Type Judgments และ Type Rules

คำศัพท์ที่จะใช้

- *Type judgment*: การตัดสินความถูกต้องของ type สำหรับ expression หรือ statement ในโปรแกรม
  - เป็นบทสรุป (conclusion) ของการทำ type checking
- *Type rules*: กฎเกณฑ์ที่จะใช้ในการทำ type judgment
  - เขียนโดยใช้ inference rules
  - กำหนดโดยผู้ออกแบบภาษาโปรแกรม
- *Context* หรือ *environment*: ข้อมูลการ binding ของวัตถุในโปรแกรมเช่นตัวแปร ฟังก์ชั่น หรือ type
  - Symbol table เป็นตัวเก็บข้อมูลพวกนี้

Typing Judgment (การตัดสิน type)

- E : T อ่านว่า E เป็น expression ที่มี type T
- ตัวอย่างของ type judgment
  - 2 : int
  - true : bool
  - 2 * (3 + 4) : int
  - “Hello” : string

Type Judgment สำหรับ Expression

- พิจารณา (b ? 2 : 3) : int
- จะตัดสินได้ว่า expression นี้มี type เป็น int จริงจะต้องได้ว่า:
  - b : bool
  - 2 : int
  - 3 : int

Notation (สัญลักษณ์)

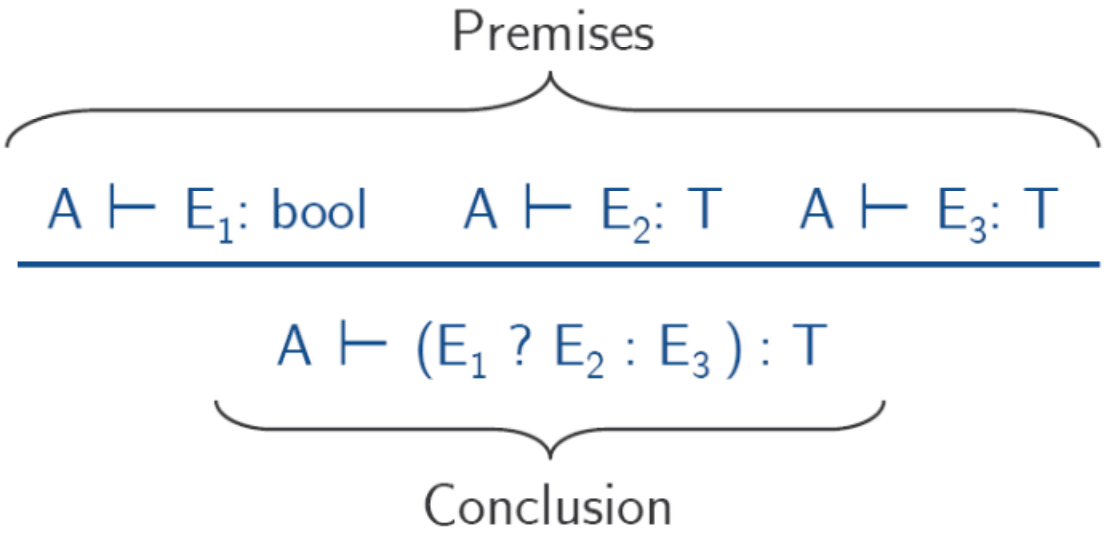
- A ⊢ E : T อ่านว่า “ใน context ของ A สามารถพิสูจน์ได้ว่า E เป็น expression ที่มี type T”
- Context ของ type เป็น set ของการทำ binding
  - โครงสร้างอะไรในคอมไพเลอร์ที่เก็บข้อมูลนี้?
    - Symbol table
- ตัวอย่างการใช้งาน
  - b : bool, x : int ⊢ b : bool
  - b : bool, x : int ⊢ (b ? 2 : x) : int

ขั้นตอนการทำ type judgment

- ใช้ inference rules เป็นหลัก
- ต้องการข้อสรุป (conclusion) ว่า:
  - b : bool, x : int ⊢ (b ? 2 : x) : int
- ต้องให้ได้ว่าสมมุติฐาน (hypothesis หรือ premises) ต่อไปนี้เป็นจริง
  - b : bool, x : int ⊢ b : bool
  - b : bool, x : int ⊢ 2 : int
  - b : bool, x : int ⊢ x : int

กฎทั่วไปของ expression ตัวอย่าง

- $A \vdash (E_1 \ ? \ E_2 : E_3) : T$ เป็นจริงเมื่อ
  - $A \vdash E_1 : bool$
  - $A \vdash E_2 : T$
  - $A \vdash E_3 : T$
- เขียนโดยใช้ notation ของ inference rule:



การพิสูจน์ expression type

- การพิสูจน์ว่า expression E มี type เป็น T (E : T) ทำได้โดยการสร้าง proof tree แสดงการทำ type derivation เพื่อตัดสิน E : T (type judgment)
- ตัวอย่างของ proof tree:

$$\dfrac{\dfrac{A \vdash \texttt{b: bool}}{A \vdash \texttt{!b: bool}} \quad \dfrac{A \vdash \texttt{2: int} \quad A \vdash \texttt{3: int}}{A \vdash \texttt{2+3 : int}} \quad A \vdash \texttt{x : int}}{\texttt{b: bool, x: int} \vdash \texttt{(!b ? 2+3 : x) : int}}$$

Type Rules

- Axiom (ไม่มี premises)

$$\frac{}{A \vdash \texttt{true} : \text{bool}}$$

- การบวก expression

$$\frac{A \vdash E_1 : \text{float} \quad A \vdash E_2 : \text{float}}{A \vdash E_1 + E_2 : \text{float}} \qquad \frac{A \vdash E_1 : \text{int} \quad A \vdash E_2 : \text{int}}{A \vdash E_1 + E_2 : \text{int}}$$

Rule สำหรับ while statement

- ให้ default type ของ statement S ทุกๆชนิดเป็น void
- กฎของ while เป็นดังนี้

$$\frac{A \vdash E : \text{bool} \qquad A \vdash S}{A \vdash \texttt{while}\ (E)\ S}\ (\text{while})$$

Assignment Statements

$$\frac{\texttt{id} : T \in A \qquad A \vdash E : T}{A \vdash \texttt{id} = E}\ \text{(variable-assign)}$$

$$\frac{A \vdash E_3 : T \qquad A \vdash E_2 : \text{int} \qquad A \vdash E_1 : \text{array}(T)}{A \vdash E_1[E_2] = E_3}\ \text{(array-assign)}$$

โปรแกรมแบบ straight-line

- กฎ: ถ้าประโยคแรกมี type ถูกต้องและประโยคที่เหลือมี type ที่ถูกต้องแล้ว ลำดับของประโยคทั้งหมดที่เกี่ยวข้องมี type ที่ถูกต้องด้วย

$$\frac{A \vdash S_1 \qquad A \vdash (S_2 ; \ldots ; S_n)}{A \vdash (S_1 ; S_2 ; \ldots ; S_n)}\ (\text{sequence})$$

Declarations (การประกาศ)

- Declarations เพิ่ม binding เข้าไปใน environment หรือ context
- เพิ่ม id และ type ที่เกี่ยวข้องเข้าไปใน symbol table

$$\frac{A \vdash \mathtt{T\ id}\ [\ =\ \mathtt{E}\ ] \qquad A, \mathtt{id} : \mathtt{T} \vdash (S_2\ ;\ \ldots\ ;\ S_n)}{A \vdash (\mathtt{T\ id}\ [\ =\ \mathtt{E}\ ];\ S_2\ ;\ \ldots\ ;\ S_n)}\ (\text{declaration})$$

Function Calls

- ถ้า E เป็น function call expression มันจะมี type ดังต่อไปนี้: $T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r$
- เราจะ type check function call $E(E_1, E_2, \ldots E_n)$ โดยใช้กฎต่อไปนี้

$$\frac{A \vdash E : T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r \qquad A \vdash E_i : T_i \quad (i \in 1..n)}{A \vdash E(E_1, \ldots, E_n) : T_r} \text{(function-call)}$$

Function Declarations

- จะต้องมีการใส่ **arguments** เข้าไปใน **symbol table**

$$A, a_1 : T_1, \ldots, a_n : T_n \vdash E : T_r$$

- จากนั้นทำ **type check** บอกว่า **declaration** นี้ใช้ได้

$$T_r \ \text{fun}\ (T_1\ a_1, \ldots, T_n\ a_n)\ \{\ \text{return}\ E;\ \}$$

Recursive Function

```
int fact(int x)  {
    if (x==0) return 1;
    else return x * fact(x - 1);
 }
```

- Type check ด้วยการพิสูจน์ว่า: $A \vdash$ `x * fact(x-1) : int`
- โดย context A ต้องมีสมาชิกต่อไปนี้: $A =$ { `fact: int`$\rightarrow$`int, x : int` }
- ต้องมี binding ต่อไปนี้อยู่ใน context:

$$\text{fun: } T_1 \times T_2 \times \ldots \times T_n \rightarrow T_r \in A$$

Return

- Statement นี้ไม่ได้ให้ค่าเพื่อใช้ใน context ปัจจุบัน
- เราจะ type check return ด้วยการเพิ่ม ret : T_r เข้าไปใน symbol table
- เริ่มจาก declaration:

$$T_r \;\; \text{fun}\ (T_1\ a_1, \ldots,\ T_n\ a_n) \;\; \{\ \text{return E;}\ \}$$

- กฎการ type check เพื่อให้แน่ใจว่า expression ที่ return E มี type ตามที่ declaration กำหนดภายใต้ context A:

$$A,\ a_1 : T_1,\ \ldots,\ a_n : T_n\ ,\ \texttt{ret} : T_r \vdash S : T_r$$

$$\frac{A \vdash E : T \qquad \texttt{ret} : T \in A}{A \vdash \texttt{return}\ E}\ (\text{return})$$

Type Checking over AST

Type Checking

- Traverse AST โดยใช้ recursive function
- Function ที่ traverse AST จะตรวจสอบ expression ที่จะ type check
- หนึ่งใน argument ของ function นี้คือ context หรือ environment ที่ให้ข้อมูลเพิ่มเติมเกี่ยวกับ expression ที่จะ type check
  - Symbol table เก็บข้อมูล context
- Function นี้จะ return ข้อมูลเกี่ยวกับ type ของ expression ที่กำลังพิจารณา

โค๊ดสำหรับการทำ type check

ข้อมูลเกี่ยวกับ type และตัวอย่างของ prototype

```
typedef struct Ty_ty_ *Ty_ty;

struct Ty_ty_ {enum {Ty_record, Ty_nil, Ty_int, Ty_string, Ty_array,
                     Ty_name, Ty_void} kind;
               union {   Ty_ty array;
                         struct {S_symbol sym; Ty_ty ty;} name;
                        } u;
              };

Ty_ty Ty_Int(void);
Ty_ty Ty_String(void);
Ty_ty Ty_Void(void);
Ty_ty Ty_Array(Ty_ty ty);
```

โค๊ดสำหรับการทำ type check

ฟังก์ชั่นที่ return ข้อมูลเกี่ยวกับ type ชนิดต่างๆ

```
static struct Ty_ty_ tyint = {Ty_int};
Ty_ty Ty_Int(void) {return &tyint;}

static struct Ty_ty_ tystring = {Ty_string};
Ty_ty Ty_String(void) {return &tystring;}

static struct Ty_ty_ tyvoid = {Ty_void};
Ty_ty Ty_Void(void) {return &tyvoid;}

Ty_ty Ty_Array(Ty_ty ty) {
        Ty_ty p = malloc(sizeof(*p));
        p->kind=Ty_array;
        p->u.array=ty;
        return p;
}
```

Function สำหรับ type check

```
Ty_ty transExp(S_table env, A_exp a) {
  switch(a->kind) {
   case A_varExp : {
    return transVar(env, a->u.var);
   }
   case A_intExp: {
    return Ty_Int();
   }
   case A_opExp: {
    A_oper oper = a->u.op.oper;
    Ty_ty left = transExp(env, a->u.op.left);
    Ty_ty right = transExp(env, a->u.op.right);
    if (oper == A_plusOp) {
       if (left->kind == Ty_int && right->kind == Ty_int) return Ty_Int();
       else if …
       else Print_Error(“Type error”)
    }
    else if …
    break;
  }
  …
```

Type check สำหรับ assignment statement

```
Ty_ty transExp(S_table env, A_exp a) {
 switch(a->kind) {
 …
 case A_assignExp: {
   // Retrieve types for the expression and variable parts of the assignment.
   Ty_ty assignExp = transExp(env, a->u.assign.exp);
   Ty_ty assignVar = transVar(env, a->u.assign.var);

    // Insert code to check if assignExp is type-compatible with assignVar

   return Ty_Void();

   break;
 }
 …
```

Type check สำหรับ if statement

```
Ty_ty transExp(S_table env, A_exp a) {
 switch(a->kind) {
 …
 case A_ifExp: {
  Ty_ty test;
  Ty_ty then;
  Ty_ty elsee;
  // Grab expressions for test, then, and else.
  test = transExp(env, a->u.iff.test);
  then = transExp(env, a->u.iff.then);
  if (a->u.iff.elsee != NULL)
    elsee = transExp(env, a->u.iff.elsee);
  else
    elsee = Ty_Nil();
 // Insert code to check if test is type-compatible with boolean

 // if then and elsee are well-typed, return Void type
  return Ty_Void();
  break;
 }
…
```

Type check สำหรับ call statement

```
case A_callExp: {
  E_enventry functionID = NULL;
  A_expList argsList = NULL;
  Ty_tyList formals;
  Ty_ty returnVal;

  // Verify that function has been defined previously.
  functionID = S_look(venv, a->u.call.func);
  if (functionID == NULL) {
    Print_Error("No function by that name");
  }
  returnVal = functionID->u.fun.result;
  if (no argument function) {
    return ActualTypeOf(returnVal));
  }
  // Check parameters and function call types.
  argsList = a->u.call.args;
  formals = functionID->u.fun.formals;
  while (argsList != NULL && formals != NULL) {
    Ty_ty fType = formals->head;
    A_exp argExp = argsList->head;
    Ty_ty argType = transExp(env, argExp);
    formals = formals->tail;
    argsList = argsList->tail;
    // Insert code to check if fType and artType are compatible
    }
  return ActualTypeOf(returnVal));
  break;
 }
```

Additional typedef

```
typedef struct Ty_tyList_ *Ty_tyList;
struct Ty_tyList_ {Ty_ty head; Ty_tyList tail;};

typedef struct A_expList_ *A_expList;
struct A_expList_ {A_exp head; A_expList tail;};
```

ฟังก์ชั่น main ของคอมไพเลอร์

```
extern A_exp absyn_root;

void SEM_transProg(A_exp exp) {
  Ty_ty exprType;
  // Create environments.
  S_table env = NULL;
  env = symTabConstruct(exp); // construct the symbol table
  exprType = transExp(env, exp); // type-checking
}

int main(int argc, char **argv) {
  if (argc!=2) {
    fprintf(stderr,"usage: parsetest filename\n");
    exit(1);
  }
  /* Step 1, get tokens from lexical analysis; Step 2, build an AST while parsing */
  parseInput(argv[1]);
  /* Step 3, traverse AST for semantic analysis from the entry point of absyn_root. */
  SEM_transProg(absyn_root);
  return 0;
}
```