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 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 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

\[ \Gamma \vdash x : \text{Int} \quad \Gamma \vdash y : \text{Int} \]

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

- 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
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 \( n \in \mathbb{Z} \)

\[
\Gamma \vdash \diamond \\
\frac{}{\Gamma \vdash n : Int} \quad (\text{\textsc{Num}})
\]

• The \( \diamond \) indicates that \( \Gamma \) 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 \(+\)

\[
\Gamma \vdash x : \text{Int} \quad \Gamma \vdash y : \text{Int} \\
\frac{}{\Gamma \vdash (x + y) : \text{Int}} \quad (\text{\textsc{Plus}})
\]
Example of type derivation: II

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

\[
\begin{array}{l}
\text{\frac{}{\phi \vdash \Diamond}} \\
\text{\frac{\phi \vdash \Diamond}{\phi \vdash \text{Int}}}
\end{array}
\]

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

\[
\begin{array}{l}
\text{\frac{}{\phi \vdash \Diamond}} \\
\text{\frac{\phi \vdash \Diamond}{\phi \vdash \text{Int}}}
\end{array}
\]
Completing the derivation ...

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

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

Types

Type Rules

A Simple Type System for the Typed Lambda Calculus

Type Inference

Unification

Postscript
Syntax

\[ \alpha, \beta ::= \]
\[ \kappa \quad \kappa \in Basic \quad \text{basic types} \]
\[ \alpha \to \beta \quad \text{function types} \]

\[ M, N ::= \]
\[ x \quad \text{variable} \]
\[ \lambda x : \alpha. M \quad \text{function} \]
\[ M \, N \quad \text{application} \]
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 $\Gamma$
• (Axiom) Empty environment is well-formed

\[
\phi \vdash \diamond \\
\text{(Env } \phi) \quad \phi \vdash \diamond
\]

• Extend the environment by assigning a type \( \alpha \) to a variable \( x \)

\[
\Gamma \vdash \alpha \quad x \notin \text{dom}(\Gamma) \\
\Gamma, x : \alpha \vdash \diamond \\
\text{(Env } x) \quad \Gamma, x : \alpha \vdash \diamond
\]
• Derivation rule for basic types (i.e. type constants)

\[ \Gamma \vdash \diamond \quad \kappa \in Basic \quad \text{(Type Const)} \]

\[ \Gamma \vdash \kappa \]

• Derivation rule for function types

\[ \Gamma \vdash \alpha \quad \Gamma \vdash \beta \quad \text{(Type Arrow)} \]

\[ \Gamma \vdash \alpha \rightarrow \beta \]
• Variable type (read as if $x : \alpha$ occurs somewhere in $\Gamma$)

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

• Function type

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

• Function Application Type

$$
\frac{\Gamma \vdash M : \alpha \to \beta \quad \Gamma \vdash N : \alpha}{\Gamma \vdash MN : \beta} \quad \text{(Val App)}
$$
Some languages support "generic" functions
- types are parametrized
- notably from the ML family

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

\[
\begin{align*}
typevar(0.5) & = \kappa_1 = \text{Float} \\
typevar(a) & = \alpha = \kappa_1 \\
typevar(b) & = \beta = \eta \\
typevar(1.0) & = \kappa_2 = \text{Float} \\
typevar(a + 1.0) & = \eta = + (\alpha, \kappa_2) \\
+ (\gamma, \gamma) & \rightarrow \gamma \\
\alpha & = \kappa_2
\end{align*}
\]
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 `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, $\text{length}$ is a function of type $\forall \alpha \text{list}(\alpha) \rightarrow \beta$
Example #2: Equations and solving them

<table>
<thead>
<tr>
<th>EXPR</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>length</code>:</td>
<td>$\beta \rightarrow \gamma$</td>
</tr>
<tr>
<td><code>x</code>:</td>
<td>$\beta$</td>
</tr>
<tr>
<td><code>if</code>:</td>
<td>$\text{bool} \times \alpha_i \times \alpha_i \rightarrow \alpha_i$</td>
</tr>
<tr>
<td><code>null</code>:</td>
<td>$\text{list}(\alpha_n) \rightarrow \text{bool}$</td>
</tr>
<tr>
<td><code>null(x)</code>:</td>
<td>$\text{bool}$</td>
</tr>
<tr>
<td><code>0</code>:</td>
<td>$\text{int}$</td>
</tr>
<tr>
<td><code>+</code>:</td>
<td>$\text{int} \times \text{int} \rightarrow \text{int}$</td>
</tr>
<tr>
<td><code>tl</code>:</td>
<td>$\text{list}(\alpha_t) \rightarrow \text{list}(\alpha_t)$</td>
</tr>
<tr>
<td><code>tl(x)</code>:</td>
<td>$\text{list}(\alpha_t)$</td>
</tr>
<tr>
<td><code>length(tl(x))</code>:</td>
<td>$\gamma$</td>
</tr>
<tr>
<td><code>1</code>:</td>
<td>$\text{int}$</td>
</tr>
<tr>
<td><code>length(tl(x)) + 1</code>:</td>
<td>$\text{int}$</td>
</tr>
<tr>
<td><code>if(...)</code>:</td>
<td>$\text{int}$</td>
</tr>
</tbody>
</table>

Note $\alpha_n$ remains in the final type, so we add a $\forall \alpha_n$, making this a polymorphic type. So $\text{length}$ is $\forall \alpha_n \text{list}(\alpha_n) \rightarrow \text{int}$.
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 $S(t_1) = S(t_2)$
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:

\[
\begin{align*}
((\alpha_1 \rightarrow \alpha_2) \times \text{list}(\alpha_3)) & \rightarrow \text{list}(\alpha_2) \\
((\alpha_3 \rightarrow \alpha_4) \times \text{list}(\alpha_3)) & \rightarrow \alpha_5
\end{align*}
\]
Applying $S(x)$ to both the equations leads to the unified equation:

$$(((\alpha_1 \rightarrow \alpha_2) \times \text{list}(\alpha_1)) \rightarrow \text{list}(\alpha_2)$$
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 ($\alpha_1, \alpha_2, \alpha_3, \ldots$)
- **Edges** connect constructors to their arguments
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
Each node is initially in its own equivalence class, indicated by a number.

Ultimately, nodes that are equivalent will have the same number.
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
• 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