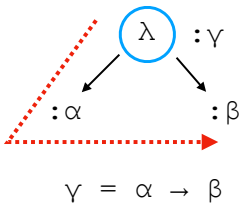
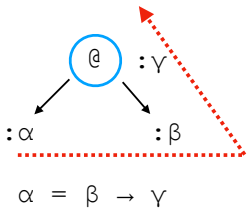
Type Inference

Type infererence is a technique used by a programming language to automatically deduce the type of a code expression. A language that features type inference allows a programmer to write code without type annotations, which is easier, while still enjoying the benefits of a typed language, namely static type checking. Since static type checking runs at compile time, a programmer can find bugs in a program before they ship that program to their customer (who finds bugs at run time).

Type inference relies on three constraint generation rules:

abstraction	application	variable
		no constraint

Procedure

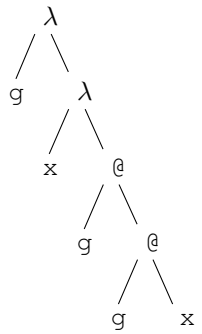
1. First, turn the program into a lambda calculus expression.
2. Label each subexpression with a type. It is easiest to do this in AST form. Write the type of any subexpression whose type is known. For the remainder, label with a type variable.
3. Using the form of each subexpression, generate type constraints.
4. Finally, using the set of type constraints, try to solve for every variable. If any variable remains after solving, it is a polymorphic type. F# denotes polymorphic types with a leading `'`, and renames them from `a` to `z` from left to right.

Example

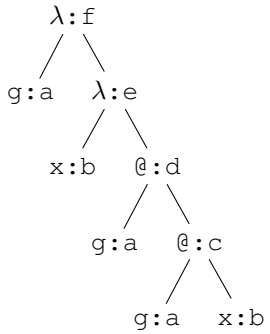
Suppose we have the F# function: `let f g x = g (g x)`

We know that `f` is a curried function with two parameters. It first applies `g` to `x` and then it applies `g` to the result of `g x`. An equivalent expression in the lambda calculus is: $\lambda g.\lambda x.g(gx)$ We omit the name of the function `f`, because in the lambda calculus, abstractions do not have names.

Here is our AST.



In this example, the type of each subexpression is not obvious. So we proceed by labeling them with variables. Observe that whenever we encounter the same variable more than once, that variable should have the same type. I like to start at the bottom of the tree, and assign variables from `a` to `z`.



Next we need to generate type constraints for each subexpression. I find it easiest to do this with the information in tabular form.

subexpression	type	constraint
`g`	`a`	n/a
`x`	`b`	n/a
`gx`	`c`	`a = b → c`
`g(gx)`	`d`	`a = c → d`
`λx.g(gx)`	`e`	`e = b → d`
`λg.λx.g(gx)`	`f`	`f = a → e`

Finally, we solve. I like to rewrite the table as I go, but since that takes up a lot of space, I will simply write my reasoning here.

1. `a = b → c` and `a = c → d` so `b → c = c → d`. Therefore `b = c` and `c = d`.
2. That means that `a = b → b`.
3. By transitivity, `b = d`.
4. Substituting what we know into `e = b → d`, then `e = b → b`.
5. Finally if `f = a → e`, then substituting again, `f = b → b → b → b`.
6. `b` is a type variable, so F# renames the type `'a → 'a → 'a → 'a`. Check using `dotnet fsi`.