Programming Languages and Techniques (CIS1200)

Lecture 12

Finite Maps and Typechecking

Chapter 10

Announcements

- Midterm 1: Friday, February 14th
  - Coverage: up to today (Chapters 1-10)
  - During lecture
    Last names: A – Z Meyerson Hall B1
  - 60 minutes; closed book, **single-sided note page allowed**
  - 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

Announcements

- Midterm 1: ~~Friday, February 14th~~ **Monday, February 17th**
  - Coverage: up to today (Chapters 1-10)
  - During lecture
    Last names: A – L ~~Meyerson Hall B1~~ Stiteler AUD (here)
    M – Z DRL A1
  - 60 minutes; closed book, **single-sided note page allowed**
  - MAKE UP EXAM, **Wednesday, February 19th** 9-10AM, must preregister
  - 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

12: Which do you prefer?

I want to see new material in class on Friday

0%

I would rather watch a recorded lecture of the new material after the exam (but before Wednesday's class)

0%

I don't have a preference

0%

Finite Maps

A case study on abstract interfaces

and concrete implementations

Motivating Scenario

- Suppose you were writing a course-management system and needed to look up the lab section for a student given the student's PennKey…
  - Students might add/drop the course
  - Students might switch lab sections
  - Students should be in only *one* lab section

- How would you do it? What data structure would you use?

Key/Value store

Key	Value
“stephanie”	15
“swap”	05
“ezaan”	10
“likat”	15
…	…

- Each key is associated with a value.
  - No two keys are identical
  - Values can be repeated
- Given the key “stephanie”, we want to find / lookup the value 15

Finite Maps

- A *finite map* (a.k.a. *dictionary*) is a collection of *entries* from distinct *keys* to *values*.
  - Operations to *add* a new entry, *test* for key membership, *get* the value bound to a particular key, *list* all entries stored in the map

- Example: we might use a finite map to look up the lab section of a CIS 1200 student

- Like sets, *finite maps* appear in many settings:
  - domain names to IP addresses
  - words to their definitions (a dictionary)
  - user names to passwords
  - …

Design Process Step 1:
Understand the problem

Signature: Finite Map

Design Process Step 2: specify the 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
  val equals  : ('k,'v) map -> ('k,'v) map -> bool

end
```

The map type is generic in *two* ways: type of keys *and* type of values

Properties of Finite Maps

For any finite map m, key k, and value v:

Design Process Step 3: write test cases

1. `get k (add k v m) = v`
2. If `k1 <> k2` then
   `get k1 (add k2 v2 (add k1 v1 m)) = v1`
3. If `mem k m = true` then
   there is a `v` such that `get k m = v`
4. If `mem k m = false` then
   `get k m = v` fails
5. `mem k (add k v m) = true`

(among others…)

Tests for Finite Map abstract type

```
;; open Assert

(* A simple map with one element. *)
let m1 : (int,string) map = add 1 "uno" empty

(* access value for key in the map *)
;; run_test "find 1 m1" (fun () -> (get 1 m1) = "uno")

(* find for value that does not exist in the map? *)
;; run_failing_test "find 2 m1" (fun () -> (get 2 m1) = "dos" )

let m2 : (int, string) map = add 1 "un" m1

(* find after redefining value, should be new value *)
;; run_test "find 1 m2" (fun () -> (get 1 m2) = "un")

(* test membership *)
;; run_test "mem test" (fun () ->
      mem 1 (add 2 "dos" (add 1 "uno" empty)))
```

Using an anonymous function avoids making up a (redundant) function name for the test

Design Process Step 3: write test cases

Finite Map Demo

Implementing the module

finiteMap.ml

Implementation: Ordered Lists

```
module Assoc : MAP = struct
  (* Represent a finite map as a list of pairs.       *)
  (* Representation invariant:                        *)
  (*   - no duplicate keys (helps get, remove)        *)
  (*   - keys are sorted (helps equals, get)          *)

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

  let empty : ('k,'v) map = []

  let rec mem (key:'k) (m : ('k,'v) map) : bool =
    begin match m with
   | [] -> false
   | (k,v)::rest ->
     (key >= k) &&
        ((key = k) || (mem key rest))
    end
```

Design Process Step 4: implement it!

Implementation: Ordered Lists

```
let rec get (key:'k) (m : ('k,'v) map) : 'v =
  begin match m with
  | [] -> failwith "key not found"
  | (k,v)::rest ->
    if key < k then failwith "key not found"
    else if key = k then v
    else get key rest
  end

let rec remove (key:'k) (m : ('k,'v) map) : ('k,'v) map =
  begin match m with
  | [] -> []
  | (k,v)::rest ->
    if key < k then m
    else if key = k then rest
    else (k,v)::remove key rest
  end
```

Language support for Abstract Types

- Programming languages support abstract types in different ways
- Key concept: Encapsulation
  - A way to specify (write down) an interface
  - A means of hiding implementation details
- Benefits: safety and flexibility
- In OCaml:
  - Interfaces are specified using a *signature* or *interface*
  - Encapsulation: the interface can *omit* information
    - type definitions
    - names of auxiliary functions
  - Clients *cannot* mention values or types not named in the interface

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 me
  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
```

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

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:

$$\underbrace{\texttt{fun (x:int) -> x + x}}_{??}$$

Typechecking Functions

To typecheck a function:

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

$\underbrace{\phantom{\texttt{fun (x:int) -> x + x}}}_{T_{arg} \texttt{ -> } T_{ans}}$

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

Typechecking Functions

To typecheck a function:

$$\underbrace{\texttt{fun (x:int) -> x + x}}_{T_{arg} \texttt{ -> } T_{ans}}$$

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:

$$\underbrace{\texttt{fun (x:int) -> }\underbrace{\underbrace{\texttt{x}}_{\texttt{int}}\texttt{ + }\underbrace{\texttt{2}}_{}}_{}}_{\texttt{int -> }T_{ans}}$$

Recursively typecheck the body of the function in a "typing context" where the argument has the input type:
`(x : int)`

Typechecking Functions

To typecheck a function:

$$\underbrace{\texttt{fun (x:int) -> }\underbrace{\underbrace{\texttt{x}}_{\texttt{int}}\texttt{ + }\underbrace{\texttt{2}}_{\texttt{int}}}_{}}_{\texttt{int -> }T_{ans}}$$

Literals like 2 have unique types:

`(2 : int)`

Typechecking Functions

To typecheck a function:

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

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

$$\underbrace{\texttt{fun (x:int) -> }\underbrace{\underbrace{\texttt{x}}_{\texttt{int}}\texttt{ + }\underbrace{\texttt{2}}_{\texttt{int}}}_{\texttt{int}}}_{\texttt{int -> }T_{ans}}$$

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

Typechecking Functions

To typecheck a function:

$$\underbrace{\texttt{fun (x:int) -> } \underbrace{\underbrace{\texttt{x}}_{\texttt{int}} \texttt{ + } \underbrace{\texttt{2}}_{\texttt{int}}}_{\texttt{int}}}_{\texttt{int -> } T_{ans}}$$

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

T_{ans} = `int`

Typechecking Functions

To typecheck a function:

$$\underbrace{\texttt{fun (x:int) -> }\underbrace{\underbrace{\texttt{x}}_{\texttt{int}}\texttt{ + }\underbrace{\texttt{2}}_{\texttt{int}}}_{\texttt{int}}}_{\texttt{int -> int}}$$

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

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

```
int -> bool -> ??
```

```
??
```

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

$$\underbrace{\texttt{(}\underbrace{\texttt{(fun (x:int) (y:bool) -> }\underbrace{\texttt{y)}}_{\texttt{bool}}}_{\texttt{int -> bool -> ??}}\texttt{ 3)}}_{\texttt{??}}\ \texttt{: ??}$$

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)  : ??
                          bool
   int -> bool -> bool
              ??
```

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

$$\underbrace{\texttt{(}\underbrace{\texttt{(fun (x:int) (y:bool) -> }\underbrace{\texttt{y}}_{\texttt{bool}}\texttt{)}}_{\texttt{int -> bool -> bool}}\ \underbrace{\texttt{3}}_{\texttt{int}}\texttt{)}}_{\texttt{??}}\ \texttt{: ??}$$

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

$$\underbrace{\texttt{(}\underbrace{\texttt{(fun (x:int) (y:bool) -> }\underbrace{\texttt{y)}}_{\texttt{bool}}}_{\texttt{int -> bool -> bool}}\ \underbrace{\texttt{3)}}_{\texttt{int}}}_{\texttt{bool -> bool}}\ \texttt{: ??}$$

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

    `(int list) tree` ⇔ `'a tree`
  - 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

    `bool tree` ⇔ `int tree`

    ERROR! `bool` ≠ `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)
```

??

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)
```

'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)
```

```
'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)
                           int '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)
```

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)
```

int 'v ('k, 'v) map

T_2

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)
```

int 'v (int, 'v) map

T_2

T'_2

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)
```

int 'v (int, 'v) map

T_2

T'_2

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

A third Application:
T''_1 = (int, 'v) map
T''_2 = (int, 'v) map

Argument and argument type already agree

'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 (int, 'v) map

T_2

T'_2

U_1 -> U_2

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

T_2

T'_2

U_1 -> U_2

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

??

'v -> ??

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
```

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

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

T_2

T'_2

U_1 -> U_2

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

U_2= (int * 'v) list

‘v -> ??

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
```

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

int 'v (int, 'v) map

T_2

T'_2

U_1 -> U_2

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

U_2= (int * 'v) list

Another Application:
U_1 = (int, 'v) map
U_2 = (int * 'v) list

```
'v -> (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`