Lecture 10: **Type Inference**

Type inference

Languages such as Haskell, ML, ocaml support inference of types in most cases

Using Amy syntax, with type inference we could write programs without type annotations:

```
def message(s, verbose) = {
  if (verbose > 1) { print(s) }
  else { print(".") }
}
```

The system would infer types of parameters and result, and check that the program type checks. If it is not possible to find types, the type checker will still complain.

- as concise code as in untyped language
- type inference still catches meaningless programs

Today we explain how to do such type inference, for simple types

Intuition and key ideas

```
def message(s, verbose) = {
  if (verbose > 1) { print(s) }
  else { print(".") }
}
```

$$\frac{> : \mathit{Int} \times \mathit{Int} \to \mathit{Bool},\ \ \mathit{verbose} : \tau_{\mathit{verbose}}, 1 : \mathit{Int}}{(\mathit{verbose} > 1) : \mathit{Bool}}$$

so $\tau_{\mathit{verbose}} = \mathit{Int}$, for application of $>$ to make sense.

$$\frac{\mathit{print} : \mathit{String} \to \mathit{Unit},\ \ s : \tau_s}{\mathit{print}(s) : \mathit{Unit}}$$

so $\tau_s = \mathit{String}$, for application of *print* to make sense.
Both if branches return Unit, and so should message
Strategy:

1. Use type variables (e.g. τ_{verbose}, τ_s) to denote unknown types
2. Use type checking rules to derive constraints among type variables (arguments have expected types)
3. Use unification algorithm to solve constraints

Language extended with tuples and functions

(Still) a language similar to one in the project.
Types are:

1. primitive types: Int, Bool, String, Unit
2. type constructors: Pair[A,B] denotes $A \times B$, Function[A,B] denotes $A \to B$ (A,B are argument types)

We can write Function[Pair[Int,Int], Bool] as $(Int \times Int) \to Bool$
This gives abstract syntax of types:

$$t := Int \mid Bool \mid String \mid Unit \mid (t_1 \times t_2) \mid (t_1 \to t_2)$$

Terms now also include pairs, anonymous functions:

$$t := x \mid c \mid f(t_1, \ldots, t_n) \mid \textbf{if } (t) \ t_1 \textbf{ else } t_2 \mid (t_1, t_2) \mid (x \Rightarrow t)$$

x denotes variables, c literals
Primitives P1,P2 for pair components, if $t = (x, y)$ then $P1(t) = x$, $P2(t) = y$. Like Scala's $t.\_1$ and $t.\_2$

Usual type rules

$$\frac{\Gamma \vdash b : \mathit{Bool} \quad \Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash (\textbf{if } (b) \ t_1 \textbf{ else } t_2) : \tau}$$

$$\frac{\Gamma \vdash f : \tau_1 \times \cdots \times \tau_n \to \tau_0 \quad \Gamma \vdash t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau_0}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

Rules for constants:

$$\frac{}{\text{"..."} : \mathit{String}} \qquad \frac{}{\mathit{true} : \mathit{Boolean}} \qquad \frac{}{\mathit{false} : \mathit{Boolean}} \qquad \ldots$$

Rules for pairs

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash (t_1, t_2) : (\tau_1, \tau_2)}$$

If the first component t_1 has type τ_1 and the second component t_2 has type τ_2 then the pair (t_1, t_2) has the type (τ_1, τ_2).

$$\frac{\Gamma \vdash t : (\tau_1, \tau_2)}{\Gamma \vdash P1(t) : \tau_1}$$

$$\frac{\Gamma \vdash t : (\tau_1, \tau_2)}{\Gamma \vdash P2(t) : \tau_2}$$

Rules for anonymous function

$$\frac{\Gamma[x := \tau_1] \vdash t : \tau_2}{\Gamma \vdash (x \Rightarrow t) : (\tau_1 \to \tau_2)}$$

What does this rule say?

Rules for anonymous function

$$\frac{\Gamma[x := \tau_1] \vdash t : \tau_2}{\Gamma \vdash (x \Rightarrow t) : (\tau_1 \to \tau_2)}$$

What does this rule say?

Anonymous function $x \Rightarrow t$ that maps x to the value given by term t has a function type.

Rules for anonymous function

$$\frac{\Gamma[x := \tau_1] \vdash t : \tau_2}{\Gamma \vdash (x \Rightarrow t) : (\tau_1 \to \tau_2)}$$

What does this rule say?

Anonymous function $x \Rightarrow t$ that maps x to the value given by term t has a function type.

The type of this function is $\tau_1 \to \tau_2$, where τ_1 is the type of x and τ_2 is the type of t.

Rules for anonymous function

$$\frac{\Gamma[x := \tau_1] \vdash t : \tau_2}{\Gamma \vdash (x \Rightarrow t) : (\tau_1 \to \tau_2)}$$

What does this rule say?
Anonymous function $x \Rightarrow t$ that maps x to the value given by term t has a function type.
The type of this function is $\tau_1 \to \tau_2$, where τ_1 is the type of x and τ_2 is the type of t.
Inside t there may be uses of x, which has some type τ_1. This is why Γ is extended with binding of x to τ_1 when type checking t.

Example

```
def translatorFactory(dx, dy) = {
  p ⇒ (P1(p) + dx, P2(p) + dy)
}
def upTranslator = translatorFactory(0, 100)
def test = upTranslator((3, 5)) // computes (3, 105)
```

Type inference can find types that correspond to this annotated program:

```
def translatorFactory(dx: Int, dy: Int) : (Int,Int) ⇒ (Int,Int) = {
  (p : (Int,Int)) ⇒ (P1(p) + dx, P2(p) + dy)
}
def upTranslator : (Int,Int) ⇒ (Int,Int) =
  translatorFactory(0, 100)
def test: (Int,Int) =
  upTranslator((3, 5))
```

Type checking showing inferred types work

```
def translatorFactory(dx: Int, dy: Int) : (Int,Int) ⇒ (Int,Int) = {
  (p : (Int,Int)) ⇒ (P1(p) + dx, P2(p) + dy)
}
def upTranslator : (Int,Int) ⇒ (Int,Int) =
  translatorFactory(0, 100)
def test: (Int,Int) =
  upTranslator((3, 5))
```

Example steps in type derivation:

$$\dfrac{\dfrac{\dfrac{\Gamma[p := (Int \times Int)] \vdash P1(p) : Int \quad \Gamma[p := (Int \times Int)] \vdash dx : Int}{\Gamma[p := (Int \times Int)] \vdash (P1(p) + dx) : Int \quad \ldots}}{\Gamma[p := (Int \times Int)] \vdash (P1(p) + dx, P2(p) + dy) : (Int \times Int)}}{\Gamma \vdash p \Rightarrow (P1(p) + dx, P2(p) + dy) : ((Int \times Int) \rightarrow (Int \times Int))}$$

How do we discover $dx : Int$? We construct the derivation tree keeping type of dx symbolic until some derivation step tells us what it must be. Here, $+$ expects two integers in $P1(p) + dx$

Deriving constraints in type inference

```
def translatorFactory(dx, dy) = {
  p ⇒ (P1(p) + dx, P2(p) + dy)
}
```

Let $\Gamma_1 = \Gamma[p := \tau_p]$

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma_1 \vdash p : \tau_p \quad \tau_p = (\tau_3, \tau_4)}{\Gamma_1 \vdash P1(p) : \tau_3 \quad \Gamma_1 \vdash dx : \tau_{dx} \quad \Gamma_1 \vdash + : (Int, Int) \to Int}}{\Gamma_1 \vdash (P1(p) + dx) : \tau_1 \quad \tau_3 = Int,\ \tau_{dx} = Int,\ \tau_1 = Int}}{\Gamma_1 \vdash (P1(p) + dx, P2(p) + dy) : \tau_r \quad \tau_r = (\tau_1, \tau_2)}}{\Gamma \vdash (p \Rightarrow (P1(p) + dx, P2(p) + dy)) : \tau_{fun} \quad \tau_{fun} = \tau_p \to \tau_r}$$

Analogously, for the second component of the pair, we derive $\tau_2 = Int$, $\tau_4 = Int$ on other branches of the derivation tree.
From these constraints it follows $\tau_p = (Int, Int)$, $\tau_r = (Int, Int)$ and

$$\tau_{fun} \quad = \quad (Int, Int) \to (Int, Int)$$

Constraints from type rules: application

Consider application syntax tree node $f(t_1, \ldots, t_n)$
Assume we have a type variable τ_t for each sub-term t and the resulting term
So we have, using notation for type annotation (as in Scala):

$$(f : \tau_f)(t_1 : \tau_1, \ldots, t_n : \tau_n) : \tau_0$$

Type rule for function application:

$$\frac{\Gamma \vdash f : \tau_1 \times \cdots \times \tau_n \to \tau_0 \quad \Gamma \vdash t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash t_n : \tau_n}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau_0}$$

then simply requires

$$\tau_f = \tau_1 \times \cdots \times \tau_n \to \tau_0$$

Instead of worrying how to compute one of the types from others, we simply write down this constraint for each application node (also for primitives)

Constraints from type rules: anonymous functions

Consider anonymous function syntax tree node $x \Rightarrow t$
Assume we have a type variable for each sub-term and the bound variable x

$$((x : \tau_x) \Rightarrow (t : \tau_t)) : \tau_{fun}$$

Type rule for anonymous functions:

$$\frac{\Gamma[x := \tau_1] \vdash t : \tau_2}{\Gamma \vdash (x \Rightarrow t) : (\tau_1 \rightarrow \tau_2)}$$

then simply requires

$$\tau_{fun} = (\tau_x \rightarrow \tau_t)$$

To derive constraints just state the equality between type variables in the tree node and the types that appear in the rule.

Further constraints

tree node	constraint
$(t_1 : \tau_1, t_2 : \tau_2) : \tau$	$\tau = (\tau_1, \tau_2)$
$P1(t : \tau) : \tau_1$	$\tau = (\tau_1, \tau_2)$ τ_2 is a fresh type var.
$P2(t : \tau) : \tau_2$	$\tau = (\tau_1, \tau_2)$ τ_1 is a fresh type var.
(**if** $(b : \tau_b)$ $t_1 : \tau_1$ **else** $t_2 : \tau_2) : \tau$	$\tau = \tau_1, \tau = \tau_2, \tau_b = \mathit{Bool}$
$x : \tau_x$	$\Gamma(x) = \tau_x$
$\mathit{false} : \tau$	$\tau = \mathit{Bool}$
$\mathit{true} : \tau$	$\tau = \mathit{Bool}$
$k : \tau$	$\tau = \mathit{Int}$
"..." : τ	$\tau = \mathit{String}$

Summary of type inference

1. Introduce type variable for each tree node
2. For each tree node use type rules to derive constraints among the type variables
3. Solve the resulting set of equations on type variables

Solving equations on simple types: unification

Types in equations have the following syntax:

$$t := \tau \mid \mathit{Int} \mid \mathit{Bool} \mid \mathit{String} \mid \mathit{Unit} \mid (t_1 \times t_2) \mid (t_1 \to t_2)$$

We assume that

- primitive types are disjoint and distinct from pairs and functions
- pairs and functions are always distinct
- two pairs are equal iff their corresponding component types are equal
- two functions are equal iff their argument and result types are equal

Idea of the algorithm: eliminate variables that are alone on left or right side; decompose pair and function equalities.

Unification algorithm

Works on a set of equations. Applies the following rules as long as they change equation set

Let a denote a type variable and τ a term distinct from a

Orient: Replace $\tau = a$ with $a = \tau$ when τ is not a type variable

Delete useless: Remove $a = a$

Eliminate: Given $a = \tau$ where τ does not contain a, replace a with τ in all remaining equations

Occurs check: Given $a = \tau$ where τ contains a, report clash (type error)

Decompose pairs: Replace $(\tau_1, \tau_2) = (\tau_1', \tau_2')$ with two equations $\tau_1 = \tau_1'$ and $\tau_2 = \tau_2'$.

Decompose functions: Replace $(\tau_1 \to \tau_2) = (\tau_1' \to \tau_2')$ with two equations $\tau_1 = \tau_1'$ and $\tau_2 = \tau_2'$.

Decomposition clash: Given $(\tau_1, \tau_2) = (\tau_1' \to \tau_2')$ or $(\tau_1 \to \tau_2) = (\tau_1', \tau_2')$, or function or pair type equal to primitive type, report clash.

Primitive types: Remove equality between identical primitive types (Int, Bool, Unit, String). Report clash given equality between distinct primitive types.

Properties of unification

Algorithm always terminates (running time almost linear given the right data structures)

If it reports clash it means that equations have no solution (there exist no annotations that make program type check)

Otherwise, the equations have one or more solutions. Note that a variable that appears on left of equation does not appear on the right (else the eliminate rule would apply). Call a variable that only appears on the right a parameter.
If there are no parameters, there is exactly one solution. Otherwise, for each assignment of types to parameters we obtain a solution.
Moreover, all solutions are obtained this way. Therefore, the result of unification algorithm describes all possible ways to assign simple types to the program.

What does the algorithm do in this case?

```
def rightNest(t) = {
  (P1(P1(t)), (P2(P1(t)), P2(t)))
}
def test1 = rightNest(((1, 2), 3))
```

What happens in this case?

```
def rightNest(t) = {
  (P1(P1(t)), (P2(P1(t)), P2(t)))
}
def test1 = rightNest(((1, 2), 3))
def test2 = rightNest((false , true), false)
```

Program fails to type check because the argument type of t becomes equal to both Int and Bool, which is inconsistent.

More flexibility through generalization

```
def rightNest(t) = {
  (P1(P1(t)), (P2(P1(t)), P2(t)))
}
def test1 = rightNest(((1, 2), 3))
def test2 = rightNest((false , true), false)
```

After completing the inference for a function (e.g. rightNest), first generalize its free type variables into a variable schema:

$$\forall a, b, c.\ ((a, b), c)) \rightarrow (a, (b, c))$$

Then, each time we use the function, replace quantified variables with fresh variables.
Use in test1:

$$((a_1, b_1), c_1)) \rightarrow (a_1, (b_1, c_1))$$

$a_1 = Int$, $b_1 = Int$, $c_1 = Int$
Use in test2:

$$((a_2, b_2), c_2)) \rightarrow (a_2, (b_2, c_2))$$

$a_2 = Bool$, $b_2 = Bool$, $c_2 = Bool$
With this new approach, the program type checks

More examples for type inference

```
def S(x, y, z) = (x(z))(y(z))

def Sb(x, y, z) = (x(z))(z(x))

def cm(f, g) = x => f(g(x))

def cr(f) = x => (y => f(x,y))

def uncr(f) =
  p => (f(P1(p)))(P2(p))

def pr(x, y) = c => (c(x))(y)

def c1(p) = p(x => (y => x))

def c2(p) = p(x => (y => y))

def e(x, y) = c1(pr(x,y))
```

Occurs check

Expression x(x) generates constraint

$$\tau_x = \tau_x \rightarrow \tau_1$$

which fails occurs check.

Similarly for expression x(z)(z(x))