CS 312
Type Checking and Type Inference

Radu Rugina
Fall 2007

Type Checking

- Type checking: check if expressions are well formed
- A program that type-checks will have no runtime type errors, regardless of inputs!
- Type-checking function:

```
(* Returns the type of exp in type environment E
 * Raises: TypeError if exp is not well-typed *)
fun tcheck(E:environment, exp:expression): typ
```

- Type environment E maps variables to their types
- tcheck traverses the AST of exp bottom-up
  - At each node, it returns the type of that node

Type Checking Implementation

```
fun tcheck(E: env, exp: expr): typ =
  case exp of
    Var(x) => Env.lookup(E, x)

  | True => Bool
  | False => Bool

  | IntConst(n) => Int

  | Plus(e1,e2) =>
      if tcheck(E,e1) = Int
         andalso tcheck(E,e2) = Int
      then Int
      else raise TypeError “Plus"
```


If-Then-Else

```
fun tcheck(E: env, exp: expr):typ =
  case exp of
    …
  | If(e1, e2, e3) =>
      let val t1 = tcheck(E, e1)
          val t2 = tcheck(E, e2)
          val t3 = tcheck(E, e3)
      in
        if t1 = Bool andalso t2 = t3
        then t2
        else raise TypeError “If"
      end
…
```


Functions

```
fun tcheck(E: env, exp: expr): typ =
  case exp of
    …
  | Fun(x,t,e) =>
      Arrow(t, tcheck(Env.update(E,x,t), e))

  | Apply(e1, e2) =>
      case (tcheck(E, e1), tcheck(E, e2)) of
      of (Arrow(t1, t2), t3) =>
          if t1 = t3 then t2
          else raise TypeError "Apply"
      | _ => raise TypeError "Apply"
 …
```


Type Checking vs Inference

- Type checking: check if e is well formed, return its type

```
(* Returns the type of exp in environment E
 * Raises: TypeError if exp is not well-typed *)
fun tcheck(E:environment, exp:expression): typ
```

- Type inference: given an expression e, compute the types of all variables in e. Use those types to determine the type of e and its sub-expression
- Goal: identify the most general (i.e. polymorphic) type of e and its sub-expressions

Type Inference

- General approach: constraint-based formulation
- Step 1: Assign fresh types (i.e., type variables)
- Step 2: Generate constraints between type variables by recursively walking the expression tree
  - Generate more fresh type variables
- Step 3: Solve constraints between type variables to infer the types

Example

```
fun map (f, l) =
   if null (l) then
      nil
   else
      cons (f (hd l), map (f, tl l)))
end
```

(An example taken from David Walker @ Princeton)

Step 1: Fresh Types for Variables

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil
   else
      cons (f (hd l), map (f, tl l)))
end
```


Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil
   else
      cons (f (hd l), map (f, tl l)))
end
```

null: b' list -> bool

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l : b' list) : bool then
      nil
   else
      cons (f (hd l), map (f, tl l)))
end
```

null: b' list -> bool

constraints
b = b' list

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil    : d list
   else
      cons (f (hd l), map (f, tl l)))
end
```

constraints
b = b' list

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil   : d list
   else
      cons (f (hd l), map (f, tl l)))
end
```

constraints
b = b' list

hd: b'' list -> b''

tl: b''' list -> b''' list

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil   : d list
   else
      cons (f (hd l :b''), map (f, tl l :b''' list)))
end
```

constraints
b = b' list
b = b'' list
b = b''' list

b = b'' list

b = b''' list

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil   : d list
   else
      cons (f (hd l :b''), map (f :a, tl l :b''' list)))
end
```

constraints
b = b' list
b = b'' list
b = b''' list

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil   : d
   else
      cons (f (hd l :b''), map (f :a, tl l :b''' list)))
end
```

constraints
b = b' list
b = b'' list
b = b''' list

f : a

map: a * b -> c

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil   : d
   else
      cons (f (hd l :b''), map (f :a, tl l :b''' list)))
end
```

constraints
b = b' list
b = b'' list
b = b''' list
a = b'' -> a'

f : a

map: a * b -> c

a = b'' -> a'

a=a
b = b''' list

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil   : d list
   else
      cons (f (hd l) :a', map (f, tl l) :c))
end
```

constraints
b = b' list
b = b'' list
b = b''' list
a = b'' -> a'

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil   : c'
   else
      cons (f (hd l) :a', map (f, tl l) :c))
end
```

cons: c' * c' list -> c' list

constraints
b = b' list
b = b'' list
b = b''' list
a = b'' -> a'

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil   : c'
   else
      cons (f (hd l) :a', map (f, tl l) :c)) :c' list
end
```

cons: c' * c' list -> c' list

a' = c'

c = c' list

constraints
b = b' list
b = b'' list
b = b''' list
a = b'' -> a'

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil   : d list
   else
      cons (f (hd l), map (f, tl l))) :c' list
end
```

constraints
b = b' list
b = b'' list
b = b''' list
a = b'' -> a'
a' = c'
c = c' list

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil   : d list
   else
      cons (f (hd l), map (f, tl l))) :c' list
end
```

d list = c' list

constraints
b = b' list
b = b'' list
b = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil
   else
      cons (f (hd l), map (f, tl l)))
   : d list
end
```

constraints
b = b' list
b = b'' list
b = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil
   else
      cons (f (hd l), map (f, tl l)))
   : d list
end
```

d list = c

constraints
b = b' list
b = b'' list
b = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list
d list = c

Step 2: Generate Constraints

```
fun map (f : a, l : b) : c =
   if null (l) then
      nil
   else
      cons (f (hd l), map (f, tl l)))
end
```

constraints
b = b' list
b = b'' list
b = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list
d list = c

Step 3: Solve Constraints

- Constraint solution provides all possible solutions to type annotations on terms

constraints
b = b' list
b = b'' list
b = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list
d list = c

solution
a = b' -> c'
b = b' list
c = c' list

```
map (f : 'b->'c
        x : 'b list)
: 'c list
=
 ...
end
```


Solving Constraints

- Iterative constraint solving using substitutions (Martelli-Montanari algorithm): compute a map from type variables to most general types
  - Iterate through the constraints
  - Replace t1->t2 = t3->t4 by t1 = t3, t2 = t4
  - Replace list t1 = list t2 by t1 = t2
  - If t1 = t2 where t1 and t2 are incompatible types (e.g. t1 is is a function type and t2 is list type) then Type Error
  - Constraint a = t, where a is a type variable
    - If a occurs in t then "Type Error"
    - If t is a then discard the constraint
    - Otherwise, add a → t to the mapping and substitute t for a in all other constraints

Example

Constraints

b = b' list
b = b'' list
b = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list
d list = c

Mappings

Example

Constraints

b = b' list
b' list = b'' list
b' list = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list
d list = c

Mappings

b → b' list

Example

Constraints

b' list = b'' list
b' list = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list
d list = c

Mappings

b → b' list

Example

Constraints

b' = b''
b'' list = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list
d list = c

Mappings

b → b' list
b' → b''

(the process continues...)

Example

Constraints

Solution

b → b' list
b' → b''
b'' → b'''
a → b''' -> a'
a' → c'
c → c' list
d → c'

Example

- The mapping essentially forms a DAG (directed acyclic graph)
- To find the type for a variable, traverse the DAG
- The types a, b, c can be expressed in terms of b''' and c':

  a = b''' -> c'
  b = b''' list
  c = c' list

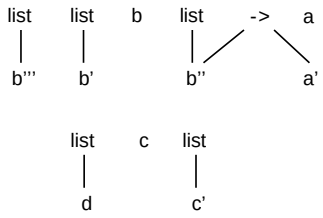
Solution

b → b' list
b' → b''
b'' → b'''
a → b''' -> a'
a' → c'
c → c' list
d → c'

Unifications

- Another way of solving constraints: via unifications
- Consider a graph where each node is either a type variable, or a function type, or a list type.
- Function types have two child types; list types have one child.
- Solve the constraint t = t' by unifying the types of t and t' in the graph
- Recursive unifications: unify(t1->t2, t3->t4) implies unify(t1,t3) and unify(t3,t4)

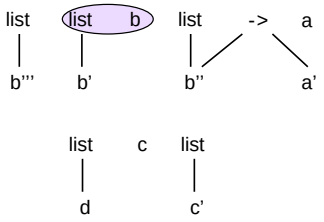
Example



Constraints

b = b' list
b = b'' list
b = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list
d list = c

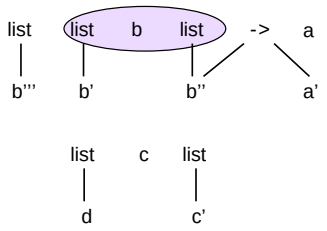
Example



Constraints

b = b' list
b = b'' list
b = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list
d list = c

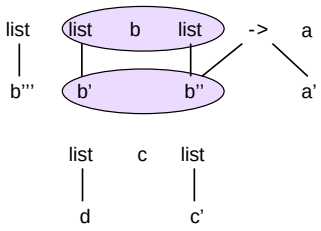
Example



Constraints

b = b' list
b = b'' list
b = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list
d list = c

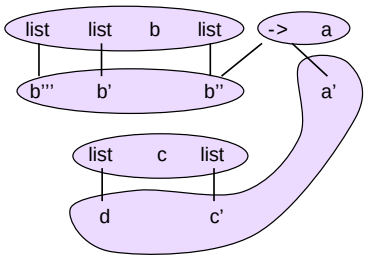
Example



Constraints

b = b' list
b = b'' list
b = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list
d list = c

Solution



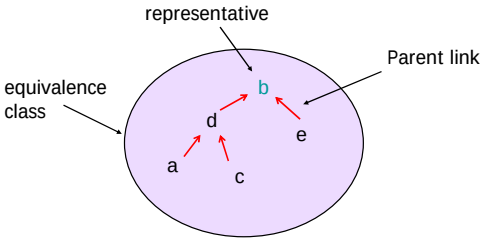
Constraints

b = b' list
b = b'' list
b = b''' list
a = b'' -> a'
a' = c'
c = c' list
d list = c' list
d list = c

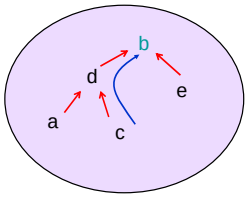
Union-Find Data Structure

- Unifications can be implemented efficiently using union-find data structures.
- Models equivalence classes:
  - Each class has a representative
  - Each node has a parent; the node without a parent is the representative
- Supports two operations:
  - Union: merges together two equivalence classes
  - Find: lookup the representative of a class.

Union-Find Data Structure



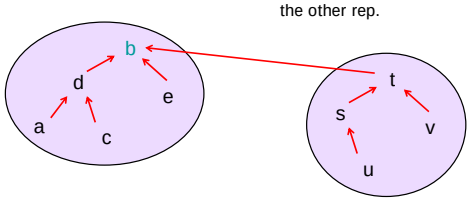
Find Operation



Find operation: traverse parent pointers up

Union Operation: O(1)

Union operation:
Choose one of the reps
Make it the parent of
the other rep.



Union-Find Data Structure