

Programming Languages and Techniques (CIS120)

Lecture 12

Typechecking
Chapter 11

Typechecking

How does OCaml* typecheck your code?

*Historical aside: the algorithm we are about to see is known as the Damas-Hindley-Milner type inference algorithm. Turing Award winner Robin Milner was, among other things, the inventor of "ML" (for "meta language"), from which OCaml gets its "ml".

OCaml Typechecking Errors

```
type ('k,'v) map = ('k * 'v) list
```

```
(* A finite map that contains no entries. *)
```

```
let empty () = []
```

```
let rec mem
```

```
begin ma
```

```
| [] ->
```

```
| (k,v):
```

```
if key
```

```
(key = k) || (mem key rest)
```

```
end
```

```
;; run_test "mem test" (fun () ->
```

```
mem "b" [("a",3); ("b",4)]
```

```
)
```

```
let rec get (key:'k) (m : ('k,'v) map) : 'v =
```

```
begin match m with
```

```
| [] -> failwith "not found"
```

✖ Signature mismatch:

...

Values do not match:

val empty : unit -> 'a list

is not included in

val empty : ('k, 'v) map

File "finiteMap.ml", line 13, characters 2-27: Expected declaration

File "finiteMap.ml", line 60, characters 6-11: Actual declaration

Typechecking

How do you determine the type of an expression?

1. Recursively determine the types of *all* sub-expressions

– Constants have “obvious” types

`3 : int          “foo” : string          true : bool`

– Identifiers may have type annotations

- `let` and function arguments
- Module signatures/interfaces

2. Expressions that *construct* structured values have compound types built from the types of sub-expressions

`(3, “foo”) : int * string`
`(fun (x:int) -> x + 1) : int -> int`
`Node(Empty, (3, “foo”), Empty) : (int * string) tree`

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

- Given a function $f : T_1 \rightarrow T_2$
- and an argument $e : T_1$ of the input type
- $(f\ e) : T_2$

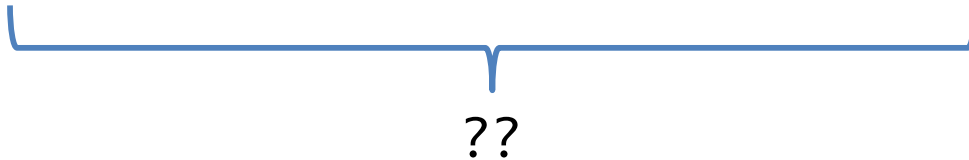
`((fun (x:int) (y:bool) -> y) 3) : ??`

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

- Given a function $f : T_1 \rightarrow T_2$
- and an argument $e : T_1$ of the input type
- $(f\ e) : T_2$

`((fun (x:int) (y:bool) -> y) 3) : ??`



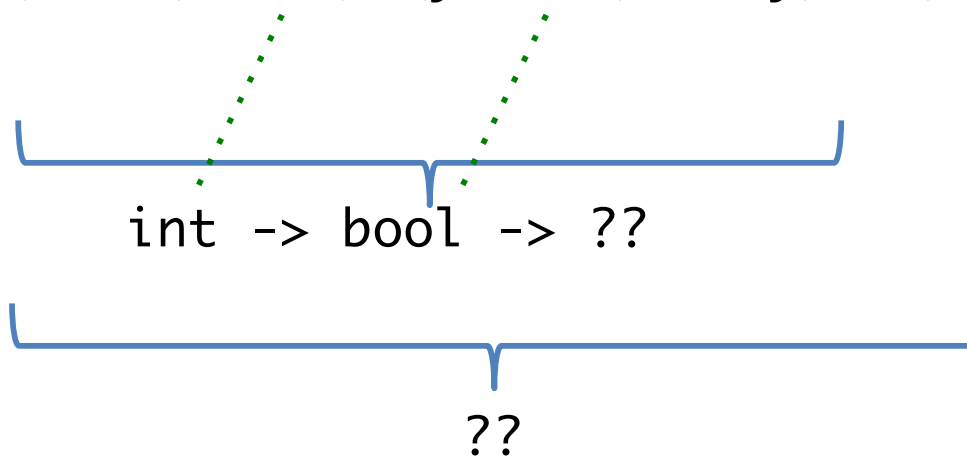
??

Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

- Given a function $f : T_1 \rightarrow T_2$
- and an argument $e : T_1$ of the input type
- $(f\ e) : T_2$

`((fun (x:int) (y:bool) -> y) 3) : ??`

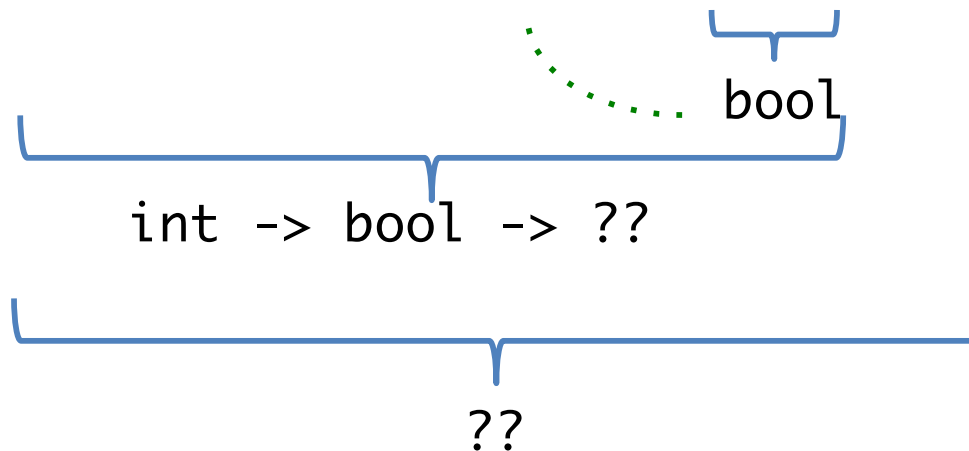


Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

- Given a function $f : T_1 \rightarrow T_2$
- and an argument $e : T_1$ of the input type
- $(f\ e) : T_2$

`((fun (x:int) (y:bool) -> y) 3) : ??`

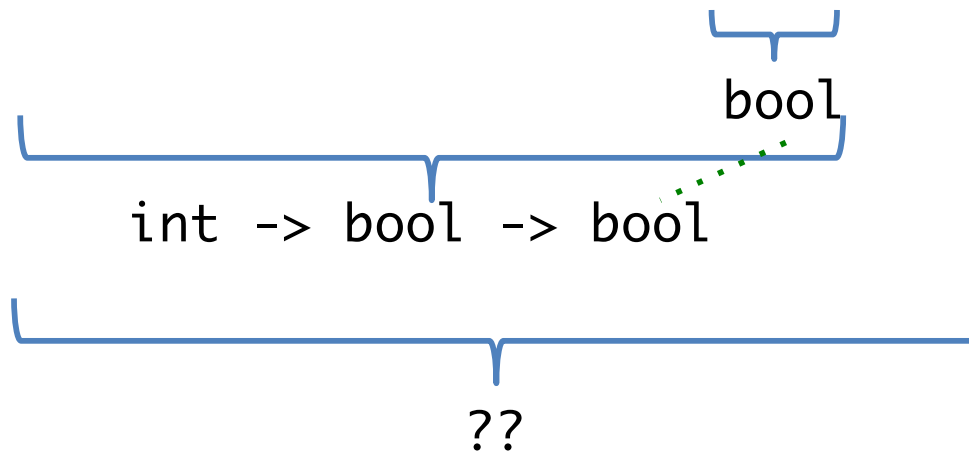


Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

- Given a function $f : T_1 \rightarrow T_2$
- and an argument $e : T_1$ of the input type
- $(f\ e) : T_2$

`((fun (x:int) (y:bool) -> y) 3) : ??`

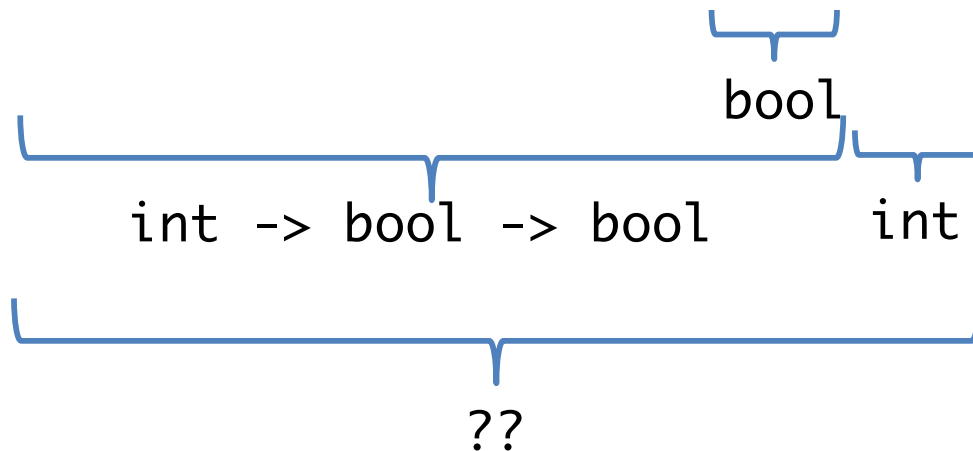


Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

- Given a function $f : T_1 \rightarrow T_2$
- and an argument $e : T_1$ of the input type
- $(f\ e) : T_2$

`((fun (x:int) (y:bool) -> y) 3) : ??`

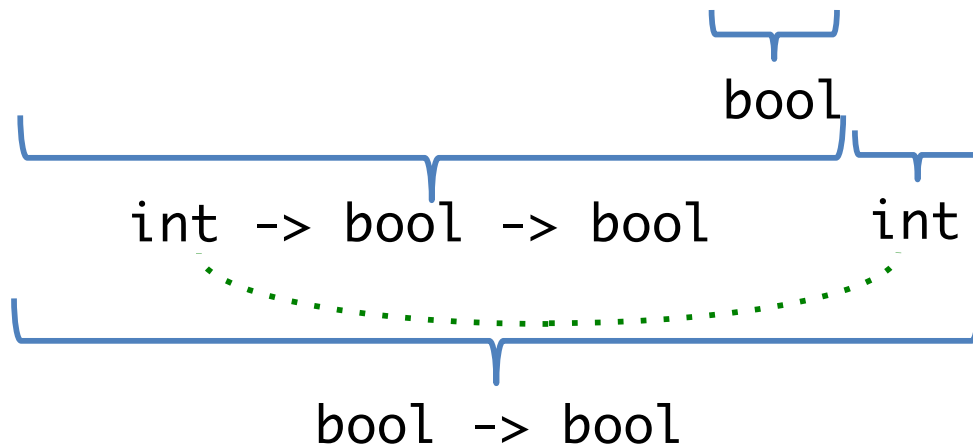


Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:

- Given a function $f : T_1 \rightarrow T_2$
- and an argument $e : T_1$ of the input type
- $(f e) : T_2$

`((fun (x:int) (y:bool) -> y) 3) : ??`



Here:

$T_1 = \text{int}$

$T_2 = \text{bool} \rightarrow \text{bool}$

Typechecking III

- What about generics? i.e. what if $f : 'a \rightarrow 'a$?
- For generic types we *unify*
 - Given a function $f : T_1 \rightarrow T_2$
 - and an argument $e : U_1$ of the input type

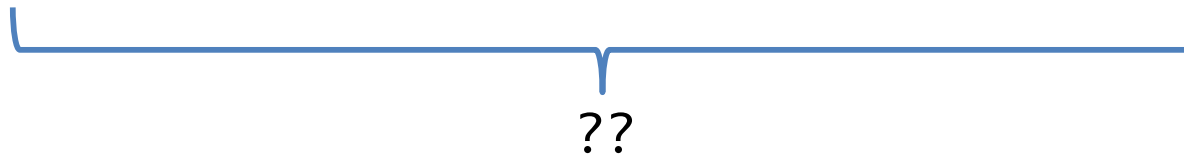
Can “match up” T_1 and U_1 to obtain information about type parameters in T_1 and U_1 based on their usage

- Obtain an *instantiation*: e.g. $'a = \text{int list}$
- *Propagate* that information to all occurrences of $'a$

Example Typechecking Problem

```
empty    : ('k, 'v) map  
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map  
entries  : ('k, 'v) map -> ('k * 'v) list
```

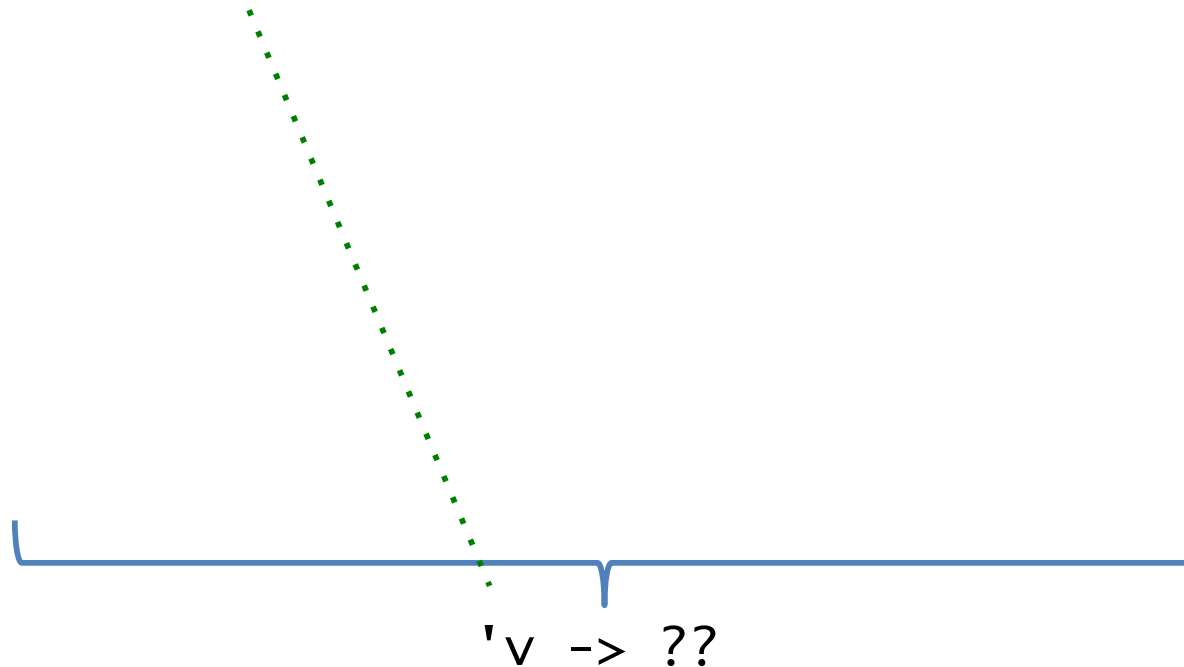
```
fun (x:'v) -> entries (add 3 x empty)
```



Example Typechecking Problem

```
empty    : ('k, 'v) map  
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map  
entries  : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```



Example Typechecking Problem

```
empty    : ('k, 'v) map  
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map  
entries  : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```

Diagram showing the argument `add 3 x empty` with blue curly braces underneath each token, indicating the scope of each argument.

Diagram showing the function `add` and its arguments `3` and `x` with blue curly braces underneath, indicating the scope of the function and its arguments.

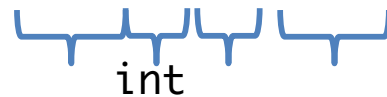
Diagram showing the function `entries` and its argument `(add 3 x empty)` with blue curly braces underneath, indicating the scope of the function and its argument.

Diagram showing the entire function signature `fun (x:'v) -> entries (add 3 x empty)` with a large blue curly brace underneath, indicating the scope of the function. Below the brace is the text `'v -> ??`.

Example Typechecking Problem

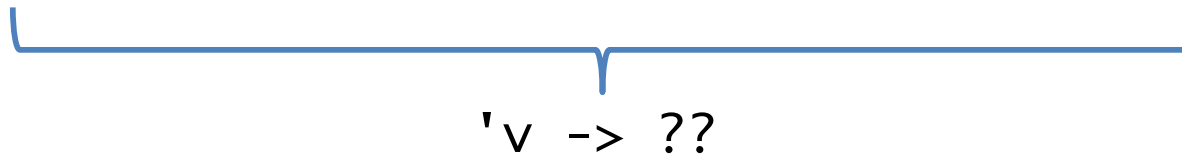
```
empty    : ('k, 'v) map  
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map  
entries  : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```


int



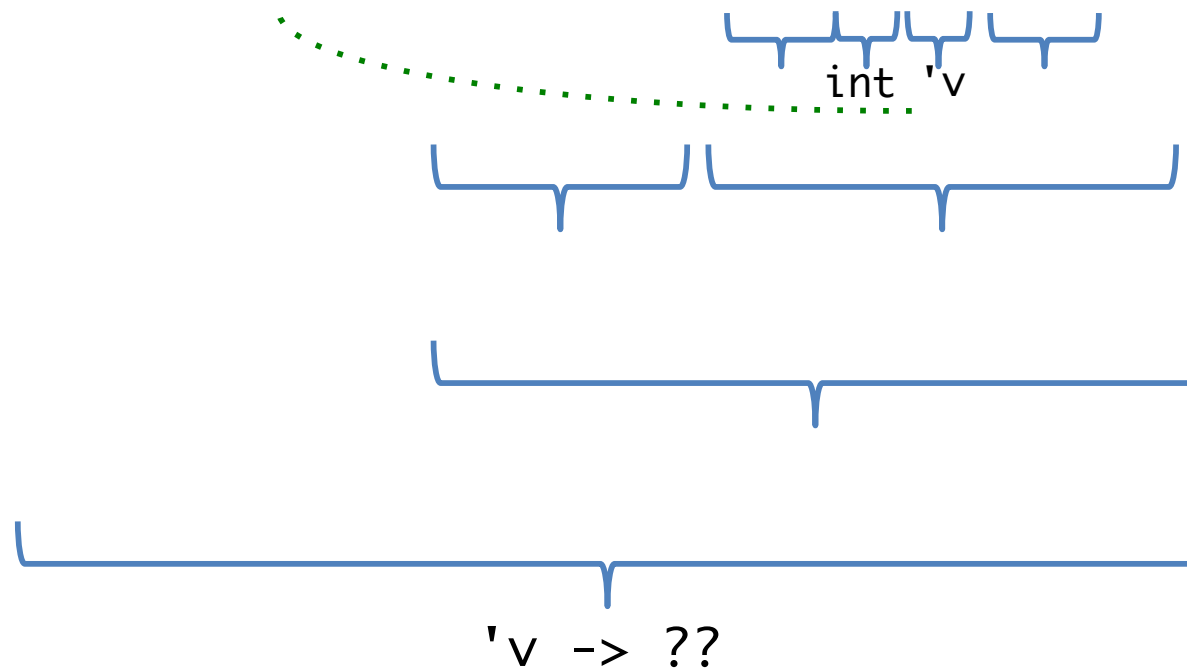



'v -> ??

Example Typechecking Problem

```
empty    : ('k, 'v) map  
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map  
entries  : ('k, 'v) map -> ('k * 'v) list
```

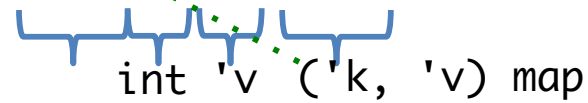
```
fun (x:'v) -> entries (add 3 x empty)
```



Example Typechecking Problem

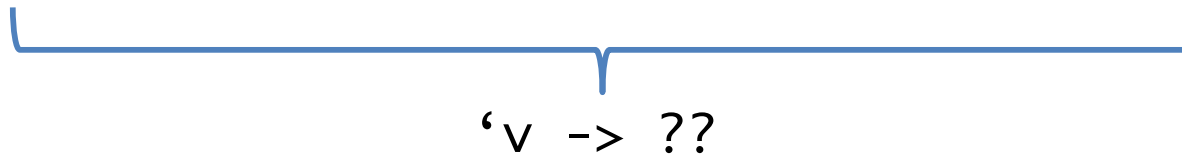
```
empty    : ('k, 'v) map  
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map  
entries  : ('k, 'v) map -> ('k * 'v) list
```

```
fun (x:'v) -> entries (add 3 x empty)
```


int 'v ('k, 'v) map






'v -> ??

Example Typechecking Problem

```
empty    : ('k, 'v) map  
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map  
entries  : ('k, 'v) map -> ('k * 'v) list
```

`fun (x:'v) -> entries (add 3 x empty)`

int 'v ('k, 'v) map

'v -> ??

Example Typechecking Problem

```
empty    : ('k, 'v) map  
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map  
entries  : ('k, 'v) map -> ('k * 'v) list
```

fun (x:'v) -> entries (add 3 x empty)

int 'v ('k, 'v) map

??

'v -> ??

Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

fun (x:'v) -> entries (add 3 x empty)

Application:

$T_1 = 'k$

$T_2 = 'v \rightarrow ('k, 'v) \text{ map} \rightarrow ('k, 'v) \text{ map}$

Instantiate: $'k = \text{int}$

$'v \rightarrow ??$

int 'v ('k, 'v) map

T_2

Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

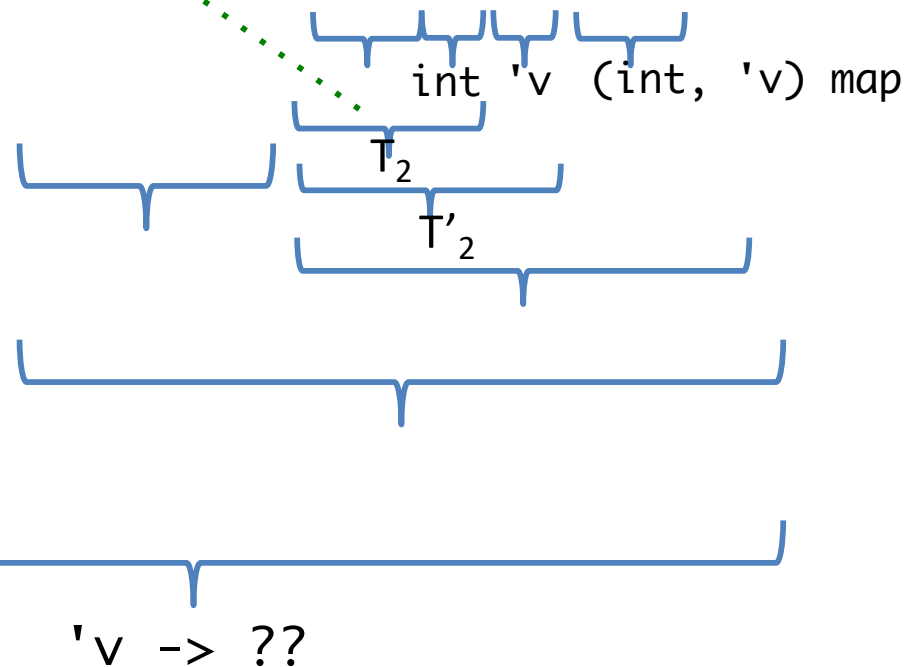
fun (x:'v) -> entries (add 3 x empty)

Another Application:

$T'_1 = 'v$

$T'_2 = (int, 'v) \text{ map} \rightarrow (int, 'v) \text{ map}$

Instantiate: $'v = 'v$



Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

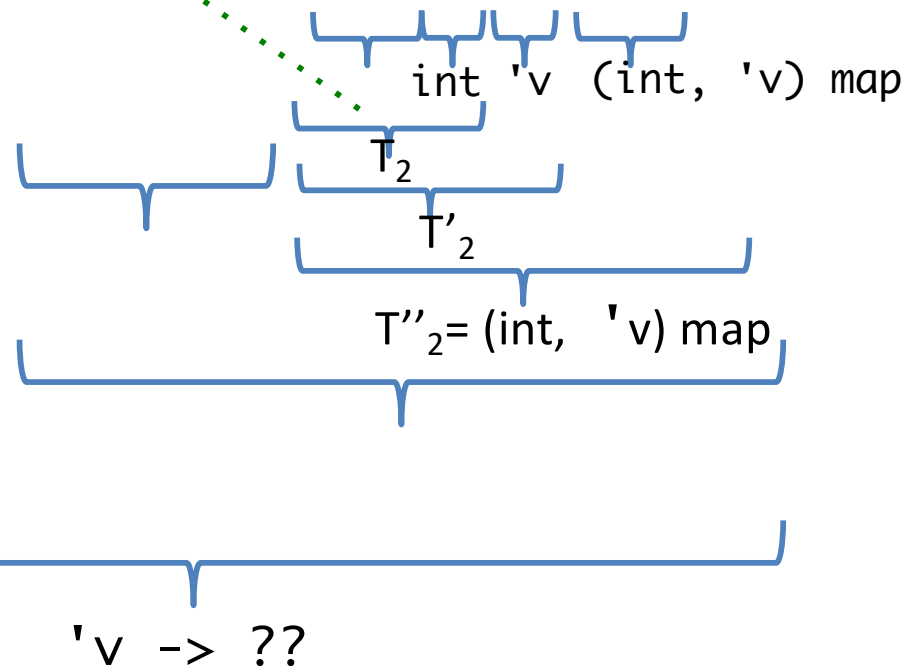
fun (x:'v) -> entries (add 3 x empty)

A third Application:

$T''_1 = (\text{int}, 'v) \text{ map}$

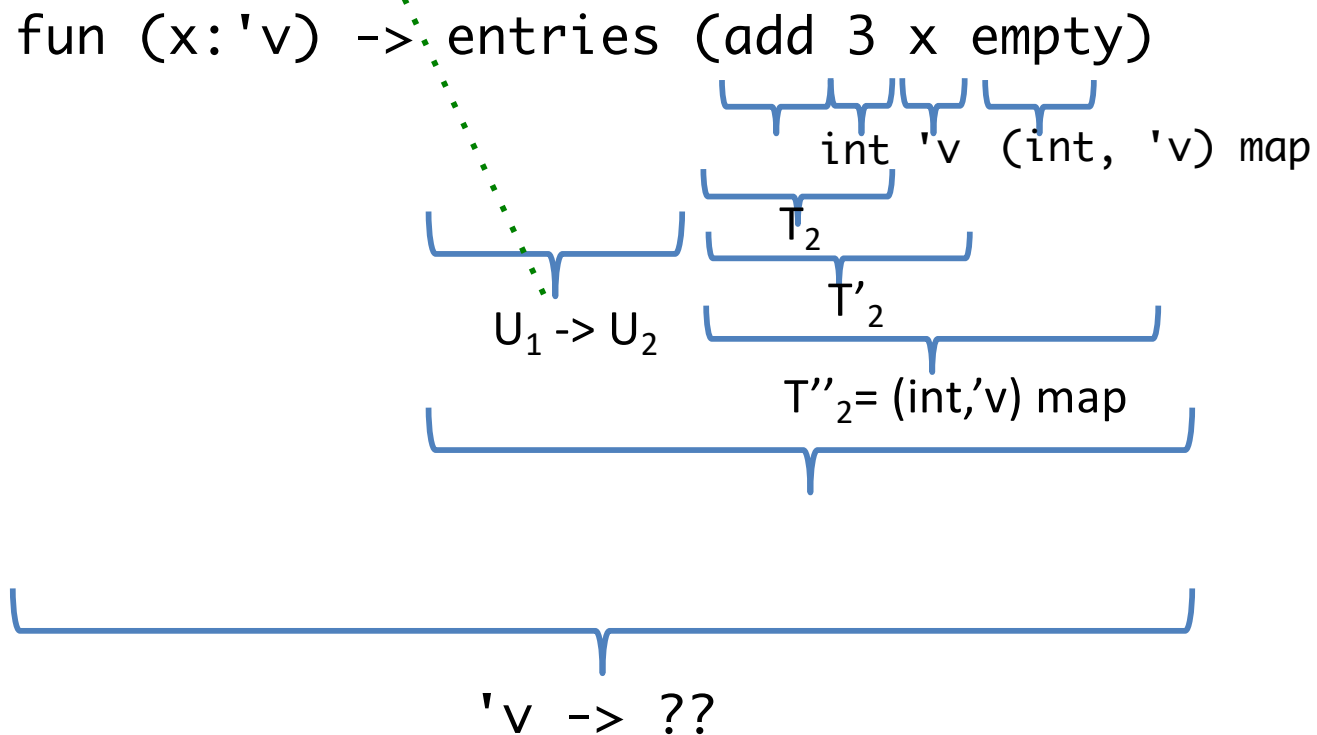
$T''_2 = (\text{int}, 'v) \text{ map}$

Argument and argument
type already agree



Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```



Example Typechecking Problem

```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

fun (x:'v) -> entries (add 3 x empty)

Another Application:

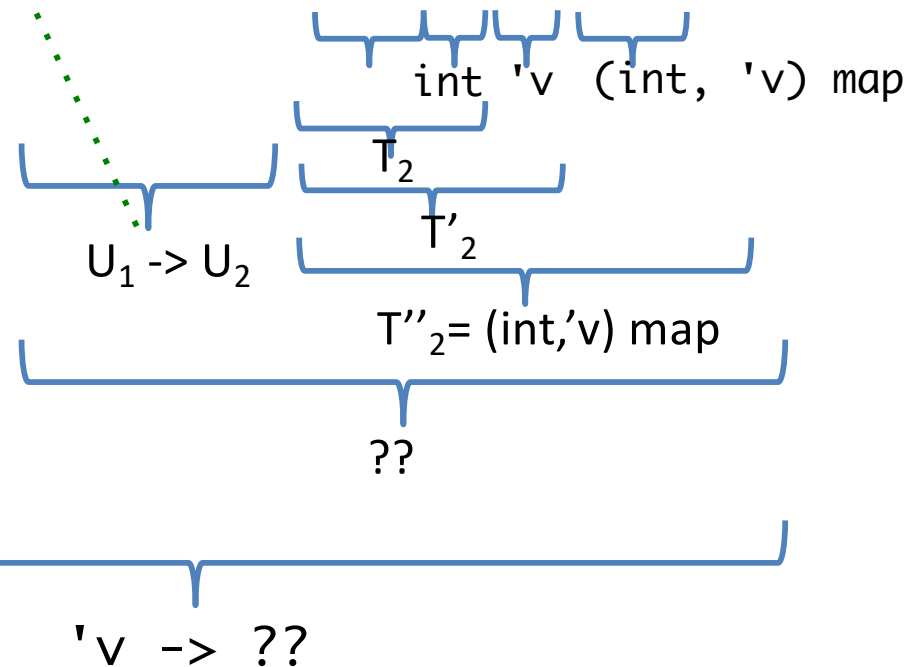
$U_1 = ('k, 'v) \text{ map}$

$U_2 = ('k * 'v) \text{ list}$

Unify U_1 with T''_2

$('k, 'v) \text{ map} \sim\sim (int, 'v) \text{ map}$

Instantiate $'k = int$



Example Typechecking Problem

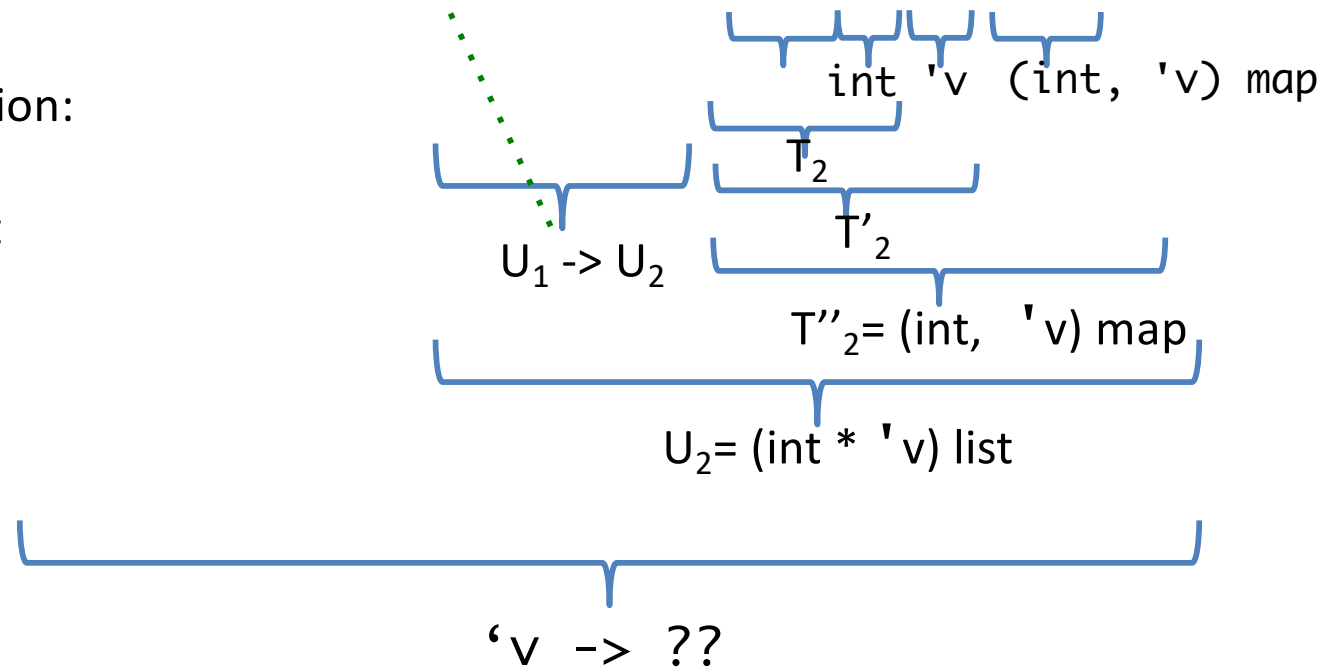
```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

fun (x:'v) -> entries (add 3 x empty)

Another Application:

$U_1 = (\text{int}, 'v) \text{ map}$

$U_2 = (\text{int} * 'v) \text{ list}$



Example Typechecking Problem

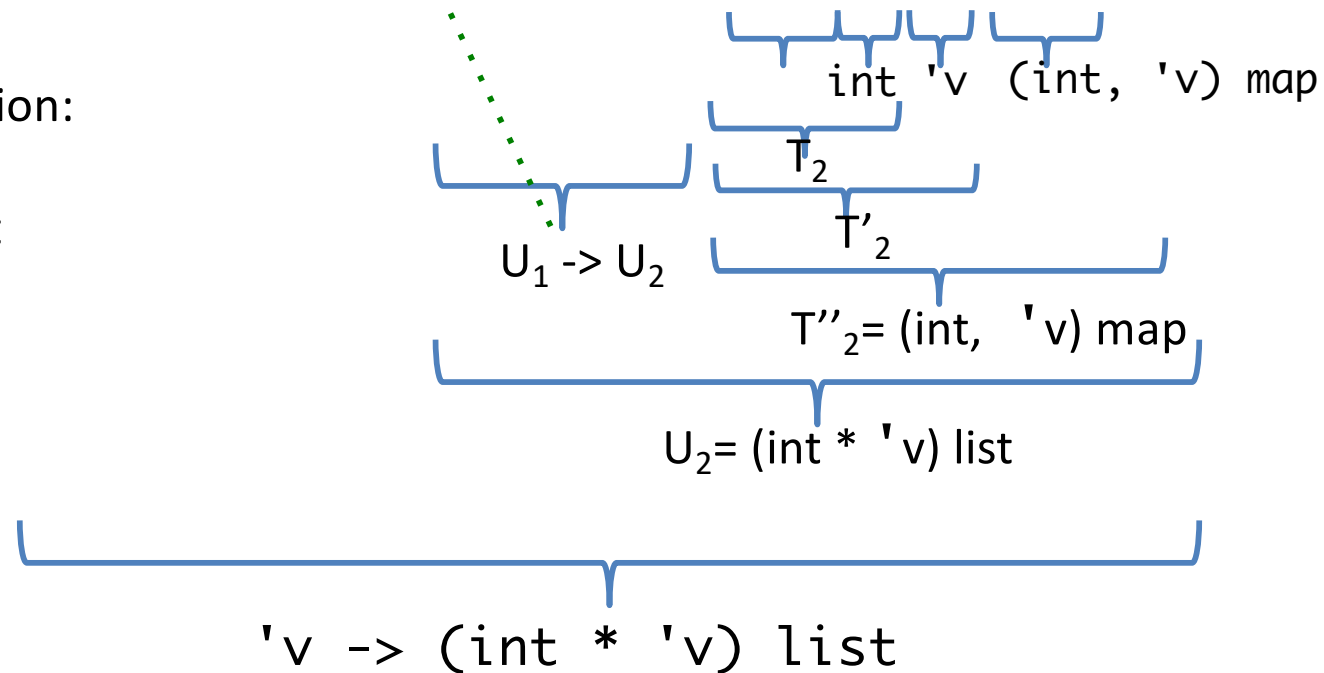
```
empty    : ('k, 'v) map
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries  : ('k, 'v) map -> ('k * 'v) list
```

fun (x:'v) -> entries (add 3 x empty)

Another Application:

$U_1 = (\text{int}, 'v) \text{ map}$

$U_2 = (\text{int} * 'v) \text{ list}$



Ill-typed Expressions?

- An expression is ill-typed if, during this type checking process, inconsistent constraints are encountered:

```
empty    : ('k, 'v) map  
add      : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map  
entries  : ('k, 'v) map -> ('k * 'v) list
```

add 3 true (add “foo” false empty)

Error: found `int` but expected `string`