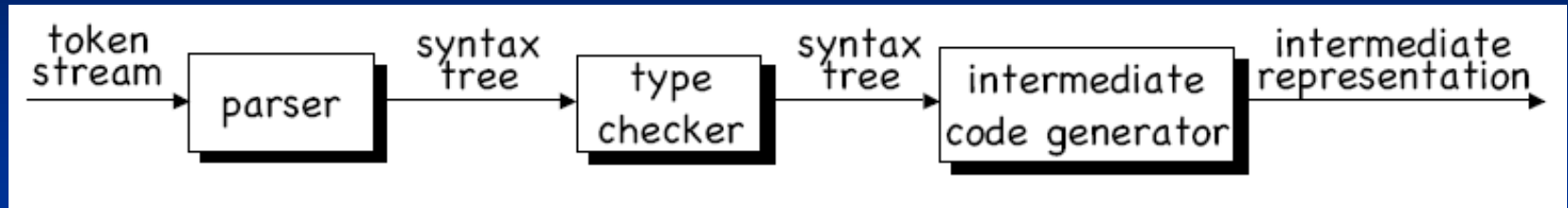


Compiler Construction

Lecture 9 Type Checking

Type Checking (Chapter 6)

Type Checking



TYPE CHECKING is the main activity in semantic analysis.

Goal: calculate and ensure consistency of the type of every expression in a program

If there are type errors, we need to notify the user.

Otherwise, we need the type information to generate code that is correct.

Type Systems and Type Expressions

Type systems

Every language has a set of types and rules for assigning types to language constructs.

Example from the C specification:

- “The result of the unary & operator is a pointer to the object referred to by the operand. If the type of the operand is ‘...’ then the type of the result is ‘pointer to ...’

Usually, every expression has a type.

Type have structure: the type ‘pointer to int’ is CONSTRUCTED from the type ‘int’

Basic vs. constructed types

Most programming languages have basic and types.

BASIC TYPES are the atomic types provided by the

- Pascal: boolean, character, integer, real
- C: char, int, float, double

CONSTRUCTED TYPES are built up from basic types.

- Pascal: arrays, records, sets, pointers
- C: arrays, structs, pointers

Type expressions

We denote the type of language constructs with TYPE EXPRESSIONS.

Type expressions are built up with TYPE CONSTRUCTORS.

1. A basic type is a type expression. The basic types are boolean, char, integer, and real. The special basic type `type_error` signifies an error. The special type `void` signifies “no type”
2. A type name is a type expression (type names are like typedefs in C)

Type expressions

3. A type constructor applied to type expressions is a type expression.
 - a. Arrays: if T is a type expression, then $\text{pointer}(T)$ is a type expression denoting the type “pointer to an object of type T ”
 $\text{Array}(I, T) \leftarrow I$: index set, T : element type
 - b. Products: if T_1 and T_2 are type expressions, then their Cartesian product $T_1 \times T_2$ is also a type expression.
 - c. Records: a record is a special kind of product in which the fields have names (examples below)
 - d. Pointers: if T is a type expression, then $\text{pointer}(T)$ is a type expression denoting the type “pointer to an object of type T ”
 - e. Functions: functions map elements of a domain D to a range R , so we write $D \rightarrow R$ to denote “function mapping objects of type D to objects of type R ” (examples below)
4. Type expressions may contain variables, whose values are themselves type expressions. \leftarrow polymorphism

Record type expressions

The Pascal code

```
type row = record
    address: integer;
    lexeme: array[1..15] of char
end;
var table: array[1..10] of row;
```

associates type expression

record((address \times integer) \times (lexeme \times array(1..15,char)))

with the variable row, and the type expression

array(1..10,record((address \times integer) \times (lexeme \times array(1..15,char)))

with the variable table

Function type expressions

The C declaration

```
int *foo( char a, char b );
```

would associate type expression

```
char × char → pointer(integer)
```

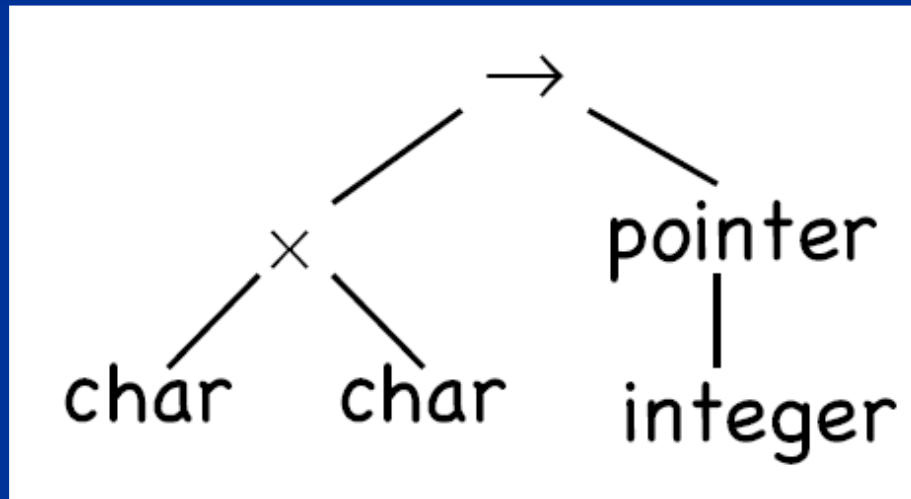
with foo. Some languages (like ML) allow all sorts of crazy function types, e.g.

```
(integer → integer) → (integer → integer)
```

denotes functions taking a function as input and returning another function

Graph representation of type expressions

The recursive structure of a type can be represented with a tree, e.g. for `char × char → pointer(integer)`:



Some compilers explicitly use graphs like these to represent the types of expressions.

Type systems and checkers

A TYPE SYSTEM is a set of rules for assigning type expressions to the parts of a program.

Every type checker implements some type system.

Syntax-directed type checking is a simple method to implement a type checker.

Static vs. dynamic type checking

STATIC type checking is done at compile time.

DYNAMIC type checking is done at run time.

Any kind of type checking CAN be done at run time.

But this reduces run-time efficiency, so we want to do static checking when possible.

A SOUND type system is one in which ALL type errors can be found statically.

If the compiler guarantees that every program it accepts will run without type errors, then the language is STRONGLY TYPED.

An Example Type Checker

Example type checker

Let's build a translation scheme to synthesize the type of every expression from its subexpressions.

Here is a Pascal-like grammar for a sequence of declarations (D) followed by an expression (E)

```
P → D ; E
D → D ; D | id : T
T → char | integer | array [ num ] of T | ↑ T
E → literal | num | id | E mod E | E [ E ] | E ↑
```

Example program: key: integer;
 key mod 1999

The type system

The basic types are char and integer.

type_error signals an error.

All arrays start at 1, so

array[256] of char

leads to type expression: array(1..256,char)

The symbol \uparrow in an declaration specifies a pointer type,
so

\uparrow integer

leads to type expression: pointer(integer)

Translation scheme for declarations

$P \rightarrow D ; E$	
$D \rightarrow D ; D$	
$D \rightarrow \mathbf{id} : T$	{ addtype(id .entry, T.type) }
$T \rightarrow \mathbf{char}$	{ T.type := char }
$T \rightarrow \mathbf{integer}$	{ T.type := integer }
$T \rightarrow \uparrow T_1$	{ T.type := pointer(T_1 .type) }
$T \rightarrow \mathbf{array} [\mathbf{num}] \mathbf{of} T_1$	{ T.type := array(1 .. num val, T_1 .type) }

Try to derive the annotated parse tree for the declaration X: array[100] of \uparrow char

Type checking for expressions

Once the identifiers and their types have been inserted into the symbol table, we can check the type of the elements of an expression:

$E \rightarrow \text{literal}$	{ E.type := char }
$E \rightarrow \text{num}$	{ E.type := integer }
$E \rightarrow \text{id}$	{ E.type := lookup(id.entry) }
$E \rightarrow E_1 \text{ mod } E_2$	{ if $E_1.\text{type} = \text{integer}$ and $E_2.\text{type} = \text{integer}$ then E.type := integer else E.type := type_error }
$E \rightarrow E_1 [E_2]$	{ if $E_2.\text{type} = \text{integer}$ and $E_1.\text{type} = \text{array}(s,t)$ then E.type := t else E.type := type_error }
$E \rightarrow E_1 \uparrow$	{ if $E_1.\text{type} = \text{pointer}(t)$ then E.type := t else E.type := type-error }

How about boolean types?

Try adding

$T \rightarrow \text{boolean}$

Relational operators $< \leq = \geq > \langle \rangle$

Logical connectives **and or not**

to the grammar, then add appropriate type checking semantic actions.

Type checking for statements

Usually we assign the type VOID to statements.

If a type error is found during type checking, though, we should set the type to `type_error`

Let's change our grammar allow statements:

$$P \rightarrow D ; S$$

i.e., a program is a sequence of declarations followed by a sequence of statements.

Type checking for statements

Now we need to add productions and semantic actions:

$S \rightarrow \text{id} := E$	{ if $\text{id.type} = E.type$ then $S.type := \text{void}$ else $S.type := \text{type_error}$ }
$S \rightarrow \text{if } E \text{ then } S_1$	{ if $E.type = \text{boolean}$ then $S.type := S_1.type$ else $S.type := \text{type_error}$ }
$S \rightarrow \text{while } E \text{ do } S_1$	{ if $E.type = \text{boolean}$ then $S.type := S_1.type$ else $S.type := \text{type_error}$ }
$S \rightarrow S_1 ; S_2$	{ if $S_1.type = \text{void}$ and $S_2.type = \text{void}$ then $S.type := \text{void}$ else $S.type := \text{type_error}$.

Type checking for function calls

Suppose we add a production $E \rightarrow E (E)$

Then we need productions for function declarations:

$T \rightarrow T_1 \rightarrow T_2$	$\{ T.type := T_1.type \rightarrow T_2.type \}$
-------------------------------------	---

and function calls:

$E \rightarrow E_1 (E_2)$	$\{ \text{if } E_2.type = s \text{ and } E_1.type = s \rightarrow t$ $\quad \text{then } E.type := t$ $\quad \text{else } E.type := type_error \}$
-----------------------------	---

Type checking for function calls

Multiple-argument functions, however, can be modeled as functions that take a single PRODUCT argument.

$$\text{root} : (\text{real} \rightarrow \text{real}) \times \text{real} \rightarrow \text{real}$$

this would model a function that takes a real function over the reals, and a real, and returns a real. In C:

```
float root( float (*f)(float), float x );
```

Type expression equivalence

Type checkers need to ask questions like:

- “if $E1.type == E2.type$, then ...”

What does it mean for two type expressions to be equal?

STRUCTURAL EQUIVALENCE says two types are the same if they are made up of the same basic types and constructors.

NAME EQUIVALENCE says two types are the same if their constituents have the SAME NAMES.

Structural Equivalence

```
boolean sequiv( s, t )
{
    if s and t are the same basic type
        return TRUE;
    else if s == array( s1, s2 ) and t == array( t1, t2 )
        return sequiv( s1, t1 ) and sequiv( s2, t2 )
    else s == s1 x s2 and t = t1 x t2 then
        return sequiv( s1, t1 ) and sequiv( s2, t2 )
    else if s == pointer( s1 ) and t == pointer( t1 )
        return sequiv( s1, t1 )
    else if s == s1 → s2 and t == t1 → t2 then
        return sequiv( s1, t1 ) and sequiv( s2, t2 )
    return false
}
```

Try: int foo(int, float)

Relaxing structural equivalence

We don't always want strict structural equivalence.

E.g. for arrays, we want to write functions that accept arrays of any length.

To accomplish this, we would modify `sequiv()` to accept any bounds:

...

```
else if s == array( s1, s2 ) and t == array( t1, t2 )  
    return sequiv( s2, t2 )
```

...

Encoding types

Recursive routines are very slow.

Recursive type checking routines increase the compiler's run time.

In the compilers of the 1970's and 1980's, compilers took too long time to run.

So designers came up with ENCODINGS for types that allowed for faster type checking.

See Example 6.1 in the text.

Name equivalence

Most languages allow association of names with type expressions. This makes type equivalence trickier.

Example from Pascal:

```
type link = ↑ cell;
```

```
var next: link;
```

```
    last: link;
```

```
    p: ↑ cell;
```

```
    q,r: ↑ cell;
```

Do next, last, p, q, and r have the same type?

In Pascal, it depends on the implementation!

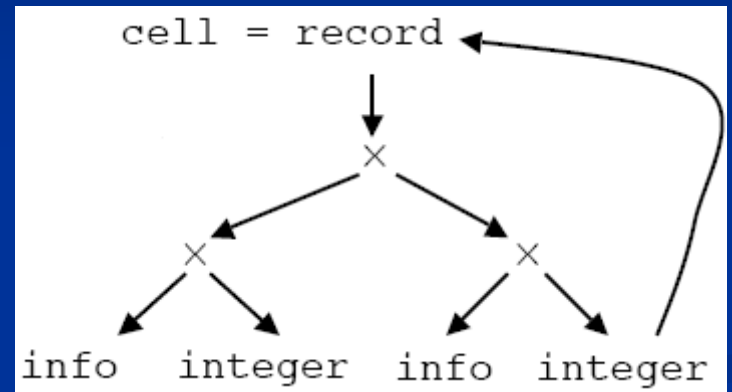
In structural equivalence, the types would be the same.

But NAME EQUIVALENCE requires identical NAMES.

Handling cyclic types

Suppose we had the Pascal declaration

```
type link = ↑ cell;  
cell = record  
    info: integer;  
    next: link;  
end;
```



The declaration of cell contains itself (via the next pointer).
The graph for this type therefore contains a cycle.

Cyclic types

The situation in C is slightly different, since it is impossible to refer to an undeclared name.

```
typedef struct _cell {  
    int info;  
    struct _cell *next;  
} cell;  
typedef *cell link;
```

But the name link is just shorthand for
(struct _cell *).

C uses name equivalence for structs to avoid recursion
(after expanding typedef's).

But it uses structural equivalence elsewhere.

Type conversion

Suppose we encounter an expression $x+i$ where x has type float and i has type int.

CPU instructions for addition could take EITHER float OR int as operands, but not a mix.

This means the compiler must sometimes convert the operands of arithmetic expressions to ensure that operands are consistent with operators.

With postfix as an intermediate language for expressions, we could express the conversion as follows:

$x\ i\ \text{intto real float}+$

where $\text{real}+$ is the floating point addition operation.

Type coercion

If type conversion is done by the compiler without the programmer requesting it, it is called **IMPLICIT** conversion or type **COERCION**.

EXPLICIT conversions are those that the programmer specifies, e.g.

```
x = (int)y * 2;
```

Implicit conversion of **CONSTANT** expressions should be done at compile time.

Type checking example with coercion

<u>Production</u>	<u>Semantic Rule</u>
$E \rightarrow \text{num}$	$E.\text{type} := \text{integer}$
$E \rightarrow \text{num} . \text{num}$	$E.\text{type} := \text{real}$
$E \rightarrow \text{id}$	$E.\text{type} := \text{lookup}(\text{id.entry})$
$E \rightarrow E_1 \text{ op } E_2$	$\begin{aligned} E.\text{type} := & \text{if } E_1.\text{type} == \text{integer and } E_2.\text{type} == \text{integer} \\ & \text{then integer} \\ & \text{else if } E_1.\text{type} == \text{integer and } E_2.\text{type} == \text{real} \\ & \text{then real} \\ & \text{else if } E_1.\text{type} == \text{real and } E_2.\text{type} == \text{integer} \\ & \text{then real} \\ & \text{else if } E_1.\text{type} == \text{real and } E_2.\text{type} == \text{real} \\ & \text{then real} \\ & \text{else type_error} \end{aligned}$