Harvard School of Engineering and Applied Sciences — CS 152: Programming Languages

Records and subtyping

Lecture 15 Tuesday, March 25, 2014

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 $\textbf{int} \times \textbf{unit} \times \textbf{bool} \times \textbf{int}$.

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 $\{\mathsf{foo}:\textbf{int}, \mathsf{bar}:\textbf{bool}\}$.

We extend the syntax, operational semantics, and typing rules of the call-by-value lambda calculus to support records.

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

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 record.

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

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 $\{\mathsf{lat}:\textbf{int}, \mathsf{long}:\textbf{int}\}$, which is different from $\{\mathsf{long}:\textbf{int}, \mathsf{lat}:\textbf{int}\}$, 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. Subtyping was first introduced in SIMULA, invented by Norwegian researchers Dahl and Nygaard, and considered the first object-oriented programming language.

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 $\tau_1 \leq: \tau_2$), 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).

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

The subsumption rule says that if e is of type τ, and τ is a subtype of τ', then e is also of 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' if 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 both seem reasonable 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 $\{\mathsf{x}\!:\!\textbf{int}, \mathsf{y}\!:\!\textbf{int}\}$, that contains two fields x and y, both integers. That is:

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

Lets also define

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

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**: **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.

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

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 $\{\mathsf{foo} : \tau_2, \mathsf{bar} : \tau_4\}$? Turns out that this is the case so long as the fields of records are immutable. More on this when we consider subtyping for references.

Also, we could relax the requirement that the order of fields must be the same. The following is a more permissive subtyping rule for records.

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

2.2 Subtyping for products

Like records, we can allow the elements of a product to be in a subtyping relation.

$$\frac{\tau_1 \leq \tau'_1 \quad \tau_2 \leq \tau'_2}{\tau_1 \times \tau_2 \leq \tau'_1 \times \tau'_2}$$

2.3 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 $\tau_1 \to \tau_2 \leq \tau'_1 \to \tau'_2$ 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 $\tau_1 \to \tau_2 \leq \tau'_1 \to \tau'_2$. 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 $\tau'_1 \leq \tau_1$. (What could go wrong if $\tau_1 \leq \tau'_1$?)

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 $\tau_2 \leq \tau'_2$.

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

$$\frac{\tau'_1 \leq \tau_1 \quad \tau_2 \leq \tau'_2}{\tau_1 \to \tau_2 \leq \tau'_1 \to \tau'_2}$$

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.

2.4 Subtyping for locations

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.

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

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

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

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 $\tau' \leq \tau$. But 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.

$$\mathsf{Person\ p\ =\ arr[0];}$$

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

$$\mathsf{arr[0] = new\ Person(``Bob");}$$

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.

2.5 Bounded polymorphism

Polymorphism and subtyping can be combined. That is, we can change type abstraction to include an upper bound on the types that can be used to instantiate the type variable: $\forall X \leq \tau_1.\, \tau_2$. This means that X can only be instantiated with a subtype of τ_1.

Note that if there is a "top type" $\top$ (i.e., every type τ is a subtype of $\top$), then $\forall X.\, \tau$ is equivalent to $\forall X \leq \top.\, \tau$.

Subtyping and parametric polymorphism are largely orthogonal. We need to change the type variable context Δ so that it contains the bounds on type variables, so Δ is now a sequence of elements of the form $X \leq \tau$.

The syntax of expressions, values and types is now as follows.

$$\begin{aligned} e &::= n \mid x \mid \lambda x\!:\!\tau.\, e \mid e_1\, e_2 \mid \Lambda X \leq \tau.\, e \mid e\,[\tau] \\ v &::= n \mid \lambda x\!:\!\tau.\, e \mid \Lambda X \leq \tau.\, e \\ \tau &::= \textbf{int} \mid \tau_1 \rightarrow \tau_2 \mid X \mid \forall X.\, \tau \end{aligned}$$

The operational semantics are the same (modulo the minor changes to syntax). Most of the typing rules remain the same. We present the rules for type abstraction, type application, and subsumption. Note that type application requires that the instantiating type is a subtype of the declared bound on the type variable. Note also that the subtyping relation now uses the type variable context Δ.

$$\frac{\Delta, X \leq \tau_1, \Gamma \vdash e\!:\!\tau_2}{\Delta, \Gamma \vdash \Lambda X \leq \tau_1.\, e\!:\!\forall X \leq \tau_1.\, \tau_2} \qquad \frac{\Delta, \Gamma \vdash e\!:\!\forall X \leq \tau_1.\, \tau_2 \quad \Delta \vdash \tau \leq \tau_1}{\Delta, \Gamma \vdash e\,[\tau]\!:\!\tau_2\{\tau/X\}} \qquad \frac{\Delta, \Gamma \vdash e\!:\!\tau \quad \Delta \vdash \tau \leq \tau'}{\Delta, \Gamma \vdash e\!:\!\tau'}$$

The rule for subtyping a type variable simply uses the type variable context Δ.

$$\frac{}{\Delta \vdash X \leq \tau}\; X \leq \tau \in \Delta$$

The subtyping rule for polymorphic types is the most interesting. Going back to the subtyping principle, $\forall X \leq \tau_1.\, \tau_2$ is a subtype of $\forall X \leq \tau_1'.\, \tau_2'$ if whenever a value of type $\forall X \leq \tau_1'.\, \tau_2'$ is expected, a value of type $\forall X \leq \tau_1.\, \tau_2$ can be used instead. Intuitively, an expression of polymorphic type $\forall X \leq \tau_1'.\, \tau_2'$ can be thought of as a function from types to terms. That is, it takes a type as an argument, and returns an expression. (By contrast, functions (also known as term abstractions) that we have seen previously are functions from terms to terms.)

A term of type $\forall X \leq \tau_1'.\, \tau_2'$ may be instantiated with any type that is a subtype of τ_1'; for $\forall X \leq \tau_1'.\, \tau_2'$ to be a subtype, it must also be able to be instantiated with any type that is a subtype of τ_1', so we want τ_1' to be a subtype of τ_1. That is, we are contravariant in the bound on the type variable. Also, when instantiated on any type X that satisfies both constraints (i.e., $X \leq \tau_1$ and $X \leq \tau_1'$), a value produced by an expression of type τ_2 should be usable at type τ_2'. So we want $\Delta, X \leq \tau_1' \vdash \tau_2 \leq \tau_2'$. The inference rule for subtyping on polymorphic types is thus the following.

$$\frac{\Delta \vdash \tau_1' \leq \tau_1 \quad \Delta, X \leq \tau_1' \vdash \tau_2 \leq \tau_2'}{\Delta \vdash \forall X \leq \tau_1.\, \tau_2 \leq \forall X \leq \tau_1'.\, \tau_2'}$$