Lecture 27
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.
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.
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:

\[
\{\text{foo} : \text{int}, \text{bar} : \text{bool}\}
\]
$l \in \mathcal{L}$

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

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

$\tau ::= \cdots | \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$
Dynamic Semantics

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

\[ \{l_1 = v_1, \ldots, l_n = v_n\}.l_i \rightarrow v_i \]
\[
\forall i \in 1..n. \quad \Gamma \vdash e_i : \tau_i \\
\Gamma \vdash \{ l_1 = e_1, \ldots, l_n = e_n \} : \{ l_1 : \tau_1, \ldots, l_n : \tau_n \} \\
\Gamma \vdash e : \{ l_1 : \tau_1, \ldots, l_n : \tau_n \} \\
\Gamma \vdash e.l_i : \tau_i
\]
Example

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

\[ \text{GETX} \triangleq \lambda p: \{x : \text{int}, y : \text{int}\}. p.x \]

\[ \text{GETX} \{x = 4, y = 2\} \]
Example

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

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

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

\[
\text{GETX} \triangleq \lambda p : \{x : \text{int}, y : \text{int}\}. p.x
\]

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

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

GETX \{y = 2, x = 4\}
Subtyping

Definition (Subtype)

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

If $\tau_1 \leq \tau_2$, we also say $\tau_2$ is the supertype of $\tau_1$. 
Subtyping

**Definition (Subtype)**

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

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

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

This typing rule says that if $e$ has type $\tau$ and $\tau$ is a subtype of $\tau'$, then $e$ also has type $\tau'$. 
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):

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

This program isn’t well-typed (yet):

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

So let’s add width subtyping:

$$k \geq 0$$

$$\{ l_1 : \tau_1, \ldots, l_{n+k} : \tau_{n+k} \} \leq \{ l_1 : \tau_1, \ldots, l_n : \tau_n \}$$
This program also doesn’t get stuck:

\[
(\lambda p : \{ x : \text{int}, y : \text{int} \}. \, p.x + p.y) \{ y = 37, x = 5 \}
\]
Record Subtyping

This program also doesn’t get stuck:

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

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

$$\pi \text{ is a permutation on } 1..n$$

$$\{l_1: \tau_1, \ldots, l_n: \tau_n\} \leq \{l_{\pi(1)}: \tau_{\pi(1)}, \ldots, l_{\pi(n)}: \tau_{\pi(n)}\}$$
Record Subtyping

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

\[(\lambda p: \{x : \{y : \texttt{int}\}\}. p.x.y) \{x = \{y = 4, z = 2\}\}\]
Record Subtyping

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

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

Let’s add depth subtyping:

$$\forall i \in 1..n. \ \tau_i \leq \tau'_i$$

$$\{l_1 : \tau_1, \ldots, l_n : \tau_n\} \leq \{l_1 : \tau'_1, \ldots, l_n : \tau'_n\}$$
Record Subtyping

Putting all three forms of record subtyping together:

\[
\forall i \in 1..n. \exists j \in 1..m. \quad l'_i = l_j \land \tau_j \leq \tau'_i
\]

\[
\{l_1:\tau_1, \ldots, l_m:\tau_m\} \leq \{l'_1:\tau'_1, \ldots, l'_n:\tau'_n\}
\]

S-RECORD
Standard Subtyping Rules

We always make the subtyping relation both reflexive and transitive.

\[ \tau \leq \tau \quad \text{S-REFL} \]

\[ \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \quad \text{S-TRANS} \]

Think of every type describing a set of values. Then \( \tau_1 \leq \tau_2 \) when \( \tau_1 \)'s values are a subset of \( \tau_2 \)'s.
Top Type

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

\[ \tau ::= \cdots | \top \]

\[ \tau \leq \top \]  

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:

\[
\frac{\tau_1 \leq \tau'_1 \quad \tau_2 \leq \tau'_2}{\tau_1 + \tau_2 \leq \tau'_1 + \tau'_2} \quad \text{S-Sum}
\]
Subtype All the Things!

We can also write subtyping rules for sums and products:

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

\[
\frac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2'}{\tau_1 \times \tau_2 \leq \tau_1' \times \tau_2'} \quad \text{S-PRODUCT}
\]
Function Types

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

\[ \text{S-FUNCTION} \]

\[ \tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2 \]
Desiderata

We’d like to have:

\[
\text{int} \rightarrow \{x: \text{int}, y: \text{int}\} \leq \text{int} \rightarrow \{x: \text{int}\}
\]
Desiderata

We’d like to have:

\[
\text{int} \rightarrow \{x: \text{int}, y: \text{int}\} \leq \text{int} \rightarrow \{x: \text{int}\}
\]

And:

\[
\{x: \text{int}\} \rightarrow \text{int} \leq \{x: \text{int}, y: \text{int}\} \rightarrow \text{int}
\]
Desiderata

We’d like to have:

\[
\text{int} \rightarrow \{x:\text{int}, y:\text{int}\} \leq \text{int} \rightarrow \{x:\text{int}\}
\]

And:

\[
\{x:\text{int}\} \rightarrow \text{int} \leq \{x:\text{int}, y:\text{int}\} \rightarrow \text{int}
\]

In general, to prove:

\[
\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2
\]

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:

\[
\frac{\tau'_1 \leq \tau_1 \quad \tau'_2 \leq \tau_2}{\frac{\tau_1 \rightarrow \tau_2 \leq \tau'_1 \rightarrow \tau'_2}{S\text{-FUNCTION}}}
\]
What should the relationship be between $\tau$ and $\tau'$ in order to have $\tau \text{ ref} \leq \tau' \text{ ref}$?
If \( r' \) has type \( \tau' \text{ref} \), then \( !r' \) has type \( \tau' \).

Imagine we replace \( r' \) with \( r \), where \( r \) has a type \( \tau \text{ref} \) that we’ve somehow decided is a subtype of \( \tau' \text{ref} \).
Example

If \( r' \) has type \( \tau' \text{ ref} \), then \( !r' \) has type \( \tau' \).

Imagine we replace \( r' \) with \( r \), where \( r \) has a type \( \tau \text{ ref} \) that we’ve somehow decided is a subtype of \( \tau' \text{ ref} \).

Then \( !r \) should still produce something can be treated as a \( \tau' \). In other words, it should have a type that is a subtype of \( \tau' \).

So the referent type should be covariant:

\[
\frac{\tau \leq \tau'}{\tau \text{ ref} \leq \tau' \text{ ref}}
\]
Example

If \( v \) has type \( \tau' \), 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 \( \tau' \).
Example

If \( v \) has type \( \tau' \), 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 \( \tau' \).

So the referent type should be contravariant!

\[
\begin{align*}
\tau' &\leq \tau \\
\tau \text{ ref} &\leq \tau' \text{ ref}
\end{align*}
\]
Reference Subtyping

In fact, subtyping for reference types must be *invariant*: a reference type \( \tau \text{ ref} \) is a subtype of \( \tau' \text{ ref} \) if and only if \( \tau \leq \tau' \) and \( \tau' \leq \tau \).

\[
\frac{\tau \leq \tau' \quad \tau' \leq \tau}{\tau \text{ ref} \leq \tau' \text{ ref}} \quad \text{S-REF}
\]
Tragically, Java’s mutable arrays use covariant subtyping!

```java
Suppose that Cow is a subtypeof Animal.

```
```java
Codethat only reads from arraystypes checks:
```
```java
Animal[] arr = new Cow[] {
   new Cow("Alfonso")
};
```
```java
Animal a = arr[0];
```
```java
but writing to the array can get into trouble:
```
```java
arr[0] = new Animal("Brunhilda");
```
Specifically, this generates an ArrayStoreException.
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:

```java
Animal[] arr = new Cow[ ] { new Cow("Alfonso") };
Animal a = arr[0];
```
Tragically, Java’s mutable arrays use covariant subtyping!

Suppose that Cow is a subtype of Animal.

Code that only reads from arrays typechecks:

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

but writing to the array can get into trouble:

```java
arr[0] = new Animal(“Brunhilda”);  
```

Specifically, this generates an ArrayStoreException.