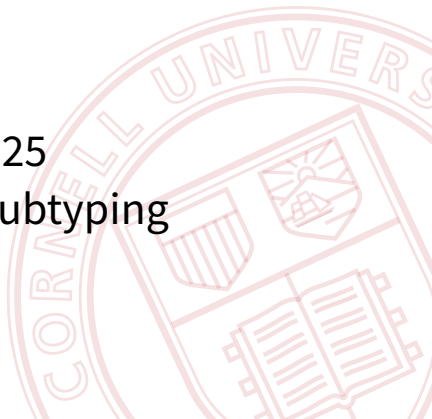# CS 4110

Programming Languages & Logics

Lecture 25
Records and Subtyping

Records

We've seen binary products (pairs), and they generalize to *n*-ary products (tuples).

Records are a generalization of tuples where we mark each field with a label.

Records

We've seen binary products (pairs), and they generalize to *n*-ary products (tuples).

Records are a generalization of tuples where we mark each field with a label.

Example:
$$\{\mathsf{foo} = 32, \mathsf{bar} = \mathsf{true}\}$$
is a record value with an integer field foo and a boolean field bar.

Records

We've seen binary products (pairs), and they generalize to *n*-ary products (tuples).

Records are a generalization of tuples where we mark each field with a label.

Example:
$$\{\text{foo} = 32, \text{bar} = \text{true}\}$$
is a record value with an integer field foo and a boolean field bar.

Its type is:
$$\{\text{foo} : \textbf{int}, \text{bar} : \textbf{bool}\}$$

$$l \in \mathcal{L}$$

$$e ::= \cdots \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l$$

$$v ::= \cdots \mid \{l_1 = v_1, \ldots, l_n = v_n\}$$

$$\tau ::= \cdots \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$

$$E ::= \ldots$$
$$| \ \{l_1 = v_1, \ldots, l_{i-1} = v_{i-1}, l_i = E, l_{i+1} = e_{i+1}, \ldots, l_n = e_n\}$$
$$| \ E.l$$

$$\overline{\{l_1 = v_1, \ldots, l_n = v_n\}.l_i \rightarrow v_i}$$

Static Semantics

$$\frac{\forall i \in 1..n. \quad \Gamma \vdash e_i : \tau_i}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

$$\frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i}$$

Example

$$\text{GETX} \triangleq \lambda p : \{x : \textbf{int}, y : \textbf{int}\}. \, p.x$$

Example

$$\text{GETX} \triangleq \lambda p : \{x : \textbf{int}, y : \textbf{int}\}. \, p.x$$

$$\text{GETX} \, \{x = 4, y = 2\}$$

Example

$$\text{GETX} \triangleq \lambda p : \{x : \textbf{int}, y : \textbf{int}\}. \, p.x$$

$$\text{GETX} \, \{x = 4, y = 2\}$$

$$\text{GETX} \, \{x = 4, y = 2, z = 42\}$$

Example

$$\text{GETX} \triangleq \lambda p : \{x : \textbf{int}, y : \textbf{int}\}.\, p.x$$

$$\text{GETX} \{x = 4, y = 2\}$$

$$\text{GETX} \{x = 4, y = 2, z = 42\}$$

$$\text{GETX} \{y = 2, x = 4\}$$

Subtyping

Definition (Subtype)

τ_1 is a *subtype* of τ_2, written $\tau_1 \leq \tau_2$, if a program can use a value of type τ_1 whenever it would use a value of type τ_2.

If $\tau_1 \leq \tau_2$, we also say τ_2 is the *supertype* of τ_1.

Subtyping

Definition (Subtype)

τ_1 is a *subtype* of τ_2, written $\tau_1 \leq \tau_2$, if a program can use a value of type τ_1 whenever it would use a value of type τ_2.

If $\tau_1 \leq \tau_2$, we also say τ_2 is the *supertype* of τ_1.

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'} \text{ SUBSUMPTION}$$

This typing rule says that if e has type τ and τ is a subtype of τ', then e also has type τ'.

Record Subtyping

We'll define a new subtyping relation that works together with the subsumption rule.

$$\tau_1 \leq \tau_2$$

Record Subtyping

This program isn't well-typed (yet):

$$(\lambda p \colon \{x \colon \textbf{int}\}.\, p.x) \; \{x = 4, y = 2\}$$

Record Subtyping

This program isn't well-typed (yet):

$$(\lambda p : \{x : \textbf{int}\}. \, p.x) \, \{x = 4, y = 2\}$$

So let's add width subtyping:

$$\frac{k \geq 0}{\{l_1 : \tau_1, \ldots, l_{n+k} : \tau_{n+k}\} \leq \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

Record Subtyping

This program also doesn't get stuck:

$$(\lambda p : \{x : \textbf{int}, y : \textbf{int}\}. \, p.x + p.y) \, \{y = 37, x = 5\}$$

Record Subtyping

This program also doesn't get stuck:

$$(\lambda p : \{x : \textbf{int}, y : \textbf{int}\}. \ p.x + p.y) \ \{y = 37, x = 5\}$$

So we can make it well-typed by adding permutation subtyping:

$$\frac{\pi \text{ is a permutation on } 1..n}{\{l_1 : \tau_1, \ldots, l_n : \tau_n\} \leq \{l_{\pi(1)} : \tau_{\pi(1)}, \ldots, l_{\pi(n)} : \tau_{\pi(n)}\}}$$

Record Subtyping

Does this program get stuck? Is it well-typed?

$$(\lambda p \colon \{x : \{y : \mathbf{int}\}\}. \, p.x.y) \, \{x = \{y = 4, z = 2\}\}$$

Record Subtyping

Does this program get stuck? Is it well-typed?

$$(\lambda p : \{x : \{y : \textbf{int}\}\}. \, p.x.y) \, \{x = \{y = 4, z = 2\}\}$$

Let's add depth subtyping:

$$\frac{\forall i \in 1..n. \quad \tau_i \leq \tau_i'}{\{l_1 : \tau_1, \ldots, l_n : \tau_n\} \leq \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}$$

Record Subtyping

Putting all three forms of record subtyping together:

$$\frac{\forall i \in 1..n.\ \exists j \in 1..m.\quad l'_i = l_j\ \wedge\ \tau_j \leq \tau'_i}{\{l_1 : \tau_1, \ldots, l_m : \tau_m\} \leq \{l'_1 : \tau'_1, \ldots, l'_n : \tau'_n\}}\ \text{S-Record}$$

Standard Subtyping Rules

We always make the subtyping relation both reflexive and transitive.

$$\frac{}{\tau \leq \tau} \text{ S-Refl} \qquad\qquad \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \text{ S-Trans}$$

Think of every type describing a set of values. Then $\tau_1 \leq \tau_2$ when τ_1's values are a subset of τ_2's.

Top Type

It's sometimes useful to define a *maximal* type with respect to subtyping:

$$\tau ::= \cdots \mid \top$$

$$\frac{}{\tau \leq \top} \text{ S-Top}$$

Everything is a subtype of $\top$, as in Java's `Object` or Go's `interface{}`.

Subtype All the Things!

We can also write subtyping rules for sums and products:

$$\frac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2'}{\tau_1 + \tau_2 \leq \tau_1' + \tau_2'} \ \text{S-Sum}$$

Subtype All the Things!

We can also write subtyping rules for sums and products:

$$\frac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2'}{\tau_1 + \tau_2 \leq \tau_1' + \tau_2'} \text{ S-Sum}$$

$$\frac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2'}{\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'} \text{ S-Product}$$

Function Types

How should we decide whether one function type is a subtype of another?

$$\frac{???}{\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2} \text{ S-Function}$$

We'd like to have:

$$\textbf{int} \rightarrow \{x\!:\!\textbf{int}, y\!:\!\textbf{int}\} \leq \textbf{int} \rightarrow \{x\!:\!\textbf{int}\}$$

Desiderata

We'd like to have:

$$\textbf{int} \rightarrow \{x\!:\!\textbf{int}, y\!:\!\textbf{int}\} \leq \textbf{int} \rightarrow \{x\!:\!\textbf{int}\}$$

And:

$$\{x\!:\!\textbf{int}\} \rightarrow \textbf{int} \leq \{x\!:\!\textbf{int}, y\!:\!\textbf{int}\} \rightarrow \textbf{int}$$

Desiderata

We'd like to have:

$$\textbf{int} \to \{x : \textbf{int}, y : \textbf{int}\} \leq \textbf{int} \to \{x : \textbf{int}\}$$

And:

$$\{x : \textbf{int}\} \to \textbf{int} \leq \{x : \textbf{int}, y : \textbf{int}\} \to \textbf{int}$$

In general, to prove:

$$\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'$$

we'll require:

- Argument types are contravariant: $\tau_1' \leq \tau_1$
- Return types are covariant: $\tau_2 \leq \tau_2'$

Function Subtyping

Putting these two pieces together, we get the subtyping rule for function types:

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'} \text{ S-Function}$$

Reference Subtyping

What should the relationship be between τ and τ' in order to have τ **ref** $\leq \tau'$ **ref**?

Example

If r' has type τ' **ref**, then $!r'$ has type τ'.

Imagine we replace r' with r, where r has a type τ **ref** that we've somehow decided is a subtype of τ' **ref**.

Example

If r' has type τ' **ref**, then $!r'$ has type τ'.

Imagine we replace r' with r, where r has a type τ **ref** that we've somehow decided is a subtype of τ' **ref**.

Then $!r$ should still produce something can be treated as a τ'. In other words, it should have a type that is a *subtype* of τ'.

So the referent type should be covariant:

$$\frac{\tau \leq \tau'}{\tau \text{ ref} \leq \tau' \text{ ref}}$$

Example

If v has type τ', then $r' := v'$ should be legal.

If we replace r' with r, then it must still be legal to assign $r := v$.
So $!r$ would then produce a value of type τ'.

Example

If v has type τ', then $r' := v'$ should be legal.

If we replace r' with r, then it must still be legal to assign $r := v$.
So $!r$ would then produce a value of type τ'.

So the referent type should be contravariant!

$$\frac{\tau' \leq \tau}{\tau \text{ ref} \leq \tau' \text{ ref}}$$

Reference Subtyping

In fact, subtyping for reference types must be *invariant:* a reference type τ **ref** is a subtype of τ' **ref** if and only if $\tau \leq \tau'$ and $\tau' \leq \tau$.

$$\frac{\tau \leq \tau' \quad \tau' \leq \tau}{\tau \text{ ref} \leq \tau' \text{ ref}} \text{ S-Ref}$$

Java Arrays

Tragically, Java's mutable arrays use covariant subtyping!

Java Arrays

Tragically, Java's mutable arrays use covariant subtyping!

Suppose that Cow is a subtype of Animal.

Code that only reads from arrays typechecks:

```
Animal[ ] arr  =  new Cow[ ] { new Cow("Alfonso") };
Animal a  =  arr[0];
```

Java Arrays

Tragically, Java's mutable arrays use covariant subtyping!

Suppose that Cow is a subtype of Animal.

Code that only reads from arrays typechecks:

```
Animal[ ] arr  =  new Cow[ ] { new Cow("Alfonso") };
Animal a  =  arr[0];
```

but writing to the array can get into trouble:

```
arr[0] = new Animal("Brunhilda");
```

Specifically, this generates an ArrayStoreException.