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

Small language with tuples and functions

Types are:

1. primitive types: Int, Bool, String, Unit
2. type constructors:
   - Pair[A,B] or (A,B) denotes set of pairs
   - Function[A,B] or $A \Rightarrow B$ denotes functions from A to B

Abstract syntax of types:

$$t := Int \mid Bool \mid String \mid Unit \mid (t_1, t_2) \mid (t_1 \Rightarrow t_2)$$

Terms include pairs and anonymous functions (x denotes variables, c literals):

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

Primitives P1,P2 for pair components, if $t = (x, y)$ then $P1(t) = x$, $P2(t) = y$.
We write them as in Scala, $t.\_1$ and $t.\_2$
For values and types, (x, y, z) is shorthand for, say, $(x, (y, z))$

Type Rules

Rule for conditionals:

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

Rules for variables:

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

Rules for constants:

$$\frac{}{\text{"...": } \mathit{String}} \qquad \frac{}{\mathit{true} : \mathit{Bool}} \qquad \frac{}{\mathit{false} : \mathit{Bool}} \qquad \cdots$$

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).

$$\frac{\Gamma \vdash t : (\tau_1, \tau_2)}{\Gamma \vdash t.\_1 : \tau_1}$$

$$\frac{\Gamma \vdash t : (\tau_1, \tau_2)}{\Gamma \vdash t.\_2 : \tau_2}$$

Functions of One argument

$$\frac{\Gamma \vdash f : \tau \Rightarrow \tau_0 \quad \Gamma \vdash t : \tau}{\Gamma \vdash f(t) : \tau_0}$$

Functions of One argument

$$\frac{\Gamma \vdash f : \tau \Rightarrow \tau_0 \quad \Gamma \vdash t : \tau}{\Gamma \vdash f(t) : \tau_0}$$

Why only one argument?

Functions of One argument

$$\frac{\Gamma \vdash f : \tau \Rightarrow \tau_0 \quad \Gamma \vdash t : \tau}{\Gamma \vdash f(t) : \tau_0}$$

Why only one argument?
Note that τ can be a tuple $(\tau_1, \ldots, \tau_n)$, so we can derive:

$$\frac{\dfrac{\Gamma \vdash t_1 : \tau_1 \quad \ldots \quad \Gamma \vdash t_n : \tau_n \quad \Gamma \vdash f : (\tau_1, \ldots, \tau_n) \Rightarrow \tau_0}{\Gamma \vdash (t_1, \ldots, t_n) : (\tau_1, \ldots, \tau_n) \quad \Gamma \vdash f : (\tau_1, \ldots, \tau_n) \Rightarrow \tau_0}}{\Gamma \vdash f((t_1, \ldots, t_n)) : \tau_0}$$

Rules for Anonymous Function

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

What does this rule say?

Rules for Anonymous Function

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

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

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

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 $\tau_1 \Rightarrow \tau_2$, where τ_1 is the type of x and τ_2 is the type of t.

Rules for Anonymous Function

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

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 $\tau_1 \Rightarrow \tau_2$, 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 ⇒ (p._1 + dx, p._2 + dy) // returns anonymous function
}
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 ⇒ (p._1 + dx, p._2 + dy) }
def upTranslator : (Int,Int) ⇒ (Int,Int) = translatorFactory(0, 100)
def test: (Int,Int) = upTranslator((3, 5))
```

Are our inferred types correct?

```
def translatorFactory(dx: Int, dy: Int): (Int,Int) ⇒ (Int,Int) = {
  p ⇒ (p._1 + dx, p._2 + dy) }
def upTranslator : (Int,Int) ⇒ (Int,Int) = translatorFactory(0, 100)
def test: (Int,Int) = upTranslator((3, 5))
```

$$\Gamma \vdash p \Rightarrow (p.\_1 + dx, p.\_2 + dy) \quad : \quad (Int, Int) \Rightarrow (Int, Int)$$

From Type Checking to Type Inference

```
def translatorFactory(dx: Int, dy: Int): (Int,Int) ⇒ (Int,Int) = {
  p ⇒ (p._1 + dx, p._2 + dy) }
def upTranslator : (Int,Int) ⇒ (Int,Int) = translatorFactory(0, 100)
def test: (Int,Int) = upTranslator((3, 5))
```

Example steps in type checking the body. Let $\Gamma' = \Gamma[p := (Int, Int)]$

$$\dfrac{\dfrac{\dfrac{\Gamma' \vdash p.\_1 : Int \qquad \Gamma' \vdash dx : Int}{\Gamma' \vdash (p.\_1 + dx) : Int \qquad \ldots}}{\Gamma' \vdash (p.\_1 + dx, p.\_2 + dy) : (Int, Int)}}{\Gamma \vdash p \Rightarrow (p.\_1 + dx, p.\_2 + dy) \quad : \quad (Int, Int) \Rightarrow (Int, Int)}$$

How did type inference 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 $p.\_1 + dx$

Deriving Constraints in Type Inference

```
def translatorFactory(dx, dy) = {
  p ⇒ (p._1 + dx, p._2 + dy)
}
```

Let $\Gamma_1 = \Gamma[p := \tau_p]$ where τ_p is to be determined later

$$
\cfrac{\cfrac{\cfrac{\cfrac{\Gamma_1 \vdash p : \tau_p \quad \tau_p = (\tau_3, \tau_4)}{\Gamma_1 \vdash p.\_1 : \tau_3 \qquad \Gamma_1 \vdash dx : \tau_{dx} \qquad \Gamma_1 \vdash + : (Int, Int) \to Int}}{\Gamma_1 \vdash p.\_1 + dx \ : \ \tau_1 \qquad \tau_3 = Int,\ \tau_{dx} = Int,\ \tau_1 = Int}}{\Gamma_1 \vdash (p.\_1 + dx, p.\_2 + dy) : \tau_r \qquad \tau_r = (\tau_1, \tau_2)}}{\Gamma \vdash (p \Rightarrow (p.\_1 + dx, p.\_2 + dy)) : \tau_{fun} \qquad \tau_{fun} = \tau_p \Rightarrow \tau_r}
$$

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

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

Constraints

Introduce type variable for each tree node. Then collect these constraints:

tree node	constraint
$(f : \tau_f)(t : \tau) : \tau_0$	$\tau_f = (\tau \Rightarrow \tau_0)$
$((x : \tau_x) \Rightarrow (t : \tau_t)) : \tau_{fun}$	$\tau_{fun} = (\tau_x \Rightarrow \tau_t)$ (x, τ_x) added to Γ' for t
$(t_1 : \tau_1, t_2 : \tau_2) : \tau$	$\tau = (\tau_1, \tau_2)$
$(t : \tau).\_1 : \tau_1$	$\tau = (\tau_1, \tau_2)$ τ_2 is a fresh type variable
$(t : \tau).\_2 : \tau_2$	$\tau = (\tau_1, \tau_2)$ τ_1 is a fresh type variable
$(\textbf{if } (b : \tau_b)\ t_1 : \tau_1 \textbf{ else } t_2 : \tau_2) : \tau$	$\tau = \tau_1, \tau = \tau_2, \tau_b = Bool$
$x : \tau_x$	$\Gamma(x) = \tau_x$
$false : \tau$	$\tau = Bool$
$true : \tau$	$\tau = Bool$
$k : \tau$	$\tau = Int$
$"..." : \tau$	$\tau = String$

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 (as in Prolog)

Types in equations have the following syntax:

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

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: eliminate variables, decompose pair and function equalities.
Algorithm works for any *term algebra* (algebra of syntactic terms)

- Pair[A,B] and Function[A,B] are two distinct binary term constructors
- Int, Bool, String are distinct nullary constructors

Unification Algorithm

Applies the following rules as long as they change equation set

Let x denote a type variable and T a type term

Orient: Replace $T = x$ with $x = T$ when τ is not a type variable

Delete useless: Remove $T = T$ (both sides syntactically identical)

Eliminate: Given $x = T$ where T does not contain x, replace x with T in all remaining equations

Occurs check: Given $x = T$ where T contains x, report clash (no solutions)

Decompose pairs: Replace $(T_1, T_2) = (T_1', T_2')$ with two equations $T_1 = T_1'$ and $T_2 = T_2'$.

Decompose functions: Replace $(T_1 \Rightarrow T_2) = (T_1' \Rightarrow T_2')$ with $T_1 = T_1'$ and $T_2 = T_2'$.

Decomposition clash (remaining cases): Given equality where two sides start with different constructors report clash (no solution). Examples: $(T_1, T_2) = (T_1' \Rightarrow T_2')$, $Int = (T_1, T_2)$, $Int = Bool$, $(T_1 \Rightarrow T_2) = String$

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.

Run the algorithm for this example

```
def rightNest(t) = {
  (t._1._1, (t._1._2, t._2))
}
def test1 = rightNest(((1, 2), 3))
```

Type variable for each sub-expression (same τ_1 for same expression, to keep it short)

$$
\begin{array}{l}
(\ \ ((t:\tau).\_1:\tau_1).\_1:\tau_2, \\
\quad (((t:\tau).\_1:\tau_1).\_2:\tau_3, (t:\tau).\_2:\tau_4):\tau_5\ \ ):\tau_6
\end{array}
$$

$$
\left|\begin{array}{l}
\color{red}{\tau = (\tau_1, \tau_{10})} \\
\tau_1 = (\tau_2, \tau_{20}) \\
\tau = (\tau_1, \tau_{30}) \\
\tau_1 = (\tau_{40}, \tau_3) \\
\tau = (\tau_{50}, \tau_4) \\
\tau_5 = (\tau_3, \tau_4) \\
\tau_6 = (\tau_2, \tau_5)
\end{array}\right|
\Rightarrow
\left|\begin{array}{l}
\tau = (\tau_1, \tau_{10}) \\
\tau_1 = (\tau_2, \tau_{20}) \\
\color{red}{(\tau_1, \tau_{10}) = (\tau_1, \tau_{30})} \\
\tau_1 = (\tau_{40}, \tau_3) \\
(\tau_1, \tau_{10}) = (\tau_{50}, \tau_4) \\
\tau_5 = (\tau_3, \tau_4) \\
\tau_6 = (\tau_2, \tau_5)
\end{array}\right|
\Rightarrow
\left|\begin{array}{l}
\tau = (\tau_1, \tau_{10}) \\
\tau_1 = (\tau_2, \tau_{20}) \\
\color{red}{\tau_1 = \tau_1}, \tau_{10} = \tau_{30} \\
\tau_1 = (\tau_{40}, \tau_3) \\
(\tau_1, \tau_{10}) = (\tau_{50}, \tau_4) \\
\tau_5 = (\tau_3, \tau_4) \\
\tau_6 = (\tau_2, \tau_5)
\end{array}\right|
\Rightarrow
$$

Applying Unification Rules Some More

$$
\left|\begin{array}{l}
\tau = (\tau_1, \tau_{10}) \\
\tau_1 = (\tau_2, \tau_{20}) \\
\tau_{10} = \tau_{30} \\
\tau_1 = (\tau_{40}, \tau_3) \\
(\tau_1, \tau_{10}) = (\tau_{50}, \tau_4) \\
\tau_5 = (\tau_3, \tau_4) \\
\tau_6 = (\tau_2, \tau_5)
\end{array}\right|
\Rightarrow
\left|\begin{array}{l}
\tau = (\tau_1, \tau_{10}) \\
\tau_1 = (\tau_2, \tau_{20}) \\
\tau_{10} = \tau_{30} \\
\tau_1 = (\tau_{40}, \tau_3) \\
\tau_1 = \tau_{50}, \tau_{10} = \tau_4 \\
\tau_5 = (\tau_3, \tau_4) \\
\tau_6 = (\tau_2, \tau_5)
\end{array}\right|
\Rightarrow
\left|\begin{array}{l}
\tau = (\tau_1, \tau_4) \\
\tau_1 = (\tau_2, \tau_{20}) \\
\tau_4 = \tau_{30} \\
\tau_1 = (\tau_{40}, \tau_3) \\
\tau_1 = \tau_{50}, \tau_{10} = \tau_4 \\
\tau_5 = (\tau_3, \tau_4) \\
\tau_6 = (\tau_2, \tau_5)
\end{array}\right|
\Rightarrow
$$

$$
\left|\begin{array}{l}
\tau = (\tau_1, \tau_4) \\
\tau_1 = (\tau_2, \tau_{20}) \\
\tau_{30} = \tau_4 \\
\tau_1 = (\tau_{40}, \tau_3) \\
\tau_{50} = \tau_1, \tau_{10} = \tau_4 \\
\tau_5 = (\tau_3, \tau_4) \\
\tau_6 = (\tau_2, \tau_5)
\end{array}\right|
\Rightarrow
\left|\begin{array}{l}
\tau = ((\tau_2, \tau_{20}), \tau_4) \\
\tau_1 = (\tau_2, \tau_{20}) \\
\tau_{30} = \tau_4 \\
(\tau_2, \tau_{20}) = (\tau_{40}, \tau_3) \\
\tau_{50} = (\tau_2, \tau_{20}), \tau_{10} = \tau_4 \\
\tau_5 = (\tau_3, \tau_4) \\
\tau_6 = (\tau_2, \tau_5)
\end{array}\right|
\Rightarrow
\left|\begin{array}{l}
\tau = ((\tau_2, \tau_{20}), \tau_4) \\
\tau_1 = (\tau_2, \tau_{20}) \\
\tau_{30} = \tau_4 \\
\tau_2 = \tau_{40}, \tau_{20} = \tau_3 \\
\tau_{50} = (\tau_2, \tau_{20}), \tau_{10} = \tau_4 \\
\tau_5 = (\tau_3, \tau_4) \\
\tau_6 = (\tau_2, \tau_5)
\end{array}\right|
\Rightarrow
$$

And More

$$
\left|\begin{array}{l}
\tau = ((\tau_2, \tau_3), \tau_4) \\
\tau_1 = (\tau_2, \tau_3) \\
\tau_{30} = \tau_4 \\
\color{red}{\tau_2 = \tau_{40}}, \tau_{20} = \tau_3 \\
\tau_{50} = (\tau_2, \tau_3), \tau_{10} = \tau_4 \\
\tau_5 = (\tau_3, \tau_4) \\
\tau_6 = (\tau_2, \tau_5)
\end{array}\right|
\Rightarrow
\left|\begin{array}{l}
\tau = ((\tau_2, \tau_3), \tau_4) \\
\tau_1 = (\tau_2, \tau_3) \\
\tau_{30} = \tau_4 \\
\tau_{40} = \tau_2, \tau_{20} = \tau_3 \\
\tau_{50} = (\tau_2, \tau_3), \tau_{10} = \tau_4 \\
\color{red}{\tau_5 = (\tau_3, \tau_4)} \\
\tau_6 = (\tau_2, \tau_5)
\end{array}\right|
\Rightarrow
\left|\begin{array}{l}
\tau = ((\tau_2, \tau_3), \tau_4) \\
\tau_1 = (\tau_2, \tau_3) \\
\tau_{30} = \tau_4 \\
\tau_{40} = \tau_2, \tau_{20} = \tau_3 \\
\tau_{50} = (\tau_2, \tau_3), \tau_{10} = \tau_4 \\
\tau_5 = (\tau_3, \tau_4) \\
\tau_6 = (\tau_2, (\tau_3, \tau_4))
\end{array}\right|
$$

The argument type is $\tau = ((\tau_2, \tau_3), \tau_4)$
The result type is $\tau_6 = (\tau_2, (\tau_3, \tau_4))$
The types τ_2, τ_3, τ_4 can be picked arbitrarily—there are infinitely many solutions.

Adding Constraints for Function Call

We have:

$$\text{rightNest} : ((\tau_2, \tau_3), \tau_4) \Rightarrow (\tau_2, (\tau_3, \tau_4))$$

Given a call rightNest(((1, 2), 3)), we add constraints equivalent to

$$(\tau_2, \tau_3), \tau_4) = ((Int, Int), Int)$$

Thus we conclude $\tau_2 = Int, \tau_3 = Int, \tau_4 = Int$. Given that

$$rightNest(((1, 2), 3)) : (\tau_2, (\tau_3, \tau_4))$$

we conclude

$$rightNest(((1, 2), 3)) : (Int, (Int, Int))$$

What happens in this case?

```
def rightNest(t) = {
  (t._1._1, (t._1._2, t._2))
}
def test1 = rightNest(((1, 2), 3))
def test2 = rightNest((false , true), false)
```

$$
\begin{array}{ll}
((\tau_2, \tau_3), \tau_4) = ((Int, Int), Int) & \text{because of test1} \\
((\tau_2, \tau_3), \tau_4) = ((Bool, Bool), Bool) & \text{because of test2}
\end{array}
$$

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

This is a pity, because we could copy rightNest into rightNest2 with the same body as rightNest, then call rightNest2((false, true), false), and everything would work. But the new program executes the same as old.

More flexibility through generalization

```
def rightNest(t) = {
  (t._1._1, (t._1._2, t._2))
}
def test1 = rightNest(((1, 2), 3))
def test2 = rightNest((false , true), false)
```

After completing the inference for rightNest, first generalize its free type variables into a variable schema:

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

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

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

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

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

$a_2 = Bool$, $b_2 = Bool$, $c_2 = Bool$

More flexibility through generalization

```
def rightNest(t) = {
  (t._1._1, (t._1._2, t._2))
}
def test1 = rightNest(((1, 2), 3))
def test2 = rightNest((false , true), false)
```

With this new approach, the program type checks and its types are inferred as follows:

```
def rightNest[A,B,C](t : ((A, B), C)) : (A, (B, C)) = {
  (t._1._1, (t._1._2, t._2))
}

def test1 : (Int, (Int, Int)) =
  rightNest[Int, Int, Int](((1, 2), 3))

def test2 : (Bool, (Bool, Bool))=
  rightNest[Bool,Bool, Bool]((false , true), false)
```