Type Inference

April 22, 2002

CS 131: Programming Languages

The Type *Checking* Problem

- Given a program where the type of every variable is specified, determine whether the program is well-typed.

```
fun f(x:bool):real =
     if x then
        3.0
     else
        2.0 * f(not x)
```

- Straightforward if we have unique (or at least *principal*) types.

The Type *Inference* Problem

- Given a program with some or all type annotations missing, can types be inserted to make the program typecheck?

```
fun f(x) =
     if x then
        3.0
     else
        2.0 * f(not x)
```

- Sometimes called type *reconstruction*.

Metavariables

- Type inference requires figuring out the unknown types.
- We will do this by using *metavariables*, which range over type annotations.

A Procedure for Type Inference

1. Allocate a *metavariable* for each missing type annotation and every subexpression

```
((fn f    => f) (fn x     => x))(3)
```

"Algorithm" continued

2. For each subexpression, figure out all the constraints that a type checker would be checking

"Algorithm" continued

3. Find values of the metavariables such that all these equational constraints are satisfied.

Solving Constraints

- What is a solution to a set of constraints?
  - A type for each metavariable, such that, when these are plugged in all the equations become true.
- Does this idea sound familiar?
  - Say, from Prolog in CS 60?
  - Or (more recently) from Resolution Theorem Proving in CS 80?

Unification

- General problem:
  - Given two "phrases" containing constants and variables, find values for the variables that makes the two phrases equal.

Unification Specification

- If asked to unify `int` with `int`, we don't have to do anything.
  - Same for any other base types.
- If asked to unify `t1->t2` with `u1->u2` it is necessary and sufficient to find values for metavariables making `t1=u1` and `t2=u2` both true.
  - Similar for `t1*t2` and `u1*u2`
- If asked to unify a metavariable with itself, we don't have to do anything.
- If asked to unify a metavariable M with any other type `t`,
  - If M already has a definition, then we just need to check that this definition unifies with `t`.
  - Otherwise we can *define* the value of M to be `t`, so long as M doesn't already have a definition *and* as long as the type `t` does not involve M.
    - Latter condition is called the "occurs check", and prevents circular definitions
    - By the way, Prolog skips this check for speed purposes.

Another Example

```
(fn x => (x x))(fn x => (x x))
```

One Last Example

```
let val id = (fn x => x)
in
   (id 3, id true)
end
```

Let-Polymorphism

- Observation: type inference would have *worked* if we hadn't used a definition

```
( (fn x => x) 3, (fn x => x) true )
```

- Suggests the following idea: whenever you see

  `let val x =` e_1 `in` e_2

  (where e_1 is a value) typecheck it as

  $e_2[x \leftarrow e_1]$

Let-Polymorphism

- Advantages: more programs typecheck
- Disadvantages: definitions are re-typechecked every time they're used.

More Efficient Implementation

- We can apply type inference to *definitions* and cache the results.
- For example, every time we do type inference on `fn x => x` we get the answer `M -> M` where `M` is a <u>fresh</u>, <u>unconstrained</u> metavariable.
  - That is, the code can be given type `int->int`, `bool->bool`, `(int*string)->(int*string)`, etc.
- Rather than substituting in the code, we can just remember this fact, which can be summarized with a "type scheme"

  $\forall\alpha.\ \alpha\text{->}\alpha$

More Efficient Implementation

- Now, whenever we see a use of `id`, instead of rechecking the code we can get the same answer by looking at the type scheme

  $\forall\alpha.\ \alpha\text{->}\alpha$

  and plugging in a *fresh* metavariable for α, yielding

  `M' -> M'`

Modifications to Type Inference

- When typechecking a variable definition, need to figure out "how polymorphic" the definition is *before* we can typecheck uses of that definition.
  - We need to completely finish type inference on the definition before going on.
- Requires interleaving constraint generation and solving
  - More efficient anyway.
  - Implementations usually don't actually "construct" constraints to be solved later, but just invoke `unify` as they go along.

Examples

```
let val diag = fn x => (x,x)
in
   (diag "A", diag 65)
end
```

```
let val K = fn x => fn y => x
in
   ((K 1) 2, (K "foo") "bar")
end
```

Let-polymorphism

- This approach to polymorphism, where only definitions can be polymorphic, is variously referred to as
  - Let-Polymorphism
  - ML Polymorphism
  - Hindley-Milner Polymorphism
- More general polymorphism is possible, but much harder to do type inference!

```
fn id => (id 3, id true)
```

Pitfall 1

- If we find that the type of a defined variable involves a metavariable without a definition, does it follow that this variable is polymorphic?

```
fun foo x =
   let val y = x
   in
       y+y
   end
```

Constrained Metavariables

- An unset metavariable with still *constrained* if it also occurs in the type of some variable already in scope.
  - And said to be *unconstrained* otherwise.
- Only safe to make definitions polymorphic in *unconstrained*, unset metavariables.

Pitfall 2

- Should this code typecheck?

```
let
   val succ = (fn n => n+1)
   val r    = ref (fn x=>x)
in
   r := succ;
   (!r)(true)
end
```

- SML '97 uses the *value restriction*: only definitions which are clearly values may be polymorphic.

Complexity Results

- Given a monomorphic expression of length n,
  - Determining whether the expression has a type (and if so what type) can be done in time $O(n)$.
  - However, the type may have length $O(2^n)$
- Given a polymorphic expression of length n,
  - Determining whether the expression has a type (and if so what type) can be done in time $O(2^n)$.
  - However, the type may have length $O(2^{2^n})$
- In practice, algorithm is much faster.