CS321 Languages and Compiler Design I

Fall 2010

Lecture 14

Syntax-Directed Type Checking

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

$$P \rightarrow D \; \texttt{;} \; S \qquad \{ \; \textit{S.env} = \textit{D.env}; \; \}$$

Actions for declarations synthesize environment attributes:

$$
\begin{array}{ll}
D \rightarrow \epsilon & \{ \; \textit{D.env} := \textit{empty} \; \} \\
D \rightarrow \texttt{id} \; \texttt{:} \; T_1 \; \texttt{;} \; D_1 & \{ \; \textit{D.env} := \textit{extend}(D_1\textit{.env}, \textit{binding}(\texttt{id}, T_1\textit{.type})) \; \} \\
T \rightarrow \texttt{bool} & \{ \; \textit{T.type} := \textit{boolean} \; \} \\
T \rightarrow \texttt{int} & \{ \; \textit{T.type} := \textit{integer} \; \} \\
T \rightarrow \texttt{array of} \; T_1 & \{ \; \textit{T.type} := \textit{array}(T_1\textit{.type}) \; \} \\
T \rightarrow \texttt{pair} \; T_1 \; T_2 & \{ \; \textit{T.type} := T_1\textit{.type} \times T_2\textit{.type} \; \}
\end{array}
$$

Expressions

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

$E \rightarrow \texttt{num}$ $\{$ *E.type := integer* $\}$

$E \rightarrow \texttt{id}$ $\{$ *E.type := lookup(E.env,*`id`*)* $\}$

$E \rightarrow (E_1, E_2)$ $\{$ E_1*.env = E.env;* E_2*.env = E.env; E.type =* E_1*.type* $\times$ E_2*.type* $\}$

$E \rightarrow E_1$ `div` E_2 $\{$ E_1*.env = E.env;* E_2*.env = E.env;*
if not (E_1*.type = integer and* E_2*.type = integer) then*
issue type error;
E.type := integer $\}$

$E \rightarrow E_1$ `or` E_2 $\{$ E_1*.env = E.env;* E_2*.env = E.env;*
if not (E_1*.type = boolean and* E_2*.type = boolean) then*
issue type error;
E.type := boolean $\}$

Issuing error might or might not stop the checking process. If it doesn't, try to choose a synthesized type value that prevents a cascade of messages from a single mistake.

More Expressions

$E \rightarrow E_1$ [E_2] { E_1.env = E.env; E_2.env = E.env;
if (E_1.type = array(T) and E_2.type = integer) then
E.type := T;
else issue type error }

$E \rightarrow E_1$.fst { E_1.env = E.env;
if $E_1 = T_1 \times T_2$ then
E.type := T_1;
else issue type error;

$E \rightarrow E_1 < E_2$ { E_1.env = E.env; E_2.env = E.env;
if not (E_1.type = integer and E_2.type = integer) then
issue type error;
E.type := boolean }

$E \rightarrow E_1 = E_2$ { E_1.env = E.env; E_2.env = E.env;
if not ((E_1.type = boolean or E_1.type = integer)
and E_1.type = E_2.type) then
issue type error;
E.type := 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	$\{$ *E_1.env = S.env;*
	if E_1.type $\neq$ lookup(S.env,`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	$\{$ *E_1.env = S.env; S_1.env = S.env;*
	if E_1.type $\neq$ boolean then
	issue type error $\}$
$S \rightarrow S_1$ `;` S_2	$\{$ *S_1.env = S.env; S_2.env = S.env;* $\}$

PROCEDURE/FUNCTION DEFINITIONS AND CALLS

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

$D \rightarrow \mathtt{id}\ (\ F_1\ )\ :\ T_1\ ;\ D_1$	$\{$ *D.env := extend(D$_1$.env, binding(*`id`*,F$_1$.type* $\rightarrow$ *T$_1$.type))* $\}$
$F \rightarrow \mathtt{id}\ :\ T_1$	$\{$ *F.type := T$_1$.type* $\}$
$F \rightarrow \mathtt{id}\ :\ T_1\ ,\ F_1$	$\{$ *F.type := T$_1$.type* $\times$ *F$_1$.type* $\}$
$E \rightarrow \mathtt{id}\ (\ A_1\ )$	$\{$ *A$_1$.env = E.env;* *if lookup(E.env,*`id`*) = T$_1$* $\rightarrow$ *T$_2$ then* *if A$_1$.type* $\neq$ *T$_1$ then* *issue type error* *E.type := T$_2$* *else* *issue type error* $\}$
$A \rightarrow E_1$	$\{$ *E$_1$.env = A.env;* *A.type := E$_1$.type* $\}$
$A \rightarrow E_1\ ,\ A_1$	$\{$ *E$_1$.env = A.env;* *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. This complicates type-checking:

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

The relationship between integer and real is a special case of **subtyping** (more later).

TYPE EQUIVALENCE

When do two identifiers have the “same” type, or “compatible” types?

E.g., if `a` has type t_1, `b` has type t_2 and `f` has type $t_2 \rightarrow t_3$, how must t_1 and t_2 be related for these to make sense?

```
a := b
f (a)
```

To maintain **type safety** we must insist at a minimum that t_1 and t_2 are **structurally equivalent**.

Structural equivalence is defined inductively:

- Primitive types are equivalent iff they are exactly the same type.
- Cartesian product types are equivalent if their corresponding component types are equivalent. (Record field names are typically ignored.)
- Disjoint union types are equivalent if their corresponding component types are equivalent.
- Mapping types (arrays and functions) are the same if their domain and range types are the same.

Equivalence (continued)

Another way to say this: two types are equal if they have the same set of values.

Recursive types are a challenge. Are these two types structurally equivalent?

```
type t1 = { a:int, b: POINTER TO t1 };
type t2 = { a:int, b: POINTER TO t2 };
```

Intuitively yes, but it's (a little) tricky for a type-checking algorithm to determine this!

TYPE NAMES

Question of equivalence is more interesting if language has type **names**, which arise for two main reasons:

- As a convenient shorthand to avoid giving the full type each time. E.g.,

```
function f(x:int * bool * real) : int * bool * real = ...
type t = int * bool * real
function f(x:t) : t = ...
```

- As a way of improving program correctness by subdividing values into types according to their meaning **within the program**.

```
type polar = { r:real, a:real };
type rect = { x:real, y:real };
function polar_add(x:polar,y:polar) : polar ...
function rect_add(x:rect,y:rect) : rect ...
var a:polar; c:rect;
a := (150.0,30.0) (* ok *)
polar_add(a,a)  (* ok *)
c := a  (* type error *)
rect_add(a,c) (* type error *)
```

For this to be useful, some structurally equivalent types must be treated as **inequivalent**.

NAME EQUIVALENCE

Simplistic idea: Two types are equivalent iff they have the same **name**.

Supports `polar`/`rect` distinction.

But pure name equivalence is very restrictive, e.g.:

```
type ftemp = real
type ctemp = real
var x:ftemp, y:ftemp, z: ctemp;
x := y; (* ok *)
x := 10.0; (* probably ok *)
x := z; (* type error *)
x := 1.8 * z + 32.0; (* probably type error *)
```

Different types now seem **too** distinct; can't even convert from one form of real to another.

NAME EQUIVALENCE (CONTINUED)

Also: what about unnamed type expressions?

```
type t = int * int
procedure f(x: int * int) = ...
procedure g(x: t) = ...
var a:t = (3,4)
g(a); (* ok *)
f(a); (* ok or not ?? *)
```

Because of these problems with pure name equivalence, most languages use **mixed** solutions.

C Type Equivalence

C uses structural equivalence for array and function types, but name equivalence for `struct`, `union`, and `enum` types. For example:

```
char a[100];
void f(char b[]);
f(a); /* ok */

struct polar{float x; float y;};
struct rect{float x; float y;};
struct polar a;
struct rect b;
a = b; /* type error */
```

A type defined by a `typedef` declaration is actually just an abbreviation for an existing type.

Note that this policy makes it easy to check equivalence of recursive types, which can only be built using `struct`s.

```
struct fred {int x; struct fred *y;} a;
struct bill {int x; struct fred *y;} b;
a = b; /* type error */
```

ML Type Equivalence

ML uses structural equivalence, except that each `datatype` declaration creates a new type unlike all others.

```
datatype polar = POLAR of real * real
datatype rect = RECT of real * real
val a = POLAR(1.0,2.0)  and  b = RECT(1.0,2.0)
if (a = b) ...  (* type error *)
```

Note that the mandatory use of constructors makes it possible to uniquely identify the types of literals.

Note that a datatype need not declare a record:

```
datatype fahrenheit = F of real
datatype celsius = C of real
val a = F 150.0  and  b = C 150.0
if (a = b) ... (* type error *)
fun convert(F x) = C(1.8 * x + 32.0) (* ok *)
```

For type abbreviation, ML offers the `type` declaration, which simply gives a new name for an existing type.

```
type centigrade = celsius
fun g(x:centigrade) = if x = b ... (* ok *)
```

JAVA TYPE EQUIVALENCE

Java uses nearly strict name equivalence, where names are either:

- One of eight built-in **primitive** types (`int`,`float`,`boolean`, etc.), or
- Declared classes or interfaces (**reference** types).

The only non-trivial type expressions that can appear in a source program are **array** types, which are compared structurally, using name equivalence for the ultimate element type. Java has no mechanism for type abbreviations.

Java types form a **subtyping** hierarchy:

- If class `A` extends class `B`, then `A` is a subtype of `B`.
- If class `A` implements interface `I`, then `A` is a subtype of `I`.
- If numeric type `t` can be coerced to numeric type `u` without loss of precision, then `t` is a subtype of `u`.

If T_1 is a subtype of T_2, then a value of type T_1 can be used wherever a value of T_2 is expected.