CSC2/455 Software Analysis and Improvement
Type Inference

Sreepathi Pai

URCS

April 15, 2019
Outline

Types

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
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 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.
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 Inference

Unification

Postscript
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

\[a = 0.5\]
\[b = a + 1.0\]

- \(\text{typeof}(0.5) = \kappa_1\)
- \(\text{typeof}(a) = \alpha\)
- \(\text{typeof}(b) = \beta\)
- \(\text{typeof}(1.0) = \kappa_2\)
- \(\text{typeof}(a + 1.0) = \eta\)
Example #1: Equations

\[
\begin{align*}
typeof(0.5) &= \kappa_1 = \text{Float} \\
typeof(a) &= \alpha = \kappa_1 \\
typeof(b) &= \beta = \eta \\
typeof(1.0) &= \kappa_2 = \text{Float} \\
typeof(a + 1.0) &= \eta = +(\alpha, \kappa_2) \\
+ (\gamma, \gamma) &\rightarrow \gamma \\
\alpha &= \kappa_2
\end{align*}
\]
Consider the ML example:

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

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

EXPR: TYPE

| length: β → γ  | UNIFY |
| x: β          |       |
| if: bool × α_i × α_i → α_i |
| null: list(α_n) → bool |
| null(x): bool |
| 0: int |
| +: int × int → int |
| tl: list(α_t) → list(α_t) |
| tl(x): list(α_t) |
| length(tl(x)): γ |
| 1: int |
| length(tl(x)) + 1: int |
| if(...): int |

Note α_n remains in the final type, so we add a ∀α_n, making this a polymorphic type. So length is ∀list(α) → 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

Types

Type Inference

Unification

Postscript
Compute a unifier to unify the equations below:

\[ ((\alpha_1 \times \alpha_2) \times \text{list}(\alpha_3)) \rightarrow \text{list}(\alpha_2) \]
\[ ((\alpha_3 \times \alpha_4) \times \text{list}(\alpha_3)) \rightarrow \alpha_5 \]
This unifies to:

\[ (((\alpha_1 \rightarrow \alpha_2) \times \text{list}(\alpha_1)) \rightarrow \text{list}(\alpha_2) \]
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 operator):
        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

Types

Type Inference

Unification

Postscript
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