Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel
Type Checking
Type Checking

• Type checking is a part of context-sensitive analysis
• A type-checker verifies that the type of a construct matches the type that is expected by its context
• Examples
  – dereference operation is only applied to pointers
  – indexing is only done for an array
  – in a method call, number of arguments and their types match the declaration
  – arguments of modulo operation are both integers
• Type checking can be
  – static: done at compile time
  – dynamic: done at execution time
• A language in which every expression can be assigned an unambiguous type is called a strongly-typed language
Type Systems

- **Base types**
  - Programming languages typically include base types for: numbers (int, float), characters, Booleans

- **Compound and constructed types**
  - Programmers need higher-level abstractions than the base types, such as lists, graphs, trees, tables, etc.
  - Programming languages provide mechanisms to combine and aggregate objects and to derive types for the resulting objects
  - arrays, structures, enumerated sets, pointers

- A type system consists of a set of base types and a set of *type-constructors*
  - array, function, pointer, product

- Using base types and type-constructors each expression in a program can be represented with a *type expression*
Example Type Expressions

- **Base types:**
  - `integer, char, float`

- **Type constructors:**
  - `array, product (×), record, pointer, function (→)`

Example type expressions in C

```c
int A[10]  // type expression for A is: array(10, integer)

int foo(char a, int *b)
    // type expression for foo is: function(product(char, pointer(integer)), integer)
    // or using notation: char × pointer(integer) → integer

struct fie {
    int a, b;
}
    // type-expression for fie is: record((a × integer) × (b × integer))
```

Note that we used product to show the field names a and b in the type expression
Type Systems

• A type system is a collection of rules for assigning type-expressions to language constructs.

• A type-expression is either a base type or is formed by applying a type constructor to a type-expression.

• Inference rules for type-expressions:
  – (Pascal) If both operands of the arithmetic operators addition, subtraction and multiplication are of type integer than the result is of type integer.
  – (C, C++) The results of the unary & operator is a pointer to the object referred to by the operand. If the operand is of type “foo”, then the type of the result is a “pointer to foo”.

• A sound type system eliminates the need for dynamic checking because it statically determines whether errors will occur or not.
Type Equivalence

• Structural equivalence
  – Two types are equivalent if they have the same structure
  – Two type-expressions are structurally equivalent if either they are the same basic type or they are formed by applying the same type constructor to structurally equivalent types

• Name equivalence
  – Each type name is viewed as a distinct type
Checking Structural Equivalence

Using type constructors: *array*, *product* ($\times$), *pointer*, and *function* ($\rightarrow$)

function sequiv(s, t)
begin
  if s and t are the same basic type then
    return true;
  else if s = array(s_1, s_2) and t = array(t_1, t_2) then
    return sequiv(s_1, t_1) and sequiv(s_2, t_2);
  else if s = s_1 \times s_2 and t = t_1 \times t_2 then
    return sequiv(s_1, t_1) and sequiv(s_2, t_2);
  else if s = pointer(s_1) and t = pointer(t) then
    return sequiv(s_1, t_1);
  else if s = s_1 \rightarrow s_2 and t = t_1 \rightarrow t_2 then
    return sequiv(s_1, t_1) and sequiv(s_2, t_2);
  else
    return false;
end
Given the following grammar for a very simple language

\[ P \rightarrow D ; S \]

\[ D \rightarrow D ; D | id : T \]

\[ T \rightarrow \text{char} | \text{int} | \text{array [num] of } T | \text{pointer } T | \text{function } T \text{ to } T \]

\[ E \rightarrow \text{literal} | \text{num} | \text{id} | E \mod E | E [E] | * E | E (E) | E = E \]

\[ S \rightarrow id := E | \text{if } E \text{ then } S | \text{while } E \text{ do } S | S ; S \]

We want to write an ad-hoc translation scheme for type-checking
We will use type-constructors to construct type-expressions
We will do type-checking using these type-expressions
Type Checking: Example

• We will use the following type constructors
  – array(I,T) : creates a type expression for an array of type T with index set I
  – pointer(T) : creates a type expression of type pointer to type T
  – function(T,T) : creates a type expression of type function from type T to type T
• We will also use the following:
  – id.entry : this attribute gives the location of the corresponding identifier in the symbol table
  – addtype(id.entry, type) : enters the type information to the symbol table
  – lookup(id.entry) : returns the type information stored in the symbol table
• If we detect an error, we will set the type of the corresponding program segment to type-error
Type Checking: How should it work?

• Let’s examine a program that is part of the language defined above

```plaintext
x: char;  //this declares a new variable “x” of type char
y: pointer char;  //this declares “y” to be of type pointer(char)
x := ‘c’;  //this assignment is a literal (which should be a char) to a char
x := * y;  //this de-references y (of type pointer(char)) to get something
           //of type char, and then assigns it to x (which is of type char)
```

• Let us now draw the parse tree for this program
• We will walk the parse tree (depth first walk –
  walk down the left-most un-touched branch and then back up)
• Show how the type-checking should progress
Type-Checking Example: Parse Tree

Direction of Traversal =

Insert code for type-checking
At the points labeled in red
Type Checking Example: Simple Example

(1) \( T.\text{type} = \text{char} \)

(2) \( \text{assert}(x \text{ not in symbol table}) \)
    
     \( \text{add } (x, \text{char}) \text{ to symbol table} \)

(3) \( T.\text{type} = \text{char} \)

(4) \( T.\text{type} = \text{pointer}(T) = \text{pointer(char)} \)

(5) \( \text{assert}(y \text{ not in symbol table}) \)
    
     \( \text{add } (y, \text{pointer(char)}) \text{ to sym} \)

(6) \( E.\text{type} = \text{char} \)

(7) \( \text{lookup}(id); \text{assert}(id.\text{exists}); \)
    
     \( \text{assert}(id.\text{type}==E.\text{type}) \)

(8) \( \text{lookup}(id); \text{assert}(id.\text{exists}); \)
    \( E.\text{type} = id.\text{type} = \text{pointer(char)} \)

(9) \( \text{assert}(E.\text{type} == \text{pointer}(z) \)
    \( \text{for some } z); \)
    \( E.\text{type} = z = \text{char} \)

(10) \( \text{lookup}(id); \text{assert}(id.\text{exists}); \)
    \( \text{assert}(id.\text{type}==E.\text{type}); \)
Type Checking Example: Declarations

\[ P \rightarrow D ; S \]

\[ D \rightarrow D ; D \]

\[ D \rightarrow \text{id} : T \quad \{ \text{addtype(id.entry, } T\text{.type); } \}\]

\[ T \rightarrow \text{char} \quad \{ T\text{.type }\leftarrow \text{ char; } \}\]

\[ T \rightarrow \text{int} \quad \{ T\text{.type }\leftarrow \text{ integer; } \}\]

\[ T \rightarrow \text{array [num] of } T_1 \quad \{ T\text{.type }\leftarrow \text{ array(1...num.val, } T_1\text{.type); } \}\]

\[ T \rightarrow \text{pointer } T_1 \quad \{ T\text{.type }\leftarrow \text{ pointer(} T_1\text{.type); } \}\]

\[ T \rightarrow \text{function } T_1 \text{ to } T_2 \quad \{ T\text{.type }\leftarrow \text{ function(} T_1\text{.type, } T_2\text{.type); } \}\]
Type Checking Example: Expressions

\[ E \rightarrow \text{literal} \quad \{ E\text{.type} \leftarrow \text{char}; \} \]

\[ E \rightarrow \text{num} \quad \{ E\text{.type} \leftarrow \text{integer}; \} \]

\[ E \rightarrow \text{id} \quad \{ E\text{.type} \leftarrow \text{lookup(id.entry)}; \} \]

\[ E \rightarrow E_1 \mod E_2 \quad \{ \text{if } (E_1\text{.type} = \text{integer} \text{ and } E_2\text{.type} = \text{integer}) \]
\quad \text{then } E\text{.type} \leftarrow \text{integer}; \]
\quad \text{else } E\text{.type} \leftarrow \text{type-error}; \} \]

\[ E \rightarrow E_1[ E_2 ] \quad \{ \text{if } (E_2\text{.type} = \text{integer} \text{ and } E_1\text{.type} = \text{array}(i,t)) \]
\quad \text{then } E\text{.type} \leftarrow t; \quad /* \text{for some } i \text{ and some } t */ \]
\quad \text{else } E\text{.type} \leftarrow \text{type-error}; \} \]
Type Checking Example: Expressions

\[ E \rightarrow * E_1 \]
\{ if \((E_1\.type = \text{pointer}(t))\) /* for some \(t\) */
  then \(E\.type \leftarrow t\);
  else \(E\.type \leftarrow \text{type-error};\) \}

\[ E \rightarrow E_1( E_2 ) \]
\{ if \((E_2\.type = s \text{ and } E_1\.type = \text{function}(s,t))\)
  then \(E\.type \leftarrow t;\) /* for some \(s\) and some \(t\) */
  else \(E\.type \leftarrow \text{type-error};\) \}

\[ E \rightarrow E_1 = E_2 \]
\{ if \((E_1\.type == E_2\.type)\)
  then \(E\.type \leftarrow \text{boolean};\)
  else \(E\.type \leftarrow \text{type-error};\) \}
Type Checking Example: Statements

\[
S \rightarrow \text{id} := E \quad \{\text{if (id.type = E.type)} \\
\quad \text{then S.type} \leftarrow \text{void}; /* we could assign type of } E \text{ here */} \\
\quad \text{else S.type} \leftarrow \text{type-error}; \}
\]

\[
S \rightarrow \text{if } E \text{ then } S_1 \quad \{\text{if (E.type = boolean)} \\
\quad \text{then S.type} \leftarrow S_1\text{.type}; \\
\quad \text{else S.type} \leftarrow \text{type-error}; \}
\]

\[
S \rightarrow \text{while } E \text{ do } S_1 \quad \{\text{if (E.type = boolean)} \\
\quad \text{then S.type} \leftarrow S_1\text{.type}; \\
\quad \text{else S.type} \leftarrow \text{type-error}; \}
\]

\[
S \rightarrow S_1 ; S_2 \quad \{\text{if (S_1\text{.type} = \text{void and } S_2\text{.type} = \text{void})} \\
\quad \text{then S.type} \leftarrow \text{void}; \\
\quad \text{else S.type} \leftarrow \text{type-error}; \}
\]
Type Inference

- Inference rules that specify mapping between operand types and result type

- \( X = Y; \) \( X \) should have a type that is \textit{compatible} with \( Y \)
  - In Pascal: an integer is compatible with the sub-range 1...100
  - In C: a float + a double results in a double
  - A float can be assigned to a double, but a double cannot be assigned to a float (the relationship is asymmetric)

- These rules need to be handled by the compiler
  - How and when they are enforced is important
Types of Typed Languages

• A language in which every expression can be assigned an unambiguous type is called a strongly typed language
  – If every expression can be typed at compile time, the language is called statically typed
  – If some expressions can only be typed at compile time, the language is dynamically typed

• Untyped languages have no notion of type at all
  – Everything is just bit patterns, floating point + has to be distinguished from integer +! (in BCPL #+ is floating point addition)

• In weakly typed languages, there is a notion of type, but it is not strictly enforced for all expressions