Compiler Construction 2011, Lecture 12

Type inference

Subset of Scala

- Int, Boolean (unless otherwise specified)
- arithmetic operations (+, -, …), Int x Int => Int
- boolean operators
- functions
- if-then-else statements

Example

Does it type-check?

```
object Main {
  val a = 2 * 3
  val b = a < 2
  val c = sumOfSquares(a)
  val d = if(b) c(3) else square(a)
}

def square(x) = x * x

def sumOfSquares(x) = {
  (y) => square(x) + square(y)
}
```

The idea

```
object Main {
  val a: TA = 2 * 3
  val b: TB = a < 2
  val c: TC = sumOfSquares(a)
  val d: TD = if(b) c(3) else square(a)
}

def square(x: TE): TF = x * x

def sumOfSquares(x: TG): TH = {
  (y: TI) => square(x) + square(y)
}
```

Find assignment
{TA -> Int, TB -> Boolean ...}

Hindley-Milner algorithm, intuitively

1. Record type constraints

```
val a: A = 3
val b: B = a
```

constraints:
{ A = Int, A = B}

2. Solve type constraints
   - obvious in the case above: {A= Int, B = Int}
   - in general use unification algorithm

3. Return assignment to type variables or failure

Type inference/reconstruction

Given partial type information, recover missing types such that program type checks.

vs. dynamically typed languages:

- compiler still has to assign some static type to each variable

vs. implicit type conversion:

- want to assign one type to each variable
- conversion is an additional technique

e.g. `val x = 2 + 3.4`

Some definitions

Definition 1 (Type substitution):

A *type substitution* **σ** is a finite mapping from type variables to types.

e.g. [A -> Int, B -> Bool] and we write **σ**X for applying this mapping to a particular type expression X

Definition 2 (Constraint set, Unification):

A *constraint set* C is a set of equations {Si = Ti}, i∈ 1…n. A substitution **σ** unifies an equation A = B, if **σ**A = **σ**B. It unifies C, if it unifies all equation.

Definition 3 (Most general unifier):

A substitution **σ** is more general than a substitution **σ**', **σ** ⊑ **σ**', if **σ**' = ɣ ○ **σ**, for some substitution ɣ. The *most general unifier* for a constraint set C is a substitution **σ** that unifies C such that **σ** ⊑ **σ**' for every substitution **σ**' unifying C.

➔ We want to find the most general substitution **σ** such that it unifies the constraint set C we obtain from the program.

Some definitions

Definition 3 (Most general unifier):

A substitution $\boldsymbol{\sigma}$ is more general than a substitution $\boldsymbol{\sigma}'$, $\boldsymbol{\sigma} \sqsubseteq \boldsymbol{\sigma}'$, if $\boldsymbol{\sigma}' = \gamma \circ \boldsymbol{\sigma}$, for some substitution γ. The *most general unifier* for a constraint set C is a substitution $\boldsymbol{\sigma}$ that unifies C such that $\boldsymbol{\sigma} \sqsubseteq \boldsymbol{\sigma}'$ for every substitution $\boldsymbol{\sigma}'$ unifying C.

Example:

```
f: Y     a: X
```

What is the most general type substitution such that the expression

`f( f( a))` type-checks?

Recording type constraints

$$\frac{\Gamma \vdash b : T_1 \quad \Gamma \vdash e_1 : T_2 \quad \Gamma \vdash e_2 : T_3}{\Gamma \vdash (\text{if (b) } e_1 \text{ else } e_2) : T_4}$$

T1 = Boolean
T2 = T3 = T4

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash (e_1 + e_2) : T_3}$$

T1 = T2 = T3
T3 = Int

$$\frac{\Gamma \vdash e_1 : T_1 ... \Gamma \vdash e_n : T_n \quad \Gamma \vdash f : (S_1 \times ... \times S_n \to S)}{\Gamma \vdash f(e_1, ..., e_n) : T}$$

S = T
T1 = S1 …

Recording type constraints

```
object Main {
  val a: TA = 2 * 3
  val b: TB = a < 2
  val c: TC = sumOfSquares(a)
  val d: TD =
   if(b) c(3): S1 else square(a): S2
}

def square(x: TE): TF = x * x

def sumOfSquares(x: TG): TH = {
  (y: TI) => (square(x) + square(y)): S3
}
```

Γ = {2: Int, 3: Int}

TA = Int
TB = Boolean

TC = TH
TA = TG

S1 = S2
TD = S2
TD = S1

TF = Int
TE = TF

TE = TG
TI = TE
TH = TI -> S3
S3 = Int
S3 = TF

Unification algorithm (Robinson '71)

Finds a solution (substitution) to a set of equational constraints.

- works for any constraint set of equalities between first-order expressions
- finds the most general solution

Definition

A set of equations is in *solved form* if it is of the form

`{x1 = t1, … xn = tn}` iff variables `xi` do not appear in terms `ti`, that is

`{x1, …, xn}` ∩ `(FV(t1)` ∪… `FV(tn)) = ∅`

In what follows,

- `x` denotes a type variable (like TA, TB before)
- `t, ti, si` denote terms, that contain type variables but are not equal to them (e.g. TA -> TB)

Unification

We obtain a solved form in finite time using the non-deterministic algorithm that applies the following rules as long as no clash is reported and as long as the equations are not in solved form.

Orient: Select `t = x, t ≠ x` and replace it with `x = t`.

Delete: Select `x = x`, remove it.

Eliminate: Select `x = t` where `x` does not occur in `t`, substitute `x` with `t` in all remaining equations.

Occurs Check: Select `x = t`, where `x` occurs in `t`, report crash.

Decomposition: Select `f(t1, …, tn) = f(s1, …, sn`, replace with `t1 = s1, …, tn = sn.`

Decomposition Clash: `f(t1, …, tn) = g(s1, …, sn), f ≠ g`, report clash.

Here, f and g can be function as in our examples, but also for example polymorphic types

`Map[A, B] = Map[C, D]` will be replaced by `A = B` and `C = D`

Solving constraints

On the board…

Compute all the types…

Example 1:

```
def foo(s: String) = s.length
def bar(x, y) = foo(x) + y
```

Example 2:

```
def baz(a, b) = a(b) :: b
```

The operator :: concatenates a list (type List[A]) with an element of the appropriate type A.

Example 3:

```
def twice(f) = (x) => f(f(x))
def succ(x) = x + 1
twice(succ)(5)
```

Example 4

a) Compute the types for the following function:

```
def count(f) ={
    (l) => {
        var c = 0
        var i = 0
        while(i < l.length){
            if(f(l(i))) c = c + 1
            i = i + 1
        }
        c
    }
}
```

b) Now consider applying this function in the following two ways:

Does the algorithm as we have it still work?

```
val list1 = List(0, 1, 2, 3, 4, 5, 7, 8)
val c1 = count((x) => x % 2 == 0)(list1)

val list2 = List(true, false, true, false)
val c2 = count((x) => x)(list2)
```