Type inference

Amer Diwan
(adapted from John Mitchell's notes)

Outline of lecture

• Why do we need type inference
  – Save programmer effort
  – More polymorphism
  – Find bugs
• A simple algorithm for type inference
• Reading: Mitchell, Section 5.4
**Saving programmer effort and program clutter**

- fun find (nil, _) = false
  | find (hd::tl, tofind) = tofind = hd orelse find(tl, tofind)

- OR

- fun find(nil: 'a list, tofind: 'a) = false
  | find((hd:'a)::(tl:'a list), tofind: 'a) = tofind = hd orelse find(tl, tofind)

- This still doesn't include return types

- Counter argument: user types are useful documentation. What do you think?

**Getting most general type**

- Let's suppose I have lists of integers and I need a function that checks if an integer is in my list
  - fun find(nil: int list, tofind: int) = false
    | find((hd:int)::(tl:int list), tofind: int) = tofind = int or else find(tl, tofind)

- Works for me but not too reusable
Getting most general type (cont.)

• fun add_to_list(alist, toadd) =
  if find(alist, toadd) then alist else toadd::alist

• Possible typing:
  fun add_to_list(alist: 'a list, toadd: 'a) =
    if find(alist, toadd) then alist else toadd::alist

• Is this right?
• Polymorphic types are hard to manually get "right".

Finding bugs

• fun reverse (nil) = nil
  | reverse(x::lst) = reverse(lst);
reverse: 'a list -> 'b list

• The type doesn't look right. What is wrong?
Summary of goals for type inference

• Give programmer the benefit of static typing without the effort
• Compute the most general type for functions to get maximum reusability
• Compute poor man's version of program “specifications”--useful for finding bugs

SML type inference example

• Example
  – fun f(x) = 2 + x;
  f = fn: int->int
• How does this work?
  – + has two types: int*int->int, real*real->real
  – 2: int has only one type
  – Thus +: int*int->int
  – From context, need x: int
  – Therefore f(x:int) = 2 + x has type int->int
SML type inference: another example

• fun f(x) = x + x
• What is the type of f?

Another presentation

• Example
  – fun f(x) = 2 + x;
    > f = fn : int->int
• How does this work?
  – Assign types to leaves
  – Propagate to internal nodes and generate constraints
  – Solve by substitution
Generating constraints

Application:
- \( f \) must have function type \( \text{domain} \to \text{range} \)
- Domain of \( f \) must be type of argument \( x \)
- Result type is range of \( f \)

Function expression:
- Type is function type \( \text{domain} \to \text{range} \)
- Domain is type of variable \( x \)
- Range is type of function body \( e \)

Solving constraints

- **Unification**
- Basic idea:
  - If a constraint says \( t = u \), \( t \) and \( u \) are type expressions, then unify values of \( t \) and \( u \)
  - If \( t \) or \( u \) is a primitive type then it is easy
  - If \( t \) and \( u \) are non-primitive types, then unify their components
  - If \( t \) is a type variable, replace uses of \( t \) with \( u \)
Solving constraints: examples

- $x = \text{int} \Rightarrow$
  
  $x = \text{int}$

- $\text{int} \to \text{'a} = \text{'b} \to \text{'b} \Rightarrow$
  
  $\text{int} = \text{'b}$ and $\text{'a} = \text{'b} \Rightarrow$
  
  $\text{int} = \text{'a}$

- $\text{int} * \text{bool} = \text{'a} * \text{'a} \Rightarrow$
  
  $\text{'a} = \text{int}$ and $\text{'a} = \text{bool} \Rightarrow$
  
  $\text{int} = \text{bool}$

  Type error!

Inferring polymorphic types

- Example
  
  - fun apply(f, x) = f(x);
    
    $f = \text{fn} : (\text{'a} \to \text{'b}) \to \text{'a} \to \text{'b}$

- How does this work
  
  - Assign types to leaves
  - Assign types to interior nodes
  - Generate constraints
  - Unify!
Example (cont.)

• Constraints:
  – t = t1->t2
    t1 = u
    t2 = s
    r = t * u -> s

• Substituting for t
  – t1 = u
    t2 = s
    r = (t1->t2) * u -> s

• Substitute u for t1 and s for t2
  – r = (u->s) * u->s

More on type inference

• Perfect type inference is undecidable
• SML type inference is exponential but seems to work
  – Programmer needs to intervene sometimes when overloaded functions are involved but otherwise it is mostly automatic
Discussion

• Is type inference a good idea?
  – Does writing down types force programmers to think?
  – Is it easier to read, write, and debug programs without types?

Next topic

• Using rich type systems to analyze programs
• Reading: Steensgaard (from web page). Feel free to ignore the results section