



# CSE341: Programming Languages Lecture 10 References, Polymorphic Datatypes, the Value Restriction, Type Inference

Ben Wood, filling in for Dan Grossman Fall 2011

## Callbacks

A common idiom: Library takes functions to apply later, when an *event* occurs – examples:

- When a key is pressed, mouse moves, data arrives
- When the program enters some state (e.g., turns in a game)
- A library may accept multiple callbacks
  - Different callbacks may need different private data with different types
  - Fortunately, a function's type does not include the types of bindings in its environment
  - (In OOP, objects and private fields are used similarly, e.g., Java Swing's event-listeners)

#### Mutable state

While it's not absolutely necessary, mutable state is reasonably appropriate here

 We really do want the "callbacks registered" and "events that have been delivered" to *change* due to function calls

For the reasons we have discussed, ML variables really are immutable, but there are mutable references (use sparingly)

- New types: t ref where t is a type
- New expressions:
  - **ref** e to create a reference with initial contents e
  - e1 := e2 to update contents
  - !e to retrieve contents (not negation)

## References example

val x = ref 42 val y = ref 42 val z = x val \_ = x := 43 val w = (!y) + (!z) (\* 85 \*) (\* x + 1 does not type-check)

- A variable bound to a reference (e.g., **x**) is still immutable: it will always refer to the same reference
- But the contents of the reference may change via :=
- And there may be aliases to the reference, which matter a lot
- Reference are first-class values
- Like a one-field mutable object, so := and ! don't specify the field

# Example call-back library

Library maintains mutable state for "what callbacks are there" and provides a function for accepting new ones

- A real library would support removing them, etc.
- In example, callbacks have type int->unit (executed for side-effect)

So the entire public library interface would be the function for registering new callbacks:

val onKeyEvent : (int -> unit) -> unit

## Library implementation

```
val cbs : (int -> unit) list ref = ref []
fun onKeyEvent f = cbs := f :: (!cbs)
fun onEvent i =
    let fun loop fs =
        case fs of
        [] => ()
        [f::fs' => (f i; loop fs')
        in loop (!cbs) end
```

## Clients

Can only register an int -> unit, so if any other data is needed, must be in closure's environment

- And if need to "remember" something, need mutable state

Examples:

### More about types

- Polymorphic datatypes, type constructors
- Why do we need the Value Restriction?
- Type inference: behind the curtain

# **Polymorphic Datatypes**

```
datatype int list =
    EmptyList
  | Cons of int * int list
datatype 'a non mt list =
    One of 'a
  | More of 'a * ('a non mt list)
datatype ('a, 'b) tree =
    Leaf of 'a
  | Node of 'b * ('a, 'b) tree * ('a, 'b) tree
val t1 = Node ("hi", Leaf 4, Leaf 8)
                   (* (int, string) tree *)
val t2 = Node("hi",Leaf true,Leaf 8)
                   (* does not typecheck *)
```

# **Polymorphic Datatypes**

datatype 'a option = NONE | SOME of 'a

- list, tree, etc. are not types; they are type constructors
- int list, (string, real) tree, etc. are types.
- Pattern-matching works on all datatypes.

# The Value Restriction Appears 🛞

If you use partial application to create a polymorphic function, it may not work due to the value restriction

- Warning about "type vars not generalized"
  - And won't let you call the function
- This should surprise you; you did nothing wrong <sup>(2)</sup> but you still must change your code
- See the written lecture summary about how to work around this wart (and ignore the issue until it arises)
- The wart is there for good reasons, related to mutation and not breaking the type system

#### Purpose of the Value Restriction

- A binding is only allowed to be polymorphic if the right-hand side is:
  - a variable; or
  - a value (including function definitions, constructors, etc.)
- ref [] is not a value, so we can only give it non-polymorphic types such as int list ref or string list ref, but not 'a list ref.

#### Downside of the Value Restriction

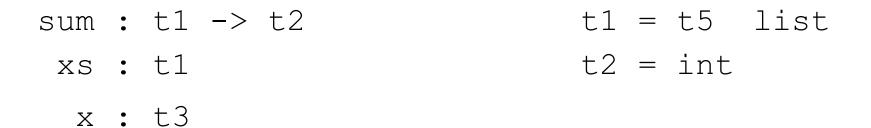
val $pr_list = List.map$ (fn x => (x,x)) (* X *)
<pre>val pr_list : int list -&gt; (int*int) list =   List.map (fn x =&gt; (x,x))</pre>
val pr_list = fn lst => List.map (fn x => (x,x)) lst
<pre>fun pr_list lst = List.map (fn x =&gt; (x,x)) lst</pre>

- The SML type checker does not know if the 'a list type uses references internally, so it has to be *conservative* and assume it could.
- In practice, this means we need to be more explicit about partial application of polymorphic functions.

sum : t1 -> t2
xs : t1

fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')

fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')



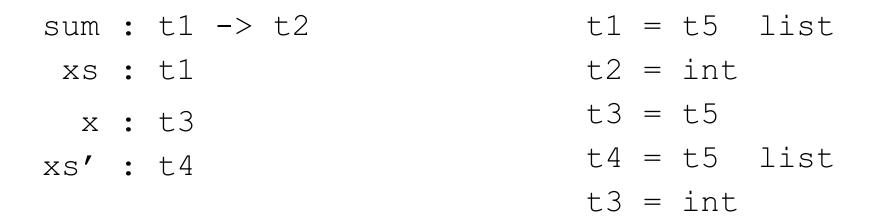




fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')



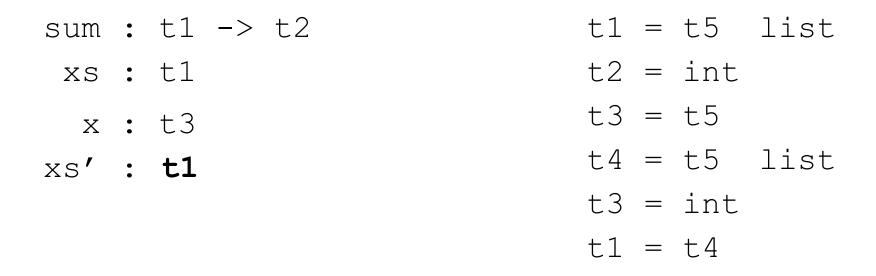
fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')



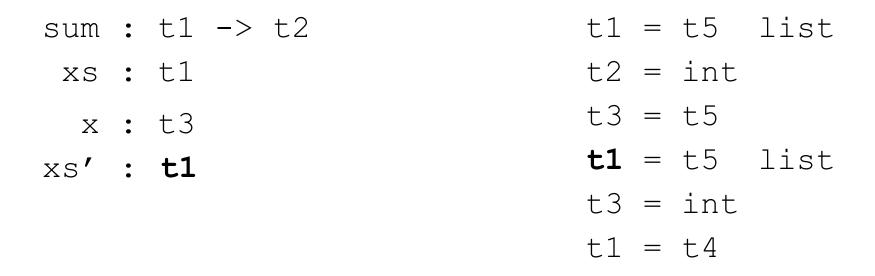
fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')



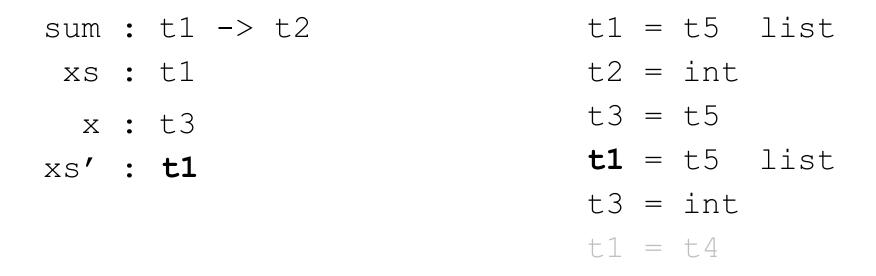
fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')



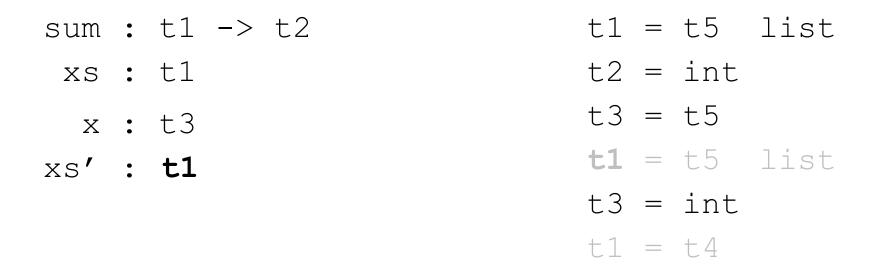
fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')



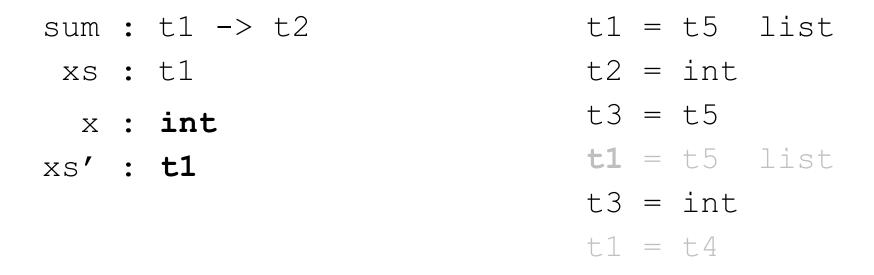
fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')

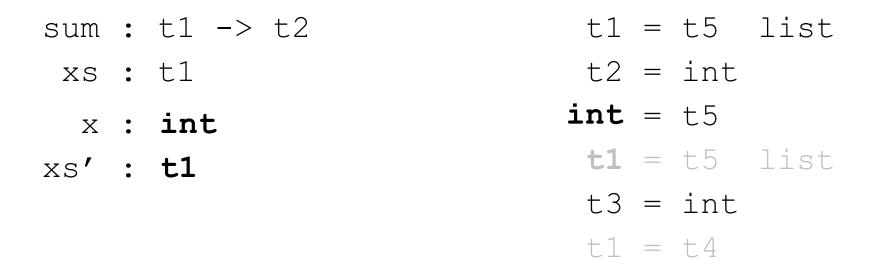


fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')

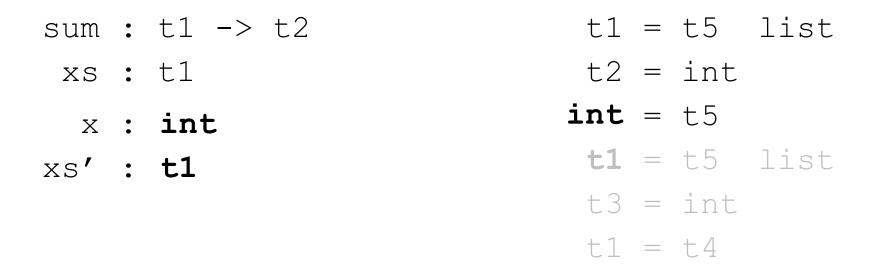


fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')

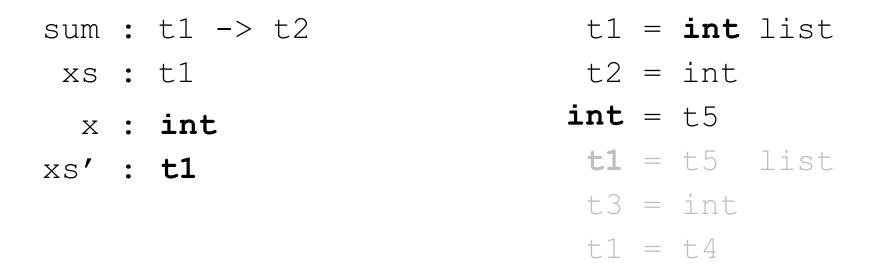




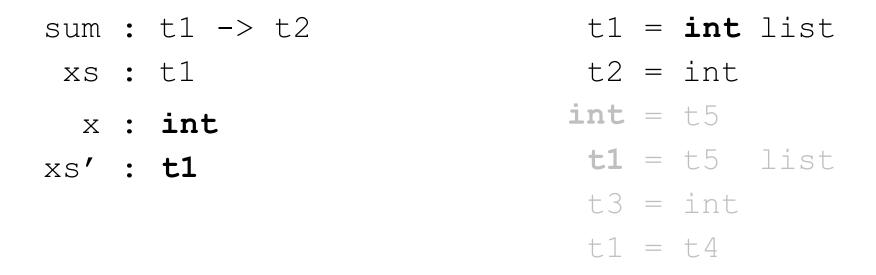
fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')



fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')



fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')



fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')

sum	•	t1 -> <b>int</b>	t1	=	int	list
XS	•	t1	t2	=	int	
Х	•	int	int	_	t5	
xs′	•	t1	<b>t1</b>	_	t5	list
			t3	_	int	
			t1		t4	

fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')

sum	•	t1 -> <b>int</b>	t1	=	int	list
XS	•	t1	t2	_	int	
Х	:	int	int	_	t5	
xs′	•	t1	<b>t1</b>	_	t5	list
			t3	_	int	
			t1	_	t4	

fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum xs')

sum	•	int	<pre>list -&gt;</pre>	int	t1	_	int	list
XS	•	int	list		t2	_	int	
X	•	int			int	_	t5	
XS'	•	int	list		<b>t1</b>	_	t5	list
					t3	_	int	
					t1		t4	

# Type inference: length

fun length xs =
 case xs of
 [] => 0
 | \_::xs' => 1 + (length xs')

length : t1 -> t2
 xs : t1

## Type inference: length

fun length xs =
 case xs of
 [] => 0
 | \_::xs' => 1 + (length xs')

fun length xs =
 case xs of
 [] => 0
 | \_::xs' => 1 + (length xs')

fun length xs =
 case xs of
 [] => 0
 | \_::xs' => 1 + (length xs')

fun length xs =
 case xs of
 [] => 0
 | \_::xs' => 1 + (length xs')

fun length xs =
 case xs of
 [] => 0
 | \_::xs' => 1 + (length xs')

fun length xs =
 case xs of
 [] => 0
 | \_::xs' => 1 + (length xs')

fun length xs =
 case xs of
 [] => 0
 | \_::xs' => 1 + (length xs')

fun length xs =
 case xs of
 [] => 0
 | \_::xs' => 1 + (length xs')

fun length xs =
 case xs of
 [] => 0
 | \_::xs' => 1 + (length xs')

fun length xs =
 case xs of
 [] => 0
 | \_::xs' => 1 + (length xs')

fun length xs =
 case xs of
 [] => 0
 | \_::xs' => 1 + (length xs')

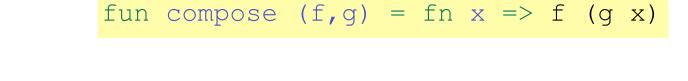
Fall 2011

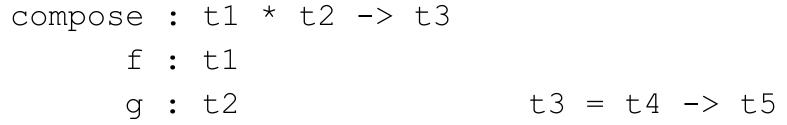
fun length xs =
 case xs of
 [] => 0
 | \_::xs' => 1 + (length xs')

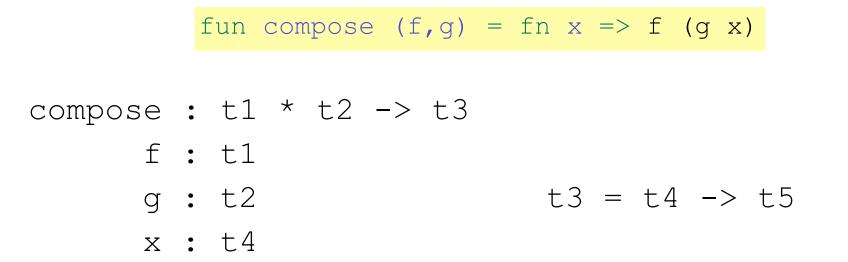
#### length works no matter what `a is.

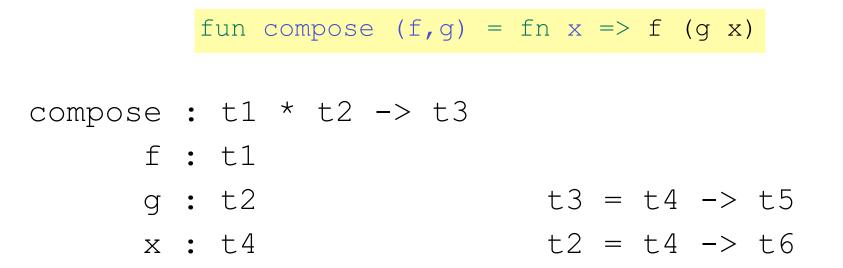
#### fun compose $(f,g) = fn x \Rightarrow f (g x)$

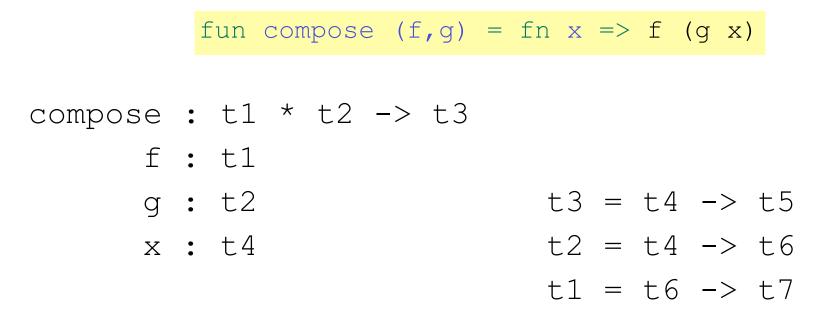
compose : t1 \* t2 -> t3 f : t1 g : t2

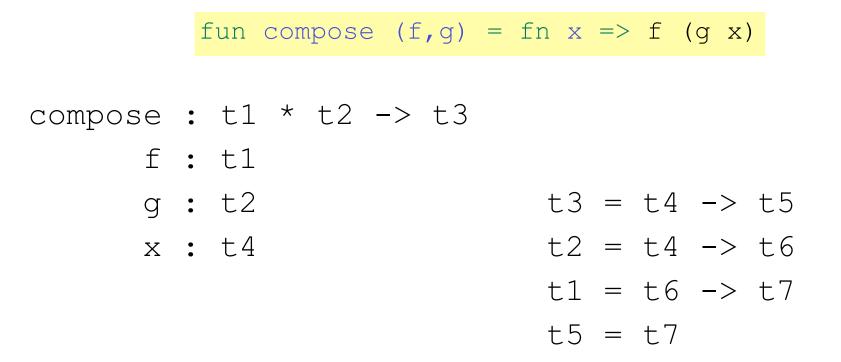


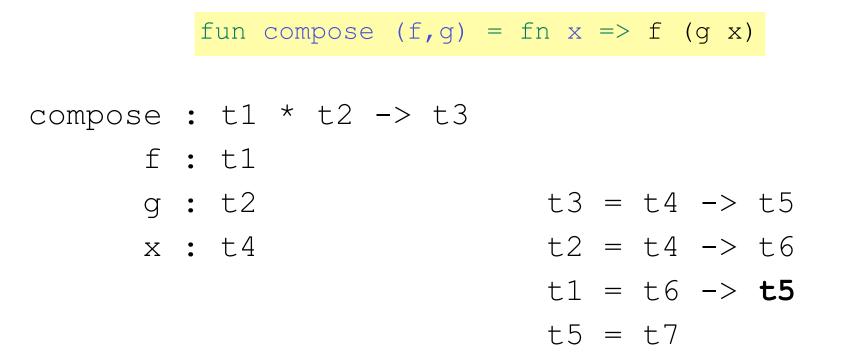


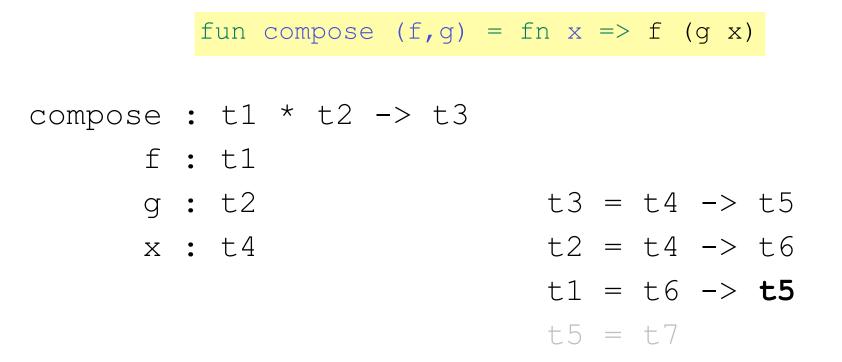












fun compose  $(f,g) = fn \ x => f \ (g \ x)$ compose :  $(t6 \ -> \ t5) \ * \ t2 \ -> \ t3$ f :  $t6 \ -> \ t5$ g :  $t2 \ t3 = t4 \ -> \ t5$ x :  $t4 \ t2 = t4 \ -> \ t6$ t1 =  $t6 \ -> \ t5$ t5 = t7

fun compose  $(f,g) = fn \ x => f \ (g \ x)$ compose :  $(t6 \ -> \ t5) \ * \ t2 \ -> \ t3$ f :  $t6 \ -> \ t5$ g :  $t2 \ t3 = t4 \ -> \ t5$ x :  $t4 \ t2 = t4 \ -> \ t6$   $t1 = t6 \ -> \ t5$ t5 = t7

fun compose (f,g) = fn x => f (g x)
compose : (t6 -> t5) \* (t4 -> t6) -> t3
f : t6 -> t5
g : t4 -> t6
t3 = t4 -> t5
x : t4
t2 = t4 -> t6
t1 = t6 -> t5
t5 = t7

fun compose (f,g) = fn x => f (g x)
compose : (t6 -> t5) \* (t4 -> t6) -> t3
f : t6 -> t5
g : t4 -> t6
t3 = t4 -> t5
x : t4
t2 = t4 -> t6
t1 = t6 -> t5
t5 = t7

fun compose  $(f,g) = fn x \Rightarrow f (g x)$ 

compose : 
$$(t6 \rightarrow t5) * (t4 \rightarrow t6) \rightarrow (t4 \rightarrow t5)$$
  
f :  $t6 \rightarrow t5$   
g :  $t4 \rightarrow t6$   
x :  $t4 \rightarrow t5$   
 $t2 = t4 \rightarrow t6$   
 $t1 = t6 \rightarrow t5$   
 $t5 = t7$ 

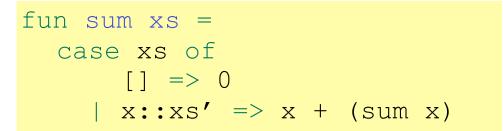
fun compose (f,g) = fn x => f (g x)

compose :  $(t6 \rightarrow t5) * (t4 \rightarrow t6) \rightarrow (t4 \rightarrow t5)$ f :  $t6 \rightarrow t5$ g :  $t4 \rightarrow t6$ x :  $t4 \rightarrow t5$ t :  $t4 \rightarrow t6$ t :  $t6 \rightarrow t5$ t :  $t6 \rightarrow t6$ t :  $t1 = t6 \rightarrow t5$ t : t5 = t7

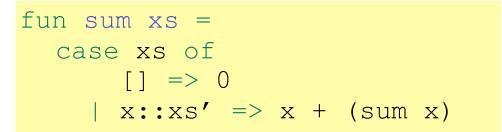
fun compose (f,g) = fn x => f (g x)
compose : ('a -> 'b) \* ('c -> 'a) -> ('c -> 'b)
f : t6 -> t5
g : t4 -> t6
t3 = t4 -> t5
x : t4 -> t5
t2 = t4 -> t6
t1 = t6 -> t5
t5 = t7

fun compose  $(f,g) = fn x \Rightarrow f (g x)$ compose : ('a -> 'b) \* ('c -> 'a) -> ('c -> 'b)f : t6 -> t5 g : t4 -> t6  $t_3 = t_4 -> t_5$ x : t4 -> t5  $t_2 = t_4 -> t_6$ t1 = t6 -> **t5** t.5 = t.7compose : ('b -> 'c) \* ('a -> 'b) -> ('a -> 'c)

sum : t1 -> t2
xs : t1



fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)



fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)



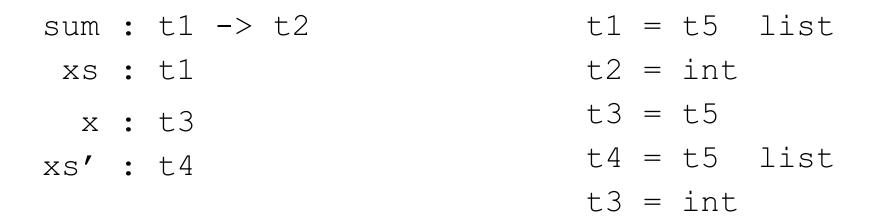
fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)



fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)



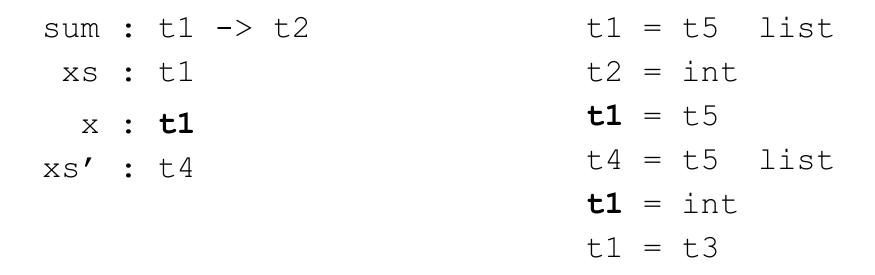
fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)



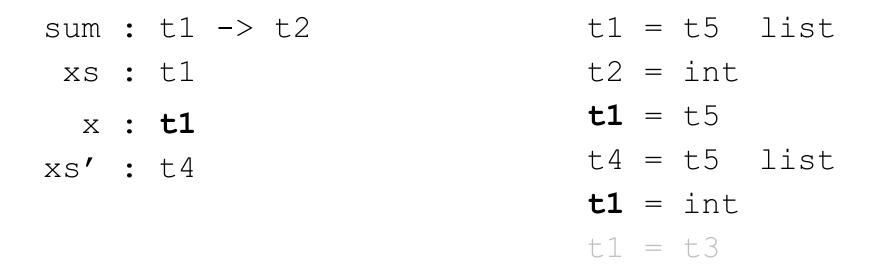
fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)



fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)



fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)



fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)

sum	•	<b>int</b> -> t2	int	=	t5	list
XS	•	int	t2	=	int	
Х	•	int	int	=	t5	
xs′	•	t4	t4	=	t5	list
			t1	=	int	
			t1	_	t3	

fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)

sum	•	<b>int-</b> > t2	int	=	t5	list
XS	•	int	t2	=	int	
Х	•	int	int	=	t5	
xs′	•	t4	t4	=	t5	list
			t1	_	int	
			t1		t3	

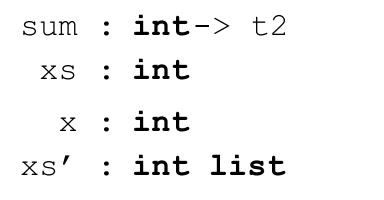
fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)

sum	•	<b>int</b> -> t2	int	=	t5	list
XS	•	int	t2	=	int	
Х	:	int	int	=	t5	
xs′	•	t5 list	t4	=	t5	list
			<b>t1</b>	_	int	
			t1	_	t3	

fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)

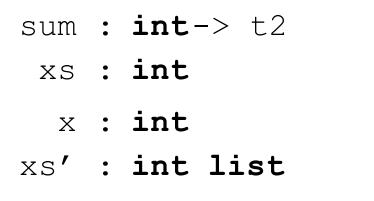
sum	•	<b>int</b> -> t2	int	=	t5	list
XS	•	int	t2	=	int	
Х	•	int	int	=	t5	
xs′	•	t5 list	t4	_	t5	list
			t1	_	int	
			t1	_	t3	

fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)



int	=	int	list
t2	=	int	
int	=	t5	
t4	_	t5	list
<b>t1</b>	_	int	
t1	_	t3	

fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)



fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)

sum : int-> int
 xs : int
 x : int
 xs' : int list
 t1 = int
 t1 = t3<
 int = int
 t1 = t3<
 int = int
 t1 = t3</pre>

fun sum xs =
 case xs of
 [] => 0
 | x::xs' => x + (sum x)

sum : int-> int
xs : int
x : int
xs' : int list

int = int list
 t2 = int
 int = t5
 t4 = t5 list
 t1 = int
 t1 = t3

# Parting comments on ML type inference

- You almost never have to write types in ML (even on parameters), with some minor caveats.
- Hindley-Milner type inference algorithm
- ML has no subtyping. If it did, the equality constraints we used for inference would be overly restrictive.
- Type variables and inference are not tied to each.
   Some languages have one without the other.
  - Type variables alone allow convenient code reuse.
  - Without type variables, we cannot give a type to compose until we see it used.