CS 320
Computer Language Processing
Exercise Set 5

April 02, 2025

Consider a type system for a simple functional language, consisting of integers, booleans, parametric pairs, and lists. The rest of the exercises will revolve around this system.

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau} \text{ (var)}$$

$$\frac{n \text{ is an integer value}}{\Gamma \vdash n : \texttt{int}} \text{ (int)}$$

$$\frac{e_1 : \texttt{int} \qquad e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \text{ (+)} \qquad \frac{e_1 : \texttt{int} \qquad e_2 : \texttt{int}}{\Gamma \vdash e_1 - e_2 : \texttt{int}} \text{ (-)}$$

$$\frac{b \text{ is a boolean value}}{\Gamma \vdash \texttt{bool}(b) : \texttt{bool}} \text{ (bool)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \wedge e_2 : \texttt{bool}} \text{ (and)} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \vee e_2 : \texttt{bool}} \text{ (or)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool}}{\Gamma \vdash \neg e_1 : \texttt{bool}} \text{ (not)}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \texttt{=} e_2 : \texttt{bool}} \text{ (eq)} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \qquad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 \texttt{<=} e_2 : \texttt{bool}} \text{ (lte)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\; e_1 \;\texttt{then}\; e_2 \;\texttt{else}\; e_3 : \tau} \text{ (ite)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : (\tau_1, \tau_2)} \text{ (pair)}$$

$$\frac{\Gamma \vdash e : (\tau_1, \tau_2)}{\Gamma \vdash \texttt{fst}(e) : \tau_1} \text{ (fst)} \qquad \frac{\Gamma \vdash e : (\tau_1, \tau_2)}{\Gamma \vdash \texttt{snd}(e) : \tau_2} \text{ (snd)}$$

$$\frac{}{\Gamma \vdash \texttt{Nil()} : \texttt{List[}\tau\texttt{]}} \text{ (nil)} \qquad \frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash e_2 : \texttt{List[}\tau\texttt{]}}{\Gamma \vdash \texttt{Cons(}e_1, e_2\texttt{)} : \texttt{List[}\tau\texttt{]}} \text{ (cons)}$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1 \texttt{ => } e : \tau_1 \texttt{ => } \tau_2} \text{ (fun)} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \texttt{ => } \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2} \text{ (app)}$$

Exercise 1 For each of the following term-type pairs (t, τ), check whether the term can be ascribed with the given type, i.e., whether there exists a derivation of $\Gamma \vdash t : \tau$ for some typing context Γ in the given system. If not, briefly argue why.

1. $\texttt{x}, \texttt{bool}$
2. $\texttt{x} + 1, \texttt{int}$
3. `(x && y) == (x <= 0)`, `bool`
4. `f => x => y => f((x, y))`:
   `((List[Int], Bool)=>Int)=>List[Int] =>Bool =>Int`
5. `Cons(x, x) : List[List[Int]]`

Solution

1. `x, Bool`. Derivation, assume `x` is a boolean:

   $$\frac{(\texttt{x}, \texttt{bool}) \in \{(\texttt{x}, \texttt{bool})\}}{\{(\texttt{x}, \texttt{bool})\} \vdash \texttt{x: Bool}}$$

   Note that this would work with any type, as there are no constraints.

2. $\texttt{x} + 1, \texttt{int}$. Derivation, assume `x` is an integer:

   $$\frac{\dfrac{(\texttt{x}, \texttt{int}) \in \{(\texttt{x}, \texttt{int})\}}{\{(\texttt{x}, \texttt{int})\} \vdash \texttt{x: Int}} \qquad \dfrac{1 \in \mathbb{N}}{\{(\texttt{x}, \texttt{int})\} \vdash \texttt{1: Int}}}{\{(\texttt{x}, \texttt{int})\} \vdash \texttt{x + 1:Int}}$$

   Due to addition constraining the type of `x`, other possible types would not work.

3. `(x && y) == (x <= 0)`, `bool`. Not well-typed. From the left-hand side, we would enforce that `x: Bool`, but on the right, we find `x: Int`. Due to this conflict, there is no valid derivation for this term.

4. `f =>x =>y =>f((x, y))`: this is the currying function. Note that it will conform to `((a, b)=> c) => a =>b =>c` for any choice of `a`, `b`, and `c`. (check)

5. `Cons(x, x) : List[List[Int]]`. Not well-typed. The *cons* rule tells us that the second argument must have the same type as the result, so `x: List[List[Int]]`, but the first argument enforces the type to be `List[Int]` (again, due to result type). As `int` $\neq$ `List[int]`, this is not well-typed.

   Note that the singular assignment of `x` to `Nil()` can make a well typed term here, but the typing must hold for *all* possible values of `x`.

$\square$

Exercise 2 A *program* is a top-level expression t accompanied by a set of user-provided function definitions. The program is well-typed if each of the function bodies conform to the type of the function, and the top-level expression is well-typed in the context of the function definitions.

For each of the following function definitions, check whether the function body is well-typed:

1. `def f(x:Int)(y:Int):Bool = x <=y`
2. `def rec(x:Int):Int = rec(x)`
3. `def fib(n:Int):Int =if n <= 1 then 1 else (fib(n - 1)+ fib(n - 2))`

Solution

1. Well-typed, apply rule *Leq*.
2. Well-typed. We need to check if the body conforms to the output type, if we know the function and its parameters have their ascribed type. So, under the context `rec: Int =>Int, x:Int`, we need to prove that `rec(x):Int`. This follows from the *app* rule.

   So, if we allow recursion and do not check for termination, we can prove unexpected things using the non-terminating programs.
3. Well-typed. We need to produce a derivation of the following:

   `fib: Int =>Int,n: Int` $\vdash$ `if n <= 1 then 1 else (fib(n - 1)+ fib(n - 2)):Int`

   i.e., given that `fib` inductively has type `Int =>Int` and the parameter `n` has type `Int`, we need to prove that the body of the function has the ascribed type `Int`.

   The derivation can be constructed by following the structure of the term on the right-hand side, the body. We set $\Gamma =$ `fib: Int =>Int,n: Int` for brevity. The `n-2` branch is skipped due to space and being the same as the `n-1` branch.

$$\dfrac{\dfrac{\dfrac{(\texttt{n},\texttt{int})\in\Gamma}{\Gamma\vdash\texttt{n: Int}}\quad\dfrac{1\in\mathbb{N}}{\Gamma\vdash\texttt{1: Int}}}{\Gamma\vdash\texttt{n <=1: Bool}}\quad\dfrac{1\in\mathbb{N}}{\Gamma\vdash\texttt{1: Int}}\quad\dfrac{\dfrac{\dfrac{(\texttt{fib, Int =>Int})\in\Gamma}{\Gamma\vdash\texttt{fib: Int =>Int}}\quad\dfrac{\dfrac{(\texttt{n},\texttt{int})\in\Gamma}{\Gamma\vdash\texttt{n: Int}}\quad\dfrac{1\in\mathbb{N}}{\Gamma\vdash\texttt{1: Int}}}{\Gamma\vdash\texttt{(n - 1):Int}}}{\Gamma\vdash\texttt{fib(n - 1):Int}}\quad\dfrac{\dots}{\Gamma\vdash\texttt{fib(n - 2):Int}}}{\Gamma\vdash\texttt{fib(n - 1)+ fib(n - 2):Int}}}{\Gamma\vdash\texttt{if n <=1 then 1 else (fib(n - 1)+ fib(n - 2)):Int}}$$

$\square$

Exercise 3 Consider the following term t:

$$t = \texttt{1 => map(1)(x =>fst(x)(snd(x))+ snd(x))}$$

where `map` is a function with type $\forall\tau,\pi.\ \texttt{List}[\tau]$ `=>` $(\tau$ `=>` $\pi)$ `=>` $\texttt{List}[\pi]$.

1. Label and assign type variables to each subterm of t.
2. Generate the constraints on the type variables, assuming t is well-typed, to infer the type of t.
3. Solve the constraints via unification to deduce the type of t.

Solution

1. We can label the subterms in the following way:

$$t : \tau = \texttt{l => map(l)(x => fst(x)(snd(x))+ snd(x))} \tag{1}$$
$$t_1 : \tau_1 = \texttt{map(l)(x => fst(x)(snd(x))+ snd(x))} \tag{2}$$
$$t_2 : \tau_2 = \texttt{x => fst(x)(snd(x))+ snd(x)} \tag{3}$$
$$t_3 : \tau_3 = \texttt{fst(x)(snd(x))+ snd(x)} \tag{4}$$
$$t_4 : \tau_4 = \texttt{fst(x)(snd(x))} \tag{5}$$
$$t_5 : \tau_5 = \texttt{snd(x)} \tag{6}$$
$$t_6 : \tau_6 = \texttt{fst(x)} \tag{7}$$
$$\texttt{l} : \tau_7 = \texttt{l} \tag{8}$$
$$\texttt{x} : \tau_8 = \texttt{x} \tag{9}$$
$$\texttt{map} : \tau_9 = \texttt{map} \tag{10}$$

   We can choose to separately label `x`, `l`, and `map`, but it does not make any difference to the result.

2. Inserting the type of `map` (thus removing τ_9), and adding constraints by looking at the top-level of each subterm, we can get the set of initial constraints, labelled by the subterm equation above they come from:

$$\tau = \tau_7 \texttt{ => } \tau_1 \tag{1}$$
$$\tau_1 = \texttt{List[}\tau_3\texttt{]} \tag{2, 4}$$
$$\tau_7 = \texttt{List[}\tau_8\texttt{]} \tag{2, 9}$$
$$\tau_2 = \tau_8 \texttt{ => } \tau_3 \tag{3}$$
$$\tau_3 = \texttt{int} \tag{4}$$
$$\tau_4 = \texttt{int} \tag{4}$$
$$\tau_5 = \texttt{int} \tag{4}$$
$$\tau_6 = \tau_5 \texttt{ => } \tau_4 \tag{5}$$
$$\tau_8 = (\tau_5', \tau_5) \tag{6}$$
$$\tau_8 = (\tau_6, \tau_6') \tag{7}$$

   for fresh type variables τ_5' and τ_6' arising from the rule for pairs.

3. The constraints can be solved step-by-step (major steps shown):

   (a) Eliminating known types (τ_3, τ_4, τ_5):

$$\tau = \tau_7 \texttt{ => } \tau_1$$
$$\tau_1 = \texttt{List[int]}$$
$$\tau_7 = \texttt{List[}\tau_8\texttt{]}$$
$$\tau_2 = \tau_8 \texttt{ => int}$$
$$\tau_6 = \texttt{int => int}$$
$$\tau_8 = (\tau_5', \texttt{int})$$
$$\tau_8 = (\tau_6, \tau_6')$$

(b) Eliminating τ_1, τ_6:

$$\tau = \tau_7 \texttt{ => List[int]}$$
$$\tau_7 = \texttt{List[}\tau_8\texttt{]}$$
$$\tau_2 = \tau_8 \texttt{ => int}$$
$$\tau_8 = (\tau_5', \texttt{int})$$
$$\tau_8 = (\texttt{int => int}, \tau_6')$$

(c) Eliminating τ_8 using either of its equations:

$$\tau = \tau_7 \texttt{ => List[int]}$$
$$\tau_7 = \texttt{List[}(\tau_5', \texttt{int})\texttt{]}$$
$$\tau_2 = (\tau_5', \texttt{int}) \texttt{ => int}$$
$$(\tau_5', \texttt{int}) = (\texttt{int => int}, \tau_6')$$

(d) Performing unification of the pair type:

$$\tau = \tau_7 \texttt{ => List[int]}$$
$$\tau_7 = \texttt{List[}(\tau_5', \texttt{int})\texttt{]}$$
$$\tau_2 = (\tau_5', \texttt{int}) \texttt{ => int}$$
$$\tau_5' = \texttt{int => int}$$
$$\texttt{int} = \tau_6'$$

(e) Eliminating τ_5' and τ_6':

$$\tau = \tau_7 \texttt{ => List[int]}$$
$$\tau_7 = \texttt{List[}(\texttt{int => int}, \texttt{int})\texttt{]}$$
$$\tau_2 = (\texttt{int => int}, \texttt{int}) \texttt{ => int}$$

(f) Eliminating τ_2, τ_7:

$$\tau = \texttt{List[}(\texttt{int => int}, \texttt{int})\texttt{] => List[int]}$$

(g) Finally, all type variables are assigned, as we eliminate τ:

$$\emptyset \text{ (no constraints left)}$$

The type of t as discovered by the unification process is:

$$\tau = \texttt{List[}(\texttt{int => int}, \texttt{int})\texttt{] => List[int]}$$

$\square$

Exercise 4 Consider the following definition for a recursive function g:

```
def g(n)(x)= if n <= 2 then (x, x)else (x, g(n - 1)(x))
```

1. Evaluate $g(3)(1)$ and $g(4)(2)$ using the definition of g. Suggest a type for the function g based on your observations.
2. Label and assign type variables to the definition parameters, body, and its subterms.
3. Generate the constraints on the type variables, assuming the definition of g is well-typed.
4. Attempt to solve the generated constraints via unification. Argue how the result correlates to your observations from evaluating g.

Solution

1. `g(3)(1)` evaluates to `(1, (1, 1))` and `g(4)(2)` evaluates to `(2, (2, (2, 2)))`. Notably, these two come from disjoint types. This suggests that the function g is not well-typed.
2. We can label the parameters, subterms, and assign a type to the function:

$$\texttt{g} : \tau \tag{1}$$
$$\texttt{n} : \tau_n \tag{2}$$
$$\texttt{x} : \tau_x \tag{3}$$
$$\mathit{body} : \tau_1 = \texttt{if n <=2 then (x, x)else (x, g(n - 1)(x))} \tag{4}$$
$$t_1 : \tau_2 = \texttt{n <=2} \tag{5}$$
$$t_2 : \tau_3 = \texttt{(x, x)} \tag{6}$$
$$t_3 : \tau_4 = \texttt{(x, g(n - 1)(x))} \tag{7}$$
$$t_4 : \tau_5 = \texttt{g(n - 1)(x)} \tag{8}$$
$$t_5 : \tau_6 = \texttt{n - 1} \tag{9}$$

3. We can generate the constraints by looking at the top-level of each subterm equation:

$$\tau = \tau_n \texttt{ => } \tau_x \texttt{ => } \tau_1 \tag{1, def}$$
$$\tau_1 = \tau_3 \tag{4}$$
$$\tau_1 = \tau_4 \tag{4}$$
$$\tau_2 = \texttt{bool} \tag{4}$$
$$\tau_n = \texttt{int} \tag{5}$$
$$\tau_3 = (\tau_x, \tau_x) \tag{6}$$
$$\tau_4 = (\tau_x, \tau_5) \tag{6}$$
$$\tau_5 = \tau_1 \tag{7, def}$$
$$\tau_6 = \texttt{int} \tag{9}$$

4. The constraints can be solved (eliminating τ_4, τ_5) to reach a set of constraints containing the recursive constraint $\tau_1 = (\tau_x, \tau_1)$. There is no type τ_1 (the output type of `g!`) satisfying this.

   This matches our previous observation where `g` produced two different sized tuples as its output.

$\square$