

Semantics

CMPT 379: Compilers

Instructor: Anoop Sarkar

anoopsarkar.github.io/compilers-class

Equality of types

- Main semantic tasks involve liveness analysis and checking equality
- Equality checking of types (basic types) is crucial in ensuring that code generation can target the correct instructions
- Coercions also rely on equality checking of types
- But what about those objects in PLs (records, functions, etc) that are not basic types?
- Can we perform any semantic checks on these as well?

Type Systems

- So far we have seen simple cases of type checking and coercion
- Basic types for data types: *boolean*, *char*, *integer*, *real*
- A basic type for lack of a type: *void*
- A basic type for a type error: *type_error*
- Based on these basic types we can build new types using type constructors

Type Constructors

- Arrays: `int p[10];`
 - type: *array(10, integer)*
 - multi-dim arrays: `int p[3][2]: array(3, array(2, integer))`
- Products/tuples: `pair<int, char> p(10,'a');`
 - type: *integer × char*
- Records: `struct { int p; char q; } data;`
 - Type: *record((p × integer) × (q × char))*
- Pointers: `int *p;`
 - Type: *pointer(integer)*

Type Constructors

- Functions: `int foo (int p, char q) { return 2; }`
 - Type: $integer \times char \rightarrow integer$
 - A function maps elements from the domain to the range
 - Function types map a domain type D to a range type R
 - A type for a function is denoted by $D \rightarrow R$
- In addition, type expressions can contain type variables
 - Example: $\alpha \times \beta \rightarrow \alpha$

Equivalence of Type Exprs

- Check equivalence of type exprs: s and t
- If s and t are basic types, then return true
- If $s = \text{array}(s_1, s_2)$ and $t = \text{array}(t_1, t_2)$ then return true if $\text{equal}(s_1, s_2)$ and $\text{equal}(t_1, t_2)$
- If $s = s_1 \times s_2$ and $t = t_1 \times t_2$ then return true if $\text{equal}(s_1, s_2)$ and $\text{equal}(t_1, t_2)$
- If $s = \text{pointer}(s_1)$ and $t = \text{pointer}(t_1)$ then return true if $\text{equal}(s_1, t_1)$

Polymorphic Functions

- Consider the following ML program:

```
fun null [] = true  
    | null (_::_) = false;  
fun tl (_::xs) = xs;  
fun length (alist) =  
    if null(alist) then 0  
    else length(tl(alist)) + 1;
```

- null* tests if a list is empty
- tl* removes first element and returns rest

Polymorphic Functions

- *length* is a polymorphic function (different from polymorphism in object inheritance)
- The function *length* accepts lists with elements of any basic type:
length(['a', 'b', 'c'])
length([1, 2, 3])
length([[1,2,3], [4,5,6]])
- The type for *length* is $list(\alpha) \rightarrow integer$
- α can stand for any basic type: *integer* or *char*

Polymorphic Functions

- Consider the following ML program:

***fun** map f [] = []*

| map f (x::xs) = (f(x)) :: map f xs;

- *map* takes two arguments: a function *f* and a list
- It applies *f* to each element of the list and creates a new list with the range of *f*
- Type of *map*: $(\alpha \rightarrow \beta) \rightarrow \text{list}(\alpha) \rightarrow \text{list}(\beta)$

Type Inference

- *Type inference* is the problem of determining the type of a statement from its body
- Similar to type checking and coercion
- But inference can be much more expressive when type variables can be used
- For example, the type of the *map* function on previous page uses type variables

Type Variable Substitution

- We can take a type variable in a type expression and substitute a value
- In $list(\alpha)$ we can substitute the type *integer* for the variable α to get $list(integer)$
- $list(integer) < list(\alpha)$ means $list(integer)$ is an instance of $list(\alpha)$
- $S(t)$ is a substitution for type expr t
- Replacing *integer* for α is a substitution

Type Variable Substitution

- $s < t$ means s is an instance of t
- Or s is more specific than t
- Or t is more general than s
- Some more examples:
 - $\text{integer} \rightarrow \text{integer} < \alpha \rightarrow \alpha$
 - $(\text{integer} \rightarrow \text{integer}) \rightarrow (\text{integer} \rightarrow \text{integer}) < \alpha \rightarrow \alpha$
 - $\text{list}(\alpha) < \beta$
 - $\alpha < \beta$

Type Expr Unification

- Incorrect type variable substitutions:
 - $integer < boolean$
 - $integer \rightarrow boolean < \alpha \rightarrow \alpha$
 - $integer \rightarrow \alpha < \alpha \rightarrow \alpha$
- In general, there are many possible substitutions
- Type exprs s and t unify if there is a substitution S that is most general such that $S(s) = S(t)$
- Such a substitution S is the *most general unifier* which imposes the fewest constraints on variables

Example of Type Inference

- Example:

```
fun length (alist) =  
    if null(aalist) then 0  
    else length(tl(aalist)) + 1;
```

- $length : \alpha_1$
- $null : list(\alpha_2) \rightarrow boolean$
- $alist : list(\alpha_2)$
- $null(aalist) : boolean$

Example (cont'd)

- $0 : \text{integer}$
- $\text{tl} : \text{list}(\alpha_3) \rightarrow \text{list}(\alpha_3)$
- $\text{tl}(\text{alist}) : \text{list}(\alpha_2)$
- $\text{length} : \text{list}(\alpha_2) \rightarrow \alpha_4$
- $\text{length}(\text{tl}(\text{alist})) : \alpha_4$
- $1 : \text{integer}$
- $+ : \text{integer} \times \text{integer} \rightarrow \text{integer}$
- $\text{if} : \text{boolean} \times \alpha_5 \times \alpha_5 \rightarrow \alpha_5$
- $\text{length} : \text{list}(\alpha_2) \rightarrow \text{integer}$

***fun** length (alist) =
if null(alist) **then** 0
else length(tl(alist)) + 1;*

$\text{list}(\alpha_2) \rightarrow \alpha_4 < \alpha_1$

$\text{integer} < \alpha_5$

$\text{integer} < \alpha_4$

Unification

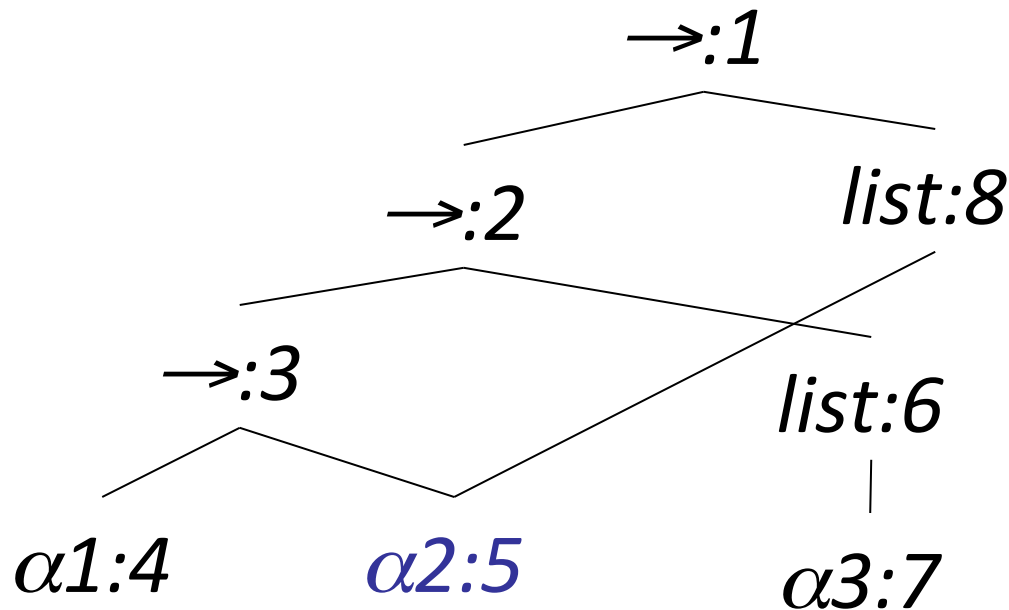
- Algorithm for finding the ***most general substitution*** S such that $S(s) = S(t)$
- Also called the ***most general unifier***
- $unify(m, n)$ unifies two type exprs m and n and returns true/false if they can be unified
- Side effect is to keep track of the *mgu* substitution for unification to succeed

Unification Algorithm

- We will explain the algorithm using an example:
 - $E: ((\alpha1 \rightarrow \alpha2) \rightarrow list(\alpha3)) \rightarrow list(\alpha2)$
 - $F: ((\alpha3 \rightarrow \alpha4) \rightarrow list(\alpha3)) \rightarrow \alpha5$
- What is the most general unifier?
 - $S_1(E) = S_1(F) ((\alpha1 \rightarrow \alpha1) \rightarrow list(\alpha1)) \rightarrow list(\alpha1)$
 - ✓ – $S_2(E) = S_2(F) ((\alpha1 \rightarrow \alpha2) \rightarrow list(\alpha1)) \rightarrow list(\alpha2)$
 - ✓ – $S_3(E) = S_3(F) ((\alpha3 \rightarrow \alpha2) \rightarrow list(\alpha3)) \rightarrow list(\alpha2)$

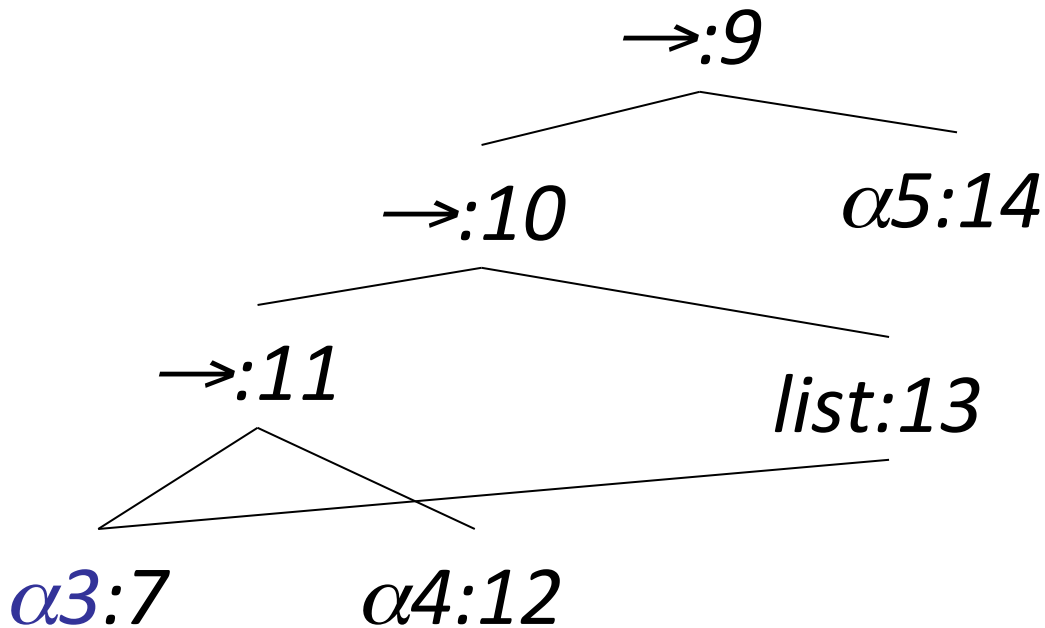
Unification Algorithm

E: $((\alpha1 \rightarrow \alpha2) \rightarrow list(\alpha3)) \rightarrow list(\alpha2)$

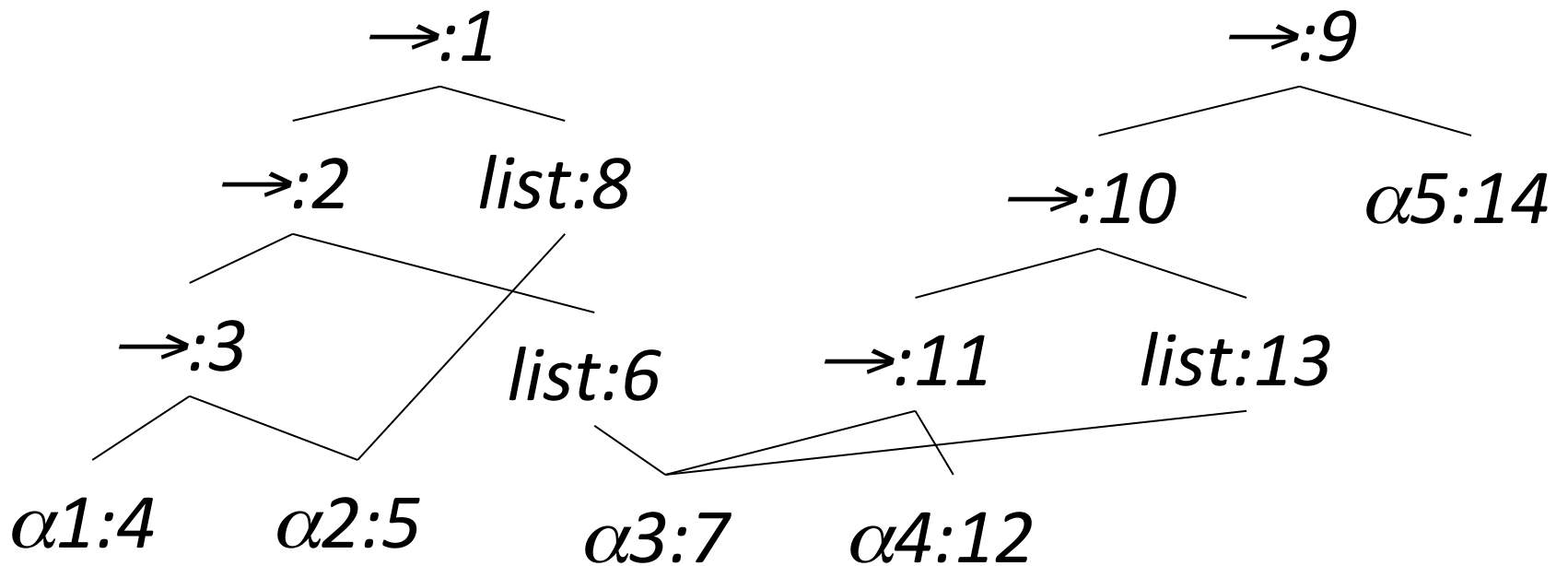


Unification Algorithm

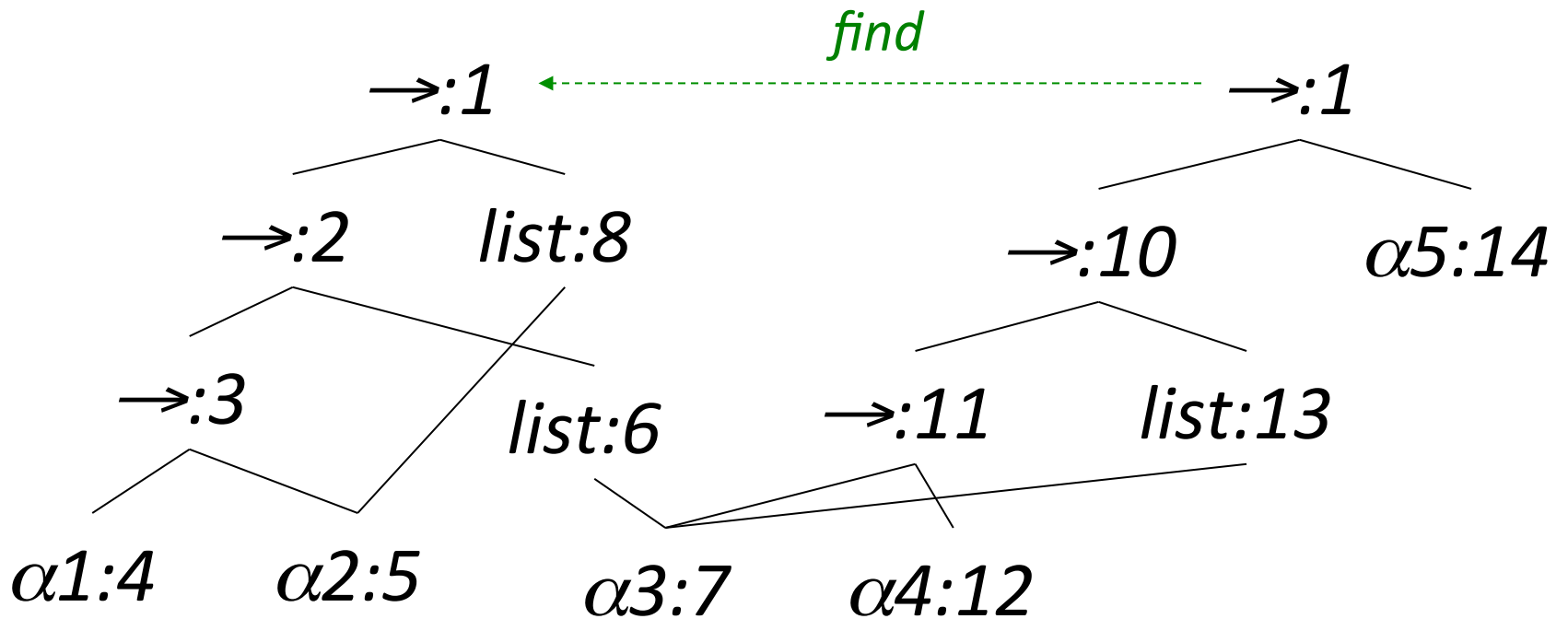
$F: ((\alpha 3 \rightarrow \alpha 4) \rightarrow \text{list}(\alpha 3)) \rightarrow \alpha 5$



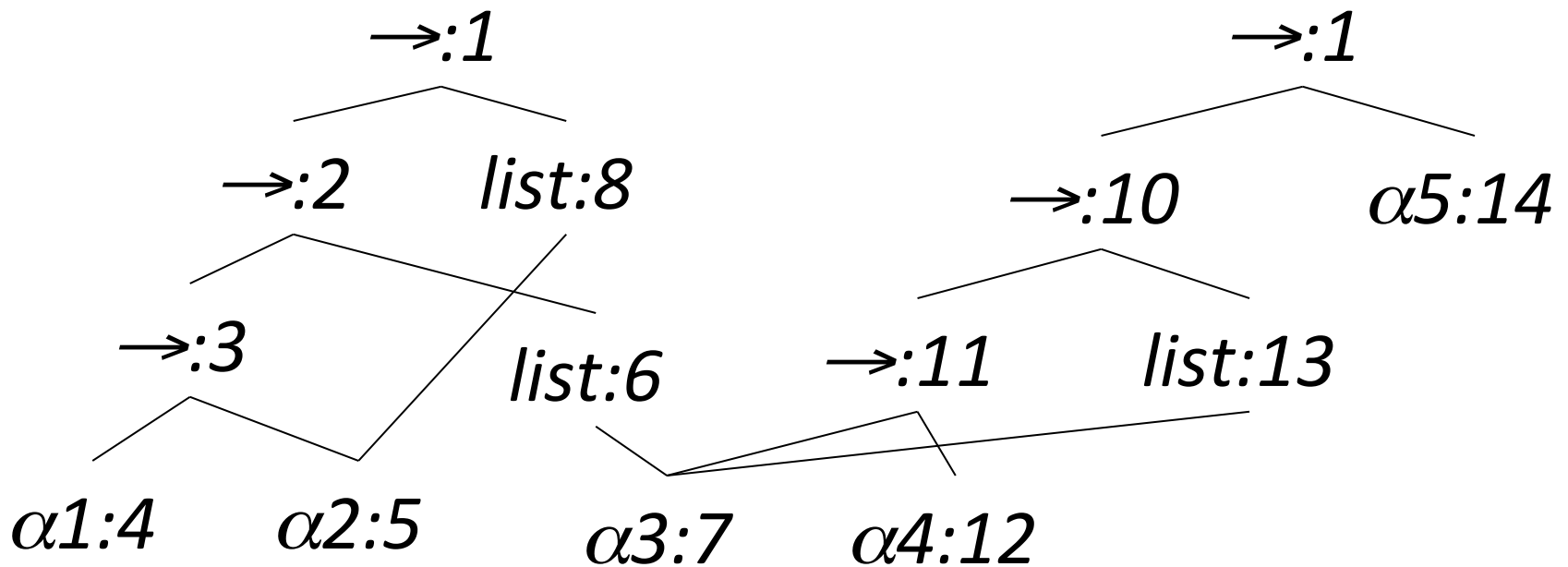
Unify(1,9)



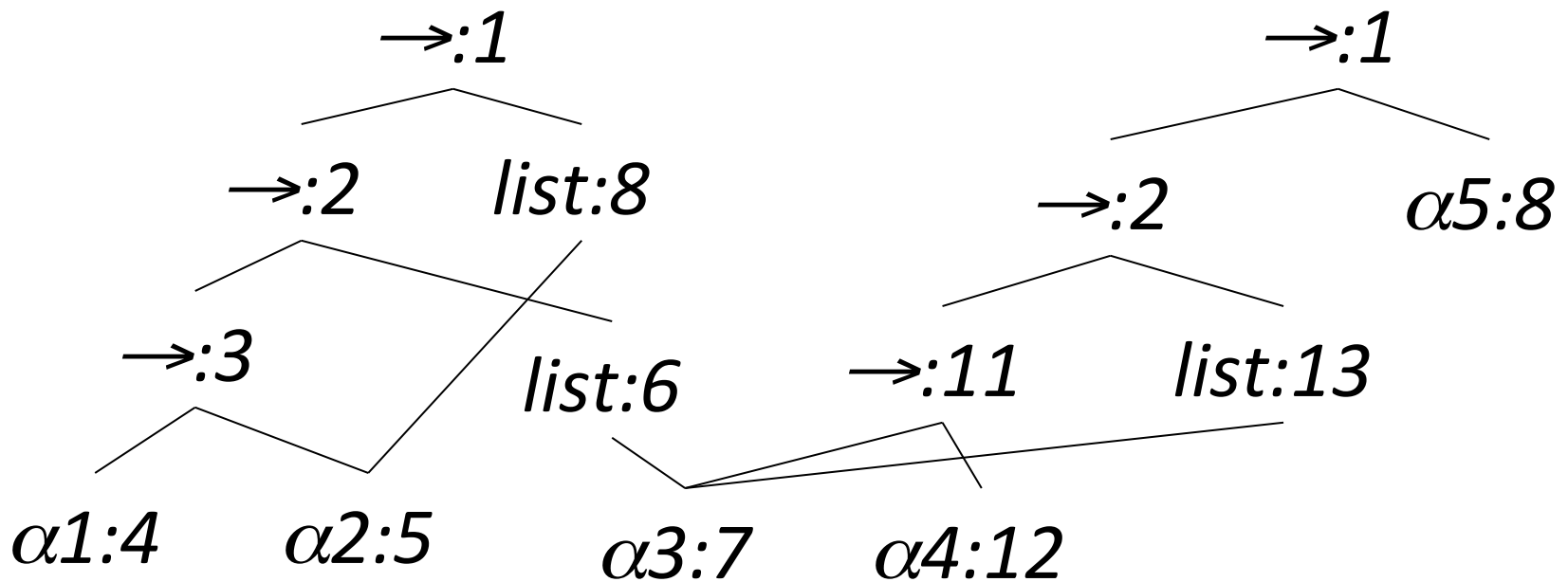
Unify(1,9)



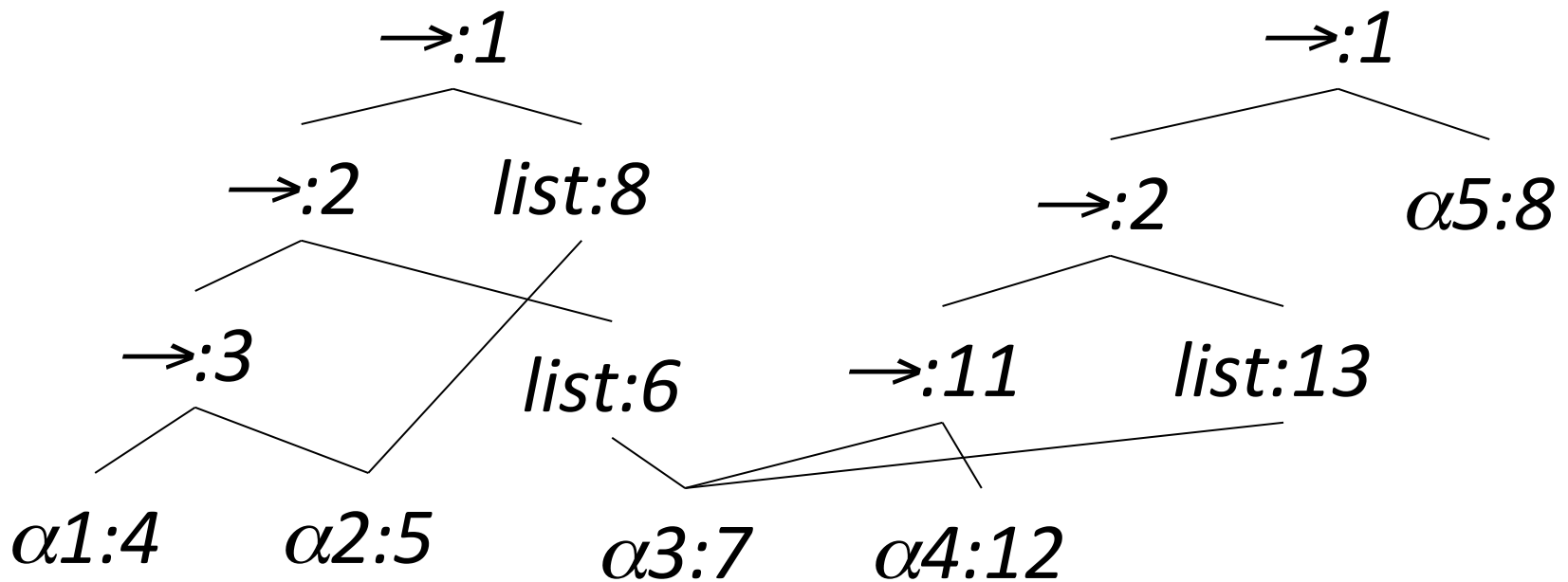
Unify(2,10) *and* Unify(8,14)



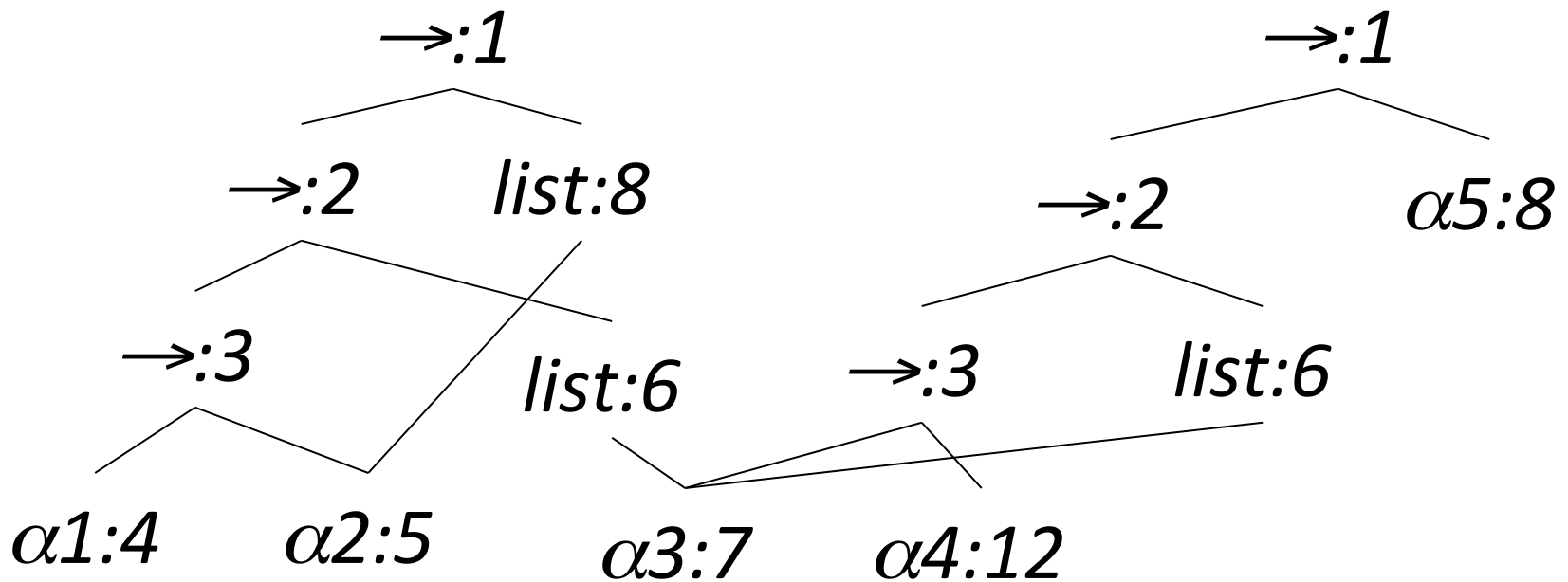
Unify(2,10) *and* Unify(8,14)



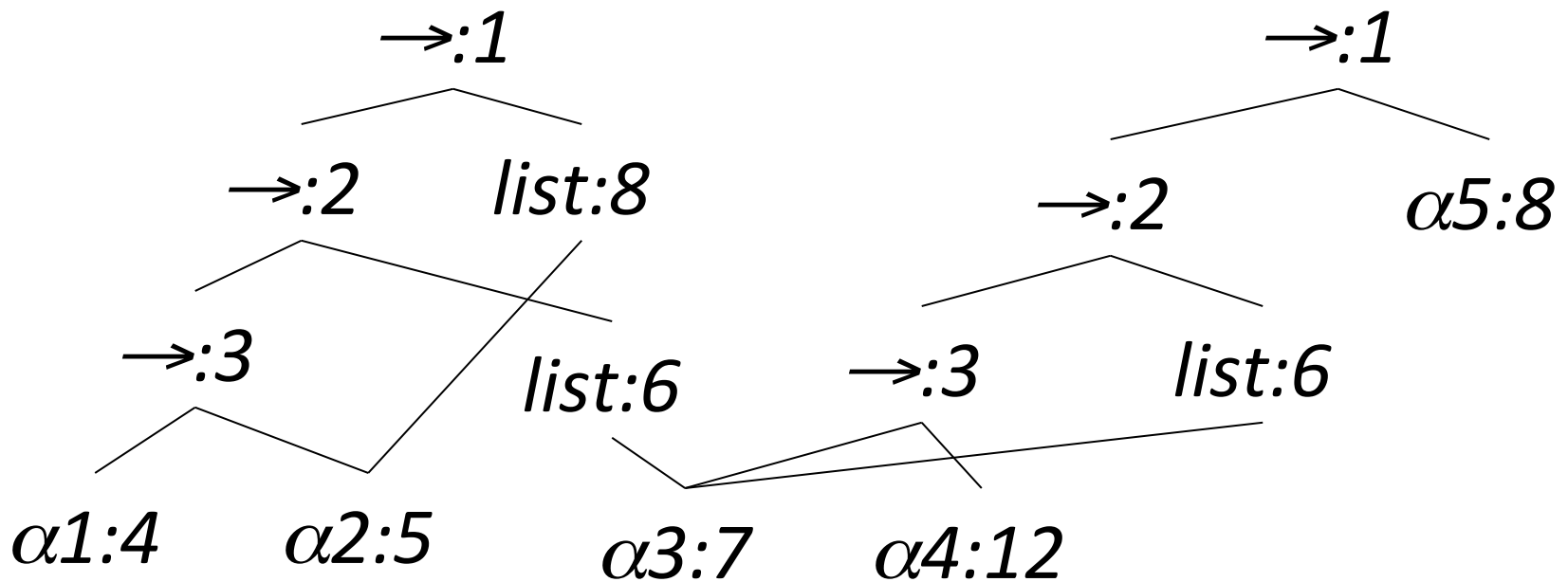
Unify(3,11) *and* Unify(6,13)



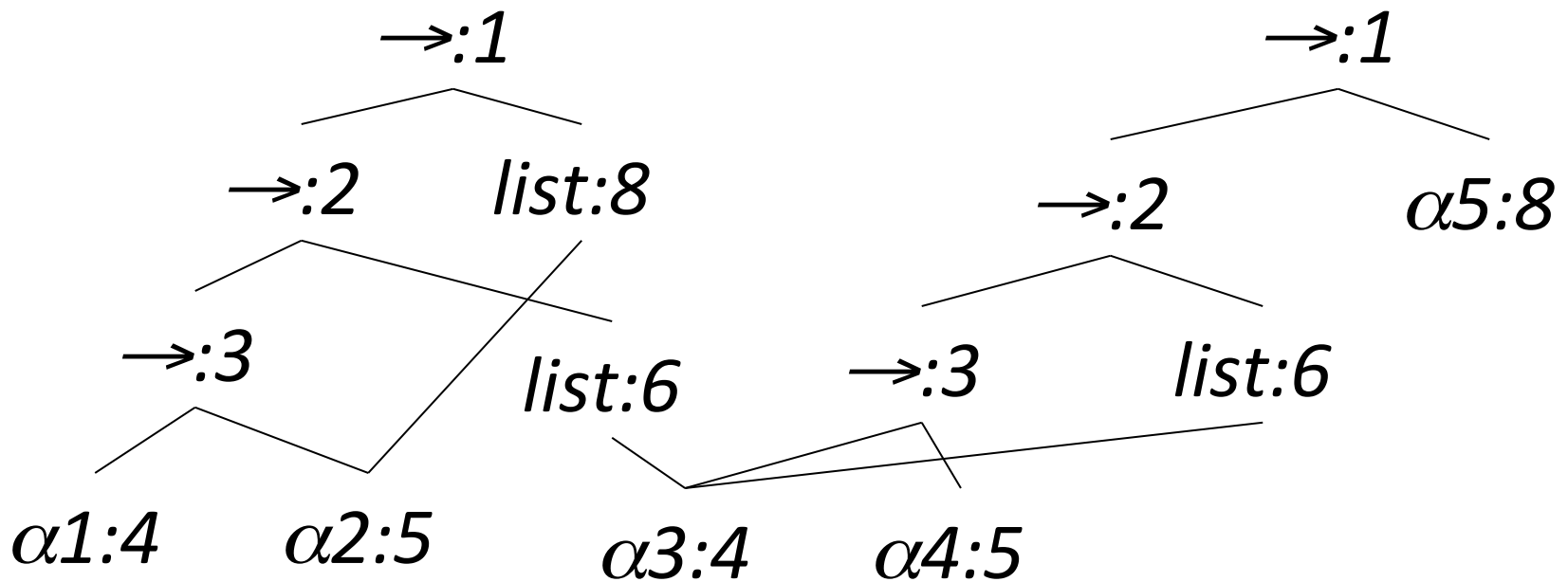
Unify(3,11) *and* Unify(6,13)



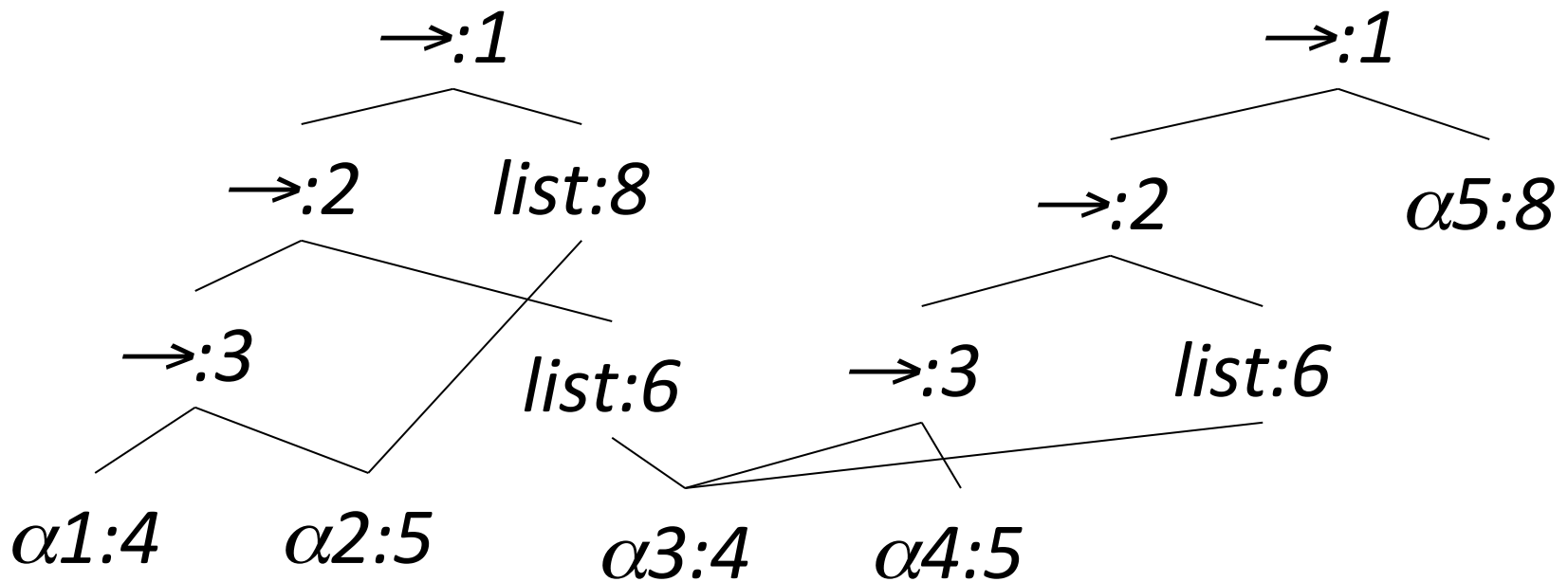
Unify(4,7) *and* Unify(5,12)



Unify(4,7) *and* Unify(5,12)



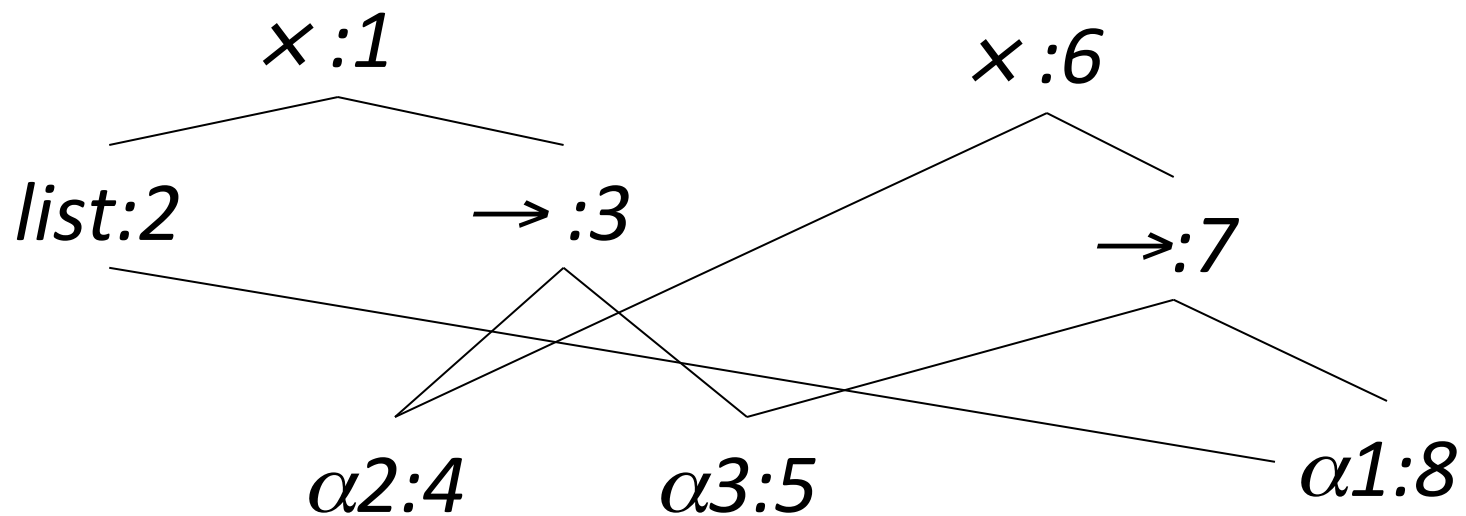
Unification success



Unification: Occur Check

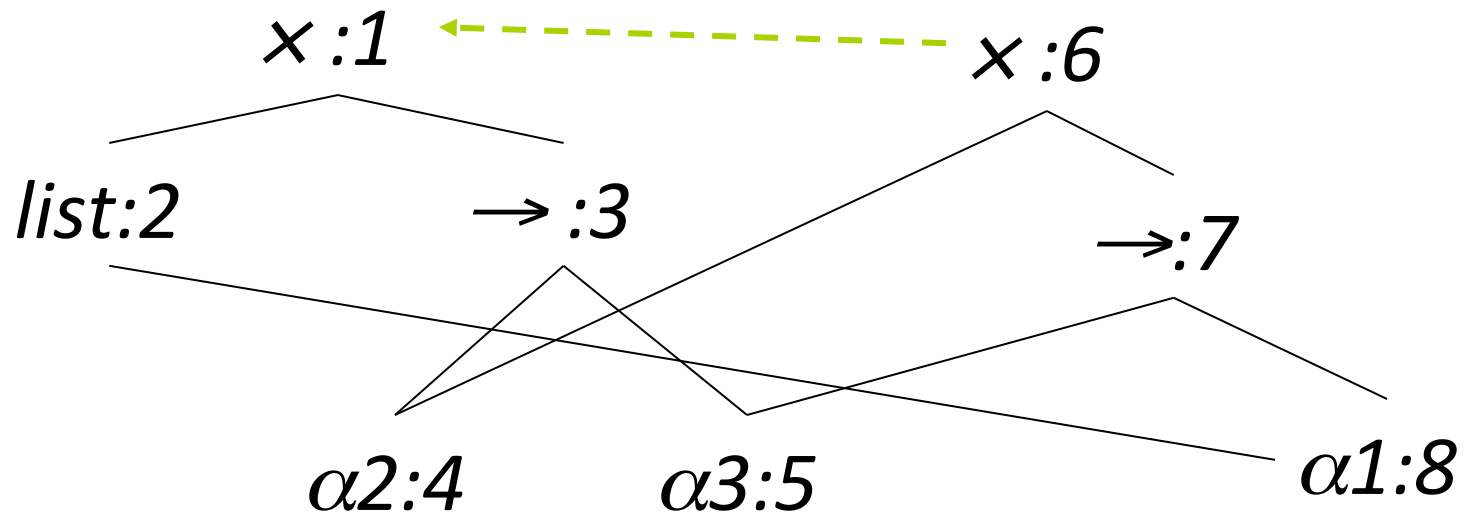
$list(\alpha 1) \times (\alpha 2 \rightarrow \alpha 3)$

$\alpha 2 \times (\alpha 3 \rightarrow \alpha 1)$



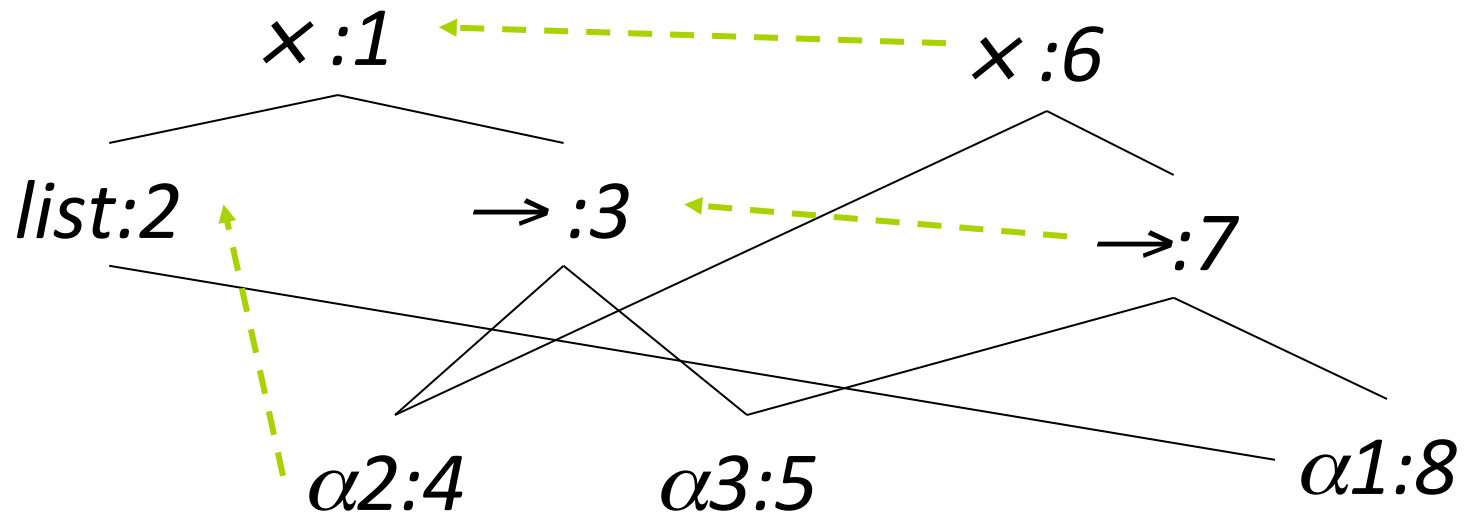
Unify(1,6)

6--1



Unify(2,4) and Unify(3,7)

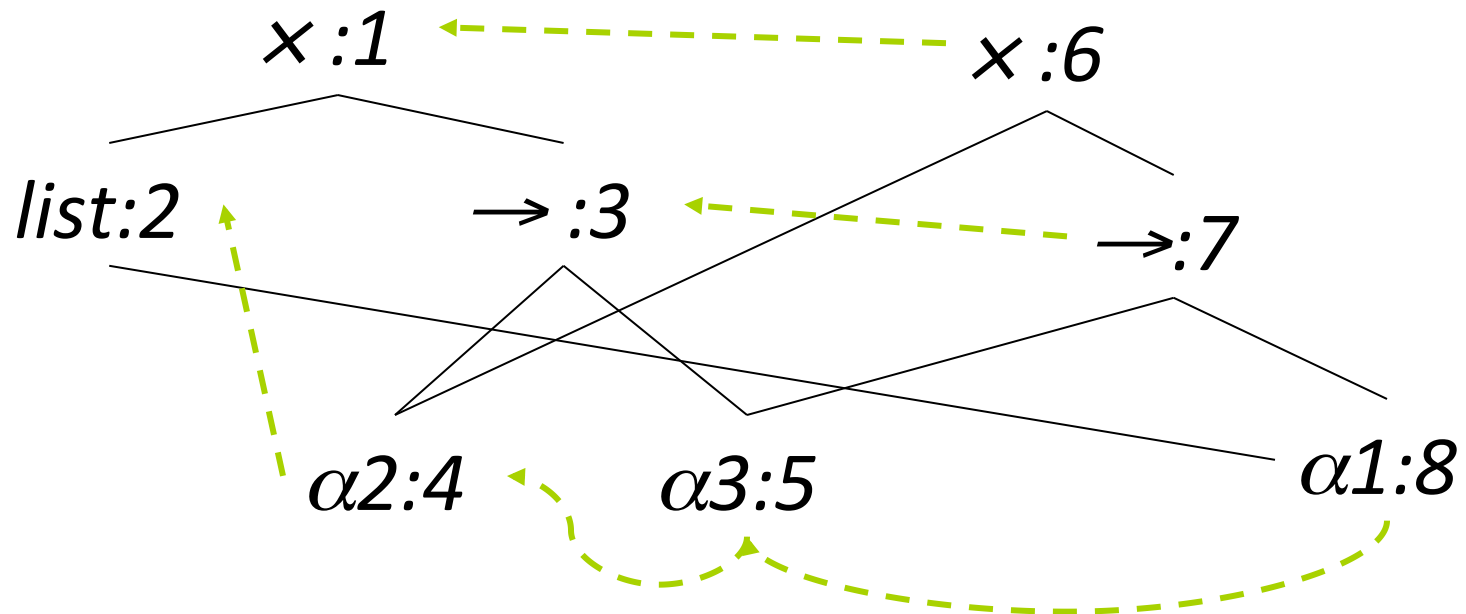
6--1, 4--2, 7--3



Unify(4,5) and Unify(5,8)

6--1, 4--2, 7--3, 5--4, 8--5

- $list(\alpha1)$
- $= list(\alpha2)$
- $= list(list(\alpha1))$



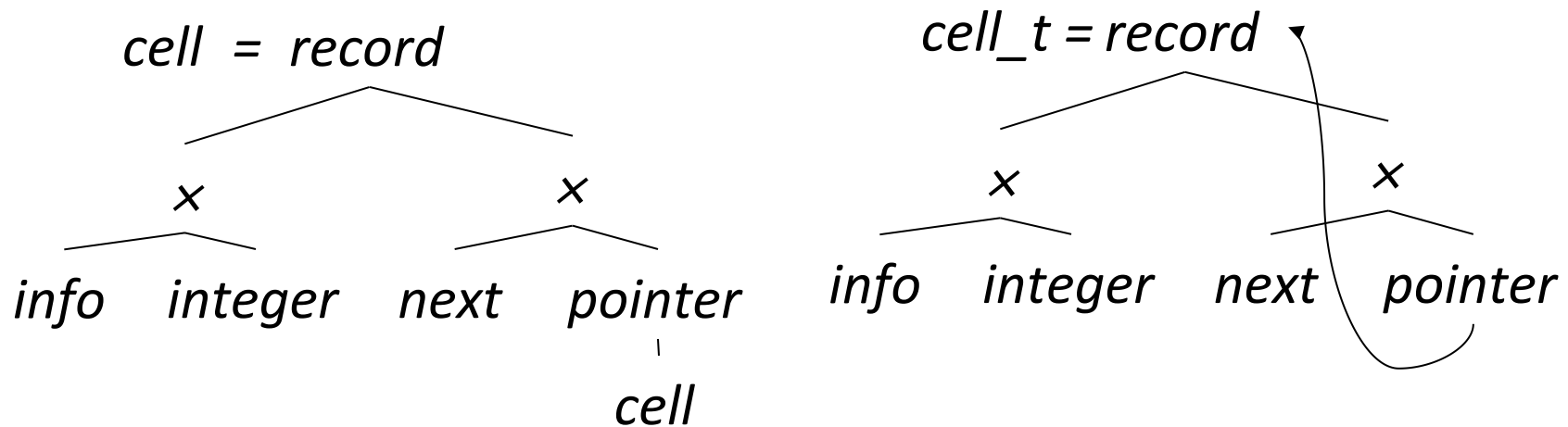
Occur Check

- Our unification algorithm creates a cycle in *find* for some inputs
- The cycle leads to an infinite loop. Note that Algorithm 6.32 in the Purple Dragon book has this bug
- A solution to this is to unify only if no cycles are created: the *occur check*
- Makes unification slower but correct

Recursive types

- Recursive types arise naturally in PLs
- For example, in pseudo-C:

```
struct cell { int info; cell_t *next; } cell_t;
```

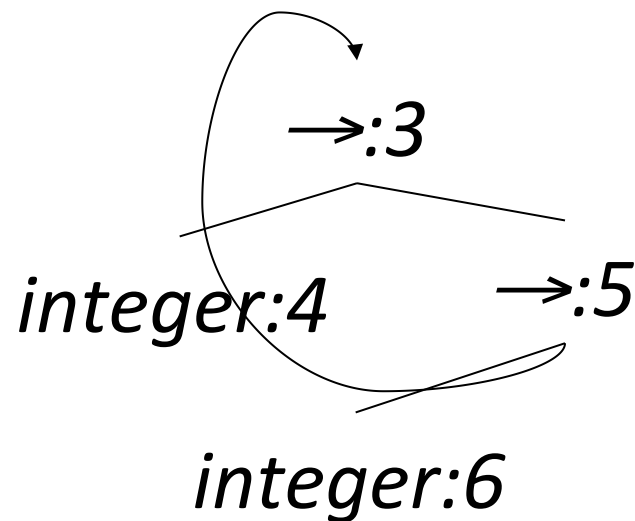
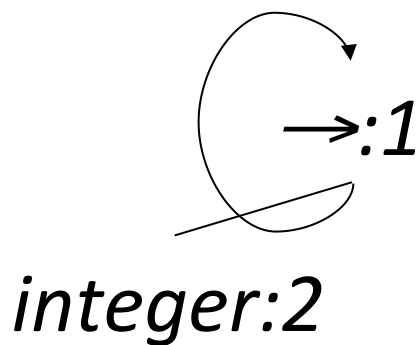


Recursive type equivalence

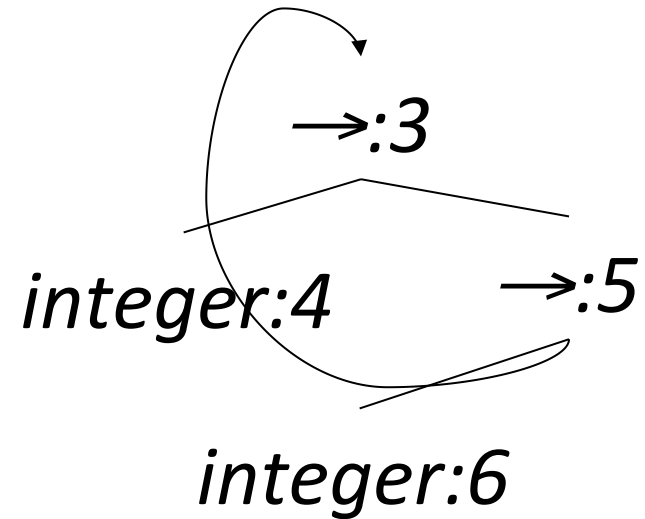
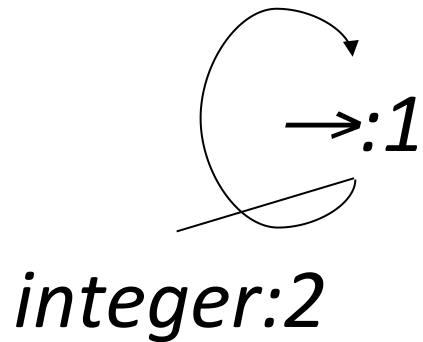
- Are these recursive type expressions equivalent:

$$\alpha1 = \text{integer} \rightarrow \alpha1$$

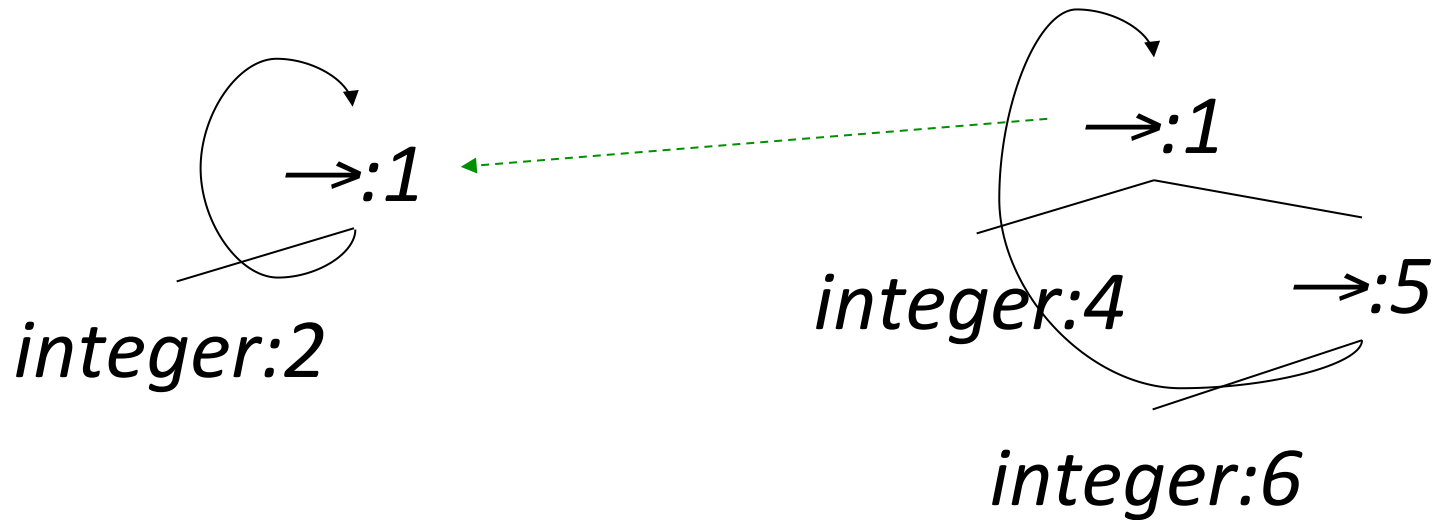
$$\alpha2 = \text{integer} \rightarrow (\text{integer} \rightarrow \alpha2)$$



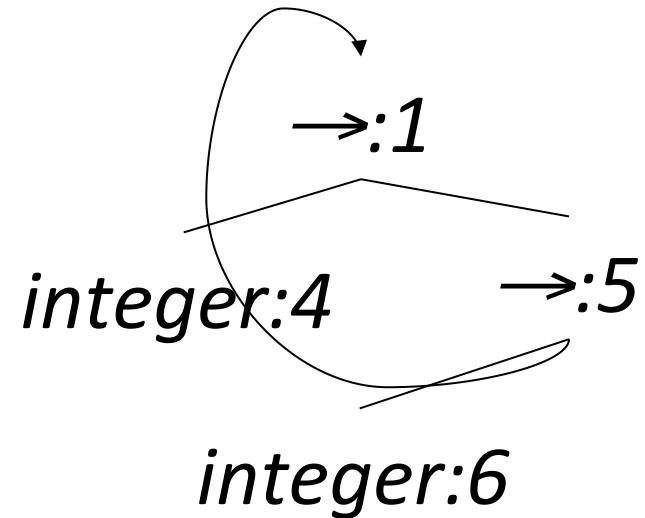
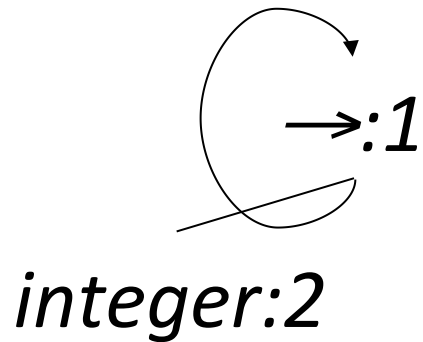
Unify(1,3)



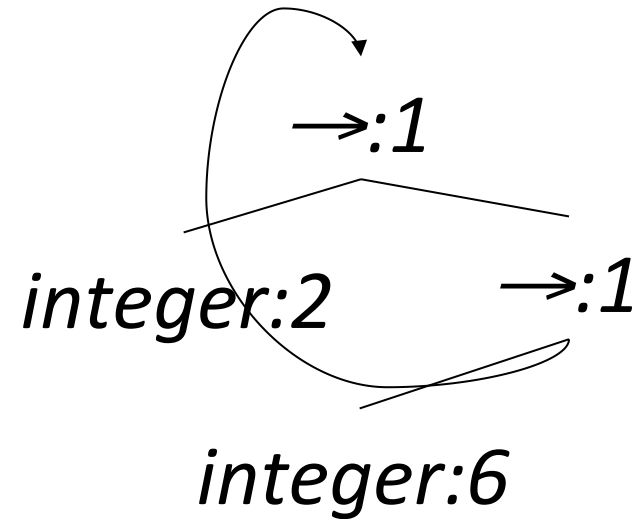
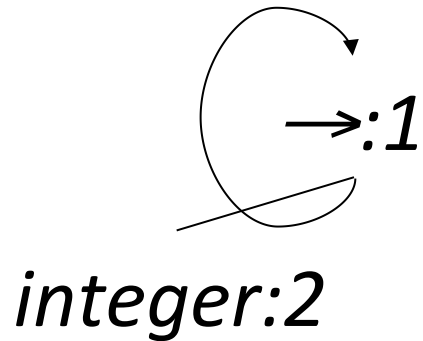
Unify(1,3)



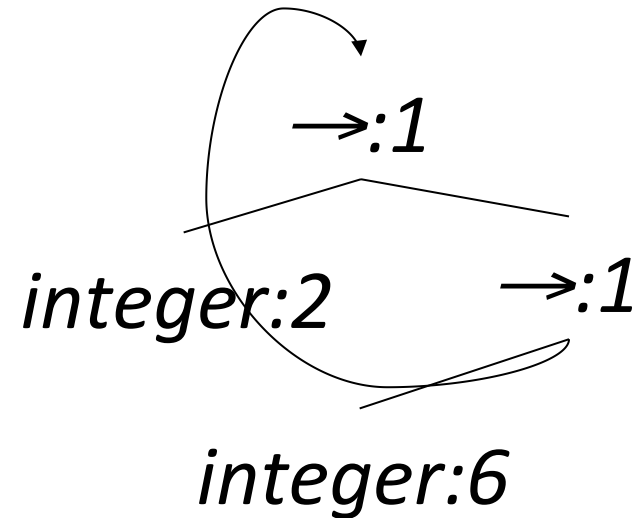
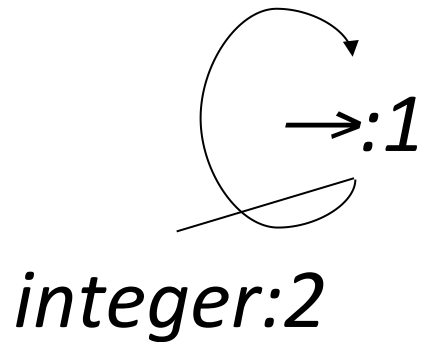
Unify(2,4) *and* Unify(1,5)



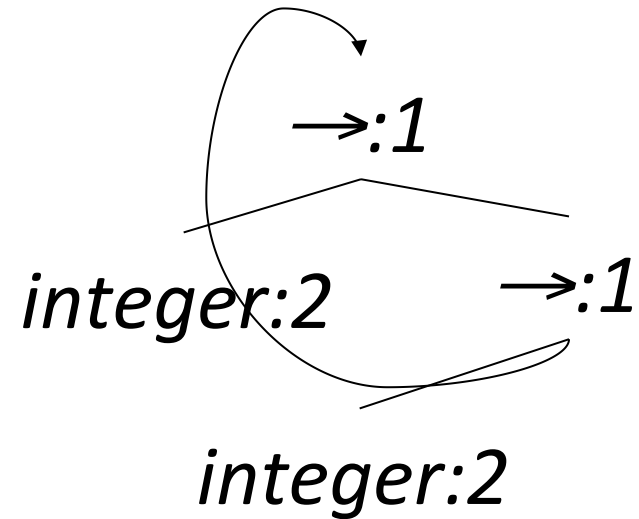
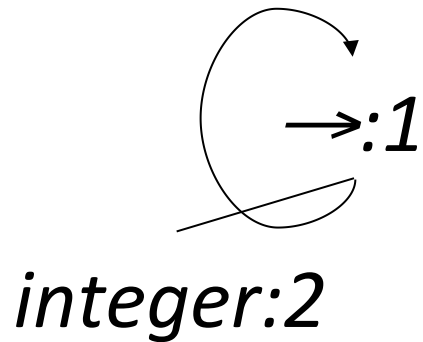
Unify(2,4) *and* Unify(1,5)



Unify(2,6) *and* Unify(1,1)



Unify(2,6) *and* Unify(1,1)



Summary

- Semantic analysis: checking various well-formedness conditions
- Most common semantic conditions involve types of variables
- Symbol tables
- Discovering types for variables and functions using inference (unification)