CSCI 2041

Sebastiaan Joosten

September 15th: More on types

Homework

- Due Sunday!!

Overview

- How does type inference work?
- Trees

Type inference in ocaml

- Two phases:
  - collect constraints
  - solve constraints

Phase 1: Collecting constraints

- Program:
- let f a
  = a - 1

- Subexpressions:
  f : ‘f
  a : ‘a
  f a : ‘fa
  (-) a 1 : ‘ma1
  (-) a : ‘ma
  - : int -> int -> int
  1 : int

- Constraints:
  ‘f ~ (‘x1 -> ‘y1)
  ‘a ~ ‘x1
  ‘fa ~ ‘y1
  ‘ma ~ (‘x2 -> ‘y2)
  int ~ ‘x2
  ‘ma1 ~ ‘y2
  (int -> int -> int) ~ (‘x3 -> ‘y3)
  ‘a ~ ‘x3
  ‘ma ~ ‘y3
- Definition constraint
  ‘fa ~ ‘ma1

- application rule: (‘x -> ‘y) -> ‘x -> ‘y

Phase 1: Collecting constraints

- Program:
- let f a
  = a - 1

- Subexpressions:
  f : ‘f
  a : ‘a
  f a : ‘fa
  (-) a 1 : ‘ma1
  (-) a : ‘ma
  - : int -> int -> int
  1 : int

- Constraints:
  ‘f ~ (‘x1 -> ‘y1)
  ‘a ~ ‘x1
  ‘fa ~ ‘y1
  ‘ma ~ (‘x2 -> ‘y2)
  int ~ ‘x2
  ‘ma1 ~ ‘y2
  (int -> int -> int) ~ (‘x3 -> ‘y3)
  ‘a ~ ‘x3
  ‘ma ~ ‘y3
- Definition constraint
  ‘fa ~ ‘ma1

- application rule: (‘x -> ‘y) -> ‘x -> ‘y

Phase 1: Collecting constraints

- Program:

- let f a
  = a - 1

- Subexpressions:
  f : ‘f
  a : ‘a
  f a : ‘fa
  (-) a 1 : ‘ma1
  (-) a : ‘ma
  - : int -> int -> int
  1 : int

- Constraints:
  ‘f ~ (‘x1 -> ‘y1)
  ‘a ~ ‘x1
  ‘fa ~ ‘y1
  ‘ma ~ (‘x2 -> ‘y2)
  int ~ ‘x2
  ‘ma1 ~ ‘y2
  (int -> int -> int) ~ (‘x3 -> ‘y3)
  ‘a ~ ‘x3
  ‘ma ~ ‘y3

- Definition constraint
  ‘fa ~ ‘ma1

- application rule: (‘x -> ‘y) -> ‘x -> ‘y

Phase 2: Solving constraints

- Step 1: pick any constraint.
  - If it is (of the form)
    'x ~ Y, or Y ~ 'x
    just replace all 'x for Y
    (provided Y does not
    contain 'x) and deactivate
    the constraint

- Constraints:
  'f ~ ('x1 -> 'y1)
  'a ~ 'x1
  'fa ~ 'y1
  'ma ~ ('x2 -> 'y2)
  int ~ 'x2
  'ma1 ~ 'y2
  (int -> int -> int) ~ ('x3 -> 'y3)
  'a ~ 'x3
  'ma ~ 'y3
- Definition constraint
  'fa ~ 'ma1

- ‘f ~ (‘x1 -> ‘y1)

Phase 2: Solving constraints

- Step 1: pick any constraint.
  - If it is (of the form)
    ‘x ~ Y, or Y ~ ‘x
    just replace all ‘x for Y
    (provided Y does not
    contain ‘x) and deactivate
    the constraint
    (repeat)

- Constraints:

  ‘a ~ ‘x1
  ‘fa ~ ‘y1
  ‘ma ~ (‘x2 -> ‘y2)
  int ~ ‘x2
  ‘ma1 ~ ‘y2
  (int -> int -> int) ~ (‘x3 -> ‘y3)
  ‘a ~ ‘x3
  ‘ma ~ ‘y3
- Definition constraint
  ‘fa ~ ‘ma1

Phase 2: Solving constraints

- ‘f ~ (‘x1 -> ‘y1)
- ‘a ~ ‘x1

- Step 1: pick any constraint.
  - If it is (of the form)
    ‘x ~ Y, or Y ~ ‘x
    just replace all ‘x for Y
    (provided Y does not
    contain ‘x) and deactivate
    the constraint

- Constraints:

  ‘fa ~ ‘y1
  ‘ma ~ (‘x2 -> ‘y2)
  int ~ ‘x2
  ‘ma1 ~ ‘y2
  (int -> int -> int) ~ (‘x3 -> ‘y3)
  ‘x1 ~ ‘x3
  ‘ma ~ ‘y3
- Definition constraint
  ‘fa ~ ‘ma1

Phase 2: Solving constraints

- ‘f ~ (‘x1 -> ‘y1)
- ‘a ~ ‘x1
- ‘fa ~ ‘y1

- Step 1: pick any constraint.
  - If it is (of the form)
    ‘x ~ Y, or Y ~ ‘x
    just replace all ‘x for Y
    (provided Y does not
    contain ‘x) and deactivate
    the constraint

- Constraints:

  ‘ma ~ (‘x2 -> ‘y2)
  int ~ ‘x2
  ‘ma1 ~ ‘y2
  (int -> int -> int) ~ (‘x3 -> ‘y3)
  ‘x1 ~ ‘x3
  ‘ma ~ ‘y3
- Definition constraint
  ‘y1 ~ ‘ma1

Phase 2: Solving constraints

- 'f ~ ('x1 -> 'y1)
- 'a ~ 'x1
- 'fa ~ 'y1
- 'ma ~ ('x2 -> 'y2)

- Step 1: pick any constraint.
  - If it is (of the form)
    'x ~ Y, or Y ~ 'x
    just replace all 'x for Y
    (provided Y does not
    contain 'x) and deactivate
    the constraint

- Constraints:

  int ~ 'x2
  'ma1 ~ 'y2
  (int -> int -> int) ~ ('x3 -> 'y3)
  'x1 ~ 'x3
  ('x2 -> 'y2) ~ 'y3
- Definition constraint
  'y1 ~ 'ma1

Phase 2: Solving constraints

- Step 1: pick any constraint.
  - If it is (of the form)
    ‘x ~ Y, or Y ~ ‘x
    just replace all ‘x for Y
    (provided Y does not
    contain ‘x) and deactivate
    the constraint

- Constraints:
  - ‘f ~ (‘x1 -> ‘y1)
  - ‘a ~ ‘x1
  - ‘fa ~ ‘y1
  - ‘ma ~ (‘x2 -> ‘y2)
  - int ~ ‘x2

  ‘ma1 ~ ‘y2
  (int -> int -> int) ~ (‘x3 -> ‘y3)
  ‘x1 ~ ‘x3
  (int -> ‘y2) ~ ‘y3
- Definition constraint
  ‘y1 ~ ‘ma1

Phase 2: Solving constraints

- Step 1: pick any constraint.
  - If it is (of the form)
    (A -> B) ~ (C -> D)
    replace it by:
    A ~ C
    B ~ D
    (then repeat)

- Constraints:
  - ‘f ~ (‘x1 -> ‘y1)
  - ‘a ~ ‘x1
  - ‘fa ~ ‘y1
  - ‘ma ~ (‘x2 -> ‘y2)
  - int ~ ‘x2
  - ‘ma1 ~ ‘y2

  (int -> int -> int) ~ (‘x3 -> ‘y3)
  ‘x1 ~ ‘x3
  (int -> ‘y2) ~ ‘y3
- Definition constraint
  ‘y1 ~ ‘y2

Phase 2: Solving constraints

- Step 1: pick any constraint.
  - If it is (of the form)
    (A -> B) ~ (C -> D)
    replace it by:
    A ~ C
    B ~ D
    (then repeat)

- Constraints:
  - ‘f ~ (‘x1 -> ‘y1)
  - ‘a ~ ‘x1
  - ‘fa ~ ‘y1
  - ‘ma ~ (‘x2 -> ‘y2)
  - int ~ ‘x2
  - ‘ma1 ~ ‘y2

  (int -> (int -> int)) ~ (‘x3 -> ‘y3)
  ‘x1 ~ ‘x3
  (int -> ‘y2) ~ ‘y3
- Definition constraint
  ‘y1 ~ ‘y2

Phase 2: Solving constraints

- Step 1: pick any constraint.
  - If it is (of the form)
    (A -> B) ~ (C -> D)
    replace it by:
    A ~ C
    B ~ D
    (then repeat)

- Constraints:
  - ‘f ~ (‘x1 -> ‘y1)
  - ‘a ~ ‘x1
  - ‘fa ~ ‘y1
  - ‘ma ~ (‘x2 -> ‘y2)
  - int ~ ‘x2
  - ‘ma1 ~ ‘y2

  int ~ ‘x3
  (int -> int) ~ ‘y3
  ‘x1 ~ ‘x3
  (int -> ‘y2) ~ ‘y3
- Definition constraint
  ‘y1 ~ ‘y2

Phase 2: Solving constraints

- Step 1: pick any constraint.
  - If it is (of the form)
    ‘x ~ Y, or Y ~ ‘x
    just replace all ‘x for Y
    (provided Y does not
    contain ‘x) and deactivate
    the constraint

- Constraints:
  - ‘f ~ (‘x1 -> ‘y1)
  - ‘a ~ ‘x1
  - ‘fa ~ ‘y1
  - ‘ma ~ (‘x2 -> ‘y2)
  - int ~ ‘x2
  - ‘ma1 ~ ‘y2

  int ~ ‘x3
  (int -> int) ~ ‘y3
  ‘x1 ~ ‘x3
  (int -> ‘y2) ~ ‘y3
- Definition constraint
  ‘y1 ~ ‘y2

Phase 2: Solving constraints

- Step 1: pick any constraint.
  - If it is (of the form)
    ‘x ~ Y, or Y ~ ‘x
    just replace all ‘x for Y
    (provided Y does not
    contain ‘x) and deactivate
    the constraint

- Constraints:
  - ‘f ~ (‘x1 -> ‘y1)
  - ‘a ~ ‘x1
  - ‘fa ~ ‘y1
  - ‘ma ~ (‘x2 -> ‘y2)
  - int ~ ‘x2
  - ‘ma1 ~ ‘y2
  - int ~ ‘x3

  (int -> int) ~ ‘y3
  ‘x1 ~ int
  (int -> ‘y2) ~ ‘y3
- Definition constraint
  ‘y1 ~ ‘y2

Phase 2: Solving constraints

- Step 1: pick any constraint.
  - If it is (of the form)
    (A -> B) ~ (C -> D)
    replace it by:
    A ~ C
    B ~ D

- Constraints:
  - ‘f ~ (‘x1 -> ‘y1)
  - ‘a ~ ‘x1
  - ‘fa ~ ‘y1
  - ‘ma ~ (‘x2 -> ‘y2)
  - int ~ ‘x2
  - ‘ma1 ~ ‘y2
  - int ~ ‘x3
  - (int -> int) ~ ‘y3
  - ‘x1 ~ int

(int -> ‘y2) ~ (int -> int)

- Definition constraint
  ‘y1 ~ ‘y2

Phase 2: Solving constraints

- Step 1: pick any constraint.
  - If it is (of the form)
    ‘x ~ Y, or Y ~ ‘x
    just replace all ‘x for Y
    (provided Y does not contain ‘x) and deactivate the constraint

- Constraints:
  - ‘f ~ (‘x1 -> ‘y1)
  - ‘a ~ ‘x1
  - ‘fa ~ ‘y1
  - ‘ma ~ (‘x2 -> ‘y2)
  - int ~ ‘x2
  - ‘ma1 ~ ‘y2
  - int ~ ‘x3
  - (int -> int) ~ ‘y3
  - ‘x1 ~ int

  int ~ int
  ‘y2 ~ int
- Definition constraint
  ‘y1 ~ ‘y2

Phase 2: Solving constraints

- Step 1: pick any constraint.
  - If it is (of the form)
    int ~ int
    replace it by:
    (no constraints)

    (i.e.: throw it away)

  - Compare: If it is (of the form)
    (A -> B) ~ (C -> D)
    replace it by:
    A ~ C
    B ~ D

- Constraints:
  - ‘f ~ (‘x1 -> ‘y1)
  - ‘a ~ ‘x1
  - ‘fa ~ ‘y1
  - ‘ma ~ (‘x2 -> ‘y2)
  - int ~ ‘x2
  - ‘ma1 ~ ‘y2
  - int ~ ‘x3
  - (int -> int) ~ ‘y3
  - ‘x1 ~ int

  int ~ int
  ‘y2 ~ int
- Definition constraint
  ‘y1 ~ ‘y2

Phase 2: Solving constraints

- Constraints:
  - ‘f ~ (‘x1 -> ‘y1)
  - ‘a ~ ‘x1
  - ‘fa ~ ‘y1
  - ‘ma ~ (‘x2 -> ‘y2)
  - int ~ ‘x2
  - ‘ma1 ~ ‘y2
  - int ~ ‘x3
  - (int -> int) ~ ‘y3
  - ‘x1 ~ int
  - ‘x2 ~ int
  -

  ‘y2 ~ int
- Definition constraint
  ‘y1 ~ ‘y2

Phase 2: Solving constraints

- Step 2: repeat on the de-activated constraints in opposite order

  (note: we could've done this as we deactivated constraints, it's typically implemented that way)

- ‘f ~ (‘x1 -> ‘y1)
- ‘a ~ ‘x1
- ‘fa ~ ‘y1
- ‘ma ~ (‘x2 -> ‘y2)
- int ~ ‘x2
- ‘ma1 ~ ‘y2
- int ~ ‘x3
- (int -> int) ~ ‘y3
- ‘x1 ~ int
- ‘y2 ~ int
- ‘y1 ~ int

Phase 2: Solving constraints

- Step 2: repeat on the de-activated constraints in opposite order

- ‘f ~ (‘x1 -> int)
- ‘a ~ ‘x1
- ‘fa ~ int
- ‘ma ~ (‘x2 -> ‘y2)
- int ~ ‘x2
- ‘ma1 ~ ‘y2
- int ~ ‘x3
- (int -> int) ~ ‘y3
- ‘x1 ~ int
- ‘y2 ~ int
- ‘y1 ~ int

Phase 2: Solving constraints

- Step 2: repeat on the de-activated constraints in opposite order

- ‘f ~ (‘x1 -> int)
- ‘a ~ ‘x1
- ‘fa ~ int
- ‘ma ~ (int -> int)
- int ~ int
- ‘ma1 ~ int
- int ~ ‘x3
- (int -> int) ~ ‘y3
- ‘x1 ~ int
- ‘y2 ~ int
- ‘y1 ~ int

Phase 2: Solving constraints

- Step 2: repeat on the de-activated constraints in opposite order

- ‘f ~ (int -> int)
- ‘a ~ int
- ‘fa ~ int
- ‘ma ~ (int -> int)
- int ~ int
- ‘ma1 ~ int
- int ~ ‘x3
- (int -> int) ~ ‘y3
- ‘x1 ~ int
- ‘y2 ~ int
- ‘y1 ~ int

Phase 2: Solving constraints

- Step 2: repeat on the de-activated constraints in opposite order

- .. we are done, let's look at the program again

- 'f ~ (int -> int)
- 'a ~ int
- 'fa ~ int
- 'ma ~ (int -> int)
- int ~ int
- 'ma1 ~ int
- int ~ 'x3
- (int -> int) ~ 'y3
- 'x1 ~ int
- 'y2 ~ int
- 'y1 ~ int

Phase 2: Solving constraints

- Program:

- let f a
  = a - 1

- Subexpressions:
  f : ‘f
  a : ‘a
  f a : ‘fa
  (-) a 1 : ‘ma1
  (-) a : ‘ma
  - : int -> int -> int
  1 : int

- ‘f ~ (int -> int)
- ‘a ~ int
- ‘fa ~ int
- ‘ma ~ (int -> int)
- int ~ int
- ‘ma1 ~ int
- int ~ ‘x3
- (int -> int) ~ ‘y3
- ‘x1 ~ int
- ‘y2 ~ int
- ‘y1 ~ int

Phase 2: Solving constraints

- Program:
- let f a
  = a - 1

- Subexpressions:
  f : int -> int
  a : int
  f a : int
  (-) a 1 : int
  (-) a : int -> int
  - : int -> int -> int
  1 : int

- ‘f ~ (int -> int)
- ‘a ~ int
- ‘fa ~ int
- ‘ma ~ (int -> int)
- int ~ int
- ‘ma1 ~ int
- int ~ ‘x3
- (int -> int) ~ ‘y3
- ‘x1 ~ int
- ‘y2 ~ int
- ‘y1 ~ int

Want to try it on something hard?

- Program:
- let f a b c = c [a b; a c]
- (don't ask me what this program does)

- Note:
- (::) operator is used, a polymorphic operator
- its type is:
  'x -> 'x list -> 'x list
- Replace occurrences of 'x with 'x1, 'x2 etc (just like we did earlier).
- Similar for []
- replace "X list ~ Y list" by "X ~ Y"

(You can take some shortcuts by applying constraints as you find them)

What else does ocaml do?

- Program:
- let f a b c = c [a b; c]
- (not a valid program)

- Ocaml will give type errors if it gets stuck in the algorithm
- One way to get stuck is if two sides of the constraint don't match:
  int ~ X -> Y
- Another way to get stuck is if a variable occurs within the other side:
  'x ~ ('x -> 'y)
  (cannot replace 'x by 'x -> 'y)

More data-types

- Trees

Data type for trees

- Several flavors of trees:
  - Trees to store data:
    data on internal node / data on leaf
    sorted / balanced / expression-trees …
  - Trees to represent structures:
    expressions / logical formula / circuits …
- Key property that makes it a tree:
  two or more ‘recursions’

Data type for trees

- type int_tree = Node of (int_tree * int * int_tree) | Leaf
- let rec sum_tree t = match t with

Data type for trees

- type int_tree = Node of (int_tree * int * int_tree) | Leaf
- let rec sum_tree t = match t with
  | Node (t1, i, t2) ->
  | Leaf ->

Data type for trees

- type int_tree = Node of (int_tree * int * int_tree) | Leaf
- let rec sum_tree t = match t with
  | Node (t1, i, t2) ->
  | Leaf -> 0

Data type for trees

- type int_tree = Node of (int_tree * int * int_tree) | Leaf

- let rec sum_tree t = match t with
  | Node (t1, i, t2) -> i +
  | Leaf -> 0

Data type for trees

- type int_tree = Node of (int_tree * int * int_tree) | Leaf

- let rec sum_tree t = match t with
  | Node (t1, i, t2) -> i + sum_tree t1 + sum_tree t2
  | Leaf -> 0

Data type for trees

- type ‘a tree = Node of ('a tree * 'a * 'a tree) | Leaf

- let rec sum_tree t = match t with
  | Node (t1, i, t2) -> i + sum_tree t1 + sum_tree t2
  | Leaf -> 0

Data type for trees

- type ‘a tree = Node of ('a tree * 'a * 'a tree) | Leaf

- let rec sum_tree t = match t with
  | Node (t1, i, t2) -> i + sum_tree t1 + sum_tree t2
  | Leaf -> 0

- let rec count t = match t with
  | Node (t1, i, t2) -> 1 + count t1 + count t2
  | Leaf -> 0