CS558 Programming Languages
Fall 2017
Lecture 8b
TYPE EQUIVALENCE

When do two identifiers have the “same” type, or “compatible” types? We need to know this to do static type checking.

- e.g., if \( x \) has type \( t_1 \) and \( e \) has type \( t_2 \), when does it make sense to allow the assignment \( x := e \)?

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

- Two types are structurally equivalent if they each describe the same set of values.

In languages that define type names, we may instead require that \( t_1 \) and \( t_2 \) be name-equivalent.

- Two types are name-equivalent if they have identical names.
- Name equivalence implies structural equivalence, but not vice-versa.

In some languages, values can be given more than one type due to subtyping, which can also be defined using structural or nominal criteria. Then we insist that \( t_2 \) be a subtype of \( t_1 \).
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 may or may not matter.)
- 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.
- Recursive types are a challenge. Are these two types structurally equivalent?

```plaintext
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!
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, e.g.

  ```
  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 *)
  ```

Whole idea here is that some structurally equivalent are treated as *inequivalent*. 
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.:

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

```c
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 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 `structs`.

```c
struct fred {int x; struct fred *y;} a;
struct bill {int x; struct fred *y;} b;
a = b; /* type error */
```
ML uses structural equivalence, except that each datatypedeclaration creates a new type unlike all others.

```ml
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 datatypeneed not declare a record:

```ml
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.

```ml
type centigrade = celsius
fun g(x:centigrade) = if x = b ... (* ok *)
```
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.
Java uses nominal subtyping, but we can also define structural subtyping based on the idea of maintaining type safety.

Let’s model object interfaces by record types with named fields and methods (using ML-like notation).

Example

```plaintext
type Shape =
  { origin: int * int,
    encloses: Shape,
    intersectWith: Shape -> Shape }

type Line =
  { origin: int * int,
    angle: int, length: int,
    encloses: Line,
    intersectWith: Shape -> Line }
```

We write \( t' <: t \) to mean \( t' \) is a subtype of \( t \). For example, we might expect that \( \text{Line} <: \text{Shape} \).
We extend our language’s type system with a subsumption rule:

\[
\frac{TE \vdash e : t' \quad t' <: t}{TE \vdash e : t} \quad \text{(Sub)}
\]

Now we must define the <: relation so that subsumption is a sound typing rule.

We assume

\[
\frac{}{t <: t} \quad \text{(Reflexive)}
\]

\[
\frac{t'' <: t' \quad t' <: t}{t'' <: t} \quad \text{(Transitive)}
\]
MORE SUBTYPING RULES

Two easy rules for records:

\[
\{l_1 : t_1, \ldots , l_n : t_n, l : t\} <: \{l_1 : t_1, \ldots , l_n : t_n\} \quad (W)
\]

\[
\{l_1 : t_1, \ldots , l_k : t_k', \ldots , l_n : t_n\} <: \{l_1 : t_1, \ldots , l_k : t_k, \ldots , l_n : t_n\} \quad (D)
\]

The (Width) rule says that we can always make a subtype by adding fields to a record. The (Depth) rule says that we can always make a subtype by replacing the type of any field with a subtype. (Also, we assume that order in records doesn’t matter.)
Making a sound rule for function subtyping is harder. A first guess:

\[
\frac{t'_1 <: t_1 \quad t'_2 <: t_2}{t'_1 \rightarrow t'_2 <: t_1 \rightarrow t_2} \quad \text{(wrong!)}
\]

But this would imply, for example, that given

\[
f : \text{Shape} \rightarrow \text{Shape}
g : \text{Line} \rightarrow \text{Line}
\]

then \(\text{typeof}(g) <: \text{typeof}(f)\), so we should be able to replace any use of \(f\) by \(g\) without destroying type-correctness.

Suppose we have \(x = f(y)\), so \(x\) and \(y\) both have type \(\text{Shape}\). If we try to replace \(f\) by \(g\):

- The result value of \(g\) is a \(\text{Line}\), which is a subtype of \(\text{Shape}\), so that’s ok to bind to \(x\).
- But the argument value of \(g\) must be a \(\text{Line}\), and \(y\) is only guaranteed to be a \(\text{Shape}\)!
The rule we really want is:

\[
\frac{t_1 <: t_1' \quad t_2' <: t_2}{t_1' \rightarrow t_2' <: t_1 \rightarrow t_2} \quad \text{(Fun)}
\]

For example, given

\[
\begin{align*}
  f &: \text{Shape} \rightarrow \text{Shape} \\
  h &: \text{Object} \rightarrow \text{Line}
\end{align*}
\]

(where \text{Object} = \{\} is the universal supertype), we could validly replace \( f \) by \( h \).

We say that the \( <: \) relation is \textbf{covariant} on the result type of functions, and \textbf{contravariant} on the argument type.
What about subtyping for mutable fields and array elements?

When can we safely replace a mutable field $x : t_x$ by a field $y : t_y$? A mutable field can appear in two contexts:

- It can be stored into, e.g. $x := e$. To safely replace $x$ by $y$, $y$ must be able to hold any value that $x$ can, i.e., we must have $t_x <: t_y$.

- It can be fetched from, e.g. $z := x$. To safely replace $x$ by $y$, the value in $y$ must be containable by $z$, which expects a value from $x$, i.e., we must have $t_y <: t_x$.

Combining these requirements, we see that $t_x$ and $t_y$ must be equal. So no non-trivial subtyping should be permitted on mutable fields and array elements.
Java appears to break this rule. If $B$ is a subtype of $A$, then $B[]$ is treated as a subtype of $A[]$. This is fine when fetching, but can be unsound when storing, because it allows an $A$ value to be stored as an element of an $B[]$, e.g.

```java
B[] bx = new B[100];
A[] ax = bx;  // permitted because $B[] <: A[]$
ax[0] = new A()  // oops!;
```

To guard against this, Java inserts an (expensive) runtime check on every array store operation to make sure that the stored value actually belongs to the same class as the array.