

Programming Languages
2nd edition
Tucker and Noonan

Chapter 6
Type Systems

I was eventually persuaded of the need to design programming notations so as to maximize the number of errors that cannot be made, or if made, can be reliably detected at compile time.
C.A.R Hoare

Contents

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

Motivation: Detecting Type Errors

The detection of type errors, either at compile time or at run time, is called *type checking*.

- Type errors occur frequently in programs.
- Type errors can't be prevented/detected by EBNF
- If undetected, type errors can cause severe run-time errors.
- A type system can identify type errors before they occur.

6.1 Type System for CLite

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;
  }
}
```

Designing a Type System

- A set of rules V in highly-stylized English
 - return true or false
 - based on abstract syntax
 - Note: standards use concrete syntax
 - Mathematically a function:
 - $V: \text{AbstractSyntaxClass} \rightarrow \text{Boolean}$
- Facilitates static type checking.
- Implementation throws an exception if invalid

Type Rule 6.1

All referenced variables must be declared.

- Type map is a set of ordered pairs
E.g., {<n, int>, <i, int>, <result, int>}
- Can implement as a hash table
- 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 6.2

All declared variables must have unique names.

```
public static void V (Declarations d) {
    for (int i=0; i<d.size() - 1; i++)
        for (int j=i+1; j<d.size(); j++) {
            Declaration di = d.get(i);
            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; ← These must all be unique
    n = 8;
    i = 1;
    result = 1;
    while (i < n) {
        i = i + 1;
        result = result * i;
    }
}
```

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.declpart);
    V (p.body, typing (p.declpart));
}
```

Rule 6.3 Example

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

Type Rule 6.4

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 (continued)

- 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.

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.

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:

The diagram shows an AST for the assignment. The root node is '=' with children 'f' (float) and '-'. The '-' node has children 'i' (int) and 'int()'. The 'int()' node has child 'c' (char).

Example (cont'd)

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

The diagram shows the transformed AST. The root node is '=' with children 'f' (float) and 'i2f'. The 'i2f' node has child 'int-', which has children 'i' (int) and 'c2i'. The 'c2i' node has child 'c' (char).

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

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

6.3 Formalizing the Clite Type System

Type map: $tm = \{ \langle v_1, t_1 \rangle, \langle v_2, t_2 \rangle, \dots, \langle v_n, t_n \rangle \}$

Created by: $typing : Declarations \rightarrow TypeMap$
 (Type Rule 6.1) $typing(d) = \bigcup_{i \in \{1, \dots, n\}} \langle d_i, v, d_i, t \rangle$

Validity of Declarations: $V : Declarations \rightarrow B$
 (Type Rule 6.2) $V(d) = \forall i, j \in \{1, \dots, n\} (i \neq j \Rightarrow d_i, v \neq d_j, v)$

Validity of a Clite Program

(Type Rule 6.3)

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

Validity of a Clite Statement

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

$V : \text{Statement} \times \text{TypeMap} \rightarrow \mathbf{B}$

- $V(s, tm) = \text{true}$ if s is a *Skip*
- = $s.\text{target} \in tm \wedge V(s.\text{source}, tm) \wedge$ if s is an *Assignment*
 $\text{typeOf}(s.\text{target}, tm) = \text{typeOf}(s.\text{source}, tm)$
- = $V(s.\text{test}, tm) \wedge \text{typeOf}(s.\text{test}, tm) = \text{bool} \wedge$ if s is a *Conditional*
 $V(s.\text{thenbranch}, tm) \wedge V(s.\text{elsebranch}, tm)$
- = $V(s.\text{test}, tm) \wedge \text{typeOf}(s.\text{test}, tm) = \text{bool} \wedge$ if s is a *Loop*
 $V(s.\text{body}, tm)$
- = $V(b_1, tm) \wedge V(b_2, tm) \wedge \dots \wedge V(b_n, tm)$ if s is a *Block*

Validity of a Clite Expression

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

$V : \text{Expression} \times \text{TypeMap} \rightarrow \mathbf{B}$

- $V(e, tm) = \text{true}$ if e is a *Value*
- = $e \in tm$ if e is a *Variable*
- = $V(e.\text{term1}, tm) \wedge V(e.\text{term2}, tm) \wedge$ if e is a *Binary* \wedge
 $\text{typeOf}(e.\text{term1}, tm) \in \{\text{float}, \text{int}\} \wedge$ $e.\text{op} \in \text{ArithmeticOp} \cup$
 $\text{typeOf}(e.\text{term2}, tm) \in \{\text{float}, \text{int}\} \wedge$ RelationalOp
 $\text{typeOf}(e.\text{term1}, tm) = \text{typeOf}(e.\text{term2}, tm)$
- = $V(e.\text{term}, tm) \wedge e.\text{op} = ! \wedge$ if e is a *Unary*
 $\text{typeOf}(e.\text{term}, tm) = \text{bool}$

Type of a Clite Expression

(Type Rule 6.6, abbreviated version)

$\text{typeOf} : \text{Expression} \times \text{TypeMap} \rightarrow \text{Type}$

- $\text{typeOf}(e, tm) = e.\text{type}$ if e is a *Value*
- = $e.\text{type}$ if e is a *Variable* $\wedge e \in tm$
- = $\text{typeOf}(e.\text{term1}, tm)$ if e is a *Binary* $\wedge e.\text{op} \in \text{ArithmeticOp}$
 = boolean if e is a *Binary* $\wedge e.\text{op} \notin \text{ArithmeticOp}$
- = $\text{typeOf}(e.\text{term}, tm)$ if e is a *Unary* $\wedge e.\text{op} = -$
 = boolean if e is a *Unary* $\wedge e.\text{op} = !$