CSC2/455 Software Analysis and Improvement
Type Inference

Sreepathi Pai

Mar 20, 2023

URCS

Outline

Types

Type Rules

A Simple Type System for the Typed Lambda Calculus

Type Inference

Unification

Postscript

Outline

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 deference 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

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 Γ
  - An empty environment is represented as ϕ
- Usually if a term M has type α in Γ, we will write it as:
  - $\Gamma \vdash M : \alpha$ (read as Γ entails that M has type α)
  - e.g. $x : \texttt{int}, y : \texttt{int} \vdash (x + y) : \texttt{int}$
  - likewise, $x : \texttt{float}, y : \texttt{float} \vdash (x + y) : \texttt{float}$
- $\Gamma \vdash M : \alpha$ is called a *judgement*

Building Block: Type Rules

$$\frac{\Gamma \vdash x : \mathit{Int} \quad \Gamma \vdash y : \mathit{Int}}{\Gamma \vdash (x + y) : \mathit{Int}} \qquad (\textsc{Plus})$$

- 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 $n \in \mathbb{Z}$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash n : \mathit{Int}} \quad (\textsc{Num})$$

- The $\diamond$ indicates that Γ is well-formed
  - It is an axiom that $\phi \vdash \diamond$, we'll call this rule $\textsc{Empty}$
  - Axioms have no premises

- Then we can add a rule for $+$

$$\frac{\Gamma \vdash x : \mathit{Int} \quad \Gamma \vdash y : \mathit{Int}}{\Gamma \vdash (x + y) : \mathit{Int}} \qquad (\textsc{Plus})$$

Example of type derivation: II

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

$$\dfrac{\dfrac{}{\phi \vdash \diamond}\textsc{Empty}}{\phi \vdash 1 : \mathit{Int}}\textsc{Num}$$

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

$$\dfrac{\dfrac{}{\phi \vdash \diamond}\textsc{Empty}}{\phi \vdash 2 : \mathit{Int}}\textsc{Num}$$

Completing the derivation ...

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

$$\frac{\phi \vdash 1 : \mathit{Int} \qquad \phi \vdash 2 : \mathit{Int}}{\phi \vdash 1 + 2 : \mathit{Int}}\textsc{Plus}$$

Outline

Syntax

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

Judgements

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

Rules - I

- (Axiom) Empty environment is well-formed

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

- Extend the environment by assigning a type α to a variable x

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

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

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

Rules - III

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

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

- Function type

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

- Function Application Type

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

Parametric Types/Polymorphism

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

α, β	$::=$		types
	κ	$\kappa \in \mathit{Basic}$	basic types
	χ		type variable
	$\alpha \to \beta$		function type
	$\forall \chi . \alpha$		universally quantified type

- A type that fits the syntax above would be $\forall \chi . \chi \to \mathit{Int}$
  - 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

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
```

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

Example #1: Equations

$$
\begin{aligned}
\text{typevar}(0.5) &= \kappa_1 = \texttt{Float} \\
\text{typevar}(a) &= \alpha = \kappa_1 \\
\text{typevar}(b) &= \beta = \eta \\
\text{typevar}(1.0) &= \kappa_2 = \texttt{Float} \\
\text{typevar}(a + 1.0) &= \eta = +(\alpha, \kappa_2) \\
+(\gamma, \gamma) &\rightarrow \gamma \\
\alpha &= \kappa_2
\end{aligned}
$$

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 α' a fixed type? Consider the two uses:
  - `length(["a", "b", "c"])`
  - `length([1, 2, 3])`

Example #2: Polymorphic Functions

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

Example #2: Equations and solving them

EXPR:	TYPE	UNIFY
length:	$\beta \to \gamma$	
x:	β	
`if`:	$\texttt{bool} \times \alpha_i \times \alpha_i \to \alpha_i$	
null :	$\text{list}(\alpha_n) \to \texttt{bool}$	
null(x) :	`bool`	$\text{list}(\alpha_n) = \beta$
0 :	`int`	$\alpha_i = \texttt{int}$
$+$:	$\texttt{int} \times \texttt{int} \to \texttt{int}$	
tl :	$\text{list}(\alpha_t) \to \text{list}(\alpha_t)$	
tl(x) :	$\text{list}(\alpha_t)$	$\text{list}(\alpha_t) = \text{list}(\alpha_n)$
length(*tl*(x)) :	γ	$\gamma = \texttt{int}$
1 :	`int`	
length(*tl*(x)) + 1 :	`int`	
`if`(...) :	`int`	

Note α_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
  - $6 = 9$, can't be unified
- Variables in equations
  - Find a substitution S that maps each type variable x in the equations to a type expression, $S[x \to e]$
  - 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

Unification Example

Compute a unifier to unify the equations below:

$$\begin{aligned} ((\alpha_1 \to \alpha_2) \times \mathrm{list}(\alpha_3)) &\to \mathrm{list}(\alpha_2) \\ ((\alpha_3 \to \alpha_4) \times \mathrm{list}(\alpha_3)) &\to \alpha_5 \end{aligned}$$

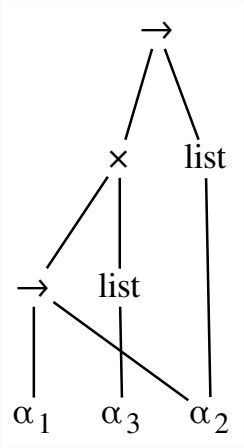
Unifier

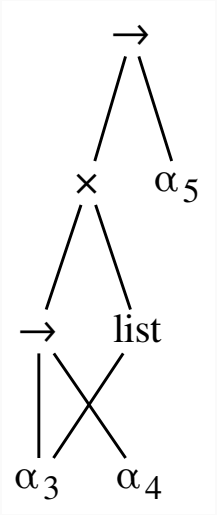
x	$S(x)$
α_1	α_1
α_2	α_2
α_3	α_1
α_4	α_2
α_5	$\text{list}(\alpha_2)$

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

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

Type Graphs

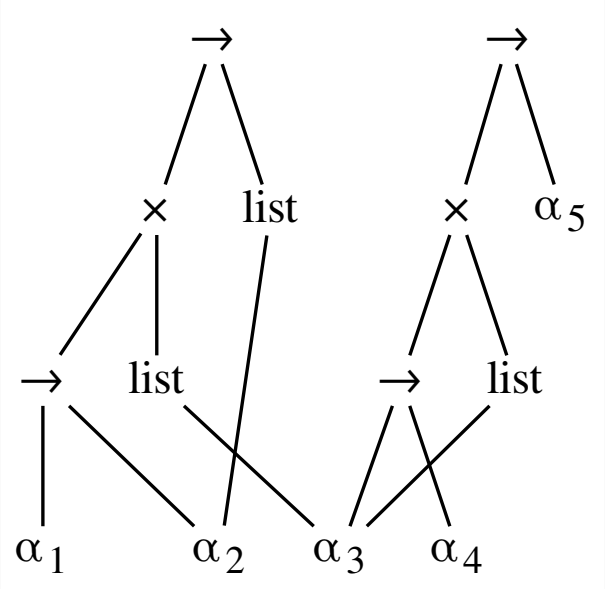




For the unification algorithm, we'll first build type graphs for the type equations we've seen:

- 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

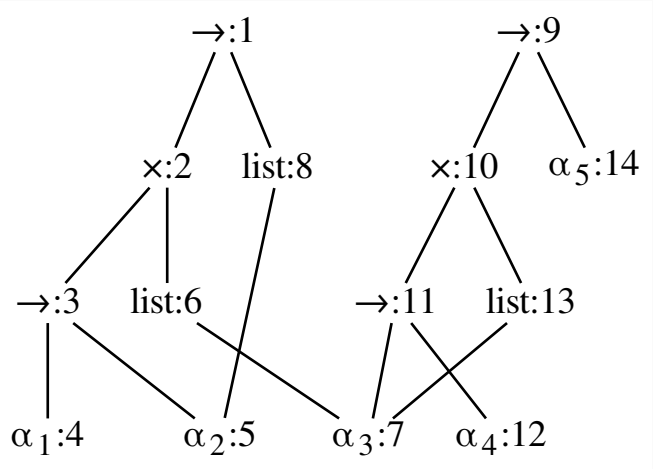


This is the actual type graph that is formed for both the type equations. The shared edges between the graphs represent shared type variables.

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



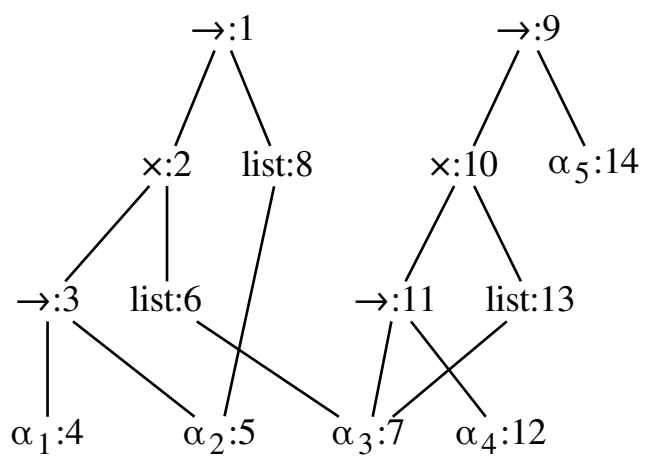
- Each node is initially in its own equivalence class, indicated by a number
- Ultimately, nodes that are equivalent will have the same number

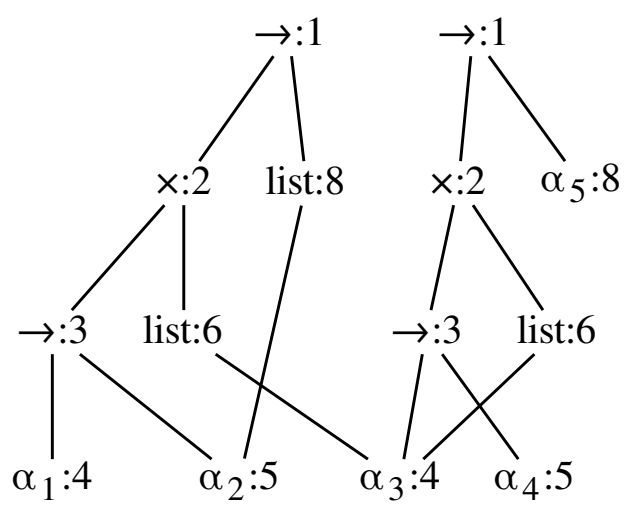
Unification Algorithm

```
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

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