# Programming Languages and Techniques (CIS1200)

Lecture 12

## Typechecking

Chapter 10

# Announcements

- Midterm 1: Friday, September 27th
  - Coverage: up to today (Chapters 1-10)
  - During lecture
    Last names: A – Z Meyerson Hall B1
  - 60 minutes; closed book, **closed notes**
  - Review Material
    - old exams on the web site ("schedule" tab)
  - Review Session
    - Tonight, 7:00-9:00pm, Towne 100 (will be recorded)
    - Review Videos available
- Dr. Sheth will have extra office hours
  - Thursday from 2-4pm, Levine 264

# Module Review

# What is a good signature?

## Fall 2022 Question 3 #324

Anonymous
15 hours ago in **Exams**

PIN STAR WATCH 29 VIEWS

Hi!

Can someone clarify the difference between ill-typed, unusable, invariant and good interfaces?

Thank you!

Comment Edit Delete Endorse •••

## 1 Answer

Luis Sangueado STAFF
12 hours ago

Sure!

**Ill-typed:** The type signature for one or more functions has an incorrect type such as trying to add an int and a string together to produce an int

**Unusable:** The provided type declaration would render an implementation unusable. For example, in HW3, not having the definition of an empty set or the `set_of_list` function would render our interface unusable since we'd never be able to create a set type!

# **Good** Signature: Finite Map

```
module type MAP = sig

  type ('k,'v) map

  val empty   : ('k,'v) map
  val add     : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val mem     : 'k -> ('k,'v) map -> bool
  val get     : 'k -> ('k,'v) map -> 'v
  val equals  : ('k,'v) map -> ('k,'v) map -> bool

end
```

# **Unusable** Signature: Finite Map

```
module type MAP = sig

  type ('k,'v) map

  val add    : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val mem    : 'k -> ('k,'v) map -> bool
  val get    : 'k -> ('k,'v) map -> 'v
  val equals : ('k,'v) map -> ('k,'v) map -> bool

end
```

Type is abstract. All we know about it is what is in the signature. All maps must be constructed from operations listed here.

```
let m = ???
```

Clients have no way of constructing a map
Using [] doesn't type check

# **Good** Signature (again): Finite Map

```
module type MAP = sig

  type ('k,'v) map

  val empty   : ('k,'v) map
  val add     : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val mem     : 'k -> ('k,'v) map -> bool
  val get     : 'k -> ('k,'v) map -> 'v
  val equals  : ('k,'v) map -> ('k,'v) map -> bool

end
```

# **Unsafe** Signature: Finite Map

```
module type MAP = sig

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

  val empty   : ('k,'v) map
  val add     : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val mem     : 'k -> ('k,'v) map -> bool
  val get     : 'k -> ('k,'v) map -> 'v
  val equals  : ('k,'v) map -> ('k,'v) map -> bool

end
```

Invariant: keys appear in order in the list, no duplicate keys

```
let m = [ (3,"dos") ; (1, "uno") ; (2, "tres") ] in
mem m 2 ?
```

Clients can call module code with maps that don't satisfy the invariant

# **Good** Signature (again): Finite Map

```
module type MAP = sig

  type ('k,'v) map

  val empty   : ('k,'v) map
  val add     : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val mem     : 'k -> ('k,'v) map -> bool
  val get     : 'k -> ('k,'v) map -> 'v
  val equals  : ('k,'v) map -> ('k,'v) map -> bool

end
```

# **Unsafe** Signature: Finite Map

```
module type MAP = sig

  type ('k,'v) map

  val empty   : ('k,'v) map
  val add     : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val mem     : 'k -> ('k,'v) map -> bool
  val get     : 'k -> ('k * 'v) list -> 'v
  val equals  : ('k,'v) map -> ('k,'v) map -> bool

end
```

Invariant: keys appear in order in the list, no duplicate keys

```
let m = [ (3,"dos") ; (1, "uno") ; (2, "tres") ] in
get m 2 ?
```

Clients can call module code with maps that don't satisfy the invariant

# **Good** Signature (again): Finite Map

```
module type MAP = sig

  type ('k,'v) map

  val empty   : ('k,'v) map
  val add     : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val mem     : 'k -> ('k,'v) map -> bool
  val get     : 'k -> ('k,'v) map -> 'v
  val equals  : ('k,'v) map -> ('k,'v) map -> bool

end
```

# **Unimplementable** Signature: Finite Map

```
module type MAP = sig

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

  val empty   : map
  val add     : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val mem     : 'k -> ('k,'v) map -> bool
  val get     : 'k -> ('k,'v) map -> 'v
  val equals  :  ('k1,'v1) map -> ('k2,'v2) map -> bool

end
```

1. Wrong implementation type (get doesn't have enough info)
2. Missing type arguments (empty) --- doesn't compile!
3. Type too generic (equals)

# .ml and .mli files

- You've already been using signatures and modules in OCaml.

- A series of type and `val` declarations stored in a file `foo.mli` is considered as defining a signature FOO
- A series of top-level definitions stored in a file `foo.ml` is considered as defining a module Foo

foo.mli

```
type t
val z : t
val f : t -> int
```

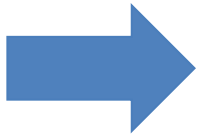
foo.ml

```
type t = int
let z : t = 0
let f (x:t) : int =
  x + 1
```

test.ml

```
;; open Foo
;; print_int
      (Foo.f Foo.z)
```

Files



```
module type FOO = sig
  type t
  val z : t
  val f : t -> int
end

module Foo : FOO = struct
  type t = int
  let z : t = 0
  let f (x:t) : int =
    x + 1
end

module Test = struct
  ;; open Foo
  ;; print_int
       (Foo.f Foo.z)
end
```

# 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 to 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 Functions

To typecheck a function:

```
fun (x:int) -> x + x
```

??

# Typechecking Functions

To typecheck a function:

```
fun (x:int) -> x + x
```

$T_{arg}$ -> $T_{ans}$

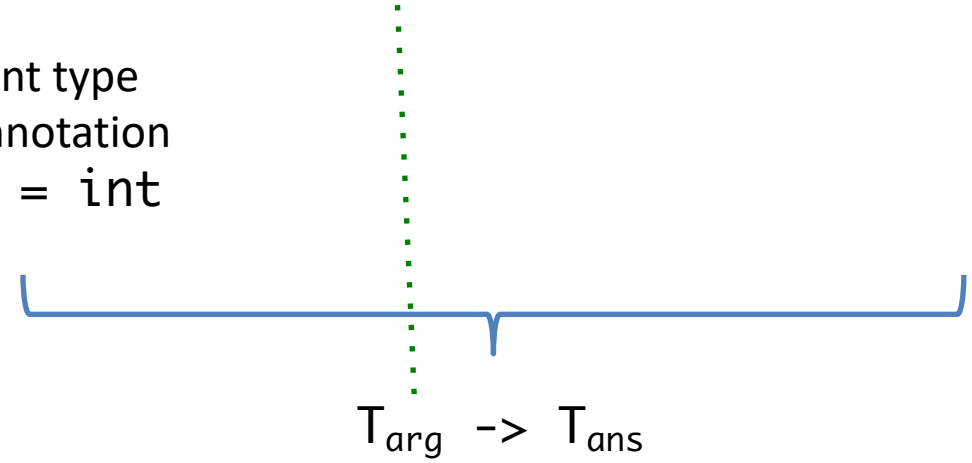
Make up "new names" for the input (argument) and output (answer) types.

# Typechecking Functions

To typecheck a function:

```
fun (x:int) -> x + x
```

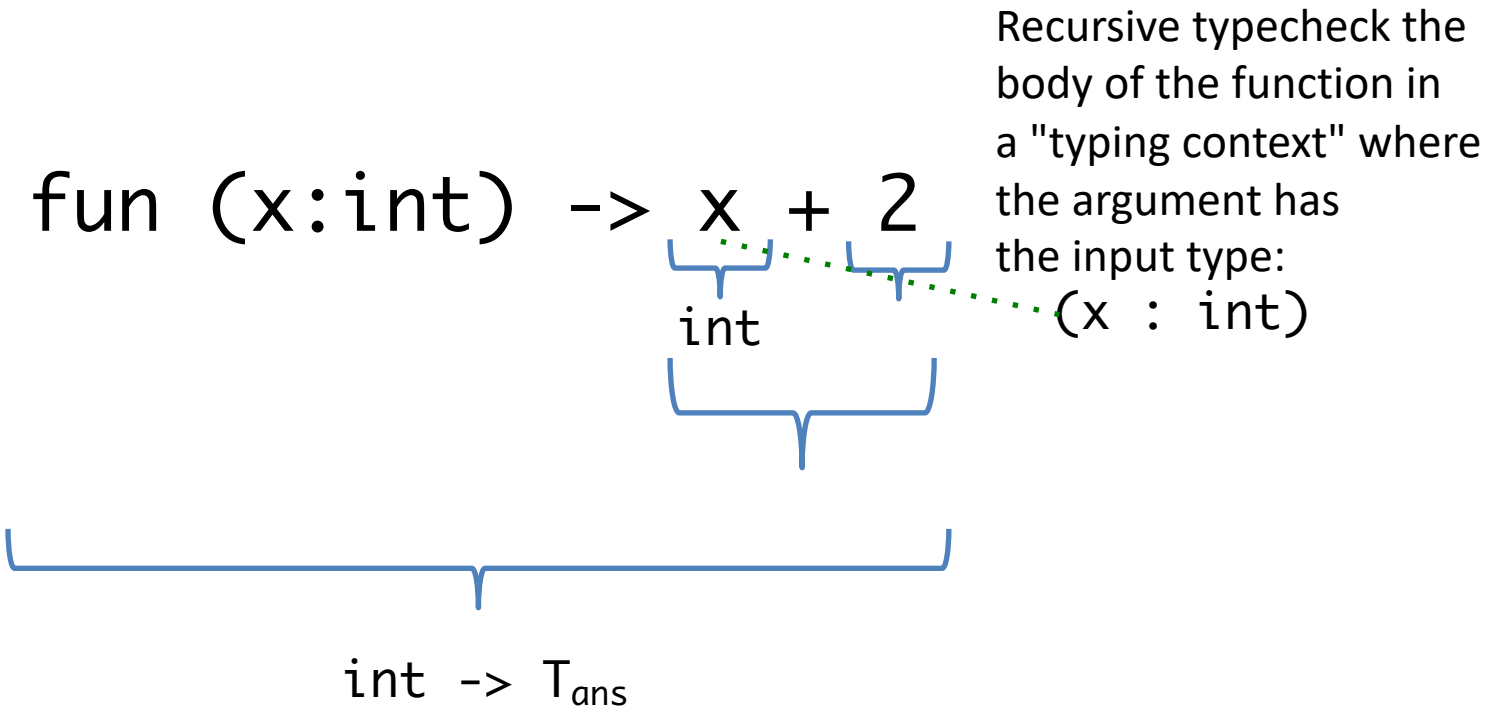
Take the argument type from the type annotation (if any*): $T_{arg}$ = `int`




*If there is no annotation, just use the "fresh" name…

# Typechecking Functions

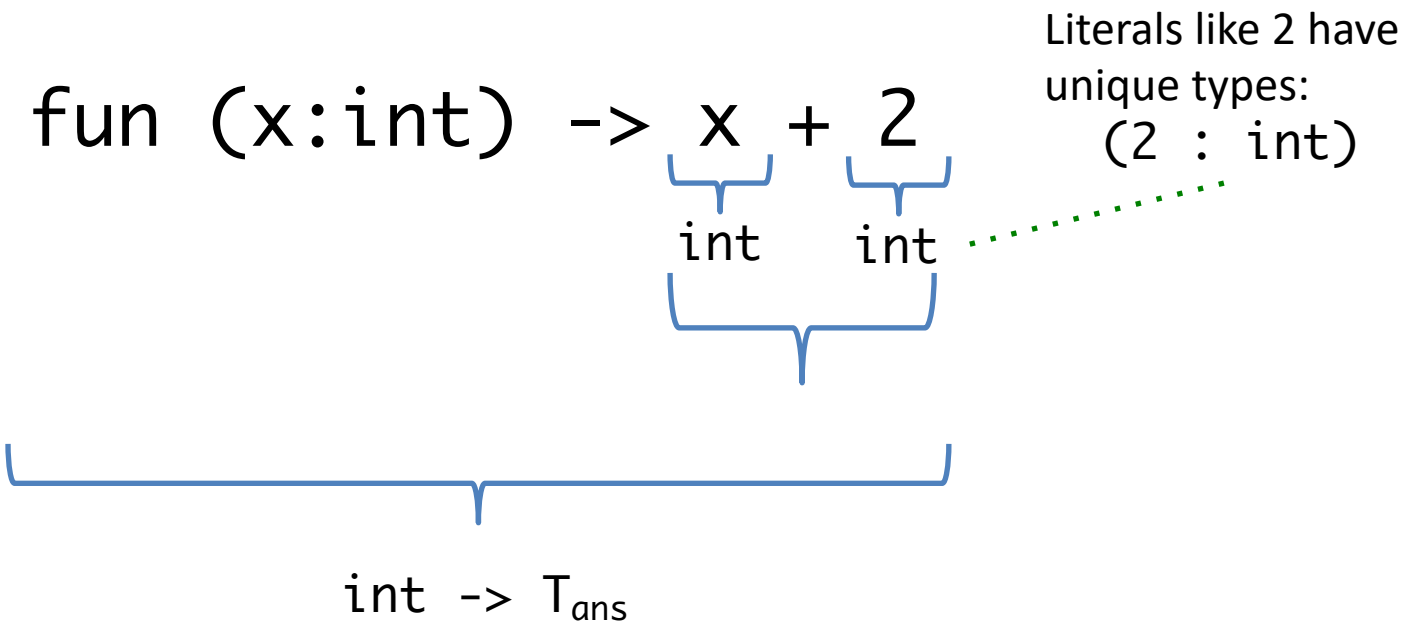
To typecheck a function:




Recursive typecheck the body of the function in a "typing context" where the argument has the input type:

# Typechecking Functions

To typecheck a function:

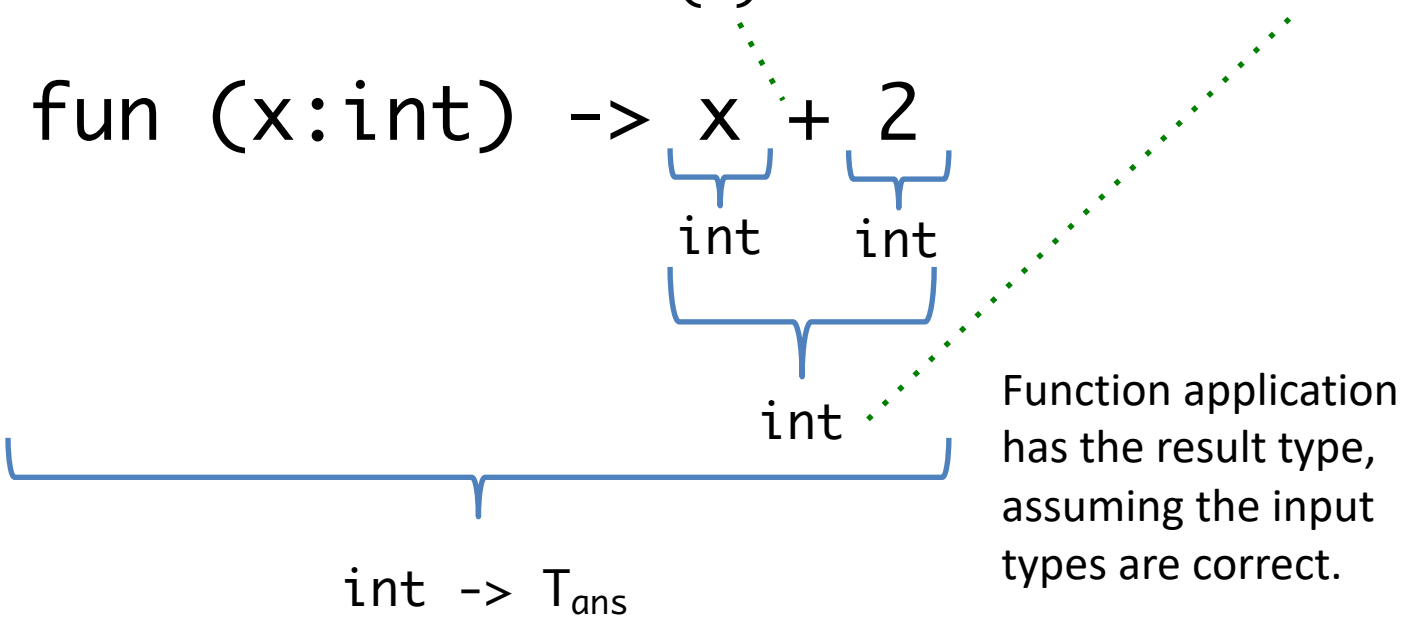


# Typechecking Functions

To typecheck a function:

Built-in operations like (+) also have types:

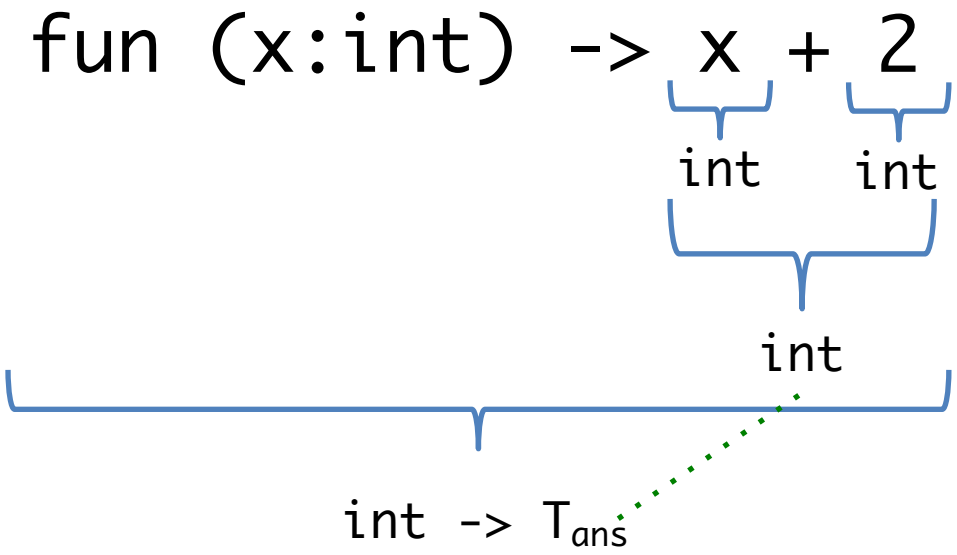
```
(+) : int -> int -> int
```




Function application has the result type, assuming the input types are correct.

# Typechecking Functions

To typecheck a function:

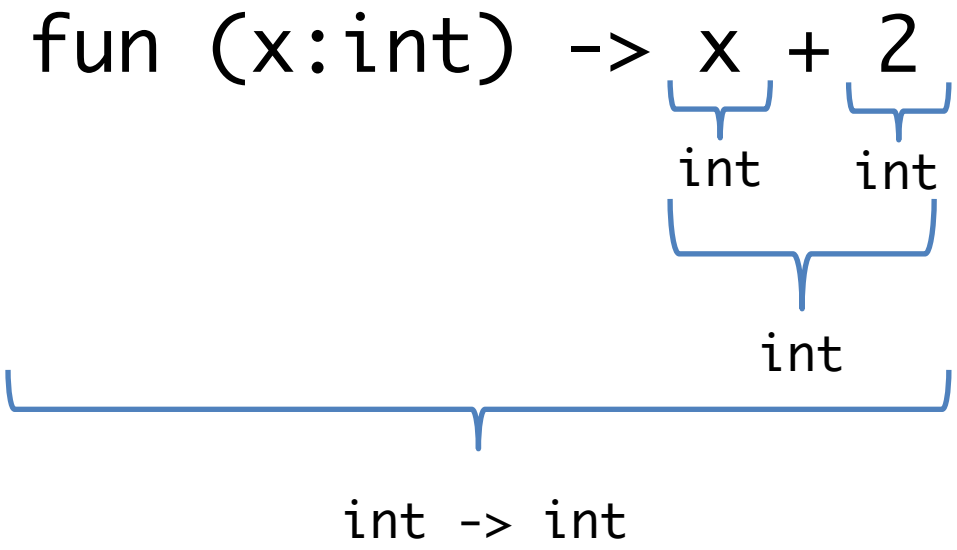



The "answer" type is the type of the function body.

$T_{ans}$ = int

# Typechecking Functions

To typecheck a function:



# Typechecking II

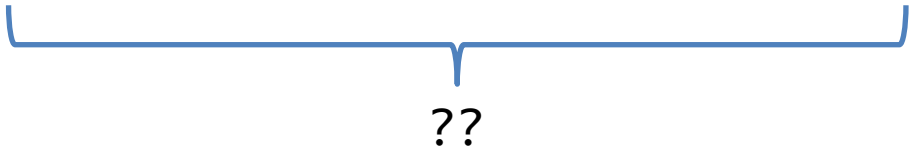
3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function `f` : $T_{arg}$ `->` $T_{ans}$
   - and an argument `e` : $T_{arg}$ of the input type
   - the application `(f e)` : $T_{ans}$ has the answer type

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

# Typechecking II

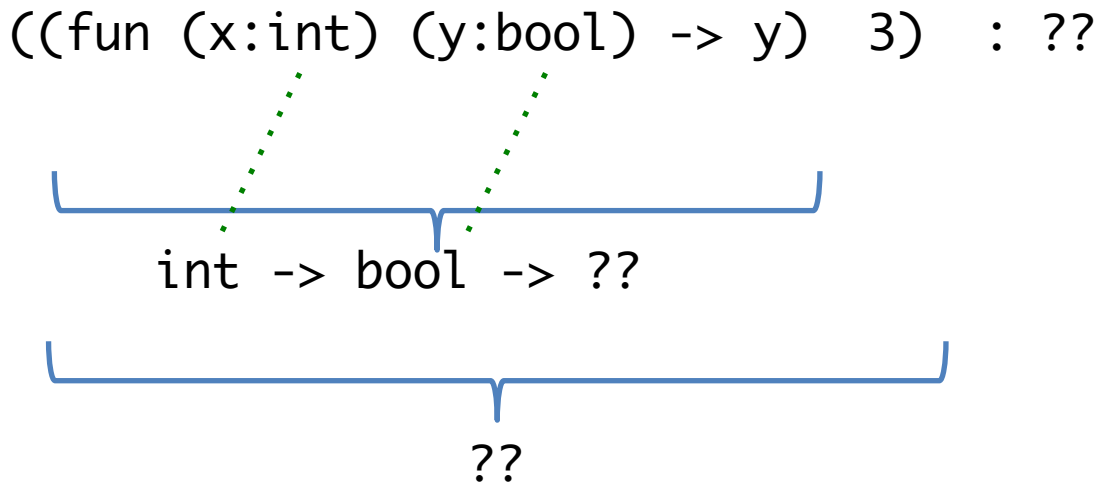
3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function `f` : $T_{arg}$ `->` $T_{ans}$
   - and an argument `e` : $T_{arg}$ of the input type
   - the application `(f e)` : $T_{ans}$ has the answer type

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



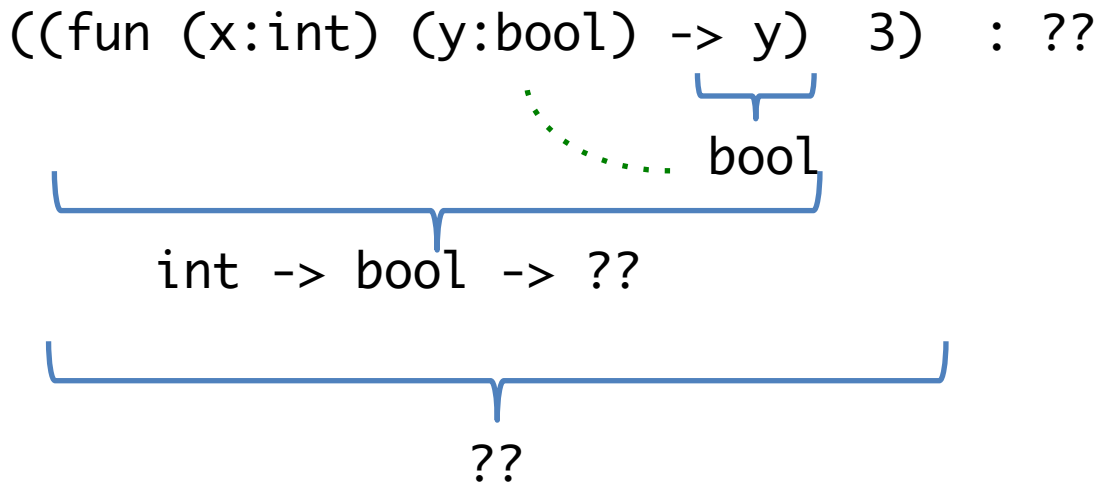
# Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function `f` : $T_{arg}$ `->` $T_{ans}$
   - and an argument `e` : $T_{arg}$ of the input type
   - the application `(f e)` : $T_{ans}$ has the answer type



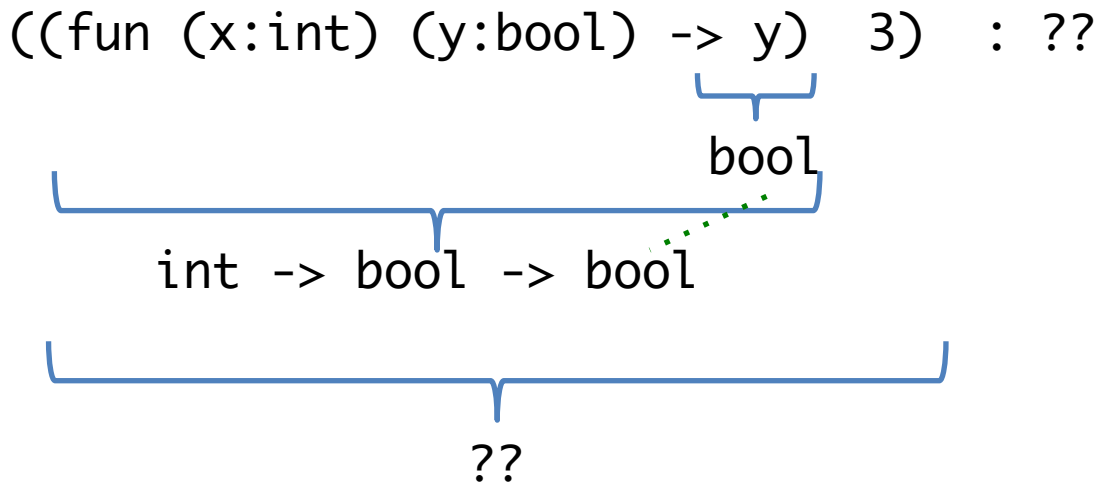
# Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function `f` : $T_{arg}$ `->` $T_{ans}$
   - and an argument `e` : $T_{arg}$ of the input type
   - the application `(f e)` : $T_{ans}$ has the answer type



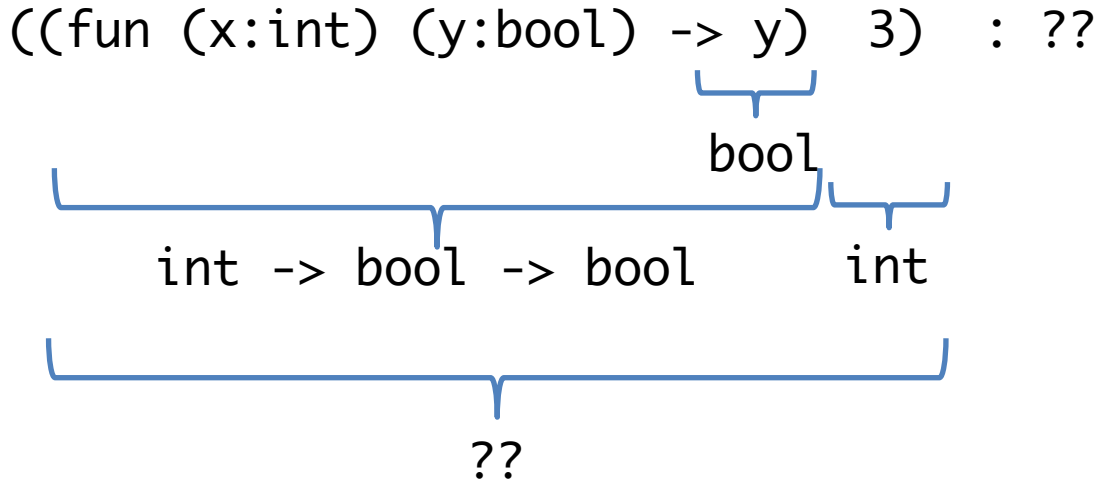
# Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function `f` : $T_{arg}$ `->` $T_{ans}$
   - and an argument `e` : $T_{arg}$ of the input type
   - the application `(f e)` : $T_{ans}$ has the answer type



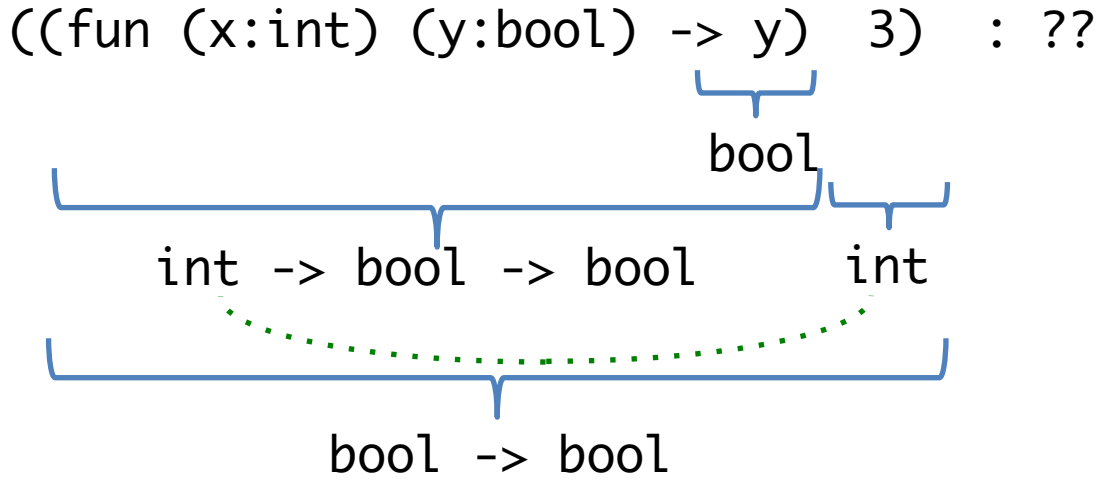
# Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function `f` : $T_{arg}$ `->` $T_{ans}$
   - and an argument `e` : $T_{arg}$ of the input type
   - the application `(f e)` : $T_{ans}$ has the answer type



# Typechecking II

3. The type of a function-application expression is obtained as the result from the function type:
   - Given a function `f` : $T_{arg}$ `->` $T_{ans}$
   - and an argument `e` : $T_{arg}$ of the input type
   - the application `(f e)` : $T_{ans}$ hasthe answer type



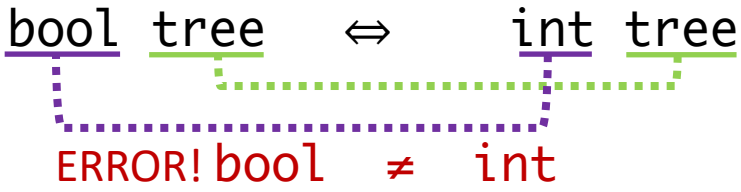

Here:
$T_{arg}$ = `int`
$T_{ans}$ = `bool -> bool`

# Typechecking III

- What about generics? i.e., what if `f : 'a ->'a`?
- For generic types we *unify*
  - Given a function `f` : $T_1$ -> $T_2$
  - 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
- *Unification:*
  - Try to match up corresponding parts of the type

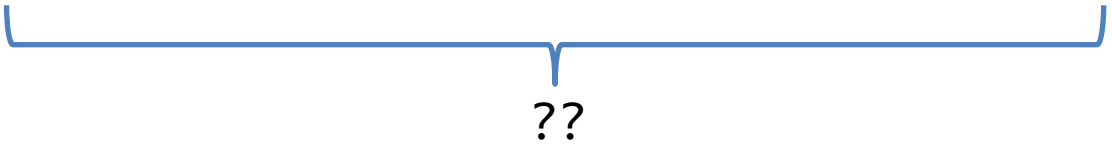

  - This produces an *instantiation*: e.g. `'a = int list`
  - *Propagate* that instantiation to all occurrences of `'a`
  - If unification fails, produce a type checking error



# Example Typechecking Problem

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

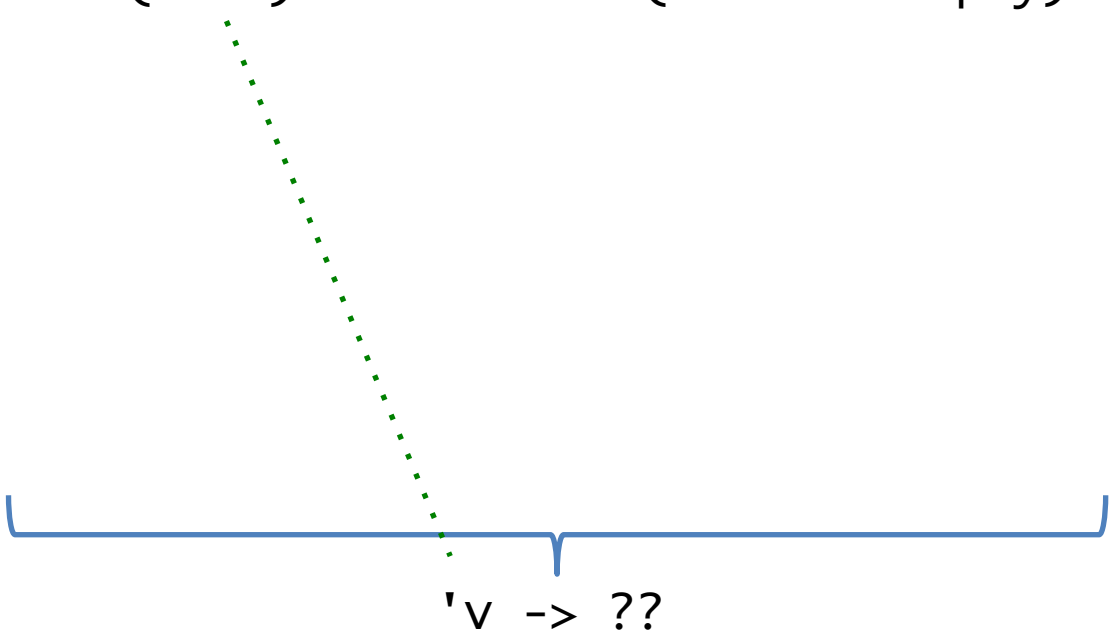
```
fun (x:'v) -> entries (add 3 x empty)
```



# Example Typechecking Problem

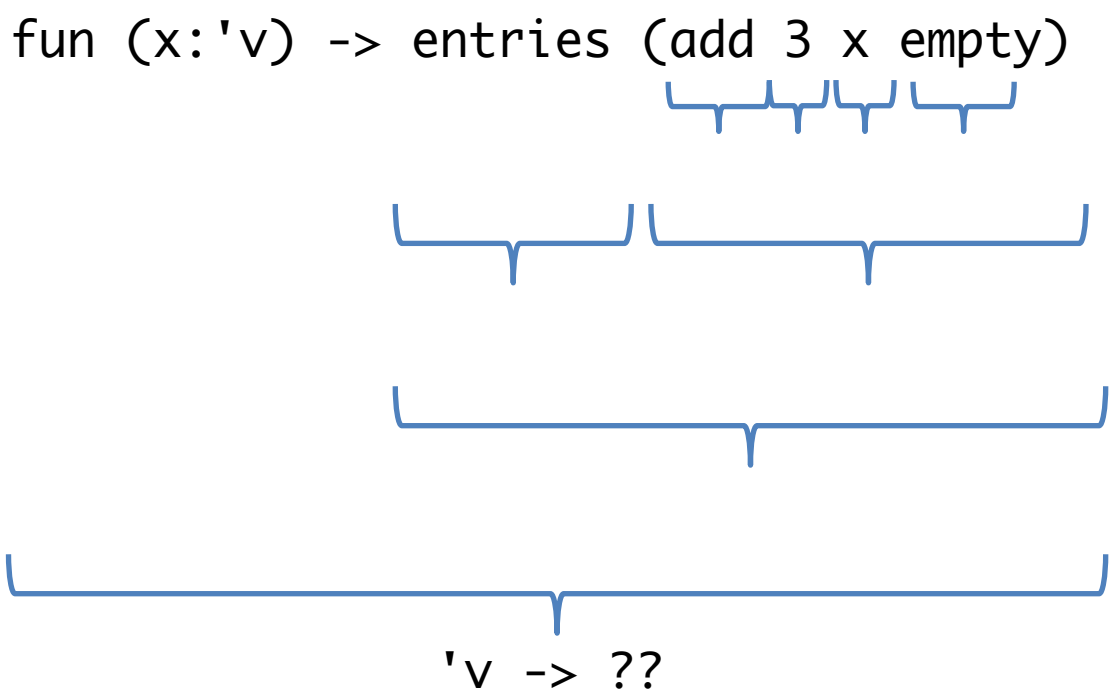
```
empty   : ('k, 'v) map
add     : 'k -> 'v -> ('k, 'v) map -> ('k, 'v) map
entries : ('k, 'v) map -> ('k * 'v) list
```

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



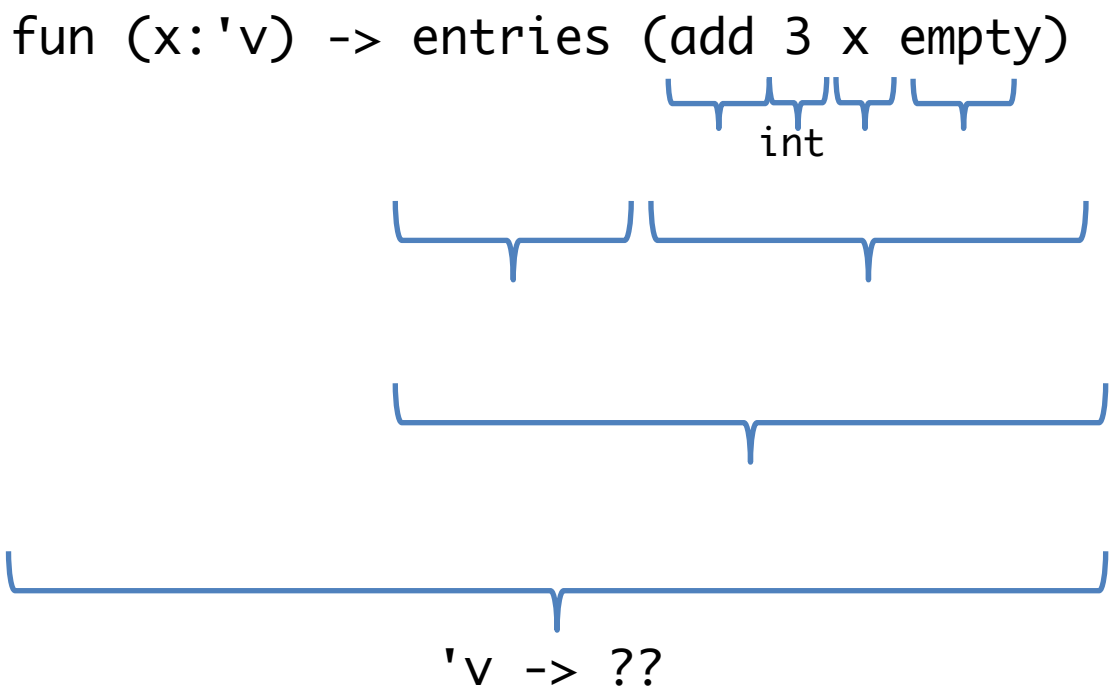
# Example Typechecking Problem

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



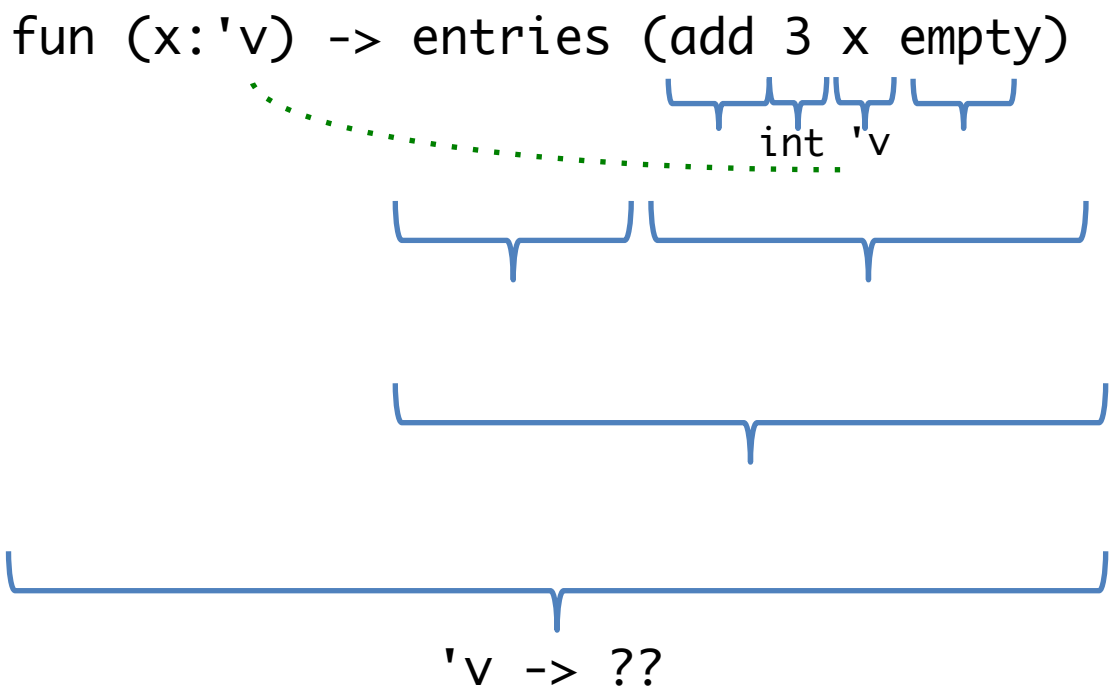
# Example Typechecking Problem

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



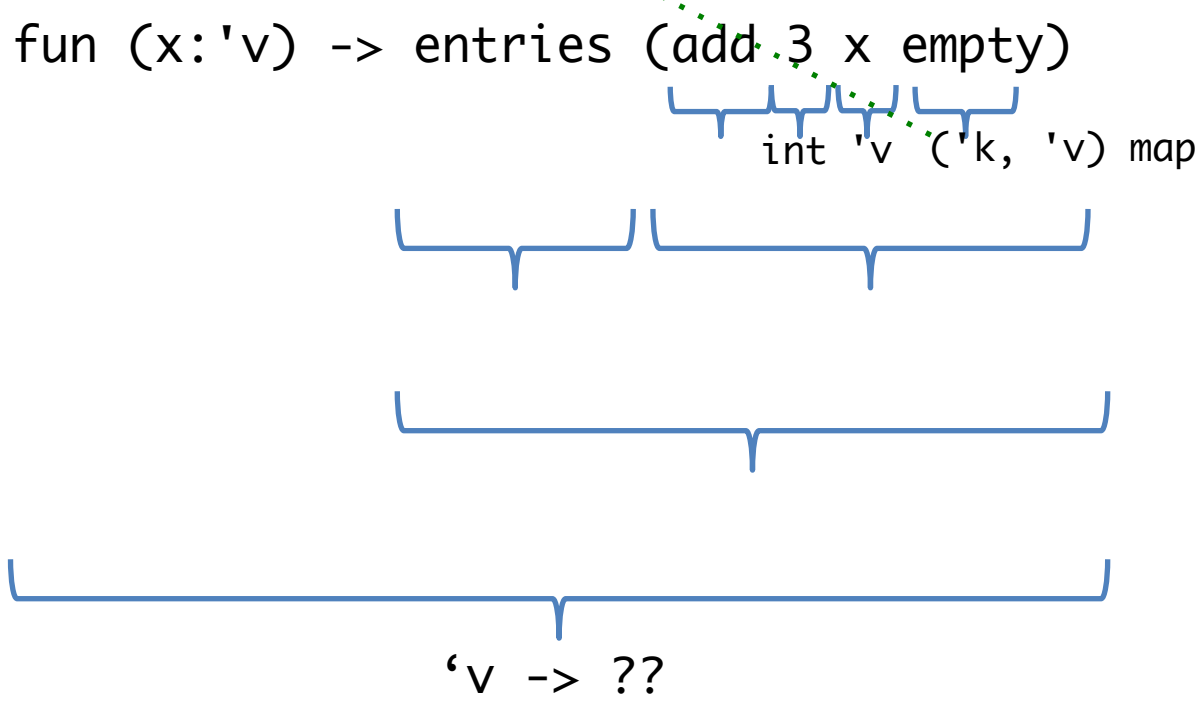
# Example Typechecking Problem

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



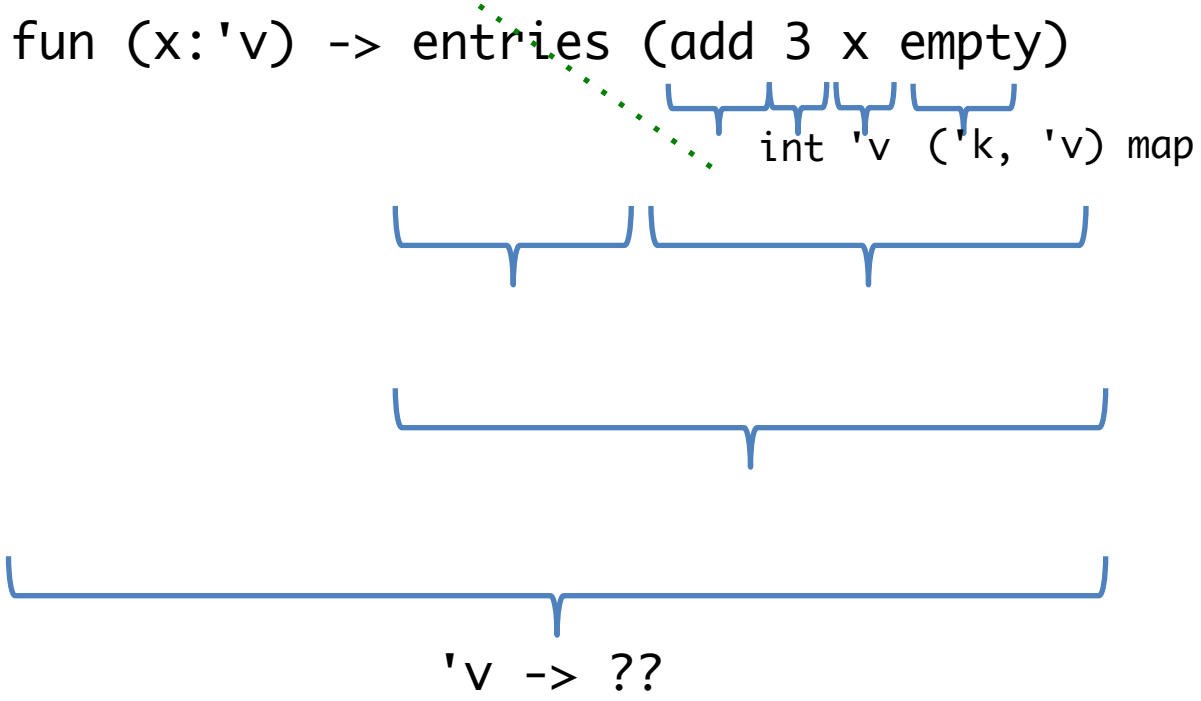
# Example Typechecking Problem

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



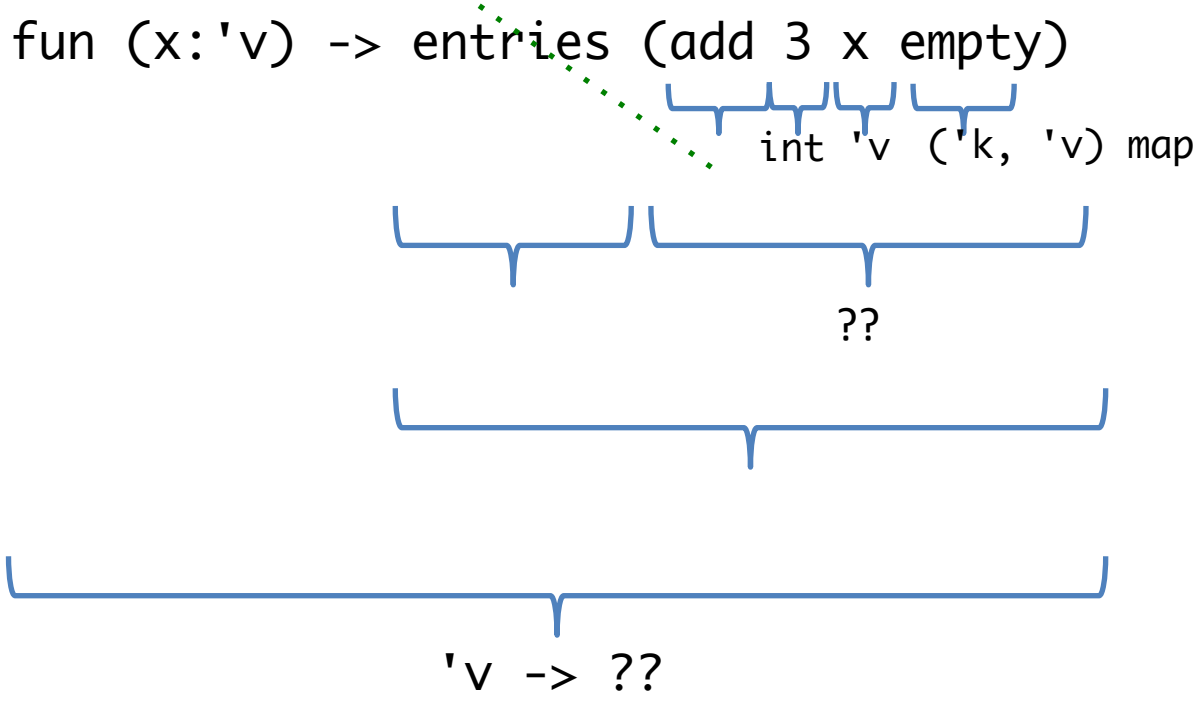
# Example Typechecking Problem

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



# 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)
```

```
                int 'v ('k, 'v) map
                  T2
```

Application:

$T_1$ = 'k

$T_2$ = 'v -> ('k,'v) map -> ('k,'v) map

Instantiate: 'k = 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)
```

int 'v ('k, 'v) map

$T_2$

Application:

$T_1$ = int

$T_2$ = 'v -> (int,'v) map -> (int,'v) map

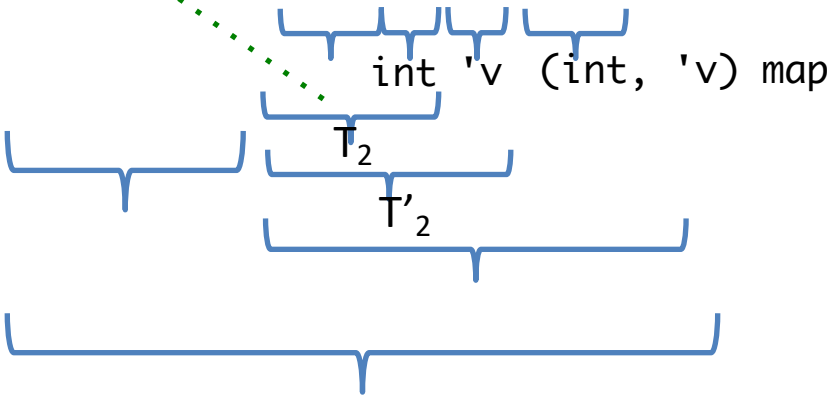
Instantiate: 'k = int

'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)
```




Another Application:

$T'_1$ = 'v

$T'_2$ = (int,'v) map -> (int,'v) map

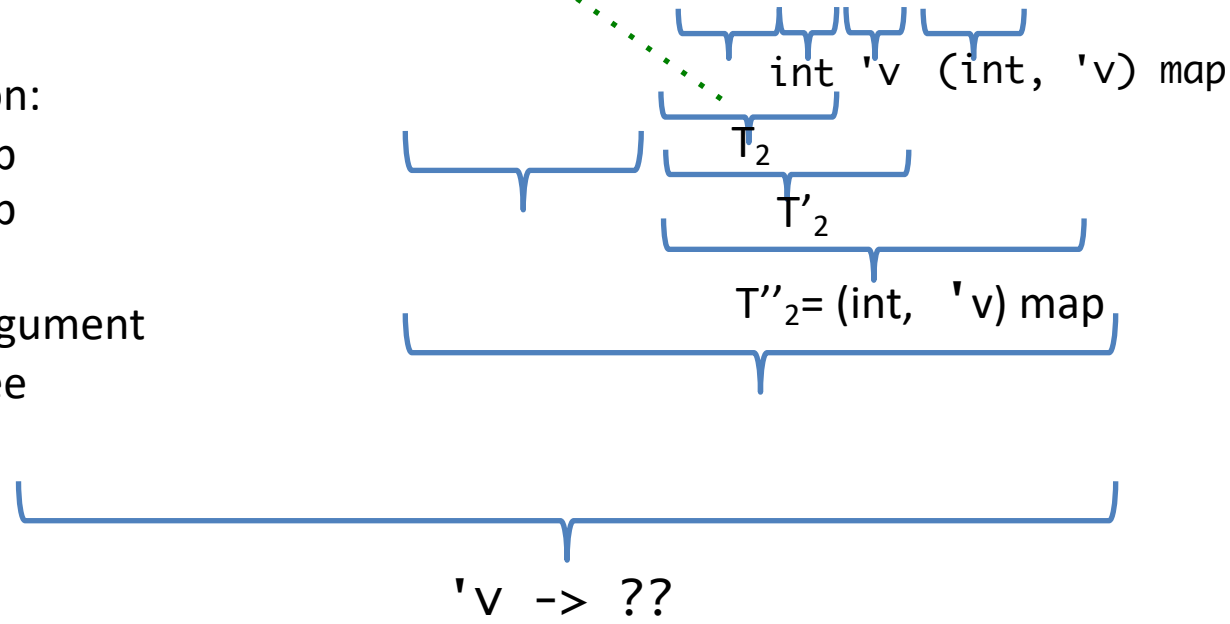
Instantiate: 'v = 'v

'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:

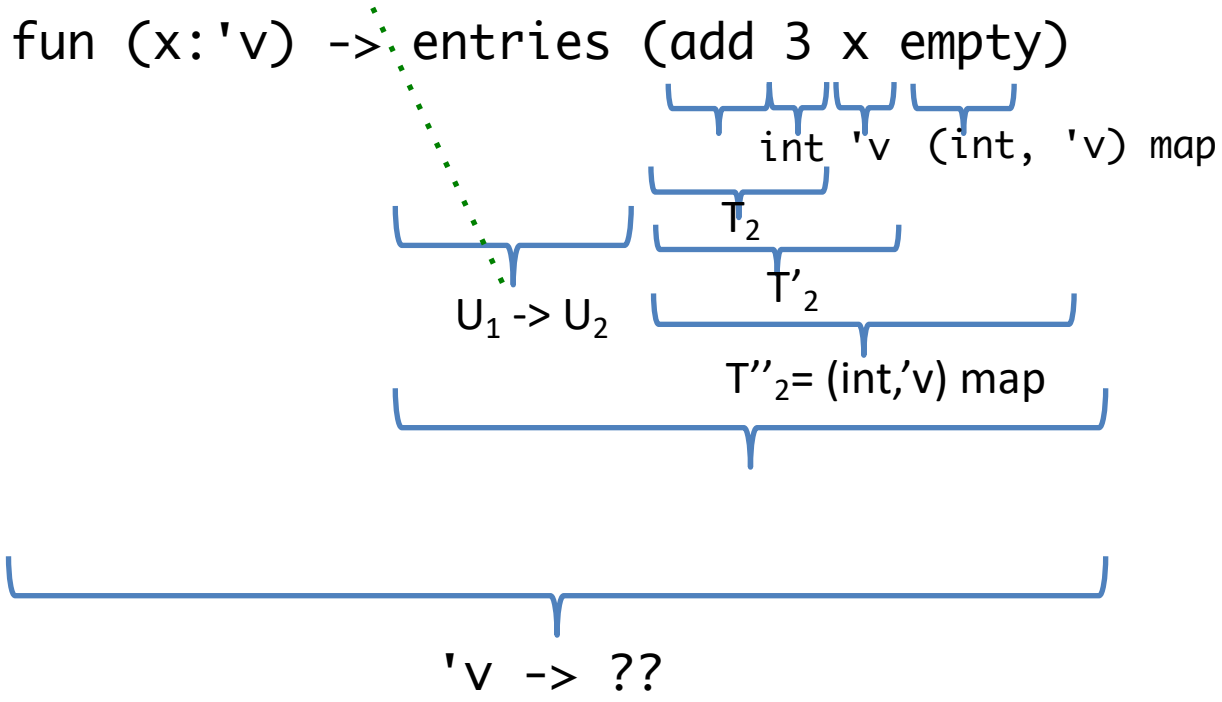
$T''_1$ = (int, 'v) map

$T''_2$ = (int, 'v) map

Argument and argument type already agree

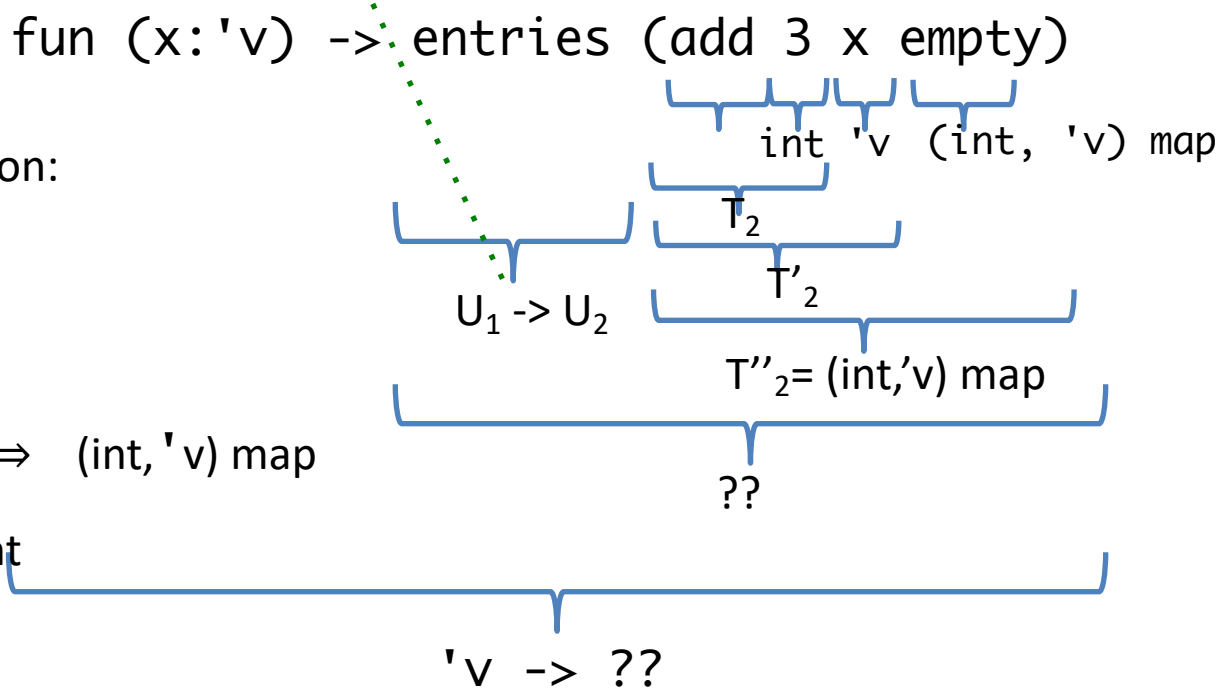
# Example Typechecking Problem

```
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
```




Another Application:

$U_1$ = ('k,'v) map

$U_2$ = ('k * 'v) list

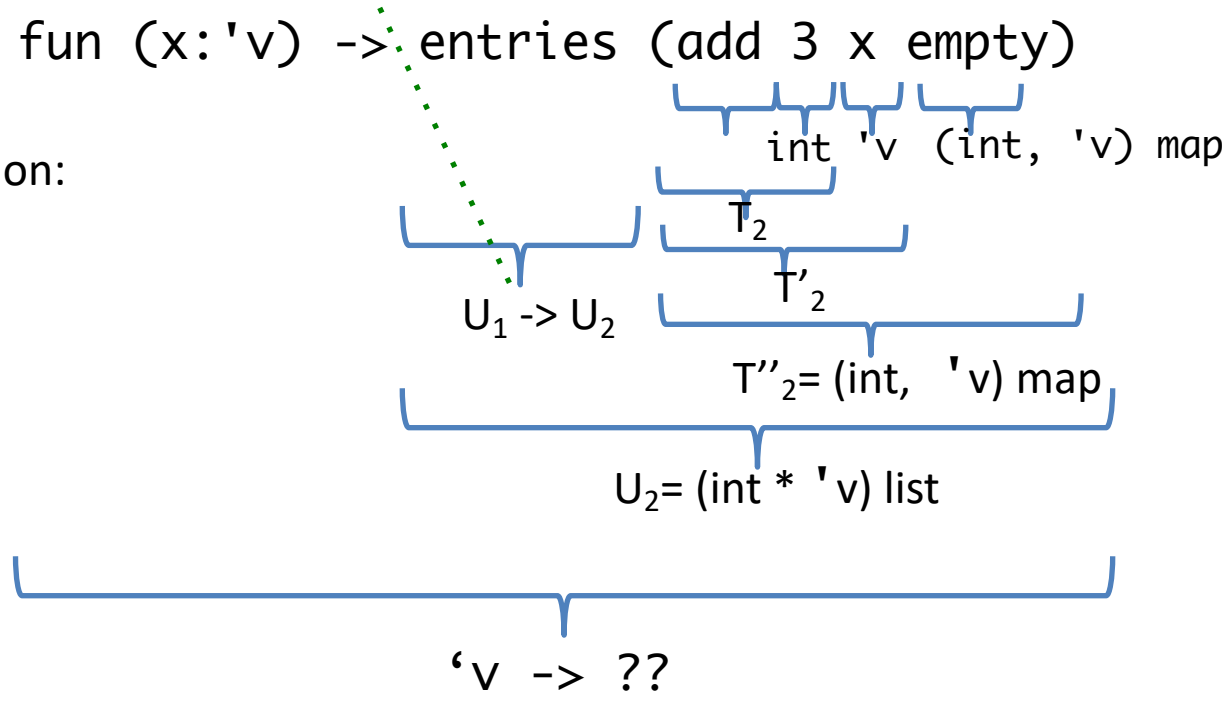
*Unify* $U_1$ with $T''_2$

('k,'v) map ⇔ (int,'v) 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
```



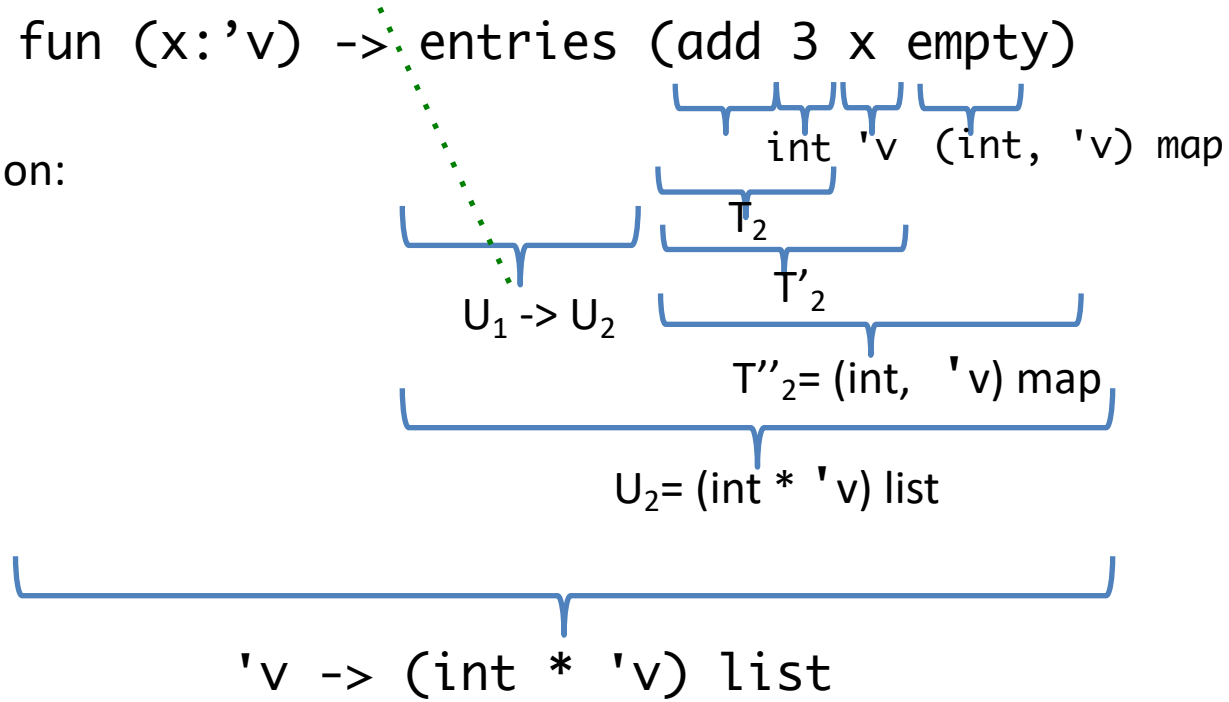

Another Application:

$U_1$ = (int, 'v) map

$U_2$ = (int * '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
```




Another Application:

$U_1$ = (int, 'v) map

$U_2$ = (int * 'v) 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`

## 12: What is the type of this expression?

0

```
let e : ______  =
  transform (fun x y -> x + y)
```

int list -> int list

0%

int list -> int list -> int list

0%

int list -> (int -> int) list

0%

None (it doesn't typecheck)

0%

What is the type of this expression?

```
let e : ______  =
  transform (fun x y -> x + y)
```

1. int list -> int list
2. int list -> int list -> int list
3. int list -> (int -> int) list
4. None (it doesn't typecheck)

Answer: 3

# Dealing with Partiality*

*A function is said to be *partial* if it is not defined for all inputs.

## 13: Which of these is a function that calculates the maximum value in a (generic) list?

0

1 — 0%

2 — 0%

3 — 0%

4 — 0%

Which of these is a function that calculates the maximum value in a (generic) list:

1.
```
let rec list_max (l:'a list) : 'a =
  begin match l with
  | [] -> []
  | h :: t -> max h (list_max t)
  end
```

2.
```
let rec list_max (l:'a list) : 'a =
  fold max 0 l
```

3.
```
let rec list_max (l:'a list) : 'a =
  begin match l with
  | h :: t -> max h (list_max t)
  end
```

4. None of the above

Answer: 4

# Quiz answer

- list_max isn't defined for the empty list!

```
let rec list_max (l:'a list) : 'a =
  begin match l with
    | [] -> failwith "empty list"
    | [h] -> h
    | h::t -> max h (list_max t)
  end
```

# Client of list_max

```
(* string_of_max calls list_max *)
let string_of_max (x:int list) : string =
  string_of_int (list_max x)
```

- Oops! `string_of_max` will fail if given `[]`
- Not so easy to debug if `string_of_max` is written by one person and `list_max` is written by another.
- Interface of list_max is not very informative

```
val list_max : int list -> int
```

# Solutions to Partiality: Option 1

- *Abort the program*:

```
failwith "an error message"
```

  - Whenever it is called, `failwith` halts the program and reports the error message it is given.
- This solution is appropriate whenever you *know* that a certain case is impossible
  - The compiler isn't smart enough to figure out that the case is impossible…
  - Often happens when there is an invariant on a data structure
- Languages (e.g. OCaml, Java) support *exception handling facilities* to let programs recover from such failures.
  - We'll talk about these when we get to Java

# Solutions to Partiality: Option 2

- Return a *default* or *error value*
  - e.g. define **`list_max []`** to be `-1`
  - Error codes used often in C programs
  - `null` used often in Java


- But…
  - What if -1 (or whatever default you choose) really *is* the maximum value?
  - Can lead to many bugs if the default isn't handled properly by the callers
  - *IMPOSSIBLE* to implement generically!
    - No way to generically create a sensible default value for every possible type
  - Sir Tony Hoare, Turing Award winner and inventor of `null` calls it his "*billion dollar mistake*"!
- *Defaults should be avoided if possible*

# Solutions to Partiality: Option 3

Return something that *cannot* be misinterpreted as a legitimate non-exceptional result …

# Optional values

Solutions to Partiality: Option 3

# Option Types

- Define a generic datatype of *optional values*

```
type 'a option =
  | None
  | Some of 'a
```

Type definition built into of OCaml. Always available.

- A “partial” function returns an option

```
let list_max (l:list) : int option = …
```

- Contrast this with “null”, a “legal” return value of any type
  - caller can accidentally forget to check whether null was used; results in NullPointerExceptions or crashes
- Modern language designs (e.g. Apple's Swift, Mozilla's Rust) distinguish between the type String (definitely not null) and String? (optional string)

# Example: list_max

- A function that returns the maximum value of a list as an option (None if the list is empty)

```
let list_max (l:'a list) : 'a option =
  begin match l with
    | [] -> None
    | x::tl -> Some (fold max x tl)
  end
```

# Revised client of list_max

```
(* string_of_max calls list_max *)
let string_of_max (l:int list) : string =
  begin match (list_max l) with
  | None -> "no maximum"
  | Some m -> string_of_int m
  end
```

- `string_of_max` will never fail
- The type of list_max makes it explicit that a *client* must check for partiality.

```
val list_max : int list -> int option
```

## 13: What is the type of this function?

0

'a list -> 'a

0%

'a list -> 'a list

0%

'a list -> 'b option

0%

'a list -> 'a option

0%

None of the above

0%

What is the type of this function?

```
let head (x: ______) : ______  =
  begin match x with
  | [] -> None
  | h :: t -> Some h
  end
```

1. ‘a list -> ‘a
2. ‘a list -> ‘a list
3. ‘a list -> ‘b option
4. ‘a list -> ‘a option
5. None of the above

Answer: 4

## 13: What is the value of this expression?

0

[1; 0]

0%

1

0%

[Some 1; None]

0%

[None; None]

0%

None of the above

0%

What is the value of this expression?

```
let head (x: 'a list) : 'a option  =
    begin match x with
    | [] -> None
    | h :: t -> Some h
    end in

[ head [1];  head [] ]
```

1. [ 1 ; 0 ]
2. 1
3. [Some 1; None]
4. [None; None]
5. None of the above

Answer: 3

# Revising the MAP interface

```
module type MAP = sig

  type ('k,'v) map

  val empty   : ('k,'v) map
  val add     : 'k -> 'v -> ('k,'v) map -> ('k,'v) map
  val remove  : 'k       -> ('k,'v) map -> ('k,'v) map
  val mem     : 'k -> ('k,'v) map -> bool
  val get     : 'k -> ('k,'v) map -> 'v option
  val entries : ('k,'v) map -> ('k * 'v) list
  val equals  : ('k,'v) map -> ('k,'v) map -> bool

end
```

`get` returns an optional 'v.
Now its type isn't a lie!

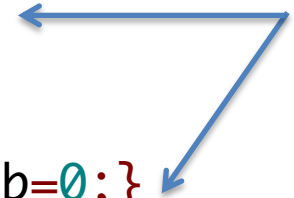
# Records

# Immutable Records

- Records are like tuples with named fields:

```
(* a type for representing colors *)
type rgb = {r:int; g:int; b:int;}

(* some example rgb values *)
let red   : rgb = {r=255; g=0;   b=0;}
let blue  : rgb = {r=0;   g=0;   b=255;}
let green : rgb = {r=0;   g=255; b=0;}
let black : rgb = {r=0;   g=0;   b=0;}
let white : rgb = {r=255; g=255; b=255;}
```




- The type rgb is a record with three fields: r, g, and b
  - fields can have any types; they don't all have to be the same
- Record values are created using this notation:

```
{field1=val1; field2=val2;…}
```

# Field Projection

- The value in a record field can be obtained by using "dot" notation: `record.field`

```
(* a type for representing colors *)
type rgb = {r:int; g:int; b:int;}

(* using 'dot' notation to project out components *)
(* calculate the average of two colors *)
let average_rgb (c1:rgb) (c2:rgb) : rgb =
  {r = (c1.r + c2.r) / 2;
   g = (c1.g + c2.g) / 2;
   b = (c1.b + c2.b) / 2;}
```

## 13: What is the type of f in the following program?

0

unit -> int

0%

unit -> unit

0%

int -> unit

0%

int -> int

0%

f is ill typed

0%

## 13: What is the type of f in the following program?

0

unit -> int

0%

unit -> unit

0%

int -> unit

0%

int -> int

0%

f is ill typed

0%

## 13: What answer does the following function produce when called?

0

17

0%

something else

0%

sometimes 17 and sometimes someth…

0%

f is ill typed

0%

## 13: What answer does the following function produce when called?

0

17

0%

something else

0%

sometimes 17 and sometimes someth…

0%

f is ill typed

0%