Computer Science 160
Translation of Programming Languages

Instructor: Christopher Kruegel

Type Checking

Type

- **Type**: for each data value in the program, there is **a collection of properties** associated with it, known as the value's *type*

- With each **type T** we associate **values** and **operators** that we can apply to values of type T.
  - Type of int also gives ranges of values $-2^{31} \leq i < 2^{31}$
  - To values of type string, we can apply operations such as *println*, but we cannot multiply two string.
- Conversely, with each **operator** with associate types that describe the nature of the operator's arguments and result.

+	integer	real	double	complex
integer	integer	real	double	complex
real	real	real	double	complex
double	double	double	double	*illegal*
complex	complex	complex	*illegal*	complex

Result Types for Addition in FORTRAN77

Benefits: Types for ***expressiveness***

- An operator that has different meanings based on the types of its arguments is "overloaded"
- Still, for the previous example. Fortran has a single addition operator, +, and uses type information to determine how it should be implemented.

Type of			Code
a	b	a + b	
integer	integer	integer	iADD r_a, r_b $\Rightarrow$ r_{a+b}
integer	real	real	i2f f_a $\Rightarrow$ r_{a_f} fADD r_{a_f}, r_b $\Rightarrow$ r_{a_f+b}
integer	double	double	i2d r_a $\Rightarrow$ r_{a_d} dADD r_{a_d}, r_b $\Rightarrow$ r_{a_d+b}
real	real	real	fADD r_a, r_b $\Rightarrow$ r_{a+b}
real	double	double	r2d r_a $\Rightarrow$ r_{a_d} dADD r_{a_d}, r_b $\Rightarrow$ r_{a_d+b}
double	double	double	dADD r_a, r_b $\Rightarrow$ r_{a+b}

Benefits: Types as two-version programming

- In languages such as Java, programs are annotated with types.
- This can be seen as a weak form of two-version programming: the programmer specifies twice what the program should do, once by the actual code, and a second time through the types.
- By saying something twice, but in somewhat different languages (Java computation vs types) the probability that we make the same mistake in both expressions is lower than if we state our intention only once.
- The key idea behind semantic analysis is to look for **contradictions between the two specifications** and reject programs with such contradictions.

Type is not everything

- Note that types can only prevent basic mistakes such as "hello" * "world".

- They cannot (usually) prevent more complicated problems, like out-of-bounds indexing of arrays.

```
int [] a = new int [ 10 ]
a [ 20 ] = 3
```

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

- An important distinction is that between type checking (old-fashioned) and type inference (modern).
- In type checking (e.g., Java), we verify that the programmer-written type-annotations are consistent with the program code.

```
def f ( x : String ) : Int = {
    if ( x = "Moon" ) true
    else false
 }
```

is easy to see as inconsistent.

Type Inference

- Types are determined from the context of the reference, rather than just by explicit statements
- Inference rules that specify mapping between operand types and result type
- The compiler can trace how values flow through variables and function arguments
- Any remaining ambiguity is treated as an error the programmer must fix by adding explicit declarations

Type inference


```
def f ( y : ??? ) : ??? = {
    if ( x = y ) y
    else x+1
}
```

What types could you give to x, y and the return value of f?

Clearly x has type integer, y has type integer, no value is returned.

Another Example

That was easy. What about this program

```
def f ( x : ??? ) : ??? = {
   while ( x.g ( y ) ) { y = y+1 };
    if ( y > z ) z = z+1
    else println ( "hello" ) ;
}
```

What types could you give to x, y, z, g and f?

y and z are integers, x must be a class A such that A has a method g which takes an integer and returns a Boolean

Finally, f returns nothing, so should be of type void

Contrasts in Type Systems

Type systems are often described by their design decisions along several dimensions

- Static vs. dynamic types
  - Time of the type binding
- Strong vs. Weak typing
  - Explicit vs. implicit type conversion

Type Binding

Type binding is an association between a name and a type attribute

Type binding time is the time at which a binding takes place.

- Compile time, e.g., bind a variable to a type in C or Java
- Link time
- Load time, e.g., references to DLLs in C/C++
- Runtime, e.g., dynamic type bindings

Static Type Binding

- In a static type system, types are fixed before the program is run (e.g., compile time)

- Compatibility checking can be done by a compiler and errors flagged

- The advantage is that the resulting code need not check for type mismatches at run time, which speeds up execution

- It typically requires adding type declarations (can be annoying), but these can also be seen as a kind of documentation (a benefit), note that this is only for explicit declarations

Dynamic Type Binding

- A variable's type can change as the program runs
- Might be re-bound on every assignment.
- Used in scripting languages (JavaScript, PHP)

- Here's a JavaScript example

```
list = [2, 4.33, 6, 8];
list = 17.3;
```

Dynamic Type Binding

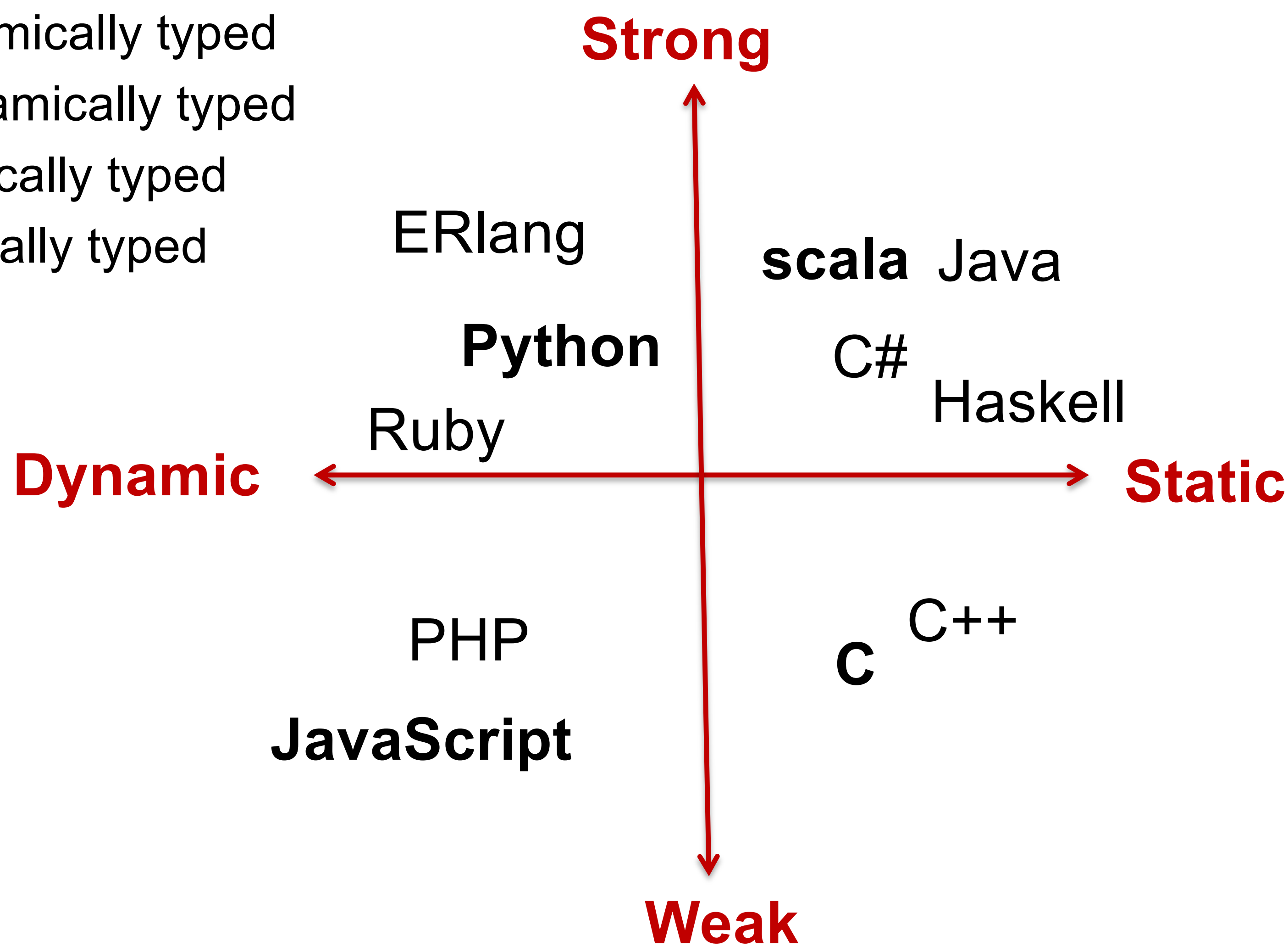
- Flexibility for the programmer
- Obviates the need for “polymorphic” types
- Development of generic functions (e.g., sort)
- But there are disadvantages as well
  - Types have to be constantly checked at run time
  - A compiler cannot detect errors via type mis-matches

Types of Typed Languages

- Definition: Strong versus weak typing
  - Strong/weak typing is about how strictly types are distinguished (e.g., implicit conversion).
  - If a language specification requires its typing rules strongly (i.e., more or less allowing only those automatic type conversions that do not lose information), one can refer to the process as *strongly* typed, if not, as we*akly* typed
  - ***S*trongly-typed** languages do not allow implicit conversions between unrelated types.
  - **Weakly-typed** languages make conversions between unrelated types *implicitly.*

Languages

1. Weakly and dynamically typed
2. Strongly and dynamically typed
3. Strongly and statically typed
4. Weakly and statically typed



Weakly and Dynamically Typed Language

A small **JavaScript** program

```
function add(a, b) {
    result = a + b;
    return result;
}

x = "10"
y = 10

result = add(x, y);

console.log(result);
```

- Any Error?
  - No Error
- What is the result?
  - 1010
- What happened?
  - 10 is implicitly converted to a string “10”, and then concatenated with the other string

Strongly and Dynamically Typed Language

A small **Python** program

```
1 def add(a, b):
2     result = a + b
3     return result
4
5 x = "10"
6 y = 10
7
8 result = add(x, y)
9 print(result)
```

- Any Error?
- What is the result?
- What happened？

```
Traceback (most recent call last):
  File "add.py", line 8, in <module>
    result = add(x, y)
  File "add.py", line 2, in add
    result = a + b
TypeError: can only concatenate str (not "int") to str
```

Note: All these discussion are based on common features of these languages. For example, we could also use type annotations in Python and enable checking statically the types.

Strongly and Statically Typed Language

A small **Scala** program

```
object Add {
  def main(args: Array[String]): Unit = {
    val x = "5"
    val y = 5

    val result = add(x, y)
    println(result)
  }

  def add(a: Int, b: Int) :Int = {
    val result = a + b
    result
  }
}
```

- Any Error?
- What is the result?
- What happened？

```
add.scala: error: type
mismatch;
found    : String
required: Int
     val result = add_(x, y)
                      ^

one error found
```

This makes the code feel dynamically typed.
But it is not. The key is **type inference**

Weakly and Statically Typed Language

A small C program

```c
#include <stdio.h>

int add(int a, int b) {
    int result = a + b;
    return result;
}

int main() {
    char x = '10';
    int y = 10;

    int result = add(x, y);
    printf("%d\n", result);
    return 0;
}
```

- Any Error?
  - No Error
- What is the result?
  - 58
- What happened？

```
add.c:9:14: warning: multi-character
 character constant [-Wmultichar]
    char x = '10';
             ^
add.c:9:14: warning: implicit conver
sion from 'int' to 'char' changes va
lue from 12592 to 48 [-Wconstant-con
version]
    char x = '10';
         ~   ^~~~
2 warnings generated.
```

Untyped Languages?

- Assembly language for instance is said to be ***untyped*** since there is no **type checking**
- The absence of type checking allows a lot of freedom, necessary for strong optimizations
- In general, you don't write assembly code except for very specific projects
- We write in a higher-level language and the compiler produces Assembly code for you

Type System

- **Type system:** The set of types in a programming language, along with the rules that use types to specify program behavior, are collectively called a type system.

  - **A set of base types, or built-in types**
    - Programming languages typically include base types for numbers (int, float), characters, Booleans, etc.

  - **Rules for constructing new types from the existing types**
    - array, function, pointer, product
    - programmers need higher-level abstractions to combine and aggregate objects and to derive types for the resulting objects.

  - **Rules for type inference:** the process of determining a type for each name and each expression in the code.

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*

Type Systems

- 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

Example Type Expressions

- Base types:
  integer, char, float
- Type constructors:
  array, *product* ($\times$), *record*, *pointer, function* ($\rightarrow$)

Example type expressions in 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: $char \times pointer(integer) \rightarrow integer$

```
struct fie {
  int a, b;
}
```

type-expression for fie is: $record((a \times integer) \times (b \times integer))$

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

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

Type Equivalence Check


```
typedef struct {
        int data[100];
        int count;
        } Stack;

typedef struct {
        int data[100];
        int count;
        } Set;

Stack x, y;
Set r, s;
```

```
x = y;
r = s;
```

Always correct

```
x = r;
```

This depends
Yes, for structural equivalence
No, for name equivalence

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

Type Checking: Example

- Given the following grammar for a very simple language

array declaration (of size num)

pointer declaration

function declaration

$P \to D$; S

$D \to D$; D | id : T

$T \to$ char | int | array [num] of T | pointer T | function T to T

integer literal

function call

$E \to$ literal | num | id | E mod E | E [E] | * E | E (E) | E = E

character literal

array access

$S \to$ id := E | if E then S | while E do S | S ; S

pointer dereference

- 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

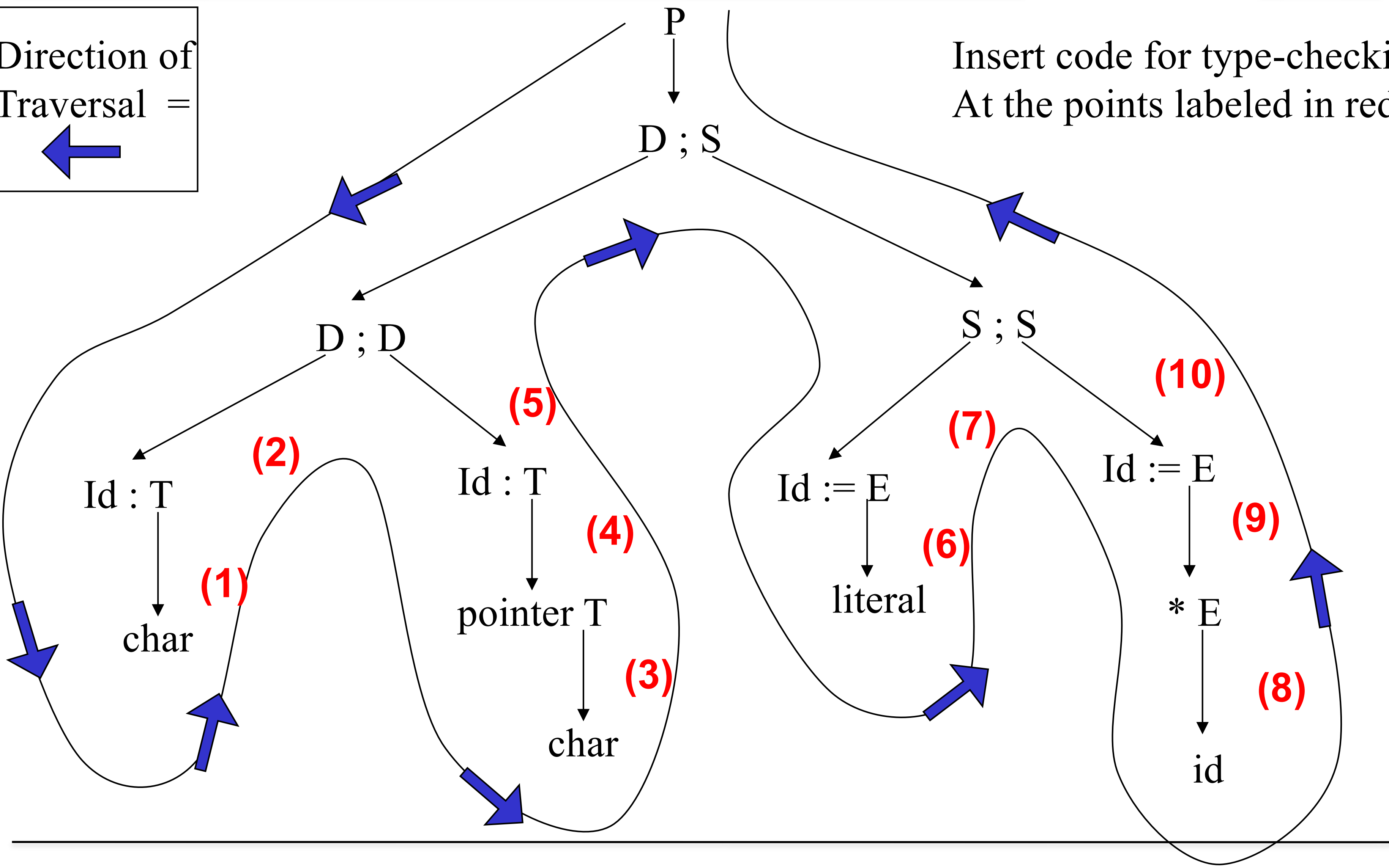
```
x: char;  //this declares a new variable "x" of type char
y: pointer char;  //this declares "y" to be of type pointer(char)
x := 'c' ; //this assignment is a literal (which should be a char) to a char
x := * y; //this de-references y (of type pointer(char)) to get something
          //of type char, 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.type = char

(2) assert(x not in symbol table)
add (x,char) to symbol table

(3) T.type = char

(4) T.type = pointer(T) = pointer(char)

(5) assert(y not in symbol table)
add (y,pointer(char)) to sym

(6) E.type = char

(7) lookup(id); assert(id.exists);
assert(id.type==E.type)

(8) lookup(id); assert(id.exists);
E.type = id.type = pointer(char)

(9) assert(E.type == pointer(z) for some z);
E.type = z = char

(10) lookup(id); assert(id.exists);
assert(id.type==E.type);

Type Checking Example: Declarations

$P \rightarrow D \,;\, S$	
$D \rightarrow D \,;\, D$	
$D \rightarrow$ id : T	{ addtype(id.entry, T.type); }
$T \rightarrow$ char	{ T.type $\leftarrow$ char; }
$T \rightarrow$ int	{ T.type $\leftarrow$ integer; }
$T \rightarrow$ array [num] of T_1	{ T.type $\leftarrow$ array(1...num.val, T_1.type); }
$T \rightarrow$ pointer T_1	{ T.type $\leftarrow$ pointer(T_1.type); }
$T \rightarrow$ function T_1 to T_2	{ T.type $\leftarrow$ function(T_1.type, T_2.type); }

Type Checking Example: Expressions

$E \rightarrow$ literal	{ E.type $\leftarrow$ char; }
$E \rightarrow$ num	{ E.type $\leftarrow$ integer; }
$E \rightarrow$ id	{ E.type $\leftarrow$ lookup(id.entry); }
$E \rightarrow E_1$ mod E_2	{ if (E_1.type = integer and E_2.type = integer) then E.type $\leftarrow$ integer; else E.type $\leftarrow$ type-error; }
$E \rightarrow E_1$ [E_2]	{ if (E_2.type = integer and E_1.type = array(i,t)) then E.type $\leftarrow$ t; /* for some i and some t */ else E.type $\leftarrow$ type-error; }

Type Checking Example: Expressions

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

Type Checking Example: Statements

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

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

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

$S \rightarrow S_1$; S_2 { if (S_1.type = void and S_2.type = void)
then S.type $\leftarrow$ void;
else S.type $\leftarrow$ type-error; }