CSE216 Programming Abstractions

Type Inference

YoungMin Kwon

Type Inference

- Type inference
  - Determining the type of an expression
- HM Type Inference algorithm
  - Assign type variables to subexpressions
  - Build type constraints using the type variables
  - Find a unifier that solves the type constraints
- TODO
  - Download reci_tiny_type_inter.zip and implement constr and unifier functions

Type Inference

```
(*constr: type constraints for kvar = expr
    kvar: type variable for expr
    expr: expression (kvar = expr)
    env : variable name to kind variable map
    return: list of constraints (kexpr = kexpr)...
*)
let rec constr kvar kexpr env =
…
    match kexpr with
    (*TODO: make constraints for literals and variables*)
    | BOOL b -> ret [(kvar, KB)]
    | NUM n  ->
    | VAR v  ->
```

Type Inference

```
(*constr: type constraints for kvar = expr
    kvar: type variable for expr
    expr: expression (kvar = expr)
    env : variable name to kind variable map
    return: list of constraints (kexpr = kexpr)...
*)
let rec constr kvar kexpr env =
…
    match kexpr with
    (*TODO: make constraints for literals and variables*)
    | BOOL b -> ret [(kvar, KB)]
    | NUM n  -> ret [(kvar, KN)]
    | VAR v  -> ret [(kvar, lookup v env)]
```

Type Inference

```
(*TODO: make constraints for primitive operators*)
| ADD (a, b) | SUB (a, b) ->
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->
    constr kv1 a env >>= fun c1 ->
    constr kv2 b env >>= fun c2 ->
    (kvar, KN)::(kv1, KN)::(kv2, KN)::c2@c1 |> ret

| EQ (a, b) | GE (a, b) ->
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->

| AND (a, b) | OR (a, b) ->
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->

| NOT a ->
    newvar ()          >>= fun kv1 ->
```

```
(*TODO: make constraints for primitive operators*)
| ADD (a, b) | SUB (a, b) ->
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->
    constr kv1 a env >>= fun c1 ->
    constr kv2 b env >>= fun c2 ->
    (kvar, KN)::(kv1, KN)::(kv2, KN)::c2@c1 |> ret

| EQ (a, b) | GE (a, b) ->
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->
    constr kv1 a env >>= fun c1 ->
    constr kv2 b env >>= fun c2 ->
    (kvar, KB)::(kv1, KN)::(kv2, KN)::c2@c1 |> ret

| AND (a, b) | OR (a, b) ->
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->
    constr kv1 a env >>= fun c1 ->
    constr kv2 b env >>= fun c2 ->
    (kvar, KB)::(kv1, KB)::(kv2, KB)::c2@c1 |> ret
| NOT a ->
    newvar ()          >>= fun kv1 ->
    constr kv1 a env >>= fun c1 ->
    (kvar, KB)::(kv1, KB)::c1 |> ret
```

Type Inference

```
(*TODO: make constraints for conditional expressions*)
| IF (c, t, f) ->
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->
    newvar ()          >>= fun kv3 ->
```

Type Inference

```
(*TODO: make constraints for conditional expressions*)
| IF (c, t, f) ->
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->
    newvar ()          >>= fun kv3 ->
    constr kv1 c env >>= fun c1 ->
    constr kv2 t env >>= fun c2 ->
    constr kv3 f env >>= fun c3 ->
    (kvar, kv2)::(kv1, KB)::(kv2, kv3)::c3@c2@c1 |> ret
```

Type Inference

```
(*TODO: make constraints for function definitions*)
| FUN (v, e) ->
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->
```

Type Inference

```
(*TODO: make constraints for function definitions*)
| FUN (v, e) ->
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->
    (v, kv1)::env      |>  fun ev ->
    constr kv2 e ev    >>= fun c1 ->
    (kvar, KF (kv1, kv2))::c1 |> ret
```

Type Inference

```
(*TODO: make constraints for function applications*)
| APP (f, a) ->
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->
```

Type Inference

```
(*TODO: make constraints for function applications*)
| APP (f, a) ->
    newvar ()          >>= fun kv1 ->
    newvar ()          >>= fun kv2 ->
    constr kv1 f env >>= fun c1 ->
    constr kv2 a env >>= fun c2 ->
    (kv1, KF (kv2, kvar))::c2@c1 |> ret
```

Type Inference

```
let rec unifier clist =
    match clist with
    | [] -> fun x -> x (*id substitution: no substitution*)
    | hd::tl ->
        match hd with
        | (KV v, ke) | (ke, KV v) ->
            if contains (KV v) ke
            then assert false (*recursive def is not supported*)
            else
                (*TODO: make a substitution for h*)
                let sub_h =                                          in
                let sub_t = tl
                        (*TODO: apply sub_h to the remaining constraints*)
                        |> List.map (fun (a, b) ->                   )
                        |> unifier in
                (*unifier is the composed substitutions*)
                fun e -> e |> sub_h |> sub_t

        (*TODO: unify function definitions*)
        | (KF (a, b), KF (c, d)) ->                                  |> unifier
…
```

Type Inference

```
let rec unifier clist =
    match clist with
    | [] -> fun x -> x (*id substitution: no substitution*)
    | hd::tl ->
        match hd with
        | (KV v, ke) | (ke, KV v) ->
            if contains (KV v) ke
            then assert false (*recursive def is not supported*)
            else
                let sub_h = substitution (KV v) ke in
                let sub_t = tl
                        |> List.map (fun (a, b) -> (sub_h a, sub_h b))
                        |> unifier in
                (*unifier is the composed substitutions*)
                fun e -> e |> sub_h |> sub_t

        | (KF (a, b), KF (c, d)) -> (a, c)::(b, d)::tl |> unifier
                                |> unifier
…
```