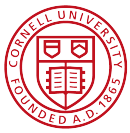
CS 4110 – Programming Languages and Logics
Lecture #27: Records and Subtyping



1 Records

We have previously seen binary products, i.e., pairs of values. Binary products can be generalized in a straightforward way to n-ary products, also called *tuples*. For example, $\langle 3, (), \mathsf{true}, 42\rangle$ is a 4-ary tuple containing an integer, a unit value, a boolean value, and another integer. Its type is $\mathbf{int} \times \mathbf{unit} \times \mathbf{bool} \times \mathbf{int}$.

Records are a generalization of tuples. We annotate each field of record with a *label*, drawn from some set of labels $\mathcal{L}$. For example, $\{\mathsf{foo} = 32, \mathsf{bar} = \mathsf{true}\}$ is a record value with an integer field labeled foo and a boolean field labeled bar. The type of the record value is written $\{\mathsf{foo} : \mathbf{int}, \mathsf{bar} : \mathbf{bool}\}$. We extend the syntax, operational semantics, and typing rules of the call-by-value lambda calculus to support records.

$$
\begin{array}{l}
l \in \mathcal{L} \\
e ::= \cdots \mid \{l_1 = e_1, \ldots, l_n = e_n\} \mid e.l \\
v ::= \cdots \mid \{l_1 = v_1, \ldots, l_n = v_n\} \\
\tau ::= \cdots \mid \{l_1 : \tau_1, \ldots, l_n : \tau_n\}
\end{array}
$$

We add new evaluation contexts to evaluate the fields of records.

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

We also add a rule to access the field of a location.

$$\frac{}{\{l_1 = v_1, \ldots, l_n = v_n\}.l_i \longrightarrow v_i}$$

Finally, we add new typing rules for records. Note that the order of labels is important: the type of the record value $\{\mathsf{lat} = -40, \mathsf{long} = 175\}$ is $\{\mathsf{lat} : \mathbf{int}, \mathsf{long} : \mathbf{int}\}$, which is different from $\{\mathsf{long} : \mathbf{int}, \mathsf{lat} : \mathbf{int}\}$, the type of the record value $\{\mathsf{long} = 175, \mathsf{lat} = -40\}$. In many languages with records, the order of the labels is not important; indeed, we will consider weakening this restriction in the next section.

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

2 Subtyping

Subtyping is a key feature of object-oriented languages. It was first introduced in the SIMULA languages by the Norwegian researchers Dahl and Nygaard.

The principle of subtyping is as follows. If τ_1 is a subtype of τ_2 (written $\tau_1 \leq \tau_2$, and also sometimes as $\tau_1 <: \tau_2$), then a program can use a value of type τ_1 whenever it would use a value of type τ_2. If $\tau_1 \leq \tau_2$, then τ_1 is sometimes referred to as the subtype, and τ_2 as the supertype.

We can express the principle of subtyping in a typing rule, often referred to as the "subsumption typing rule" (since the supertype subsumes the subtype).

$$\textsc{Subsumption}\ \frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'}$$

This rule says that if e has type τ and τ is a subtype of τ', then e also has type τ'. Recall that we provided an intuition for a type as a set of computational entities that share some common property. Type τ is a subtype of type τ' is every computational entity in the set for τ can be regarded as a computational entity in the set for τ'.

So what types are in a subtype relation? We will define inference rules and axioms for the subtype relation $\leq$. The subtype relation is both reflexive and transitive. These properties are intuitive if we think of subtyping as a subset relation. We add inference rules that express this.

$$\frac{}{\tau \leq \tau} \qquad\qquad \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}$$

2.1 Subtyping for records

Consider records and record types. A record consists of a set of labeled fields. Its type includes the types of the fields in the record. Let's define the type **Point** to be the record type $\{\mathsf{x} : \mathbf{int}, \mathsf{y} : \mathbf{int}\}$, that contains two fields x and y, both integers. That is:

$$\mathbf{Point} = \{\mathsf{x} : \mathbf{int}, \mathsf{y} : \mathbf{int}\}.$$

Lets also define

$$\mathbf{Point3D} = \{\mathsf{x} : \mathbf{int}, \mathsf{y} : \mathbf{int}, \mathsf{z} : \mathbf{int}\}$$

as the type of a record with three integer fields x, y and z. Because **Point3D** contains all of the fields of **Point**, and those have the same type as in **Point**, it makes sense to say that **Point3D** is a subtype of **Point**—i.e., **Point3D** $\leq$ **Point**.

Think about any code that used a value of type **Point**. This code could access the fields x and y, and that's pretty much all it could do with a value of type **Point**. A value of type **Point3D** has these same fields, x and y, and so any piece of code that used a value of type **Point** could instead use a value of type **Point3D**.

We can write a subtyping rule for records that allows the subtype to have more fields than the supertype. This is sometimes called "width" subtyping for records.

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

But why not let the corresponding fields be in a subtyping relation? For example, if $\tau_1 \leq \tau_2$ and $\tau_3 \leq \tau_4$, then is $\{\mathsf{foo} : \tau_1, \mathsf{bar} : \tau_3\}$ a subtype of $\{\mathsf{foo} : \tau_2, \mathsf{bar} : \tau_4\}$? (Note that this is only correct because the fields of records are immutable—more on this when we consider subtyping rules for references.) Also, why not relax the requirement that the order of fields be the same?

The following rule allows both "depth" and "permutation" subtyping for records (along with the "width" subtyping rule we saw before).

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

2.2 Top

Many languages a type $\top$ (pronounced "top") that is a supertype of every other type.

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

The $\top$ type can be used to model types such as Java's `Object`.

2.3 Subtyping for sums and products

Like records, we can extend the subtyping relation to handle products and sums.

$$\text{S-Product}\ \frac{\tau_1 \le \tau'_1 \quad \tau_2 \le \tau'_2}{\tau_1 \times \tau_2 \le \tau'_1 \times \tau'_2} \qquad \text{S-Sum}\ \frac{\tau_1 \le \tau'_1 \quad \tau_2 \le \tau'_2}{\tau_1 + \tau_2 \le \tau'_1 + \tau'_2}$$

2.4 Subtyping for functions

Consider two function types $\tau_1 \to \tau_2$ and $\tau'_1 \to \tau'_2$. What are the subtyping relations between τ_1,, τ_2, τ'_1,, and τ'_2 that should be satisfied in order for $\tau_1 \to \tau_2 \le \tau'_1 \to \tau'_2$ to hold?

Consider the following expression:

$$G \triangleq \lambda f : \tau'_1 \to \tau'_2.\, \lambda x : \tau'_1.\, f\ x.$$

This function has type

$$(\tau'_1 \to \tau'_2) \to \tau'_1 \to \tau'_2.$$

Now suppose we had a function $h : \tau_1 \to \tau_2$ such that $\tau_1 \to \tau_2 \le \tau'_1 \to \tau'_2$. By the subtyping principle, we should be able to give h as an argument to G, and G should work fine. Suppose that v is a value of type τ'_1. Then $G\ h\ v$ will evaluate to $h\ v$, meaning that h will be passed a value of type τ_1. Since h has type $\tau_1 \to \tau_2$, it must be the case that $\tau'_1 \le \tau_1$. (What could go wrong if $\tau_1 \le \tau'_1$?)

Furthermore, the result type of $G\ h\ v$ should be of type τ'_2 according to the type of G, but $h\ v$ will produce a value of type τ_2, as indicated by the type of h. So it must be the case that $\tau_2 \le \tau'_2$.

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

$$\text{S-Function}\ \frac{\tau'_1 \le \tau_1 \quad \tau_2 \le \tau'_2}{\tau_1 \to \tau_2 \le \tau'_1 \to \tau'_2}$$

Note that the subtyping relation between the argument and result types in the premise are in different directions! The subtype relation for the result type is in the same direction as for the conclusion (primed version is the supertype, non-primed version is the subtype); it is in the opposite direction for the argument type. We say that subtyping for the function type is *covariant* in the result type, and *contravariant* in the argument type.

2.5 Subtyping for references

Suppose we have a location l of type τ **ref**, and a location l' of type τ' **ref**. What should the relationship be between τ and τ' in order to have τ **ref** $\leq \tau'$ **ref**?

Let's consider the following program R, that takes a location x of type τ' **ref** and reads from it.

$$R \triangleq \lambda x{:}\tau' \textbf{ ref}.\ !x$$

The program R has the type τ' **ref** $\to \tau'$. Suppose we gave R the location l as an argument. Then $R\ l$ will look up the value stored in l, and return a result of type τ (since l is type τ **ref**. Since R is meant to return a result of type τ' **ref**, we thus want to have $\tau \leq \tau'$.

So this suggests that subtyping for reference types is covariant.

But now consider the following program W, which takes a location x of type τ' **ref**, a value y of type τ', and writes y to the location.

$$W \triangleq \lambda x{:}\tau' \textbf{ ref}.\, \lambda y{:}\tau'.\, x := y$$

This program has type τ' **ref** $\to \tau' \to \tau'$. Suppose we have a value v of type τ', and consider the expression $W\ l\ v$. This will evaluate to $l := v$, and since l has type τ **ref**, it must be the case that v has type τ, and so $\tau' \leq \tau$. This suggests that subtyping for reference types is contravariant!

In fact, subtyping for reference types must be *invariant*: reference type τ **ref** is a subtype of τ' **ref** if and only if $\tau = \tau'$. Indeed, to be sound, subtyping for any mutable location must be invariant. Interestingly, in the Java programming language, arrays are mutable locations but have covariant subtyping!

Suppose that we have two classes Person and Student such that Student extends Person (that is, Student is a subtype of Person). The following Java code is accepted, since an array of Student is a subtype of an array of Person, according to Java's covariant subtyping for arrays.

```
Person[ ] arr = new Student[ ] { new Student("Alice") };
```

This is fine as long as we only read from arr. The following code executes without any problems, since arr[0] is a Student which is a subtype of Person.

```
Person p = arr[0];
```

However, the following code, which attempts to update the array, has some issues.

```
arr[0] = new Person("Bob");
```

Even though the assignment is well-typed, it attempts to assign an object of type Person into an array of Students! In Java, this produces an ArrayStoreException, indicating that the assignment to the array failed.