CSE216 Programming Abstractions

Type Systems

YoungMin Kwon

Why Types

- Types provide implicit context
  - a + b: integer addition or floating point addition
  - new: allocate memory and call proper constructor
- Types limit the set of permitted operations
  - Reduce mistakes in programming
    - E.g.: Prevent adding a number to a Boolean (1 + true)
- With types, programs are easier to read

```
(*what if I knew the type of delay, curr, and stream…*)
let rec after_delay delay curr stream =
```

- Help compilers optimizing performances

Type Systems

- A type system consists of
  - Mechanism to
    - Define types
    - Associate them with language constructs
  - A set of rules for
    - Type equivalence
    - Type compatibility
    - Type inference

Type Checking

- Type checking
  - Process of ensuring that a program obeys the language's type compatibility rules
- Strongly typed language
  - Prohibits the application of any unsupported operations to objects
  - Static type checking: type checking is performed at compile time
  - Dynamic type checking: type checking is performed at run-time

What is Type

- Denotational point of view
  - A set of values known as a domain
    - E.g.: {1, 2, 3, …}, {‘a’, ‘b’, ‘c’, …}, {true, false},…
  - Types are domains and the meaning of an expression is a value from the domain
  - Programmers may think user defined types as mathematical operations on sets
    - E.g.: Cartesian products for tuples

What is Type

- Structural point of view
  - A type is either
    - A primitive type (int, char, boolean, …) or
    - A composite type (tuple, record, list…)
  - Programmers may think in terms of the way it is built from simpler types
- Abstraction-based point of view
  - A type is an interface consisting of a set of operations
  - Programmers may think in terms of its meaning or purpose

Polymorphism

- Parametric polymorphism
  - Code takes a type as a parameter
  - Generics or templates in Java, C++
- Subtype polymorphism
  - A code designed to work with type T also works with T's subtypes
  - Most object oriented languages
- Combination of subtype and parametric polymorphism
  - Useful for containers: List<T> or Stack<T>

Type Checking

- Type equivalence
  - Whether two types are the same
- Type compatibility
  - When an object of a type can be used in a certain context
  - Type conversion, type coercion, non-converting type cast
- Type inference
  - Given the types of the subexpressions, what is the type of the expression as a whole?

Type Equivalence

- Type equivalence
  - Whether two types are the same
- Structural equivalence
  - Two types are the same if they consists of the same components in the same way
  - E.g.: Algol-68, Modula-3, C, ML
- Name equivalence
  - Lexical occurrence of type definitions
  - E.g.: Java, C#

Type Equivalence

```
type R1 = record
    a, b: integer
end
```

```
type R2 = record
    a: integer;
    b: integer
end
```

```
type R3 = record
    b: integer;
    a: integer
end
```

- Example
  - In many languages R1 and R2 are structurally equivalent
  - In many languages R3 is not equivalent to R1 or R2.

Type Equivalence

```
type student = record
    name, address: string
    id: integer
end

x: student;
y: school;
…
x := y
```

```
type school = record
    name, address: string
    id: integer
end
```

- Problem with structural equivalence
  - Cannot distinguish types that the programmer may think of as different
- Name equivalence
  - If the programmer distinguishes the types, they are probably meant to be different

Type Conversion and Casts

- In a program, values of specific types are expected

```
a := expression
```

expression should have the same type as a

```
a + b
```

a and b are both integers or they are both floats

```
foo(v1: type1, v2: type2)
…
foo(expr1, expr2)
```

expr1 should be type1 and expr2 should be type2

Type Conversion and Casts

- Type conversion cases
  - Types are structurally equivalent, but the language uses name equivalence
    - Conversion is purely conceptual
  - Types have different sets of values, but the intersecting values are represented in the same way
    - Signed int $\leftrightarrow$ unsigned int
    - Runtime check: If the current value is in the intersect, use it ($1 \rightarrow 1$). If not, runtime error ($-1 \rightarrow ?$).

Type Conversion and Casts

- Type conversion cases (Cont'd)
  - Types have different representations, but some correspondence can be defined among their values
    - int $\leftrightarrow$ float
    - Machine instructions for the conversion

Non-converting Type Casts

- Particularly, in systems programming
  - Change the type of a value without changing the underlying implementation
  - E.g. malloc
    - Represent heap as a large array of bytes
    - Reinterpret portions of the memory as pointers and integers

Type Compatibility

- Most language requires type compatibility rather than type equality
  - a + b: a and b must be compatible with some type that supports addition
  - foo(a, b): a and b must be compatible with the formal parameters (subtypes)

Type Compatibility

- Type compatibility varies from language to language
- In Ada, type S is compatible with type T if
  - S and T are equivalent
  - One is a subtype of the other or both are subtypes of the same base type
  - Both are arrays with the same number and type of elements

Coercion

- Type coercion
  - When necessary, a language performs an automatic, implicit conversion to the expected type
- Coercion is a controversial subject
  - Type conversion without programmer's explicit cast → it can weaken type security
  - Natural way to support abstraction and extensibility → easy to use new types with existing ones

Type Checking in SPL

```
type kind = Boolean
          | Number
          | Function of kind * kind
          | Error;;

type expr = B of bool   (*Boolean*)
          | N of int    (*number*)
          | V of string (*variable*)
          (*arithmetic exprs*)
          | Add of expr * expr | Sub of expr * expr
          (*predicates*)
          | Equ of expr * expr | Leq of expr * expr
          (*logical exprs*)
          | And of expr * expr | Or  of expr * expr | Not of expr
          (*conditional expr*)
          | If  of expr * expr * expr
          (*function definition: parameter, body*)
          | Fun of (kind * string) * expr
          (*function application: operator, operand*)
          | App of expr * expr;;
```

Sub (N 1, B true) is a valid expr, but it is an unsupported opr.

```
(*env has name-kind bindings*)
let rec lookup name env =
    match env with
    | [] -> Error
    | (k, n)::rest -> if name = n
                      then k
                      else lookup name rest in

(*kind returns the kind of expr in env*)
let rec kind expr env =
    match expr with
    | B b -> Boolean
    | N n -> Number
    | V v -> lookup v env
    | Add (e1, e2) | Sub (e1, e2) ->
        if kind e1 env = Number && kind e2 env = Number
        then Number
        else Error
    | Equ (e1, e2) | Leq (e1, e2) ->
        if kind e1 env = Number && kind e2 env = Number
        then Boolean
        else Error
    | Not e ->
        if kind e env = Boolean
        then Boolean
        else Error
```

```
| And (e1, e2) | Or (e1, e2) ->
    if kind e1 env = Boolean && kind e2 env = Boolean
    then Boolean
    else Error
| If (e1, e2, e3) ->
    let t2 = kind e2 env in
    let t3 = kind e3 env in
    if kind e1 env = Boolean && t2 = t3
    then t2
    else Error

| Fun ((k, v), e) -> (*not checking the return type*)
    let t = kind e ((k, v)::env) in (*kind of e in extended env*)
    if t != Error
    then Function (k, t)
    else Error
| App (e1, e2) -> match kind e1 env with
    | Function (tp, tb) ->
        let t2 = kind e2 env in
        if tp = t2 && tp != Error
        then tb
        else Error
    | _ -> Error in
```

Type Inference

- Type inference
  - Determining the type of an expression
  - Examples
    - The result of an arithmetic operator usually has the same type as the operand
    - The result of a comparison is Boolean
    - The result of a function call is the type of the function body
    - The result of an assignment has the same type as the left-side

HM Type Inference Algorithm

HM: Hindley and Milner

- Type inference example

```
# let inc = fun x -> (+) 1 x;;
val inc : int -> int = <fun>
```

- Step 1: assign preliminary types
  - Assign type variables for unknown types

Subexpression	Preliminary type
`fun x -> (+) 1 x`	A *(whole expr)*
`x`	B *(param)*
`(+) 1 x`	C *(function body)*
`(+) 1`	D *(sub-expr)*
`(+)`	`int -> (int -> int)`
`1`	`int`
`x`	E *(sub-expr)*

HM Type Inference Algorithm

- Step 2: collect type constraints

Subexpression	Preliminary type	Constraints
`fun x -> (+) 1 x`	`A`	`A = B -> C`
`x`	`B`	
`(+) 1 x`	`C`	
`(+) 1`	`D`	`D = E -> C`
`(+)`	`int -> (int -> int)`	`int -> D = int -> (int -> int)`
`1`	`int`	
`x`	`E`	`E = B`

HM Type Inference Algorithm

- Step 3: solve type constraints
  - Find a type assignment to type variables that can satisfy all type constraints

  - `D = int -> int`
  - `E = int, C = int`
  - `B = int`
  - `A = int -> int`

Constraints
`A = B -> C`
`D = E -> C`
`int -> D = int -> (int -> int)`
`E = B`

HM Type Inference Algorithm

- Type Constraint collection
  - Assign a fresh type variable to
    - Each function parameter
    - Let D(x) be the type variable for a function parameter x
    - Each subexpression of an expression
    - Let U(e) be the type variable for a subexpression e

Type Constraints

- Generate type constraints
  - For a constant c: `U(c) = type of c`
    - E.g. Let U(1) be A, then A = int
  - For a variable x: `U(x) = D(x)`
    - E.g. Let D(x) be A, U(x) be B, then A = B
  - For function application e1 e2:
    `U(e1) = U(e2) -> U(e1 e2)`
    - E.g. Let U(f x) be A, U(f) be B, U(x) be C, then B = C -> A
  - For a function definition fun x -> e:
    `U(fun x -> e) = D(x) -> U(e)`
    - E.g. Let U(fun x -> y) be A, D(x) be B, U(y) be C, then A = B -> C

...

```
(*kind expression*)
type kexpr = KB | KN
           | KV of int (*kind variable: k0, k1, k2, …*)
           | KF of kexpr * kexpr (*ke1 -> ke2*)

(*type constraints for kvar = expr
    kvar: type variable for expr
    expr: expression (kvar = expr)
    env : variable to type variable map
    return: list of constraints (kexpr = kexpr)...
*)
let rec constr kvar expr env =
    let open State in
    match expr with
    | N n -> ret [(KV kvar, KN)]
    | Add (e1, e2) | Sub (e1, e2) ->
        newvar() >>= fun v1 ->
        newvar() >>= fun v2 ->
        constr v1 e1 env >>= fun c1 ->
        constr v2 e2 env >>= fun c2 ->
        ret ((KV kvar, KN)::(KV v1, KN)::(KV v2, KN)::c2@c1)
…
```

Unification

- How to solve type constraints
  - Substitution: substitute a type variable in a type expr with an associated type expr

```
(*substitute kvar in ke with kexp*)
let subst kvar kexp =
    let rec subst' ke =
        match ke with
        | KB -> KB
        | KN -> KN
        | KV kv -> if kv = kvar then kexp else KV kv
        | KF (ke1, ke2) -> KF (subst' ke1, subst' ke2) in
    subst' in
```

  - A composition of substitutions is a substitution

```
fun ke -> subs2 (subs1 ke)
```

Unification

- Unifier
  - A substitution U is a unifier of a constraint e1 = e2 if (U e1) = (U e2)
- HM type inference algorithm
  - Given an expression, generate a set C of type constraints
  - Find a unifier U that unifies all constraints in C

Unification

```
(*find a unifier for the constraints in cl*)
let rec unify cl =
    match cl with
    | [] -> fun x -> x (*id: no substitution*)
    | hd::tl ->
        match hd with
        | (KV kv, ke) ->
            if contains kv ke
            then assert false (*recursive def is not supported*)
            else
                let s = subst kv ke in
                let u = unify (List.map (fun (a,b) -> (s a, s b))
                                        tl) in
                fun e -> u (s e) (*return the composite substitution*)
        | (ke, KV kv) -> unify ((KV kv, ke)::tl) (*switch the order*)
        | (KF (a,b), KF (c,d)) -> unify ( (a,c)::(b,d)::tl )
        | (a, b) -> if a = b
                    then unify tl         (*hd is already unified*)
                    else assert false in (*cannot unify*)
```

Assignment 8

- In this assignment, you are required to implement a constraint generator of HM type inference algorithm
  - Download spl_infer.ml
  - Implement constr function
  - Due date 6/2/2020

Assignment 8: Test Results

```
let test1 () =
    let e1 = If (Leq (Add (N 1, V "x"), Sub (N 2, V "x")),
                 B true,
                 B false) in
    let e2 = Fun ("x", e1) in
    let e3 = Fun ("y", e2) in
    let e4 = App (e3, B true) in
    let e5 = App (e4, N 3) in
    let e6 = App (e4, B true) in
    [
        unifier (constraints e2) (KV 0);
        unifier (constraints e3) (KV 0);
        unifier (constraints e4) (KV 0);
        unifier (constraints e5) (KV 0);
      (*unifier (constraints e1) (KV 0);(*assert false*)*)
      (*unifier (constraints e6) (KV 0);(*assert false*)*)
    ]
```

Assignment 8: Test Results

```
let test2 () =
    let open SyntaticSugar in
    let max = "x" @ "y" @
                If (!"x" <= !"y", !"y", !"x") in

    let c = constraints max in
    let s = unifier c in
    s (KV 0)

(*
expected results
- : kexpr list = [KF (KN, KB);
                  KF (KV 1, KF (KN, KB));
                  KF (KN, KB);
                  KB]
- : kexpr = KF (KN, KF (KN, KN))
*)
let _ = test1 ()
let _ = test2 ()
```