Type Checking

MOST OF THE MATERIALS OF THIS PRESENTATION ARE FROM THE DRAGON BOOK.

Beyond Grammar

Determine potential software errors during compile time.

Not necessary, consider Scheme which does not go beyond a straight grammar check.

It generally makes the process of writing code more efficient.

Example

What is wrong with the following code (from author’s slides):

```
fie(a,b,c,d) {
    int a, b, c, d;
     …
}
fee() {
    int f[3],g[0], h, i, j, k;
    char *p;
    fie(h,i,“ab”,j, k);
    k = f * i + j;
    h = g[17];
    printf(“<%s,%s>.\n”,p,q);
    p = 10;
}
```

Example

What is wrong with the following code (from author’s slides):

```
fie(a,b,c,d) {
    int a, b, c, d;
     …
}
fee() {
    int f[3],g[0], h, i, j, k;
    char *p;
    fie(h,i,“ab”,j, k);
    k = f * i + j;
    h = g[17];
    printf(“<%s,%s>.\n”,p,q);
    p = 10;
}
```

- number of args to fie()
- declared g[0], used g[17]
- “ab” is not an int
- wrong dimension on use of f
- undeclared variable q
- 10 is not a character string

Examples

Type Checks:

- Report an error if an operator is applied to an incompatible operand.
- Example: array variable and function variable are added together

Flow-of-control checks:

- Statements that cause flow of control to leave a construct must have some place to transfer control to.
- Example: Break statement in C causes control to leave smallest enclosing while statement. An error occurs if such an enclosing statement does not exist.

Uniqueness checks:

- Example: Labels in case statements must be distinct.

Name-related checks:

- A name may have to appear two or more times.
- Example: In Ada, a loop may have a name that appears at the beginning and end of block.

Position of Type Checker

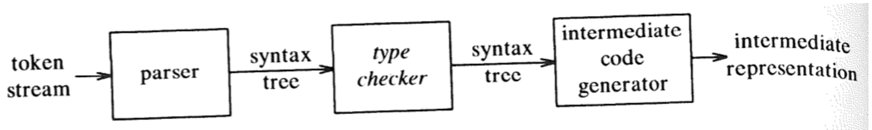


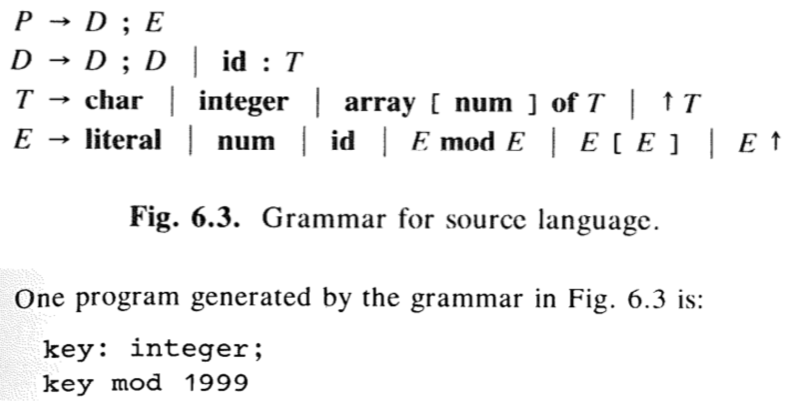
Fig. 6.1. Position of type checker.

Type Systems

A *type system* is a collection of rules for assigning type expressions to the various parts of a program.

A *type checker* implements and type system.

A Simple Language

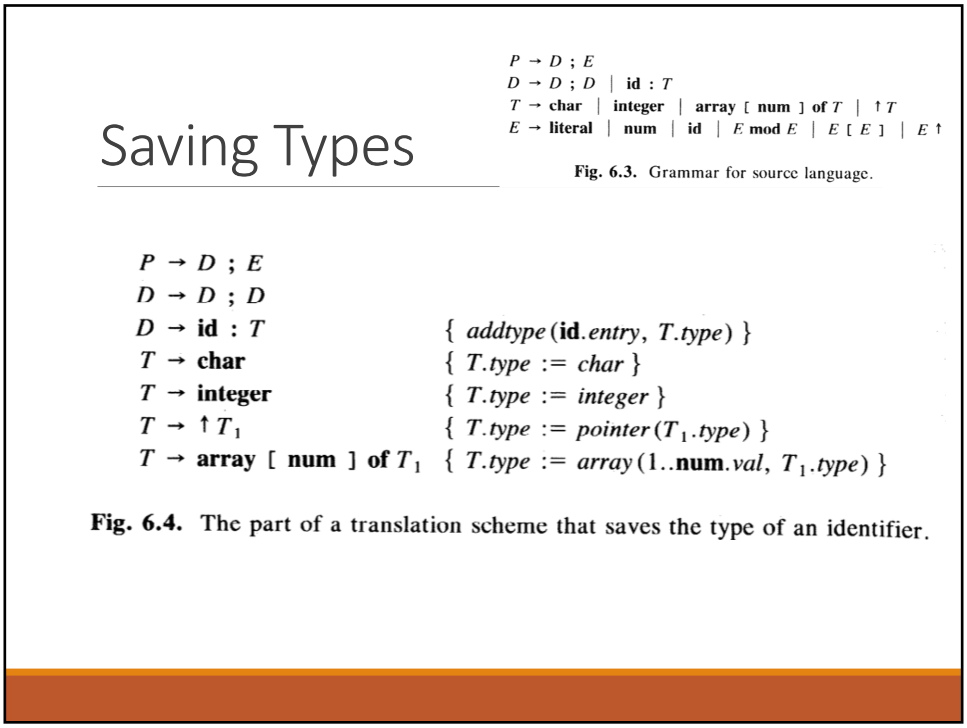


$$P \rightarrow D \; ; \; E$$
$$D \rightarrow D \; ; \; D \mid \textbf{id} : T$$
$$T \rightarrow \textbf{char} \mid \textbf{integer} \mid \textbf{array} \; [\; \textbf{num} \;] \; \textbf{of} \; T \mid \uparrow T$$
$$E \rightarrow \textbf{literal} \mid \textbf{num} \mid \textbf{id} \mid E \; \textbf{mod} \; E \mid E \; [\; E \;] \mid E \uparrow$$

Fig. 6.3. Grammar for source language.

One program generated by the grammar in Fig. 6.3 is:

```
key: integer;
key mod 1999
```



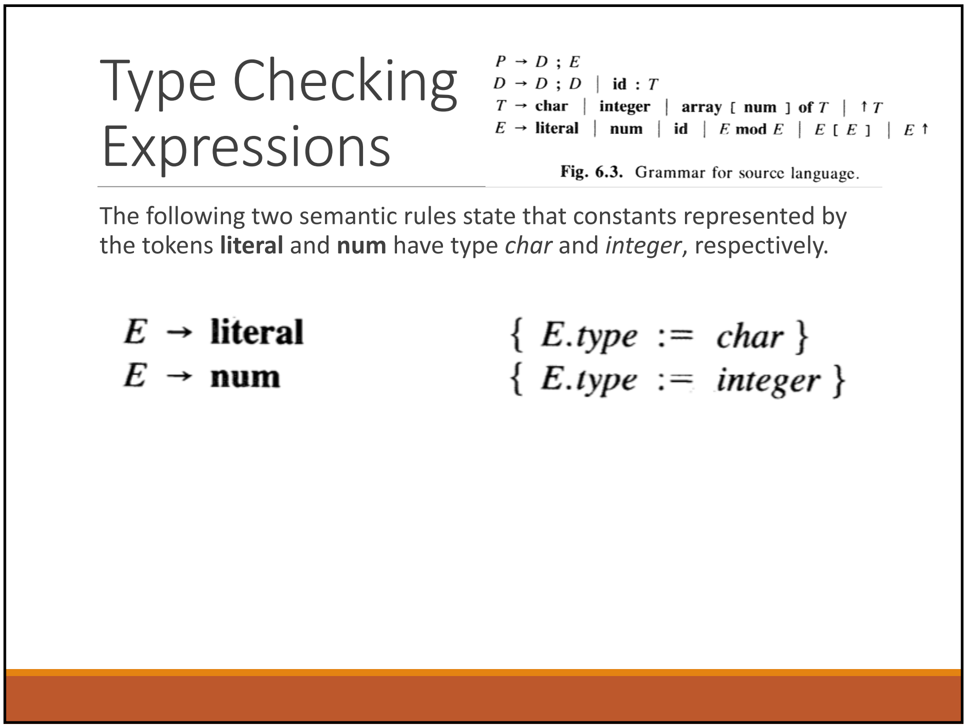
Saving Types

$P \rightarrow D \; ; \; E$
$D \rightarrow D \; ; \; D \;\mid\; \textbf{id} : T$
$T \rightarrow \textbf{char} \;\mid\; \textbf{integer} \;\mid\; \textbf{array} \; [\; \textbf{num} \;] \; \textbf{of} \; T \;\mid\; \uparrow T$
$E \rightarrow \textbf{literal} \;\mid\; \textbf{num} \;\mid\; \textbf{id} \;\mid\; E \; \textbf{mod} \; E \;\mid\; E \; [\; E \;] \;\mid\; E \uparrow$

Fig. 6.3. Grammar for source language.

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

Fig. 6.4. The part of a translation scheme that saves the type of an identifier.



Type Checking Expressions

$P \rightarrow D \; ; \; E$
$D \rightarrow D \; ; \; D \;\mid\; \textbf{id} : T$
$T \rightarrow \textbf{char} \;\mid\; \textbf{integer} \;\mid\; \textbf{array} \; [\; \textbf{num} \;] \; \textbf{of} \; T \;\mid\; \uparrow T$
$E \rightarrow \textbf{literal} \;\mid\; \textbf{num} \;\mid\; \textbf{id} \;\mid\; E \; \textbf{mod} \; E \;\mid\; E \; [\; E \;] \;\mid\; E \uparrow$

Fig. 6.3. Grammar for source language.

The following two semantic rules state that constants represented by the tokens **literal** and **num** have type *char* and *integer*, respectively.

$E \rightarrow \textbf{literal}$	$\{ \; E.type := char \; \}$
$E \rightarrow \textbf{num}$	$\{ \; E.type := integer \; \}$

Type Checking Expressions

$P \to D \; ; \; E$
$D \to D \; ; \; D \mid \textbf{id} : T$
$T \to \textbf{char} \mid \textbf{integer} \mid \textbf{array} \; [\; \textbf{num} \;] \; \textbf{of} \; T \mid \uparrow T$
$E \to \textbf{literal} \mid \textbf{num} \mid \textbf{id} \mid E \; \textbf{mod} \; E \mid E \; [\; E \;] \mid E \uparrow$

Fig. 6.3. Grammar for source language.

We use a lookup(e) function to fetch the type saved in the symbol-table:

$E \to \textbf{id} \qquad \{ \; E.type := lookup(\textbf{id}.entry) \; \}$

Type Checking Expressions

$P \to D \; ; \; E$
$D \to D \; ; \; D \mid \textbf{id} : T$
$T \to \textbf{char} \mid \textbf{integer} \mid \textbf{array} \; [\; \textbf{num} \;] \; \textbf{of} \; T \mid \uparrow T$
$E \to \textbf{literal} \mid \textbf{num} \mid \textbf{id} \mid E \; \textbf{mod} \; E \mid E \; [\; E \;] \mid E \uparrow$

Fig. 6.3. Grammar for source language.

The expression formed by applying the **mod** operator to two subexpressions of type *integer* has a resulting type of *integer*; otherwise, it's a type error.

$$\begin{aligned} E \to E_1 \; \textbf{mod} \; E_2 \qquad \{ \; E.type := \; & \textbf{if} \; E_1.type = integer \; \textbf{and} \\ & \quad E_2.type = integer \; \textbf{then} \; integer \\ & \textbf{else} \; type\_error \; \} \end{aligned}$$

Type Checking Expressions

$$
\begin{aligned}
P &\rightarrow D \;;\; E \\
D &\rightarrow D \;;\; D \;\mid\; \mathbf{id} : T \\
T &\rightarrow \mathbf{char} \;\mid\; \mathbf{integer} \;\mid\; \mathbf{array}\;[\;\mathbf{num}\;]\;\mathbf{of}\;T \;\mid\; \uparrow T \\
E &\rightarrow \mathbf{literal} \;\mid\; \mathbf{num} \;\mid\; \mathbf{id} \;\mid\; E\;\mathbf{mod}\;E \;\mid\; E\;[\;E\;] \;\mid\; E\uparrow
\end{aligned}
$$

Fig. 6.3. Grammar for source language.

In an array reference $E_1[E_2]$, the index expression E_2 must have type *integer*.

The result is the element type obtained from *array(s, t),* where **s** is the range of the indices and **t** is the type of the array elements.

$$
E \rightarrow E_1 \;[\; E_2 \;] \qquad \{\; E.type := \mathbf{if}\; E_2.type = integer \;\mathbf{and}\; E_1.type = array(s,\, t) \;\mathbf{then}\; t \;\mathbf{else}\; type\_error \;\}
$$

Type Checking Expressions

$$
\begin{aligned}
P &\rightarrow D \;;\; E \\
D &\rightarrow D \;;\; D \;\mid\; \mathbf{id} : T \\
T &\rightarrow \mathbf{char} \;\mid\; \mathbf{integer} \;\mid\; \mathbf{array}\;[\;\mathbf{num}\;]\;\mathbf{of}\;T \;\mid\; \uparrow T \\
E &\rightarrow \mathbf{literal} \;\mid\; \mathbf{num} \;\mid\; \mathbf{id} \;\mid\; E\;\mathbf{mod}\;E \;\mid\; E\;[\;E\;] \;\mid\; E\uparrow
\end{aligned}
$$

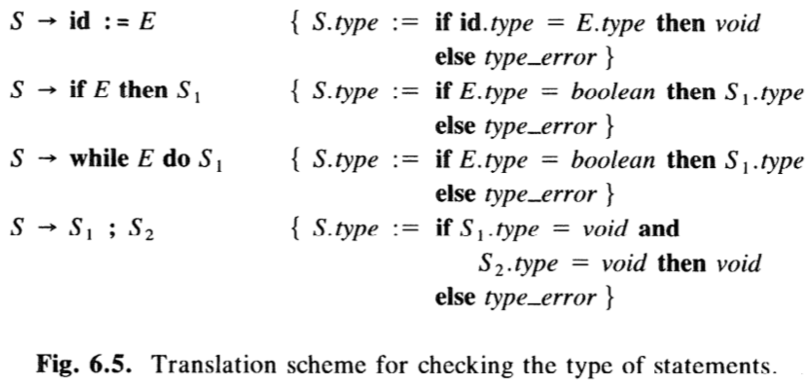
Fig. 6.3. Grammar for source language.

Within expressions, the postfix operator ↑ yields the object pointed to by its operand.

The type of $E\uparrow$ is the type *t* of the object pointed to by the pointer *E*.

$$
E \rightarrow E_1 \uparrow \qquad \{\; E.type := \mathbf{if}\; E_1.type = pointer(t) \;\mathbf{then}\; t \;\mathbf{else}\; type\_error \;\}
$$

Type Checking Statements



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

Fig. 6.5. Translation scheme for checking the type of statements.

Type Checking of Functions

Function application: E -> E(E)

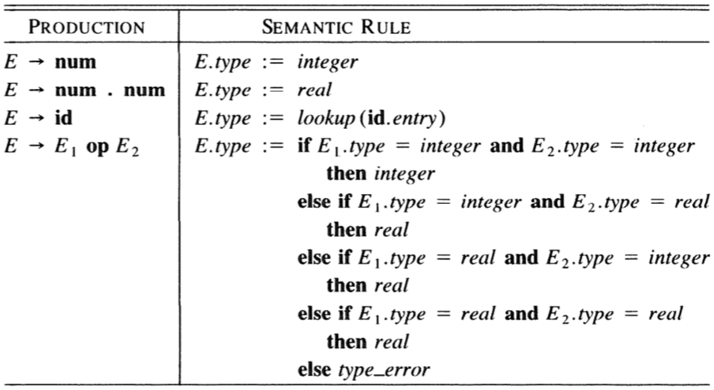
Associating a type with a function:

$T \rightarrow T_1$ '→' T_2 { $T.type := T_1.type \rightarrow T_2.type$ }

Type checking of function application:

$E \rightarrow E_1$ (E_2) { $E.type$:= **if** $E_2.type = s$ **and** $E_1.type = s \rightarrow t$ **then** t **else** $type\_error$ }

Type Coercion



PRODUCTION	SEMANTIC RULE
$E \rightarrow$ **num**	$E.type$:= *integer*
$E \rightarrow$ **num . num**	$E.type$:= *real*
$E \rightarrow$ **id**	$E.type$:= *lookup* (**id**.*entry*)
$E \rightarrow E_1$ **op** E_2	$E.type$:= **if** $E_1.type$ = *integer* **and** $E_2.type$ = *integer* **then** *integer* **else if** $E_1.type$ = *integer* **and** $E_2.type$ = *real* **then** *real* **else if** $E_1.type$ = *real* **and** $E_2.type$ = *integer* **then** *real* **else if** $E_1.type$ = *real* **and** $E_2.type$ = *real* **then** *real* **else** *type_error*

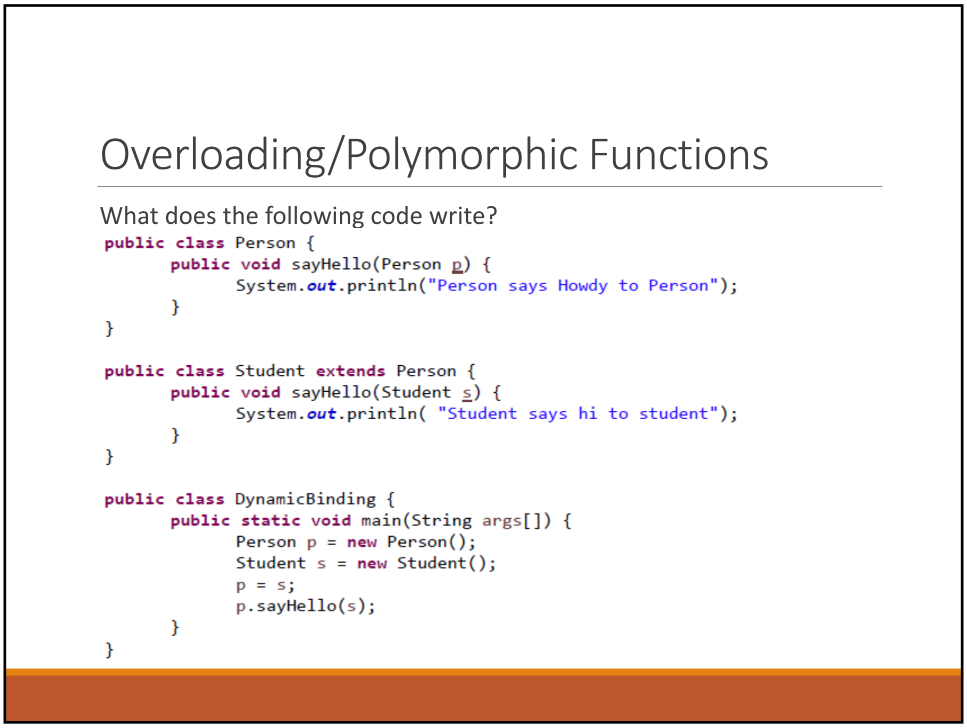
Fig. 6.9. Type-checking rules for coercion from integer to real.

Overloading

Why do we do it?

It is handy, consider the + operator for `int`s and `String`s

To extend functionality in case of polymorphism.



Overloading/Polymorphic Functions

What does the following code write?

```
public class Person {
        public void sayHello(Person p) {
                System.out.println("Person says Howdy to Person");
        }
}

public class Student extends Person {
        public void sayHello(Student s) {
                System.out.println( "Student says hi to student");
        }
}

public class DynamicBinding {
        public static void main(String args[]) {
                Person p = new Person();
                Student s = new Student();
                p = s;
                p.sayHello(s);
        }
}
```