CSC312 Principles of Programming Languages :

Type System

Ch. 6 Type System

6.1 Type System for Clite
6.2 Implicit Type Conversion
6.3 Formalizing the Clite Type System

Type System

Type? Type error?

Type checking: The detection of type errors, either at compile time or at run time.

Type system: provides a means of defining new types and determining the right way to use types.

Defining a Type System

Informally: a set of rules in stylized English, along with an algorithm that implements them.

Example: CLite type system.

Formally: A set of boolean-valued functions.

CLite Properties

Static binding

Single function: main

Single scope: no nesting, no globals

Name resolution errors detected at compile time

- *Each declared variable must have a unique identifier*
- *Identifier must not be a keyword (syntactically enforced)*
- *Each variable referenced must be declared.*

Example Clite Program (Fig 6.1)

```
// compute the factorial of integer n
void main ( ) {
   int n, i, result;
   n = 8;
   i = 1;
   result = 1;
   while (i < n) {
        i = i + 1;
        result = result * i;
   }
}
```

What type rules do you think would be reasonable?

How to check whether the program violates the rules?

Data Structure: Type Map

- Type map is a set of ordered pairs

  E.g., {<n, int>, <i, int>, <result, int>}
- Can implement as a hash table (e.g., dictionary)
- Two related functions
  - Function typing creates a type map
  - Function typeOf retrieves the type of a variable:

    typeOf(id) = type

The typing Function creates a type map

```
public static TypeMap typing (Declarations d) {
    TypeMap map = new TypeMap( );
    for (Declaration di : d) {
        map.put (di.v, di.t);
    }
    return map;
}
```

Type Rule 6.1

All referenced variables must be declared.

if (!typeOf(id))   print "undefined variable"+id

Type Rule 6.2

All declared variables must have unique names.

```
public static void V (Declarations d) {
    for (int i=0; i<d.size() - 1; i++){
        Declaration di = d.get(i);
         for (int j=i+1; j<d.size(); j++) {
            Declaration dj = d.get(j);
            check( ! (di.v.equals(dj.v)),
                    "duplicate declaration: " + dj.v);
        }
    }
}
```

Rule 6.2 example

```
// compute the factorial of integer n
void main ( ) {
   int n, i, result;
   n = 8;
   i = 1;
   result = 1;
   while (i < n) {
        i = i + 1;
        result = result * i;
   }
}
```

These must all be unique

Type Rule 6.3

A program is valid if

- *its Declarations are valid and*
- *its Block body is valid with respect to the type map for those Declarations*

```
public static void V (Program p) {
    V (p.decpart);
    V (p.body, typing (p.decpart));
}
```

Rule 6.3 Example

```
// compute the factorial of integer n
void main ( ) {
   int n, i, result;
   n = 8;
   i = 1;
   result = 1;
   while (i < n) {
        i = i + 1;
        result = result * i;
   }
}
```

These must be valid.

Type Rule 6.4

what kind of type conversions?

Validity of a Statement:

- *A Skip is always valid*
- *An Assignment is valid if:*
  - Its target *Variable* is declared
  - Its source *Expression* is valid
  - If the target *Variable* is float, then the type of the source *Expression* must be either float or int
  - Otherwise if the target *Variable* is int, then the type of the source *Expression* must be either int or char
  - Otherwise the target *Variable* must have the same type as the source *Expression*.

Type Rule 6.4 (Conditional)

- *A Conditional is valid if:*
  - Its test *Expression* is valid and has type bool
  - Its thenbranch and elsebranch *Statements* are valid

Type Rule 6.4 (Loop)

- *A Loop is valid if:*
  - Its test *Expression* is valid and has type bool
  - Its Statement body is valid

Type Rule 6.4 (Block)

- *A Block is valid if all its Statements are valid.*

Rule 6.4 Example

```
// compute the factorial of integer n
void main ( ) {
   int n, i, result;
   n = 8;
   i = 1;
   result = 1;
   while (i < n) {
        i = i + 1;
        result = result * i;
   }
}
```

This assignment is valid if:
- n is declared,
- 8 is valid, and
- the type of 8 is int or char (since n is int).

Rule 6.4 Example

```
// compute the factorial of integer n
void main ( ) {
   int n, i, result;
   n = 8;
   i = 1;
   result = 1;
   while (i < n) {
        i = i + 1;
        result = result * i;
   }
}
```

This loop is valid if
 i < n is valid,
 i < n has type bool, and
 the loop body is valid

Type Rule 6.5

Validity of an Expression:

- *A Value is always valid.*
- *A Variable is valid if it appears in the type map.*
- *A Binary is valid if:*
  - Its *Expressions* term1 and term2 are valid
  - If its *Operator* op is arithmetic, then both *Expressions* must be either int or float
  - If op is relational, then both *Expressions* must have the same type
  - If op is && or ||, then both *Expressions* must be bool
- *A Unary is valid if:*
  - Its *Expression* term is valid,
  - …

Type Rule 6.6

The type of an Expression e is:

- *If e is a Value, then the type of that Value.*
- *If e is a Variable, then the type of that Variable.*
- *If e is a Binary* op term1 term2*, then:*
  - *If* op *is arithmetic, then the (common) type of* term1 *or* term2
  - *If* op *is relational, && or ||, then* bool
- *If e is a Unary* op term*, then:*
  - *If* op *is ! then* bool
  - ...

Rule 6.5 and 6.6 Example

```
// compute the factorial of integer n
void main ( ) {
   int n, i, result;
   n = 8;
   i = 1;
   result = 1;
   while (i < n) {
       i = i + 1;
       result = result * i;
   }
}
```

This *Expression* is valid since:
op is arithmetic (*) and
the types of i and result are int.
Its result type is int since:
the type of i is int.

Examples on Gee Language

No declarations.

Only number and boolean types of values to consider.

But no test case or assignments will contain the Boolean constants True, False.
E.g.,

```
v = True; // no
v = (x> 0); // yes
if (x==False) // no
if (x>y)    // yes
```

```
#  expression operators
relation    = "==" | "!=" | "<" | "<=" | ">" | ">="

#  expressions
expression = andExpr { "or" andExpr }
andExpr    = relationalExpr { "and" relationalExpr }
relationalExpr = addExpr [ relation addExpr ]
addExpr    = term { ("+" | "-") term }
term       = factor { ("*" | "/") factor }
factor     = number | string | ident |  "(" expression ")"

# statements
stmtList =  {  statement  }
statement = ifStatement |  whileStatement  |  assign
assign = ident "=" expression  eoln
ifStatement = "if" expression block   [ "else" block ]
whileStatement = "while"  expression  block
block = ":" eoln indent stmtList undent

#  goal or start symbol
script = stmtList
```

Examples on Gee Language

```
class Assign( Statement ):
     def __init__(self, var, expr):
        self.var = str(var)
        self.expr = expr
    def __str__( self ):
          return "= " + self.var + " " + str(self.expr)
    def meaning(self, state):
          state[self.var] = self.expr.value(state)
          return state
    def tipe(self, tm):
          tp = self.expr.tipe()
          if ( tp == "")
             ??
          if self.var is not in tm
             ??
          else
             ??
```

An Assignment is valid if:

- Its source *Expression* is valid
- If the target *Variable* has been defined, it must have the same type as the source *Expression*.

Examples on Gee Language

```
class Assign( Statement ):
     def __init__(self, var, expr):
        self.var = str(var)
        self.expr = expr
    def __str__( self ):
           return "= " + self.var + " " + str(self.expr)
    def meaning(self, state):
           state[self.var] = self.expr.value(state)
           return state
    def tipe(self, tm):     # tm is the type map
           tp = self.expr.tipe();
           if (tp == "")
              error ("variable undefined!")
           if self.var is not in tm
              tm[self.var] = tp;
           else
              if (tm[self.var] != tp)
                 error ("type mismatch!")
```

An Assignment is valid if:

- Its source *Expression* is valid
- If the target *Variable* has been defined, it must have the same type as the source *Expression*.

6.2 Implicit Type Conversion

Clite Assignment supports implicit widening conversions

We can transform the abstract syntax tree to insert explicit conversions as needed.

The types of the target variable and source expression govern what to insert.

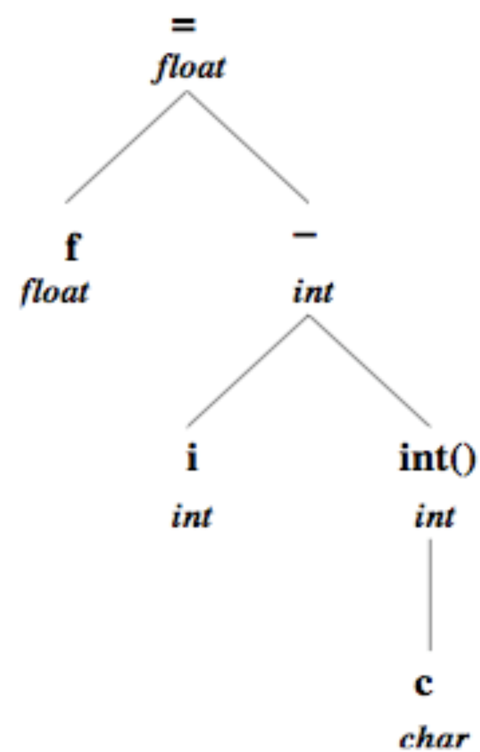
Example: Assignment of int to float

Suppose we have an assignment

f = i - int(c);

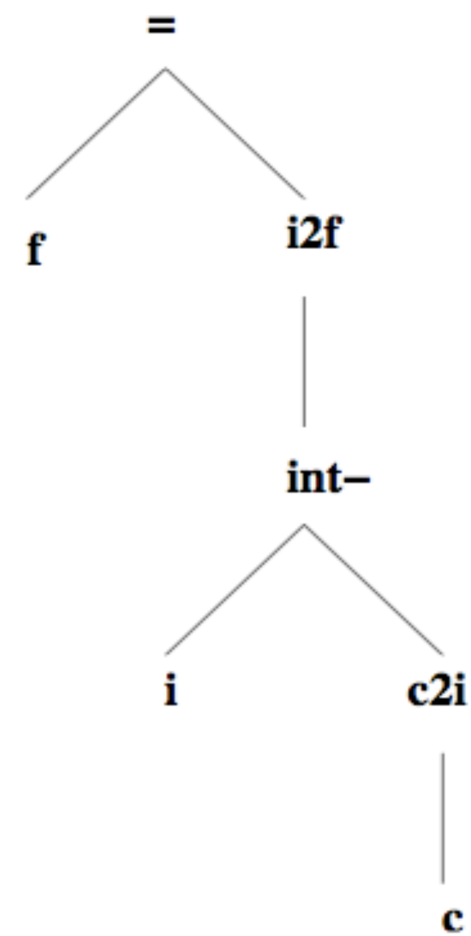
(f, i, and c are float, int, and char variables).

The abstract syntax tree is:



Example (cont'd)

So an implicit widening is inserted to transform the tree to:



Here, **c2i** denotes conversion from char to int, and **i2f** denotes conversion from int to float.

Note: **c2i** is an explicit conversion given by the operator int() in the program.

6.3 Formalizing a Type System

A set of formal rules, written as logic functions with boolean returning values (true or false).

Nothing deep; just a different way to express those rules!

But, the formalism offers rigor and a convenient basis for automatic inferences.

Some logic/mathematic notations

U : union. Example:

$$\bigcup_{i=1,2} \langle name_i, type_i \rangle$$

$$\Downarrow$$

$$\{ \langle name_1, type_1 \rangle, \langle name_2, type_2 \rangle \}$$

Formalizing the Clite Type System

Type map: $tm = \{<v_1, t_1>, <v_2, t_2>, \ldots, <v_n, t_n>\}$

Created by: $typing : Declarations \to TypeMap$

$$typing(d) = \bigcup_{i \in \{1,\ldots,n\}} < d_i.v, d_i.t >$$

Some logic/mathematic notations

$\forall$: for any. Example:

$\forall d \in$ { live animals }, d only eats meat $\Rightarrow$ d is a carnivore.

$\forall i, j \in \{1, 2, ..., k\}$, sibling($Kid_i$,$Kid_j$) $\Rightarrow$ $lastName_i$=$lastName_j$ $\wedge$ $parent_i$=$parent_j$

Type Rule 6.2

All declared variables must have unique names.

$$V : Declarations \to \mathrm{Boolean}$$

$$V(d) = \forall i, j \in \{1,\ldots,n\}(i \neq j \Rightarrow d_i.v \neq d_j.v)$$

Validity of a Clite Program

$$V : Program \rightarrow \mathrm{B}$$
$$V(p) = V(p.\mathrm{decpart}) \wedge V(p.\mathrm{body},\ typing(p.\mathrm{decpart}))$$

(Type Rule 6.3)

A program is valid if

- *its Declarations are valid and*
- *its Block body is valid with respect to the type map for those Declarations*

Type Rule 6.4

Validity of a Statement:

- *A Skip is always valid*
- *An Assignment is valid if: (simplified from our prior def.)*
  - Its target *Variable* is declared
  - Its source *Expression* is valid
  - The target *Variable* must have the same type as the source *Expression*.

Validity of a Clite Statement

(Type Rule 6.4, simplified version for an *Assignment*)

$$V : Statement \times TypeMap \to \mathrm{B}$$

$$V(s, tm) = true \qquad \text{if } s \text{ is a } Skip$$

$$\begin{aligned} &= s.\text{target} \in tm \wedge V(s.\text{source}, tm) \wedge \qquad \text{if } s \text{ is an } Assignment \\ &\quad typeOf(s.\text{target}, tm) = typeOf(s.\text{source}, tm) \end{aligned}$$

Type Rule 6.4 (Conditional)

- *A Conditional is valid if:*
  - Its test *Expression* is valid and has type bool
  - Its thenbranch and elsebranch *Statements* are valid
- *A Loop is valid if:*
  - Its test *Expression* is valid and has type bool
  - Its Statement body is valid
- *A Block is valid if all its Statements are valid.*

Validity of a Clite Statement

(Type Rule 6.4, simplified version for an *Assignment*)

$$V : Statement \times TypeMap \rightarrow \mathrm{B}$$

$$V(s, tm) = true \qquad \text{if } s \text{ is a } Skip$$

$$= s.\text{target} \in tm \wedge V(s.\text{source}, tm) \wedge typeOf(s.\text{target}, tm) = typeOf(s.\text{source}, tm) \qquad \text{if } s \text{ is an } Assignment$$

$$= V(s.\text{test}, tm) \wedge typeOf(s.\text{test}, tm) = bool \wedge V(s.\text{thenbranch}, tm) \wedge V(s.\text{elsebranch}, tm) \qquad \text{if } s \text{ is a } Conditional$$

$$= V(s.\text{test}, tm) \wedge typeOf(s.\text{test}, tm) = bool \wedge V(s.\text{body}, tm) \qquad \text{if } s \text{ is a } Loop$$

$$= V(b_1, tm) \wedge V(b_2, tm) \wedge \ldots \wedge V(b_n, tm) \qquad \text{if } s \text{ is a } Block$$

Validity of a Clite Expression

(Type Rule 6.5, abbreviated versions for *Binary* and *Unary*)

$$
\begin{array}{ll}
V : Expression \times TypeMap \rightarrow \mathrm{B} & \\
V(e, tm) = true & \text{if } e \text{ is a } Value \\
\quad = e \in tm & \text{if } e \text{ is a } Variable \\
\quad = V(e.\mathrm{term1}, tm) \wedge V(e.\mathrm{term2}, tm) \wedge & \text{if } e \text{ is a } Binary \wedge \\
\quad\quad typeOf(e.\mathrm{term1}, tm) \in \{float, \mathrm{int}\} \wedge & e.\mathrm{op} \in ArithmeticOp \cup \\
\quad\quad typeOf(e.\mathrm{term2}, tm) \in \{float, \mathrm{int}\} \wedge & \quad RelationalOp \\
\quad\quad typeOf(e.\mathrm{term1}, tm) = typeOf(e.\mathrm{term2}, tm) & \\
\quad = V(e.\mathrm{term}, tm) \wedge e.\mathrm{op} = \, ! \, \wedge & \text{if } e \text{ is a } Unary \\
\quad\quad typeOf(e.\mathrm{term}, tm) = bool &
\end{array}
$$

Type of a Clite Expression

(Type Rule 6.6, abbreviated version)

$$
\begin{aligned}
&typeOf : Expression \times TypeMap \to Type \\
&typeOf(e, tm) = e.type && \text{if } e \text{ is a } Value \\
&\quad = e.type && \text{if } e \text{ is a } Variable \wedge e \in tm \\
&\quad = typeOf(e.\text{term1}, tm) && \text{if } e \text{ is a } Binary \wedge e.\text{op} \in ArithmeticOp \\
&\quad = boolean && \text{if } e \text{ is a } Binary \wedge e.\text{op} \notin ArithmeticOp \\
&\quad = boolean && \text{if } e \text{ is a } Unary \wedge e.\text{op} = \,!
\end{aligned}
$$