Lecture 13: Subtyping

Polyvios Pratikakis

Computer Science Department, University of Crete

Type Systems and Static Analysis

Subtyping

- Usually found in Object Oriented languages
- One form of *polymorphism*: a program can have more than one types
- So far, each language feature we saw is compositional: can be added without affecting the rest of the language
- Subtyping is not: we might need to change the type rules for other features
- Roughly: if all expressions of type T also have type T', then T is a subtype of T'
- Alternatively: if we can always substitute an expression of type T' with an expression of type T in any context and still have a valid program, T is a subtype of T'

Background

- Simply typed lambda calculus with numbers and records:

$$
\begin{array}{rcl}
e & ::= & x \mid \lambda x : T.e \mid e\ e \mid n \mid \{l_1 = e_1, \ldots, l_n = e_n\} \\
 & \mid & \text{case } e \text{ of } \{l_1(x) => e_1 \mid \ldots \mid l_n(x) \Rightarrow e_n\} \\
v & ::= & n \mid \lambda x : T.e \mid \{l_1 = v_1, \ldots, l_n = v_n\} \\
T & ::= & T \rightarrow T \mid Nat \mid \{l_1 : T_1, \ldots, l_n : T_n\}
\end{array}
$$

- Type rule for function application:

$$
[\text{T-APP}]\ \frac{\Gamma \vdash e_1 : T \rightarrow T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash (e_1\ e_2) : T'}
$$

- Not allowed: $(\lambda x : \{foo : Nat\}.(x.foo))\ \{foo = 5, bar = 42\}$
- Even though it is always safe!

Subsumption

- It is *always* safe to pass a struct with more fields
- If the function can be typed assuming its argument *x* has type $\{foo : Nat\}$, then it only accesses the *foo* field of record *x*
- It won't hurt if *x* has additional fields
- We say $\{foo : Nat, bar : Nat\}$ is a *subtype* of $\{foo : Nat\}$
  - Also written as $\{foo : Nat, bar : Nat\} <: \{foo : Nat\}$
- To use the subtype relation $<:$ during type-checking, we add one more type rule:

$$[\text{T-SUB}]\ \frac{\Gamma \vdash e : T_1 \qquad T_1 <: T_2}{\Gamma \vdash e : T_2}$$

- It says we can use a subtype instead of a supertype

Defining subtype

- We define a relation $<:$ between types as usual
  - Inductively, using inference rules
- Each rule produces a judgement $T <: T'$
- The relation is the smallest set of subtyping judgements produced by the inference rules
- The same as all definitions so far

Subtyping relation

- The subtyping relation is *reflexive*:

$$[\text{S-REFL}]\ \frac{}{T <: T}$$

- The subtyping relation is *transitive*:

$$[\text{S-TRANS}]\ \frac{T_1 <: T_2 \qquad T_2 <: T_3}{T_1 <: T_3}$$

- Both, from the intuition of safely substituting a subtype for a supertype

Subtyping relation (cont'd)

- A record type is a subtype of another if it has more fields:

$$[\text{S-WIDE}]\ \frac{}{\{l_1 : T_1, \ldots, l_{n+k} : T_{n+k}\} <: \{l_1 : T_1, \ldots, l_n : T_n\}}$$

- or if all its fields are subtypes:

$$[\text{S-DEEP}]\ \frac{T_i <: T'_i, \text{ for each } 0 \leq i \leq n}{\{l_1 : T_1, \ldots, l_n : T_n\} <: \{l_1 : T'_1, \ldots, l_n : T'_n\}}$$

- or if the fields are reordered:

$$[\text{S-PERM}]\ \frac{}{\begin{array}{c}\{l_1 : T_1, \ldots, l_i : T_i, \ldots, l_j : T_j, \ldots, l_n : T_n\} <: \\ \{l_1 : T_1, \ldots, l_j : T_j, \ldots, l_i : T_i, \ldots, l_n : T_n\}\end{array}}$$

Subtyping relation (cont'd)

- A function type is a subtype of another if it can be used instead
  - The subtype should accept all arguments the supertype accepts (contravariant)
  - The subtype shouldn't return anything not returned by the supertype (covariant)

$$[\textsc{S-Fun}]\frac{T_2 <: T_1 \qquad T'_1 <: T'_2}{T_1 \to T'_1 <: T_2 \to T'_2}$$

- One supertype to rule them all (like `java.lang.Object`):

$$[\textsc{T-Top}]\frac{}{T <: \top}$$

Metatheory

- Inversion lemma of the subtyping relation
  - If $T <: T_1 \to T_2$ then T has the form $T'_1 \to T'_2$ with $T_1 <: T'_1$ and $T'_2 <: T_2$
  - If $T <: \{l_1 : T_1, \ldots, l_n : T_n\}$ then T has the form $\{k_1 : T'_1, \ldots, k_n : T'_n\}$ with at least the labels $l_1, \ldots, l_n$ and for all $0 \leq i \leq n$, if $k_j = l_i$ then $T'_j <: T_i$.
- Inversion lemma of the typing relation
  - If $\Gamma \vdash (\lambda x : T.e) : T_1 \to T_2$, then $T_1 <: T$ and $\Gamma, x : T \vdash e : T_2$
  - If $\Gamma \vdash \{k_1 = e_1, \ldots, k_n = e_n\} : \{l_1 : T_1, \ldots, l_m : T_m\}$ then for each $i \in 0..m$ there is a $j \in 0..n$ such that $l_i = k_j$ and $\Gamma \vdash e_j : T_i$

Metatheory (cont'd)

- Substitution and preservation remain the same (their proof changes)
- Substitution lemma
  - If $\Gamma, x : T_1 \vdash e : T_2$ and $\Gamma \vdash e' : T_1$ then $\Gamma \vdash e'[e/x] : T_2$
- Preservation theorem
  - If $\Gamma \vdash e : T$ and $e \to e'$ then $\Gamma \vdash e' : T$

Metatheory (cont'd)

- Canonical forms lemma
  - If v is a value and $\emptyset \vdash v : T_1 \to T_2$ then v has the form $\lambda x : T.e$
  - If v is a value and $\emptyset \vdash v : \{l_1 : T_1, \ldots, l_n : T_n\}$ then v has the form $\{k_1 = v_1, \ldots, k_m = v_m\}$ where for all l_i there is a $k_j = l_i$
- Progress theorem
  - If $\emptyset \vdash e : T$ then either e is a value or there is some e' with $e \to e'$

Subtyping and casts

- Ascription: explicitly stating the type of an expression—in ML, written $(e : T)$
- Also called *casting* in languages like C/C++, Java, C#, etc.—written $(T)e$
- Two very different forms of casting
  - Up-cast: T is a supertype of the typechecker's type for e
  - Down-cast: T is a subtype of the typechecker's type for e
- Down-cast is unsafe
  - What happens if at runtime e does not have type T?
  - Down-casts usually compiled into *run-time checks* that raise a dynamic exception
  - Alternatively, down-casts only allowed as a test (like `instanceof`), providing an "else" case

Subtyping and references

- References are like implicit function arguments
- ...and also like implicit function results
- They have to be both covariant and contravariant!
- References are *invariant* under subtyping to preserve type safety:

$$\frac{T_1 <: T_2 \qquad T_2 <: T_1}{Ref^{T_1} <: Ref^{T_2}}$$

- This restriction is caused by the two operations supported
  - Read causes a covariant constraint
  - Write causes a contravariant constraint

Subtyping and arrays

- Arrays are like references: can read and write the contents
- Like references, we need invariant subtyping for type-safety

$$\frac{T_1 <: T_2 \qquad T_2 <: T_1}{T_1[\,] <: T_2[\,]}$$

Arrays in Java

- Interestingly, Java permits covariant subtyping for arrays

$$\frac{T_1 <: T_2}{T_1[\,] <: T_2[\,]}$$

- But, consider:

```
Integer[] x = new Integer[10];
Object[] y = (Object[]) x;
y[3] = new Object();
x[3].intValue();                    // OOPS, no such method!
```

- Bad design, big performance hit to keep safe:
  - Every array assignment is equivalent to a downcast
  - Must check every assignment to every array at runtime!