CS301 W'99 Lecture Notes
Lecture 14

Syntax-Directed Type Checking

Consider a simple language of declarations, statements, and expressions.

$P \rightarrow D$ `;` S

Actions for declarations trigger symbol table entries (or synthesize environment attributes):

$D \rightarrow D_1$ `;` D_2	
$D \rightarrow D_1$ `id :` T_1	$\{$ *addtype(*`id`*, T_1.type)* $\}$
$T \rightarrow$ `bool`	$\{$ *T.type := boolean* $\}$
$T \rightarrow$ `int`	$\{$ *T.type := integer* $\}$
$T \rightarrow$ `array[num]of` T_1	$\{$ *T.type := array(*`num`*.val, T_1.type)* $\}$
$T \rightarrow T_1$`*`	$\{$ *T.type := pointer(T_1.type)* $\}$

Actions for expressions **check** for compatible operands and **synthesize** attribute type:

$E \rightarrow$ `num`	$\{$ *E.type := integer* $\}$
$E \rightarrow$ `id`	$\{$ *E.type := lookuptype(*`id`*)* $\}$
$E \rightarrow E_1$ `div` E_2	$\{$ *if E_1.type = integer and E_2.type = integer then E.type := integer else issue type error* $\}$

More Expressions

$E \rightarrow E_1$ `[` E_2 `]`	$\{$ *if E_1.type = array(I,T) and E_2.type = integer then E.type := T else issue type error* $\}$
$E \rightarrow$ `*`E_1	$\{$ *if E_1.type = pointer(T) then E.type = T else issue type error* $\}$
$E \rightarrow E_1$ `<` E_2	$\{$ *if E_1.type = integer and E_2.type = integer then E.type := boolean else issue type error* $\}$
$E \rightarrow E_1$ `or` E_2	$\{$ *if E_1.type = boolean and E_2.type = boolean then E.type := boolean else issue type error* $\}$
$E \rightarrow E_1$ `=` E_2	$\{$ *if (E_1.type = boolean or E_2.type = integer) and E_1.type = E_2.type then E.type := boolean else issue type error* $\}$

As shown, these actions abort immediately if there is an error. Alternatively, can continue to look for more errors; in this case, synthesized value should be chosen to avoid cascades of messages from a single mistake (e.g., last three always return *boolean*).

Checking Statements

In most languages, statements don't have a type, so no point in synthesizing an attribute. Actions just check component types:

$S \rightarrow$ `id :=` E_1	$\{$ *if E_1.type $\neq$ lookuptype(*`id`*) then issue type error* $\}$ *(Must also check that* `id` *is an l-value that can be assigned into.)*
$S \rightarrow$ `if` E_1 `then` S_1	$\{$ *if E_1.type $\neq$ boolean then issue type error* $\}$
$S \rightarrow S_1$ `;` S_2	$\{ \}$

Procedure/Function Definitions and Calls

Describe type of function as $type_1 \times type_2 \times \ldots \times type_n \rightarrow type$

$D \rightarrow$ `id (` F_1 `) :` T_1	$\{$ *addtype(*`id`*,F_1.type $\rightarrow$ T_1.type)* $\}$
$F \rightarrow T_1$	$\{$ *F.type := T_1.type* $\}$
$F \rightarrow T_1$ `,` F_1	$\{$ *F.type := T_1.type $\times$ F_1.type* $\}$
$E \rightarrow$ `id (` A_1 `)`	$\{$ *if lookuptype(*`id`*) = $T_1 \rightarrow T_2$ then if A_1.type = T_1 then E.type := T_2 else issue type error else issue type error* $\}$
$A \rightarrow E_1$	$\{$ *A.type := E_1.type* $\}$
$A \rightarrow E_1$ `,` A_1	$\{$ *A.type := E_1.type $\times$ A_1.type* $\}$

Type Conversions

Implicit conversions (or "**coercions**") occur as a result of applying semantic rules of the language, e.g., perhaps evaluating

```
r + i
```

where `r` is a real and `i` is an integer, causes implicit conversion of the fetched value of `i` to a real before the additon. Note that there's no effect on the permanent contents of `i`.

Explicit conversions (programmer-specified) include operators like `ord` and `chr` in Pascal or casts in C.

Sometimes these just change **type** at compile time and don't affect runtime code, but details may depend on target machine, e.g.,

```
int i;
char *p;
p = (char *) i;
```

generates no code on a SPARC, but changes representation on a CRAY-2, and may move `i` to an "a-register" on M68K.

Type-checking with Conversions

Conversions complicate type-checking (as well as code generation), e.g.:

$E \rightarrow E_1$ + E_2 { *case* (E_1,E_2) *of*
(integer,integer): E.type := integer
(integer,real):
(real,integer):
(real,real): E.type := real
otherwise: issue type error }