aside: Ad Hoc Polymorphism

```
let add (x : float) (y : float) = x +. y
let add (x : string) (y : string) = x ^ y
(* This doesn't work in OCaml... *)
```

Ad hoc polymorphism is essentially **function overloading**

Functions can be defined and used for different types of inputs

Then we can define code against *interfaces* (this is common in object oriented programming)

# Parametric Polymorphism

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

# Parametric Polymorphism

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

Parametric polymorphism allows for functions which are agnostic to the types of its inputs (this is what OCaml does)

# Parametric Polymorphism

```
let id = fun x -> x
let a = id 0
let b = id (0 = 0)
let c = id id
```

Parametric polymorphism allows for functions which are agnostic to the types of its inputs (this is what OCaml does)

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

There are many subtleties to this...

# Subtlety 1: Type Annotations

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

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

# Subtlety 1: Type Annotations

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

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

Parametric polymorphism is *not* just removing type annotations

# Subtlety 1: Type Annotations

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

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

Parametric polymorphism is *not* just removing type annotations

It's also not the same as having no type system

# Subtlety 1: Type Annotations

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

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

Parametric polymorphism is *not* just removing type annotations

It's also not the same as having no type system

There are type systems *with* polymorphism that *require* type annotations

# Subtlety 2: Type Inference

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

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

# Subtlety 2: Type Inference

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

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

Polymorphism is *not* the same has having type inference

# Subtlety 2: Type Inference

```
let rec rev ('a list) : 'a list =
  match l with
  | [] -> []
  | x :: l -> rev l @ [x]

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

Polymorphism is *not* the same has having type inference

In OCaml, polymorphism is deeply connected with it's type inference system, but they are distinct (we can choose to annotated all our OCaml code)

# Subtlety 3: Dispatch

```
let to_string (x : 'a) : string = ...
(* This is not possible in OCaml *)
```

Parametric polymorphism cannot be used for *dispatch*

We can't write a polymorphic function that "checks the type" to see what to do

# Implementing Polymorphism

# Implementing Polymorphism

There are a couple approaches to implementing parametric polymorphism:

# Implementing Polymorphism

There are a couple approaches to implementing parametric polymorphism:

» **OCaml (Hindley–Milner):** Infer the "most general" polymorphic type

# Implementing Polymorphism

There are a couple approaches to implementing parametric polymorphism:

- **OCaml (Hindley-Milner):** Infer the "most general" polymorphic type
- **System F (2nd-Order $\lambda$-Calculus):** take types as arguments!

# Implementing Polymorphism

There are a couple approaches to implementing parametric polymorphism:

- **OCaml (Hindley–Milner):** Infer the "most general" polymorphic type
- **System F (2nd-Order $\lambda$–Calculus):** take types as arguments!

Either way, we have to introduce the notion of a *type variable*

# Type Variables

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

# Type Variables

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

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

# 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

# Quantification

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

# Quantification

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

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

# 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

# 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

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

# 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

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

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

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

$\{\} \vdash \text{let } y = 2 \text{ in } y + y : \text{int}$ (let)
- $\{\} \vdash 2 : \text{int}$ (intLit)
- $\{y : \text{int}\} \vdash y + y : \text{int}$ (int Add)
  - $\{y : \text{int}\} \vdash y : \text{int}$ (var)
  - $\{y : \text{int}\} \vdash y : \text{int}$ (var)

# 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

$\underbrace{'a .\ 'b .}_{\text{quantified}}\ 'a \to 'b \to 'a$

**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)

constraints

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

The type inference process follows the rough procedure:

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

# Type Inference (High Level)

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

The type inference process follows the rough procedure:

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 $\Gamma$

# Type Inference (High Level)

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

The type inference process follows the rough procedure:

today

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 $\Gamma$

# Example (by Intuition)

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

$?_1 \to ?_3 \to ?_4$

$?_1 \doteq \boxed{int \to ?_4}$

$?_3 \doteq \boxed{int}$

$\{\} \vdash \text{fun } f \to \text{fun } x \to f\,(x+1) : ?_1 \to ?_3 \to ?_4$ $\quad (int \to ?_4) \to int \to ?_4$

$\quad\{f : ?_1\} \vdash \text{fun } x \to f\,(x+1) : ?_3 \to ?_4$

$\quad\quad\{f : ?_1,\ x : ?_3\} \vdash f\,(x+1) : ?_4$ $\qquad ?_1 \doteq int \to ?_4$

$\quad\quad\quad\{f : ?_1,\ x : ?_3\} \vdash f : ?_1$ $\quad$ (var)

$\quad\quad\quad\{f : ?_1,\ x : ?_3\} \vdash x + 1 : int$

$\quad\quad\quad\quad\{f : ?_1,\ x : ?_3\} \vdash x : int$ $\qquad ?_3 \doteq int$

$\quad\quad\quad\quad\{\ \dots \quad \dots\ \} \vdash 1 : int$

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

type variable

type quantification

$$\alpha . \underbrace{\beta . \underbrace{\alpha \to \beta \to \gamma}_{\langle mty \rangle}}_{\langle ty \rangle}$$

# Hindley-Milner Light (Mathematical)

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

type variable

type quantifier

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

int → bool ✓ bool → α ✗

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

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

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

# 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

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

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

$\sigma$ 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 $\forall \alpha.\ int \to \beta$

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

$\forall \alpha. \forall \beta.\ int \to \beta$

# Free Variables (Monotypes)

$$
\begin{aligned}
FV(\mathsf{int}) &= \varnothing \\
FV(\mathsf{bool}) &= \varnothing \\
FV(\alpha) &= \{\alpha\} \\
FV(\tau_1 \to \tau_2) &= FV(\tau_1) \cup FV(\tau_2)
\end{aligned}
$$

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*

$$[\boxed{int \to bool}/\alpha](\beta \to \boxed{\alpha} \to \boxed{\alpha}) = \beta \to (int \to \alpha) \to (int \to \alpha)$$

① $[\sigma/\alpha]\, int = int$

② $[\sigma/\alpha]\, bool = bool$

③ $[\sigma/\alpha]\, \beta = \begin{cases} \sigma & \alpha = \beta \\ \beta & \text{o.w.} \end{cases}$

④ $[\sigma/\alpha](\tau_1 \to \tau_2) = [\sigma/\alpha]\tau_1 \to [\sigma/\alpha]\tau_2$

# 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

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

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**

# 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

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

<u>The idea:</u> 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

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

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* $\tau$?

# 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)

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

$$\frac{}{\{\} \vdash 2 : \mathsf{int} \dashv \varnothing}$$

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

# $\text{HM}^-$ (Typing Operators)

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

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

$e_1 + e_2$ 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!

$$\frac{\Gamma \vdash e_1 : \tau_1 \dashv C_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv C_2 \qquad \Gamma \vdash e_3 : \tau_3 \dashv C_3 \quad (\text{if})}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau_2 \dashv \tau_2 \doteq \tau_3,\ \tau_1 \doteq \text{bool},\ C_1, C_2, C_3}$$

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

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

$$\frac{\Gamma \vdash e_1 : \text{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$$

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

$\{x : \alpha, y : \beta\} \vdash \text{if } x \text{ then } x \text{ else } y : \alpha \dashv$ $\alpha \doteq \text{bool}$, $\beta \doteq \alpha$

- $\{\ldots\} \vdash x : \alpha \dashv \emptyset$ (var)
- $\{\ldots\} \vdash x : \alpha \dashv \emptyset$ (var)
- $\{\ldots\} \vdash y : \beta \dashv \emptyset$ (var)

# HM$^-$ (Typing Functions)

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

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

$$\frac{\alpha \text{ is fresh} \qquad \Gamma \vdash e_1 : \tau_1 \dashv C_1 \qquad \Gamma \vdash e_2 : \tau_2 \dashv C_2}{\Gamma \vdash e_1\ e_2 : \alpha \dashv \tau_1 \doteq \tau_2 \to \alpha, C_1, C_2}$$

# HM$^{-}$ (Typing Application)

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

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...)

# 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)

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

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

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

# Up Next

<u>We still need to:</u>

» 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**

<u>We *wont*:</u>

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

# Summary

By restricting our type quantification, we get a system that has decidable and efficient **type inference**

Hindley–Milner style type inference requires us to figure out a collection of **constraints** that need to be unified