

# CMSC 330: Organization of Programming Languages

---

## Type Inference and Unification

# Type Checking vs Type Inference

---

- ▶ Type checking: use declared types to check types are correct

```
let apply (f: ('a->'b)) (x: 'a) : 'b = f x
```

- ▶ Type inference:

```
let apply f x = f x
```

- Infer the most general types that could have been declared, and type checks the code without the type information

# The Type Inference Algorithm

---

- ▶ Input: A program without types
- ▶ Output: A program with type for every expression, which is annotated with its most general type

# Why do we want to infer types?

---

- ▶ Reduces syntactic overhead of expressive types
  - // C++ Declare a vector of vectors of integers  
`std::vector<std::vector<int>> matrix;`
- ▶ Guaranteed to produce most general type
- ▶ Widely regarded as important language innovation
- ▶ Illustrative example of a flow-insensitive static analysis algorithm

# History

---

- ▶ Original type inference algorithm
  - Invented by Haskell Curry and Robert Feys for the simply typed lambda calculus in 1958
- ▶ In 1969, Hindley
  - extended the algorithm to a richer language and proved it always produced the most general type
- ▶ In 1978, Milner
  - independently developed equivalent algorithm, called algorithm W, during his work designing ML
- ▶ In 1982, Damas proved the algorithm was complete.
  - Currently used in many languages: ML, Ada, Haskell, C# 3.0, F#, Visual Basic .Net 9.0. Have been plans for Fortress, Perl 6, C++0x,...

# Type Inference: Basic Idea

---

## ► Example

```
fun x → 2 + x
-: int → int
```

- What is the type of the expression?
  - + has type:  $\text{int} \rightarrow \text{int} \rightarrow \text{int}$
  - 2 has type:  $\text{int}$
  - Since we are applying + to x we need  $x : \text{int}$
  - Therefore,  $\text{fun } x \rightarrow 2 + x$  has type  $\text{int} \rightarrow \text{int}$

# Type Inference: Basic Idea

---

## ► Example

```
fun f → f 3
  -: (int → a) → a
```

## ► What is the type of the expression?

- 3 has type: `int`
- Since we are applying `f` to `3` we need `f : int → a` and the result is of type `a`
- Therefore, `fun f → f 3` has type `(int → a) → a`

# Type Inference: Basic Idea

---

- ▶ Example

`fun f → f (f 3)`

- ▶ What is the type of the expression?

# Type Inference: Basic Idea

---

- ▶ Example

`fun f → f (f "hi")`

- ▶ What is the type of the expression?

# Type Inference: Basic Idea

---

- ▶ Example

**`fun f → f (f 3, f 4)`**

- ▶ What is the type of the expression?

# Type Inference: Complex Example

---

```
let square = fun z → z * z in
  fun f → fun x → fun y →
    if (f x y) then (f (square x) y)
    else (f x (f x y))
```

# Type Inference: Complex Example

---

```
let square = fun z → z * z in  
  fun f → fun x → fun y →  
    if (f x y) then (f (square x) y)  
    else (f x (f x y))
```

```
* : int → (int → int)
```

```
z : int
```

# Type Inference: Complex Example

---

```
let square = fun z → z * z in  
  fun f → fun x → fun y →  
    if (f x y) then (f (square x) y)  
    else (f x (f x y))
```

```
* : int → (int → int)
```

```
z : int
```

```
square : int → int
```

# Type Inference: Complex Example

```
let square = fun z → z * z in
  fun f → fun x → fun y →
    if (f x y) then (f (square x) y)
    else (f x (f x y))
```

`z : int`

`square : int → int`

```
(f      x      y) .      :bool
(f (square x) y)      :bool
(f      x      (f x y)) :bool
```

`y : bool`



```
x: t
(square x) : t
```

`t = int`

# Type Inference: Complex Example

---

```
let square = fun z → z * z in  
  fun f → fun x → fun y →  
    if (f x y) then (f (square x) y)  
    else (f x (f x y))
```

`z : int`

`square : int → int`

`x: int`

`y : bool`

`f : ('a → 'b → bool), x: 'a, y: 'b`

`a: int`

`b: bool`

# Type Inference: Complex Example

---

```
let square = fun z → z * z in  
  fun f → fun x → fun y →  
    if (f x y) then (f (square x) y)  
    else (f x (f x y))
```

`z : int`

`square : int → int`

`f : ('a → 'b → bool), x: 'a, y: 'b`

`a: int    b: bool`

**Type:** `(int → bool → bool) → int → bool → bool`

# Unification

---

- ▶ Unification is an algorithmic process of solving equations between symbolic expressions
- ▶ Unifies two terms
- ▶ Used for pattern matching and type inference
- ▶ Simple examples
  - $\text{int} * x$  and  $y * (\text{bool} * \text{bool})$  are **unifiable**
    - $y = \text{int}$
    - $x = (\text{bool} * \text{bool})$
  - $\text{int} * \text{int}$  and  $\text{int} * \text{bool}$  are **not unifiable**

# Type Inference Algorithm

---

- ▶ Visit the AST and generate constraints:
  - From environment: literals (2), built-in operators (+), known functions (tail)
  - From form of parse tree: e.g., application and abstraction nodes
- ▶ Solve constraints using *unification*
- ▶ Determine types of top-level declarations

# Step 1: Parse Program

---

- Parse program text to construct parse tree

**fun x → 2 + x**

**Fun:**

```
x: g = fresh_guess ()  
(g → infer (x+2) (extend env x:g))
```

**(x+2) :**

```
unify (infer x env) TInt;  
unify (infer 2 env) TInt;  
TInt
```

# Inferring Polymorphic Types

---

► Example: `fun f → f 2`

**Fun:**

```
f: t = fresh_guess ()  
(t → infer (f 2) (extend env f:t))
```

**(f 2):**

```
let t1 = infer f env in (* f : t)  
let t2 = infer 2 env (* 2:int *)  
let t3 = fresh_guess () in  
unify t1 (t2 → t3));  
t3 (* int -> 'a)
```

# Using Polymorphic Functions

---

```
(fun f-> f 2) (fun d-> d +d)
```

```
let e1 = Fun ("f", App (ID "f", Int 2)) ; ;
```

```
let e2 = Fun ("x", Binop (Add, ID "x", ID "x")) ; ;
```

```
let e3 = App (e11, e12) ; ;
```

```
e1: ((int->'a) -> 'a)
```

```
e2: (int->int)
```

```
e3: int
```

# Most General Type

---

- ▶ Type inference produces the *most general type*

```
let rec map f lst =  
  match lst with  
  [] -> []  
  | hd :: tl -> f hd :: (map f tl)  
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- ▶ Functions may have many less general types

```
val map : (t_1 -> int, [t_1]) -> [int]  
val map : (bool -> t_2, [bool]) -> [t_2]  
val map : (char -> int, [cChar]) -> [int]
```

- ▶ Less general types are all instances of **most general type**, also called the *principal type*

# Complexity of Type Inference Algorithm

---

- ▶ When Hindley/Milner type inference algorithm was developed, its complexity was unknown
- ▶ In 1989, Kanellakis, Mairson, and Mitchell proved that the problem was exponential-time complete
- ▶ Usually linear in practice though...
  - Running time is exponential in the depth of polymorphic declarations

# Type Inference: Key Points

---

- ▶ Type inference computes the types of expressions
  - Does not require type declarations for variables
  - Finds the most general type by solving constraints
  - Leads to polymorphism
- ▶ Sometimes better error detection than type checking
  - Type may indicate a programming error even if no type error
- ▶ Some costs
  - More difficult to identify program line that causes error
  - Natural implementation requires uniform representation sizes
- ▶ Idea can be applied to other program properties
  - Discover properties of program using same kind of analysis

# Example: Swap Two Values

---

## ► OCaml

```
let swap (x, y) =  
  let temp = !x in  
    (x := !y; y := temp)  
val swap : 'a ref * 'a ref -> unit = <fun>
```

## ► C++

```
template <typename T>  
void swap(T& x, T& y){  
    T tmp = x;  x=y;  y=tmp;  
}
```

Declarations both swap two values polymorphically, but they are compiled very differently

# Implementation

---

- ▶ OCaml
  - `swap` is compiled into one function
  - Typechecker determines how function can be used
- ▶ C++
  - `swap` is compiled differently for each instance  
(details beyond scope of this course ...)
- ▶ Why the difference?
  - OCaml ref cell is passed by pointer. The local `x` is a pointer to value on heap, so its size is constant
  - C++ arguments passed by reference (pointer), but local `x` is on the stack, so its size depends on the type

# Polymorphism vs Overloading

---

## ► Parametric polymorphism

- Single algorithm may be given many types
- Type variable may be replaced by any type
- if  $f:t \rightarrow t$  then  $f:int \rightarrow int$ ,  $f:bool \rightarrow bool$ , ...

## ► Overloading

- A single symbol may refer to more than one algorithm
- Each algorithm may have different type
- Choice of algorithm determined by type context
- Types of symbol may be arbitrarily different
- In ML,  $+$  has types  $int * int \rightarrow int$ ,  $real * real \rightarrow real$ , no others
- Haskell permits more general overloading and requires user assistance

# Varieties of Polymorphism

---

- ▶ **Parametric polymorphism** A single piece of code is typed generically
  - Imperative or first-class polymorphism
  - ML-style or let-polymorphism
- ▶ **Ad-hoc polymorphism** The same expression exhibit different behaviors when viewed in different types
  - Overloading
  - Multi-method dispatch
  - intentional polymorphism
- ▶ **Subtype polymorphism** A single term may have many types using the rule of subsumption allowing to selectively forget information

# Summary

---

- ▶ Types are important in modern languages
  - Program organization and documentation
  - Prevent program errors
  - Provide important information to compiler
- ▶ Type inference
  - Determine best type for an expression, based on known information about symbols in the expression
- ▶ Polymorphism
  - Single algorithm (function) can have many types