CSC2/455 Software Analysis and Improvement
Type Inference

Sreepathi Pai

URCS

April 15, 2019

Outline

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

One perspective on 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
- 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.

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

Inferring types

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

Example #1

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

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

Example #1: Equations

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

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 polymorphic type. So `length` is $\forall\text{list}(\alpha) \to \texttt{int}$

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, $S[x \rightarrow e]$
  - 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

Unification Example

Compute a unifier to unify the equations below:

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

Unifier

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

This unifies to:

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

Type Graphs

- Internal nodes are constructors
- Leaf nodes are type variables
- Edges connect constructors to their arguments

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
- Non-variable nodes must be of same type to be unifiable
- The same node is trivially unifiable

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 o
         union(s, t)
         return unify(s1, t1) and unify(s2, t2)

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

     return False
```

Figure 6.32 in the Dragon Book.

Example Figure

See Figure 6.31 in the Dragon Book (we're going to work through it)

Outline

References

- Chapter 6 of the Dragon Book
  - Section 6.5
- Martelli and Montanari, 1982, An Efficient Unification Algorithm
- Good introductory tutorials in Python:
  - Unification
  - Type Inference