Lesson 6: type inference

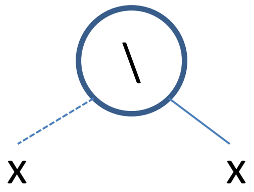
Haskell infers the types of the programs.

We see now how this is done.

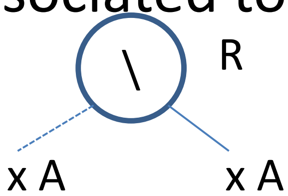
we start from the most simple function, the identity:

id x = x → id = \x -> x

we represent it with the following tree:

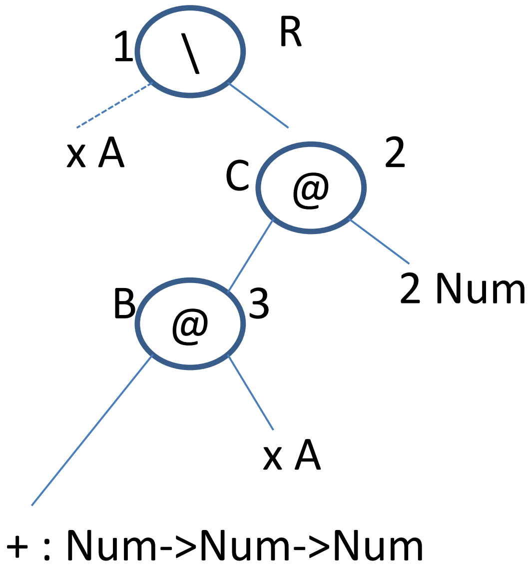


we add a type variable to each node, when the same program variable appears more than once, then the same variable is associated to all occurrences, as it happens with x for id



corresponding to the root, we write a constraint that specifies that the type of the function id, i.e. R, must be a function from the parameter type to the type of the body of the function. Thus we obtain: R= A -> A and thus id :: A -> A

2nd example: sum x = x + 2 → sum = \x-> x + 2 the tree must now represent also the application of + that we consider curried, hence with type Num -> Num -> Num



the nodes @ model function application. There are 2 such nodes for the application of + to each parameter, since + it is curried

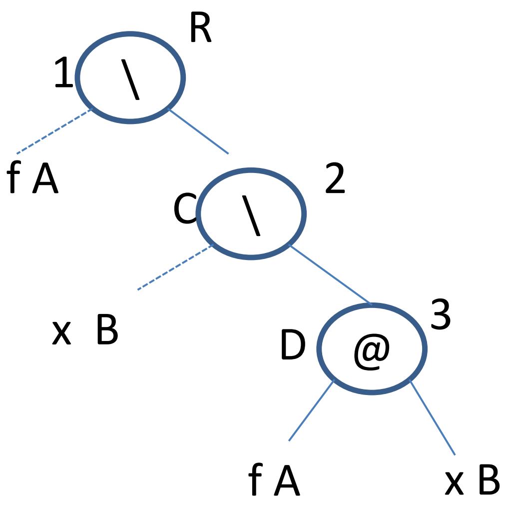
each internal node produces a constraint, the nodes are numbered to make this correspondence more clear:

(1) R = A -> C
(2) B = Num -> C
(3) Num -> Num -> Num = A -> B since + is curried its type is Num -> (Num -> Num), hence from (3) => A=Num and B = Num -> Num. If we substitute this value for B in (2) we get:

Num -> Num = Num -> C from which, C=Num, follows, substituting the values of A and C in (1) we obtain R = Num -> Num

3rd example: higher order functions

apply f x = f x → apply = \f -> \x -> f x



1) R = A -> C
2) C = B -> D
3) A = B -> D

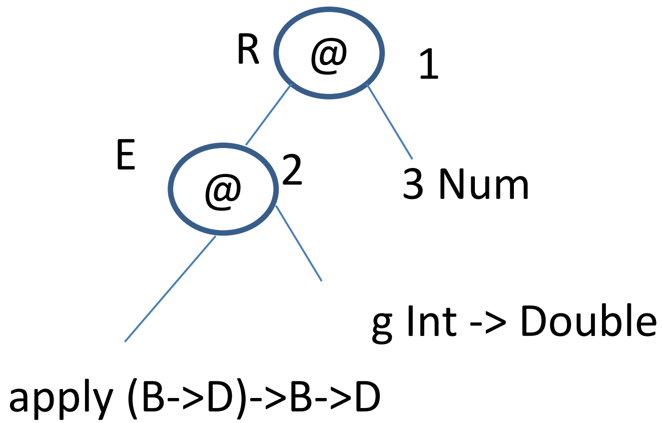
substituting (2) and (3) in (1) we obtain:

R = (B -> D) -> B -> D

the parentheses derive from the fact that A = B -> D is the argument type of the root, thus the final type of apply is (B -> D) -> B -> D

which is different from B -> D -> B -> D that would be B -> (D -> (B -> D))

4th example : apply g 3 is not a function definition but a function call, g must be defined before the call, hence it must have a type. Suppose g :: Int -> Double



(1) E = Num -> R

(2) (B->D)->B->D = (Int -> Double) -> E

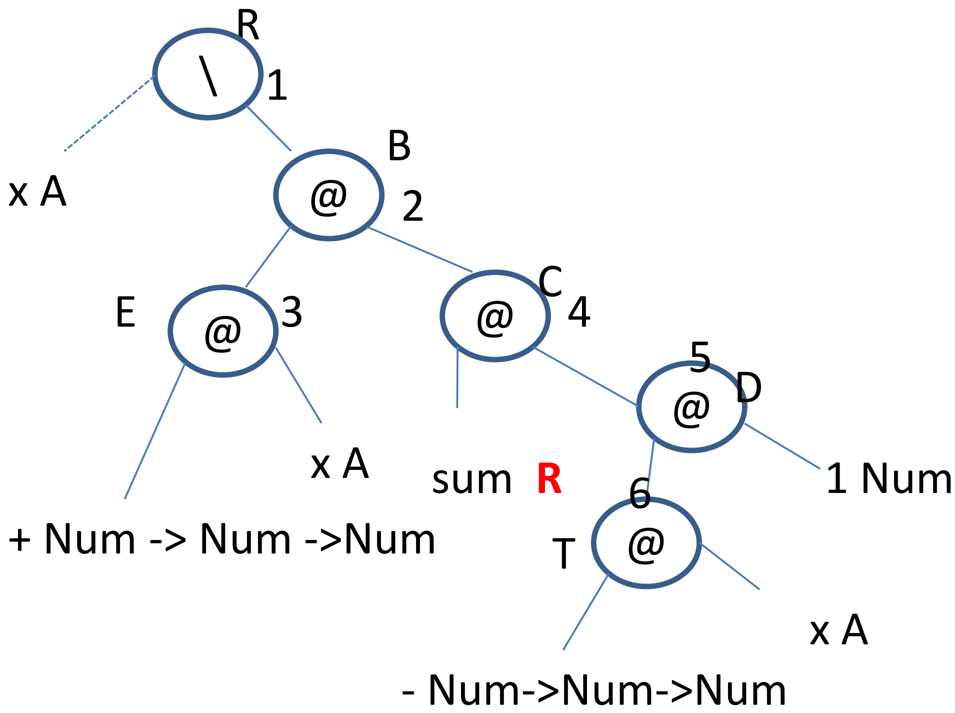
from (2) we get, Int -> Double =B->D and E=B->D ,

from which we deduce that B= Int and D= Double,
    and also that E= Int -> Double

substituting in (1) we get, Int -> Double = Num -> R
    from which we derive R = Double and the type of 3 becomes Int (in place of the more generic Num).

Hence the type of the value delivered by apply g 3 is Double

5th example: recursive function: sum x = x + (sum (x-1))



1) R=A->B
2) E=C->B
3) Num->Num->Num=A->E
4) R=D->C
5) T=Num->D
6) Num->Num->Num=A->T

from (6) A=Num and T=Num->Num that, substituted in (5) gives D=Num

From (3) we obtain, E=Num->Num and from (2) C=Num and B=Num substituting in (1) and (4) we get R=Num->Num in both cases

6th example: a function with multiple clauses

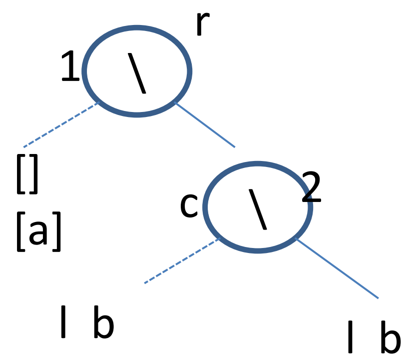
append [] l = l

append (x:xs) l = x: (append xs l)

type inference must be done for each case sepaparely and the inferred types are equalized at the end

Notice that the definition uses pattern matching and thus we need to deal with the types of the patterns

case 1: the pattern is [] :: [a]



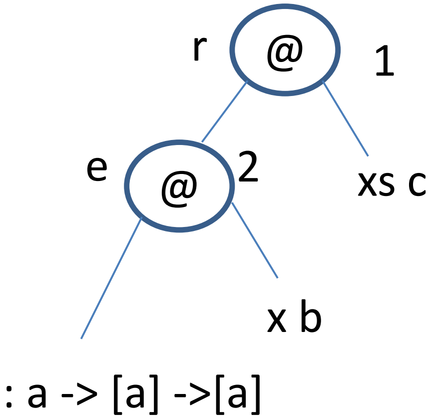
1) r= [a] -> c
2) c=b->b

substituting (2) in (1)

r=[a]->b->b

case 2: append (x:xs) l = x: (append xs l) the pattern is (x:xs)
we need to compute the type of the pattern and also that of x and xs

Lemma

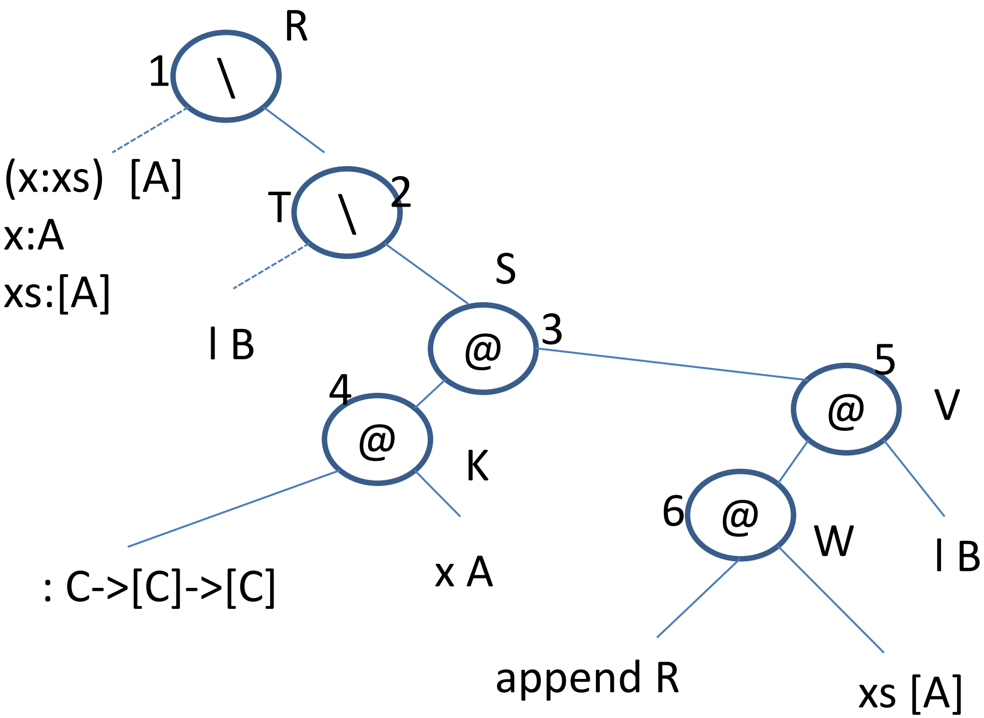


1) e=c->r
2) a->[a]->[a] = b->e

from (2) b=a and e=[a]->[a]
substituting in (1) c=[a] and r=[a]

thus the whole pattern has type [a], x::a and xs::[a]

case 2: append (x:xs) l = x: (append xs l)



1) R=[A]->T
2) T=B -> S
3) K=V->S
4) C->[C]->[C]=A->K
5) W=B->V
6) R=[A]->W

from (4) we get C=A and K=[A]->[A] and substituting K in (3) we have, V=[A] and S=[A], from (5)

W=B->[A] and from (6), R=[A]->B->[A]

we need to continue to make sure that this type is compatible with (1).

From (1) [A]->B->[A]=[A]->T, hence T=B->[A]

substituting in (2)

B->[A]=B->[A] thus ok

also (3) becomes

[A]->[A]=[A]->[A] again ok

similarly all contraints become trivially verified

hence R= [A]->B->[A]

equalize the types inferred in the two cases:

R= [A]->B->[A]=r=[a]->b->b

we get a=A b=B=[A]

thus the final type of append is:

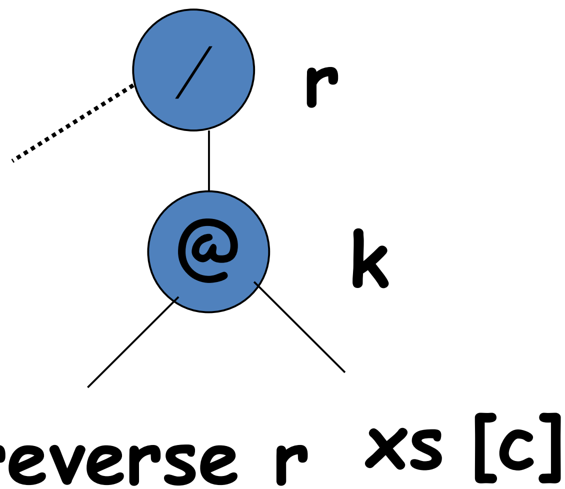
[A]->[A]->[A]

exercise

```
reverse [] = []
reverse  (x:xs)=reverse xs
reverse:: [a]→ [b]
```

(1) [a] → [b]

(2)



x:xs [c]

x c

xs [c]

r=[c] -> k

[c]->k = [a]->[b]

exercise

Consider the following Haskell function and show how its type is inferred:

```
k  f  [] = 1
k f  (x:y) = f x (k f y)
```

another exercise

Given the following Haskell datatype :

data TREE a = LEAF a | NODE a (TREE a) (TREE a);

(i) infer the type of k given below:

```
k  (LEAF x)  g = LEAF x
k  ( NODE x y z)  g = NODE  (g x) (k y g)  (k z g)
```

(ii) and now this one?

```
k  (LEAF x)  g = LEAF (g x)
k ( NODE x y z)  g = NODE  (g x) (k y g) (k z g)
```

more cases

(iii) and now?

k (LEAF x) g = g x // I deleted the LEAF call

k (NODE x y z) g = NODE (g x) (k y g) (k z g)

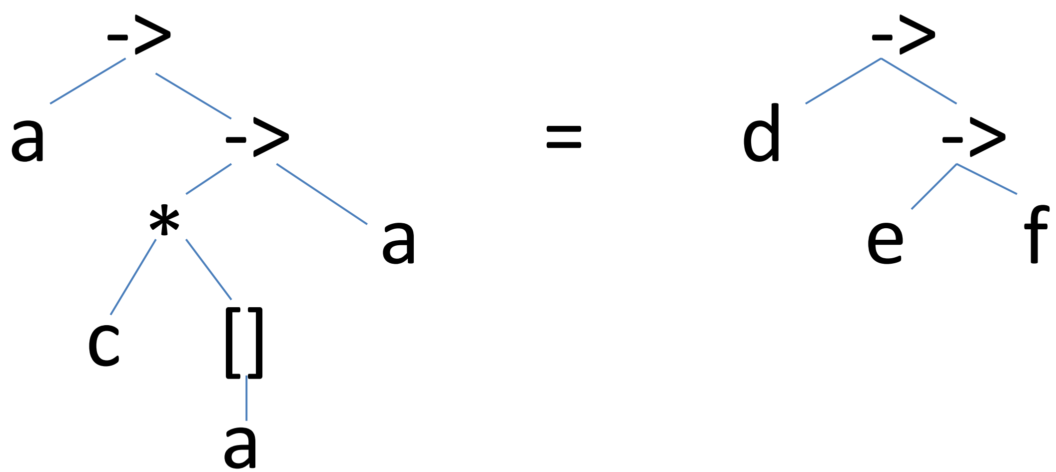
(iv) and this one?

k (LEAF x) g = x // no longer g

k (NODE x y z) g = NODE (g x) (k y g) (k z g)

unification “by hand”

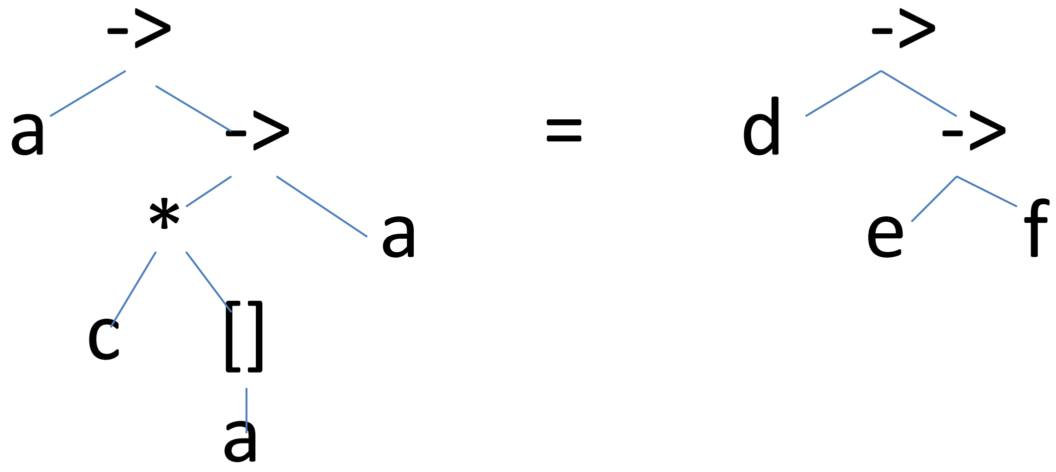
a ->((c * [a]) -> a) = d ->(e->f)



->(a, ->(*(c, [a]), a))   = ->(d, ->(e, f))

unification “by hand”

a ->((c * [a]) -> a) = d ->(e->f)



solution   a=d, e = *(c,[d]), f=d

solution of an equation set E

E is unifiable if there is a solution for it

easier to consider ground solutions == without variables in the right parts

Gsol(E)

An equation set is in **solved form** when,

1) each equation has the form x=t, where x != t

2) the equations have different left hand sides

3)these variables do not appear in the righthand sides

Observe ➔ the lefthand sides occurr exacly once

unification: $E=E_1 \rightarrow \ldots\ldots \rightarrow E_n=E'$

1. $f(t_1,..,t_n)=f(s_1,..,s_n) \rightarrow \{t_1=s_1,..,t_n=s_n\}$

2. $f(t_1,..,t_n)=g(s_1,..,s_k) \rightarrow$ stop with failure

3. $x = x$ or $a = a$ $\rightarrow$ delete

4. $t=x$ where t is not a variable $\rightarrow$ $x=t$

5. $x=t$, where t is not x and x has other occurrences in E, then, if x appears in t $\rightarrow$ stop with failure, otherwise substiture every other occurrence of x with t

example

$\{g(x)=g(g(z)), f(a,z)=f(a,y)\}$

Correctness of the unification algorithm,

1. the algorithm terminates

2. if it does not fail and produces E’, E’ is in solved form and thus is solvable and Gsol(E)=Gsol(E’)

3. if it fails then E is unsolvable.

Proof: it teminates:

each case (1) and (3) strictly decreases the number of symbols in the lhs of the equations
(4) can be done at most once per equation

After a finite number of applications of cases (1), (3) and (4) either it fails or it applies (5) whch may either fail or it eliminates all occurrences of one variable but one. Therefore (5) cannot be applied a second time to that variable and thus, (5) can be applied at most as many times as there are variables in E.

2) for each step $E\_i \rightarrow E\_(i+1)$ that succeeds, it is true that, $Gsol(E\_i)=Gsol(E\_(i+1))$

(1) $f(u)=f(v) \rightarrow u=v$

(4) $t=x \rightarrow x=t$

for (5) applied with success to $x=t$, we reason as follows:

for each solution σ of E_i, it must be that σ (x) = σ (t) and therefore, wherever in E_i there is an occurrence of x, we can substitute it with t mantaining the solution σ.

when the algorithm terminates without failing, the equation set E' that is produced is in solved form, because otherwise, the algorithm would continue

3) failure : E_i -> fail

with (2): if f(...)= g(...), then obviously E_n has no solution and thus, by part (2), also E is unsolvable

with (5): E_n contains x=t, where t is not empty and contains x ➔ x=t has no finite solution ➔ E is unsolvable

In case of success, E’ is in solved form and thus is a solution of itself and of E

E’ is a **most general unifier** for E:
for each other solution E’’ of E, E’’=E’ + extra instantiations

most general unifiers are not unique, ...but almost there can be many most general unifiers of E, but there are always a finite number of them and they are «almost equal» in the following sense:

{w=f(v), x=u, y=u, z=v} e {w=f(z), x=y, u=y, v=z} are equivalent

{y=u, y=u} and {x=y, u=y} both state the same thing: that x,y and u are equal !! The same is true for {z=v} and {v=z}

→ equivalent classes of equal variables, like {x,y,u} and {z,v}
have different set of equations that represent them