Type Checking vs Inference

- Type checking: given an expression \( e \), and a type environment \( E \) for the free variables in \( e \), check if \( e \) is well formed and return its type

\[(\text*{Returns the type of exp in environment E}\)*
\[
\text{raises: TypeError if exp is not well-typed}\)
\]

\[
\text{fun type_check(E:environment, exp:expression): typ}
\]

- 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 \( e \)

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

\[
\text{fun map (f, l) =}
\]
\[
\text{if null (l) then}
\]
\[
\text{nil}
\]
\[
\text{else}
\]
\[
\text{cons (f (hd l), map (f, tl l))}
\]
\[
\text{end}
\]

(An example taken from David Walker @ Princeton)

Step 1: Fresh Types for Variables

\[
\text{fun map (f : a, l : b) : c =}
\]
\[
\text{if null (l) then}
\]
\[
\text{nil}
\]
\[
\text{else}
\]
\[
\text{cons (f (hd l), map (f, tl l))}
\]
\[
\text{end}
\]

Step 2: Generate Constraints

\[
\text{fun map (f : a, l : b) : c =}
\]
\[
\text{if null (l) then}
\]
\[
\text{nil}
\]
\[
\text{else}
\]
\[
\text{null: b list -> bool}
\]
\[
\text{cons (f (hd l), map (f, tl l))}
\]
\[
\text{end}
\]
Step 2: Generate Constraints

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

Step 2: Generate Constraints

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

Step 2: Generate Constraints

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

Step 2: Generate Constraints

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

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

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

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
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

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

Step 3: Solve Constraints

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

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

Mappings

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

Constraints

\[ b = b' list \]
\[ b' list = b'' list \]
\[ b'' list = b''' list \]
\[ a = b'' -> a' \]
\[ a' = c' \]
\[ c = c' list \]
\[ d list = c' list \]
\[ d list = c \]

Mappings

\[ b -> b' list \]
\[ b' list = b'' list \]
\[ b'' list = b''' list \]
\[ a = b'' -> a' \]
\[ a' = c' \]
\[ c = c' list \]
\[ d list = c' list \]
\[ d list = c \]

Example

Constraints

\[ b' = b'' \]
\[ b'' list = b''' list \]
\[ a = b'' -> a' \]
\[ a' = c' \]
\[ c = c' list \]
\[ d list = c' list \]
\[ d list = c \]

Mappings

\[ b -> b' list \]
\[ b' list = b'' list \]
\[ b'' list = b''' list \]
\[ a = b'' -> a' \]
\[ a' = c' \]
\[ c = c' list \]
\[ d list = c' list \]
\[ d list = c \]

Example

Constraints

\[ b = b' list \]
\[ b' list = b'' list \]
\[ b'' list = b''' list \]
\[ a = b'' -> a' \]
\[ a' = c' \]
\[ c = c' list \]
\[ d list = c' list \]
\[ d list = c \]

Solution

\[ b -> b' list \]
\[ b' list = b'' list \]
\[ b'' list = b''' list \]
\[ a = b'' -> a' \]
\[ a' = c' \]
\[ c = c' list \]
\[ d list = c' list \]
\[ d list = c \]

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'

Solution

\[ b -> b' list \]
\[ b' list = b'' list \]
\[ b'' list = b''' list \]
\[ a = b'' -> a' \]
\[ a' = c' \]
\[ c = c' list \]
\[ d list = c' list \]
\[ d -> c' \]

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: \( \text{unify}(t_1 -> t_2, t_3 -> t_4) \) implies \( \text{unify}(t_1, t_3) \) and \( \text{unify}(t_3, t_4) \)
Example

```
Constraints
b = b' list
b = b'' list
a = b'' -> a'

list c list

list = c
```

Example

```
Constraints
b = b' list
b = b'' list
a = b'' -> a'

list c list
d list = c
```

Solution

```
Constraints
b = b' list
b = b'' list
a = b'' -> a'

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

Union Operation: O(1)

Union-Find Data Structure