CS410/510
Compiling Advanced Languages
Spr2000 Lecture Notes
Subtyping

Typing rules for records and sums

If our IR is to be strongly typed, we need to add new typing rules for records and sums. As usual, the goal is to rule out programs that do “bad” things.

For records, the main “bad” things are:

- building a record with duplicate labels; or
- trying to select a non-existent label from a record.

$$\frac{E \triangleright M_i : \tau_i \quad (l_i \ \mathit{distinct})}{E \triangleright \{l_1 \texttt{ = } M_1 \texttt{,} \ldots \texttt{,} l_n = M_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \text{(RECORD)}$$

$$\frac{E \triangleright M : \{\ldots, l : \tau, \ldots\}}{E \triangleright M \texttt{.} l : \tau} \text{(SELECT)}$$

For sums (assuming no default arm), the “bad” things are:

- trying to switch on a labeled value for which there is no arm;
- giving multiple switch arms for the same label;

Also, we want all switch arms to have the same type.

$$\frac{E \triangleright M : \tau}{E \triangleright \texttt{<} l \texttt{=} M \texttt{>} :< l : \tau >} \text{(LABEL)}$$

$$\frac{M :< l_1 : \tau_1, \ldots, l_n : \tau_n > \quad E \triangleright M_i : \tau_i \to \tau \quad (l_i \ \mathit{distinct})}{E \triangleright \texttt{switch } M \texttt{ of } l_1 \texttt{ => } M_1 \texttt{ | } \ldots \texttt{ | } l_n \texttt{ => } M_n : \tau} \text{(SWITCH}$$

Subtyping

These rules are sound, but can be irritatingly restrictive. For example, we cannot type a term like

```
(λ r:{x:int}.  r.x) {x=0, y=1}
```

even though evaluating it obviously won't cause anything "bad" to happen. The point is that a record containing both `x` and `y` fields can be **safely substituted** in any context that requires a record containing just an `x` field.

To formalize this, we introduce the idea of **subtyping**.

We write $\tau' <: \tau$, pronounced "τ' is a subtype of τ," to mean that "a τ' is just as good as a τ" (or, "every τ' is also a τ") in any context.

The proper definition of the $<:$ relation depends on the particular set of typing rules we're interested in, and we often some flexiblity in the definition. But in all cases, we must make sure that $<:$ is defined so that the **(typing) rule of subsumption** is sound:

$$\frac{E \rhd M : \tau' \qquad \tau' <: \tau}{E \rhd M : \tau} \text{(SUBSUMPTION)}$$

In principle, to add subtyping to our existing typing system, we just add this rule and define the subtyping relation. (In practice, we must do more in order to build a practical type **checker** for the new system.)

Defining the Subtype Relation

We can consider the proper definition of $<:$ separately on each type constructor.

A **record** A can substitute for another record B if

• A has all the same labels as B (and possibly more) – not necessarily in the same order; and

• Each field of A holds a subtype of the corresponding field in B.

These considerations suggest the following subtyping rule:

$$\frac{\{l_1, \ldots, l_n\} \subseteq \{k_1, \ldots, k_m\} \quad k_j = l_i \Rightarrow \tau'_j <: \tau_i}{\{k_1 : \tau'_1, \ldots, k_m : \tau'_m\} <: \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \text{(SUB-RECORD)}$$

A **summand** A can substitute for another summand B if

• Any label that A might have is one that B might also have.

• The type of value carried by each possible label for A is a subtype of the corresponding type of value for B.

$$\frac{\{k_1, \ldots, k_m\} \subseteq \{l_1, \ldots, l_n\} \quad k_j = l_i \Rightarrow \tau'_j <: \tau_i}{< k_1 : \tau'_1, \ldots, k_m : \tau'_m > \quad <: \quad < l_1 : \tau_1, \ldots, l_n : \tau_n >} \text{(SUB-SUM)}$$

Depending on the definitions of primitive constants, further subtype relationships might hold on **base types** , e.g.,

$$\texttt{Nat} <: \texttt{Int}$$

More Subtyping

Functions are trickier: a function A can substitute for a function B if A can handle any argument that might be passed to B, and A only returns results that might be returned by B.

$$\frac{\tau_1 <: \mu_1 \quad \mu_2 <: \tau_2}{\mu_1 \to \mu_2 \ <: \ \tau_1 \to \tau_2} \text{(SUB-ARROW)}$$

Note that the sense of the subtype relation is **reversed** on the argument position, but maintained on the result position; so $\to$ is said to be **contravariant** on its left argument and **covariant** on its right.

As we'll see, this contravariance phenomenon causes problems!

We ordinarily complete our definition of the subtype relation by taking the **reflexive**, **transitive** closure of the constructor-specific rules:

$$\tau <: \tau$$

$$\frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

(In fact, these may be provable from the existing laws.)

So defined, $<:$ is a **partial order**.

Subtyping of References

Finally, consider **mutable** records, say with just one element (i.e., ML **ref**'s). What kind of subtyping can we permit? We must consider both retrieval and update operations.

A reference A can safely substitute for a reference B only if:

• Any value that can be read from A could also be read from B (i.e., the type of values held in A is a subtype of those held in B).

• Any value that could be stored into B can also be stored into A (i.e., the type of values held in B is a subtype of the those held in A)

This implies that

$$\frac{\tau' = \tau}{\texttt{ref}\ \tau' <: \texttt{ref}\ \tau}\ \text{(SUB-REF)}$$

In other words, there is no useful subtyping relation for **ref**, which is said to be **invariant** on its argument.

The same phenomenon extends to arrays.

Object-oriented Languages and Subtyping

In class-based OO languages, it is common that explicit **subclassing** implies subtyping. In addition, there may be other ways to establish subtypes. For example, in Java, if we have

```
class C { ... };
interface I { ... };

class C1 extends C implements I { ... };
```

then an object of class `C1` can be used wherever an object of class `C` or of interface `I` is expected, i.e., loosely speaking

`C1` $<:$ `C`

`C` $<:$ `I`

Fundamentally, objects are implemented by **records**, so a safe type system for an OO language requires that record subtyping rules be respected.

Class **extension** (adding new fields or methods) is straightforward: it just corresponds to adding new labels in a subtype.

Overriding of class fields or methods is more problematic, because of contravariance problems.

• Since **fields** are (normally) mutable, their types need to be invariant in any subclass. In languages like Java this is expressed as an inability to override field definitions at all (since overriding with a field of the **same** type accomplishes nothing).

More Subclasses and Subtypes

- When overriding a function in class `C`

`Bar f(Foo x)`

by a function in subclass `C1`

`Bar1 f(Foo1 x)`

we must have that `Foo1` is a **supertype** of `Foo` and `Bar1` is a **subtype** of `Bar`. Since argument contravariance is seldom useful, most OO languages (including C++ and Java) require that overriding functions take **identical** argument types. Java requires the same of the result type, but C++ allows subclassing on the result.

- Java **arrays** are **covariant**! That is, the following code is legal:

```
class C1 extends C {...};
foo (C x[]) {
     C z = x[10];
     C w = ...;
     x[20] = w; }
C1 y[] = ...;
foo(y);
```

While the array fetch is fine (the result will be a `C1` and hence a `C`), the store is dangerous: `w` is a `C`, but not necessarily a `C1`!

To avoid this danger, Java must insert a **runtime check** on every array store to make sure that the value being stored belongs to a subclass of the actual array type.