

# **CSC2/455 Software Analysis and Improvement Type Inference**

---

Sreepathi Pai

Mar 18, 2024

URCS

# Outline

Types

Type Rules

A Simple Type System for the Typed Lambda Calculus

Type Inference

Unification

Postscript

# Outline

Types

Type Rules

A Simple Type System for the Typed Lambda Calculus

Type Inference

Unification

Postscript

# Typing in Languages Made Simple

- Compiler knows the type of every expression
  - Static typing
- Values “carry” their type at runtime
  - Dynamic typing
- Programs with type errors do not compile (or throw exceptions at runtime)
  - Strongly typed
- Programs with type errors carry on merrily
  - PHP (older versions only?)

# Type Systems

- Poor (Limited expressivity)
  - assembly, C
- Rich
  - C++
  - Ada
- Richest (High expressivity)
  - ML/OCaml
  - Haskell

# Why have rich type systems?

- General purpose programming languages impose a set of constraints
  - `int` may not be stored into a `char`
- Applications and APIs impose a set of logical constraints
  - Mass of an object can never be negative
  - `free(ptr)` must not be called twice on the same `ptr` value
- Application programmers must check these constraints manually
  - Although encapsulation in OOP helps
- Can we get the compiler to check *application*-level constraints for us?
  - without knowing anything about the application?
  - i.e. a general-purpose facility to impose logical application-defined constraints

# Rust

- Rust is a systems programming language from Mozilla
  - Replacement for C/C++
  - No garbage collector
  - "Bare-metal" programming ability
- Unlike C, Rust provides *memory safety*
  - No NULL pointer dereference errors
  - No use-after-free
  - No double-free
  - etc.
- Rust uses its type system to impose these constraints
  - Rust checks types statically, so programs with these errors fail to compile.
  - Rust's mechanism is not purely type-based, it also uses additional analyses

# Compilers and Type Systems

Compilers perform the following type-related tasks:

- Type checking
  - Does the program obey the typing rules of the language?
- Type inference
  - What is the type of each expression, variable, function, etc.?



# Outline

Types

Type Rules

A Simple Type System for the Typed Lambda Calculus

Type Inference

Unification

Postscript

# Formalizing Programming Languages

- Syntax of a programming language
  - Usually specified as Backus-Naur Form (BNF)
  - Consists of statements, expressions, etc.
- Semantics of a programming language
  - Multiple methods: denotational, operational, axiomatic
  - We'll see more of semantics in later parts of this course
- Type system
  - Assigns types to (syntactic) terms
  - Consists of type rules
  - Types must ultimately make semantic sense (e.g. an `int` always contains an integer)

## Building Block: Type Environments

- Static Typing Environment (or *Context*)
  - Map of variables to types
  - Denoted by  $\Gamma$
  - An empty environment is represented as  $\phi$
- Usually if a term  $M$  has type  $\alpha$  in  $\Gamma$ , we will write it as:
  - $\Gamma \vdash M : \alpha$  (read as  $\Gamma$  entails that  $M$  has type  $\alpha$ )
  - e.g.  $x : \text{int}, y : \text{int} \vdash (x + y) : \text{int}$
  - likewise,  $x : \text{float}, y : \text{float} \vdash (x + y) : \text{float}$
- $\Gamma \vdash M : \alpha$  is called a *judgement*

## Building Block: Type Rules

$$\frac{\Gamma \vdash x : \text{Int} \quad \Gamma \vdash y : \text{Int}}{\Gamma \vdash (x + y) : \text{Int}} \quad (\text{PLUS})$$

- The part above the line are the premises
- The part below the line is the conclusion
- If the premises are true, then the conclusion is also true
  - Identical to inference rules in logic

# Using Type Rules

- Type rules are “formal proof systems”
  - Like formal logic
- Goal is to “derive” a type using only the type rules
  - The derivation is the proof of a type

## Example of type derivation: I

- Let  $n \in \mathbb{Z}$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash n : \text{Int}} \quad (\text{NUM})$$

- The  $\diamond$  indicates that  $\Gamma$  is well-formed
  - It is an axiom that  $\phi \vdash \diamond$ , we'll call this rule `EMPTY`
  - Axioms have no premises
- Then we can add a rule for  $+$

$$\frac{\Gamma \vdash x : \text{Int} \quad \Gamma \vdash y : \text{Int}}{\Gamma \vdash (x + y) : \text{Int}} \quad (\text{PLUS})$$

## Example of type derivation: II

- Derivation for  $1 + 2$  is a *Int*
- First show that  $\Gamma \vdash 1 : \text{Int}$

$$\frac{\frac{\text{---} \text{EMPTY}}{\phi \vdash \diamond}}{\phi \vdash 1 : \text{Int}} \text{NUM}$$

- Similarly, show that  $\Gamma \vdash 2 : \text{Int}$

$$\frac{\frac{\text{---} \text{EMPTY}}{\phi \vdash \diamond}}{\phi \vdash 2 : \text{Int}} \text{NUM}$$

## Completing the derivation ...

- Since we have  $\phi \vdash 1 : Int$  and  $\phi \vdash 2 : Int$ , we can now apply PLUS to complete our derivation:

$$\frac{\phi \vdash 1 : Int \quad \phi \vdash 2 : Int}{\phi \vdash 1 + 2 : Int} \text{PLUS}$$



# Outline

Types

Type Rules

A Simple Type System for the Typed Lambda Calculus

Type Inference

Unification

Postscript

# Syntax

$\alpha, \beta$	$::=$		types
	$\kappa$	$\kappa \in \mathit{Basic}$	basic types
	$\alpha \rightarrow \beta$		function types
$M, N$	$::=$		terms
	$x$		variable
	$\lambda x : \alpha. M$		function
	$M N$		application

# Judgements

- $\Gamma \vdash \diamond$ 
  - $\Gamma$  is a well-formed environment
- $\Gamma \vdash \alpha$ 
  - $\alpha$  is a well-formed type in  $\Gamma$
- $\Gamma \vdash M : \alpha$ 
  - $M$  is a well-formed term of type  $\alpha$  in  $\Gamma$

# Rules - I

- (Axiom) Empty environment is well-formed

$$\frac{}{\phi \vdash \diamond} (\text{Env } \phi)$$

- Extend the environment by assigning a type  $\alpha$  to a variable  $x$

$$\frac{\Gamma \vdash \alpha \quad x \notin \text{dom}(\Gamma)}{\Gamma, x : \alpha \vdash \diamond} (\text{Env } x)$$

## Rules - II

- Derivation rule for basic types (i.e. type constants)

$$\frac{\Gamma \vdash \diamond \quad \kappa \in \mathit{Basic}}{\Gamma \vdash \kappa} \text{(Type Const)}$$

- Derivation rule for function types

$$\frac{\Gamma \vdash \alpha \quad \Gamma \vdash \beta}{\Gamma \vdash \alpha \rightarrow \beta} \text{(Type Arrow)}$$

## Rules - III

- Variable type (read as if  $x : \alpha$  occurs somewhere in  $\Gamma$ )

$$\frac{\Gamma', x : \alpha, \Gamma'' \vdash \diamond}{\Gamma', x : \alpha, \Gamma'' \vdash x : \alpha} (\text{Val } x)$$

- Function type

$$\frac{\Gamma, x : \alpha \vdash M : \beta}{\Gamma \vdash \lambda x : \alpha. M : \alpha \rightarrow \beta} (\text{Val Fun})$$

- Function Application Type

$$\frac{\Gamma \vdash M : \alpha \rightarrow \beta \quad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta} (\text{Val App})$$

# Parametric Types/Polymorphism

- Some languages support “generic” functions
  - types are parametrized
  - notably from the ML family

$\alpha, \beta$	$::=$		types
$\kappa$		$\kappa \in \textit{Basic}$	basic types
$\chi$			type variable
$\alpha \rightarrow \beta$			function type
$\forall \chi. \alpha$			universally quantified type

- A type that fits the syntax above would be  $\forall \chi. \chi \rightarrow \textit{Int}$ 
  - Indicates the type of a function that accepts any type and returns *Int*

## More than basic types

- Product types
  - $\alpha \times \beta$
- Union (or sum) types
  - $\alpha + \beta$
- Records, Variants, References, etc.



# Outline

Types

Type Rules

A Simple Type System for the Typed Lambda Calculus

**Type Inference**

Unification

Postscript

# Inferring types

- Most languages assign types to values
- Some require programmers to specify the type for variables
  - C, C++ (until recently)
- Some infer types of each variable automatically
  - even for polymorphic types
  - famous example: (Standard) ML

## Steps for type inference

- Treat unknown types as *type variables*
  - We will use Greek alphabets for type variables
  - Note: distinct from program variables
- Write a set of equations involving type variables
  - These equations are obtained from the typing rules
- Solve the set of equations

## Example #1

```
a = 0.5  
b = a + 1.0
```

- $\text{typevar}(0.5) = \kappa_1$
- $\text{typevar}(a) = \alpha$
- $\text{typevar}(b) = \beta$
- $\text{typevar}(1.0) = \kappa_2$
- $\text{typevar}(a + 1.0) = \eta$

## Example #1: Equations

$$\text{typevar}(0.5) = \kappa_1 = \text{Float}$$

$$\text{typevar}(a) = \alpha = \kappa_1$$

$$\text{typevar}(b) = \beta = \eta$$

$$\text{typevar}(1.0) = \kappa_2 = \text{Float}$$

$$\text{typevar}(a + 1.0) = \eta = +(\alpha, \kappa_2)$$

$$+(\gamma, \gamma) \rightarrow \gamma$$

$$\alpha = \kappa_2$$

## Example #2

Consider the ML example:

```
fun length(x) =  
  if null(x) then 0 else length(tl(x)) + 1;
```

- Clearly, `length` is a function of type  $\alpha' \rightarrow \beta$ , where  $\text{typeof}(x) = \alpha'$
- Is  $\alpha'$  a fixed type? Consider the two uses:
  - `length(["a", "b", "c"])`
  - `length([1, 2, 3])`

## Example #2: Polymorphic Functions

- The type  $\alpha'$  can be written as  $\text{list}(\alpha)$
- So, `length` is a function of type  $\forall\alpha \text{list}(\alpha) \rightarrow \beta$

## Example #2: Equations and solving them

EXPR: TYPE	UNIFY
$length: \beta \rightarrow \gamma$	
$x: \beta$	
$if: bool \times \alpha_i \times \alpha_i \rightarrow \alpha_i$	
$null: list(\alpha_n) \rightarrow bool$	
$null(x): bool$	$list(\alpha_n) = \beta$
$0: int$	$\alpha_i = int$
$+: int \times int \rightarrow int$	
$tl: list(\alpha_t) \rightarrow list(\alpha_t)$	
$tl(x): list(\alpha_t)$	$list(\alpha_t) = list(\alpha_n)$
$length(tl(x)): \gamma$	$\gamma = int$
$1: int$	
$length(tl(x)) + 1: int$	
$if(...): int$	

Note  $\alpha_n$  remains in the final type, so we add a  $\forall \alpha_n$ , making this a



# Unify?

Unification is a procedure to symbolically manipulate equations to make them “equal”.

- No variables in equations, only constants
  - $5 = 5$ , is unified
  - $4 = 5$ , can't be unified
- Variables in equations
  - Find a substitution  $S$  that maps each type variable  $x$  in the equations to a type expression,  $S[x \rightarrow e]$
  - Let  $S(t)$  be the equation resulting from replacing all variables  $y$  in  $t$  with  $S[y]$
  - Then,  $S$  is a unifier for two equations  $t_1$  and  $t_2$ , if  $S(t_1) = S(t_2)$

# Outline

Types

Type Rules

A Simple Type System for the Typed Lambda Calculus

Type Inference

**Unification**

Postscript

## Unification Example

Compute a unifier to unify the equations below:

$$((\alpha_1 \rightarrow \alpha_2) \times \text{list}(\alpha_3)) \rightarrow \text{list}(\alpha_2)$$

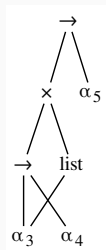
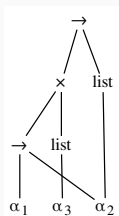
$$((\alpha_3 \rightarrow \alpha_4) \times \text{list}(\alpha_3)) \rightarrow \alpha_5$$

$x$	$S(x)$
$\alpha_1$	$\alpha_1$
$\alpha_2$	$\alpha_2$
$\alpha_3$	$\alpha_1$
$\alpha_4$	$\alpha_2$
$\alpha_5$	$\text{list}(\alpha_2)$

Applying  $S(x)$  to both the equations leads to the unified equation:

$$((\alpha_1 \rightarrow \alpha_2) \times \text{list}(\alpha_1)) \rightarrow \text{list}(\alpha_2)$$

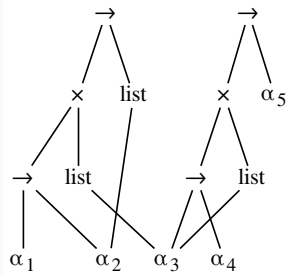
# Type Graphs



For the unification algorithm, we'll first build type graphs for the type equations we've seen:

- Internal nodes are constructors ( $\rightarrow$ ,  $\times$ ,  $\text{list}$ )
- Leaf nodes are type variables ( $\alpha_1$ ,  $\alpha_2$ ,  $\alpha_3$ , ...)
- Edges connect constructors to their arguments

## Actual Type Graph

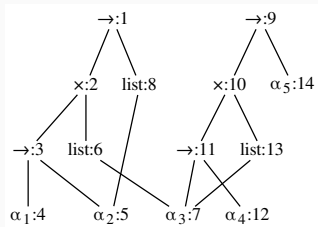


This is the actual type graph that is formed for both the type equations. The shared edges between the graphs represent shared type variables.

# High-level Unification Algorithm

- Goal is to generate equivalence classes
  - Two nodes are in the same equivalence class if they can be unified
  - Equivalence classes are identified by a representative node
- A node is trivially unifiable with itself
- Non-variable nodes must be of same type to be unifiable
- Basic algorithm is an asymmetric variant of the union–find data-structure

# Setup



- Each node is initially in its own equivalence class, indicated by a number
- Ultimately, nodes that are equivalent will have the same number



# Unification Algorithm

```
def unify(node m, node n):
    s = find(m)
    t = find(n)

    if (s == t): return True

    if (s and t are the same basic type): return True

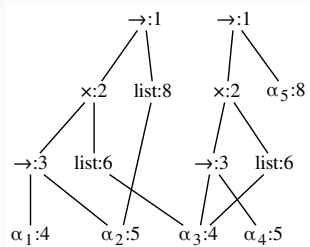
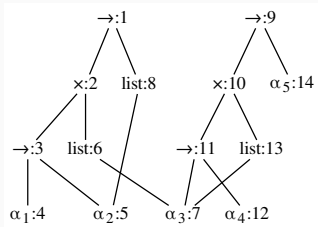
    if (s(s1, s2) and t(t1, t2) are binary op-nodes with
        the same operator):
        union_asym(s, t) # speculative
        return unify(s1, t1) and unify(s2, t2)

    if (s or t is a variable):
        union_asym(s, t)
        return True

    return False
```

Figure 6.32 in the Dragon Book.

# Unification



# Outline

Types

Type Rules

A Simple Type System for the Typed Lambda Calculus

Type Inference

Unification

Postscript

# References

- A self-contained introduction to type systems
  - Luca Cardelli, Type Systems, Handbook of Computer Science and Engineering, 2nd Ed
- An updated version (available only through the library)
  - Stephanie Weirich, Type Systems, Handbook of Computer Science and Engineering, 3rd Ed
- Algorithm is from Chapter 6 of the Dragon Book
  - Section 6.5
- Martelli and Montanari, 1982, An Efficient Unification Algorithm
- Good introductory tutorials with Python code:
  - Unification
  - Type Inference