

Programming Languages, CPSC 604—Slides 10

Subtyping

Jaakko Järvi

March 2, 2009

Outline

- 1 Subtype relation
- 2 Subtyping and other extensions
- 3 About subtyping
- 4 Summary

Subtyping intuitively

- What is subtyping, anyway?
- What does it mean that S is a subtype of T ?

$$S <: T$$

Subtyping intuitively

- What is subtyping, anyway?
- What does it mean that S is a subtype of T ?

$$S <: T$$

- **Substitutability** (Liskov substitution principle):

If for each object o_1 of type S there is an object o_2 of type T such that for all programs P defined in terms of T , the behavior of P is unchanged when o_1 is substituted for o_2 then S is a subtype of T .

and a later form of the same thing (**subtype requirement**):

Let $\phi(x)$ be a property provable about objects x of type T . Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T .

Subtyping from the type checker's point of view

- In practice, this is what type checkers do:
 - If S is a subtype of T , written as, $S <: T$, then any expression of type S can be used in any context that expects an expression of type T , and no type error will occur.

Subtyping from type checker's point of view

- In practice, this is what type checkers do:
 - If S is a subtype of T , written as, $S <: T$, then any expression of type S can be used in any context that expects an expression of type T , and no type error will occur.
- Which definition defines a smaller relation?

Why subtyping?

- Consider this function in pseudo-C

```
void foo( struct { int a; } r) {  
    r.a = 0;  
}  
struct K { int a; int b; }  
K k;
```

```
foo(k); // error
```

- Intuitively, it would be safe to pass `k` in, but...
- a system without subtyping does not allow it.

Subsumption

- Substitutability is captured with the following **subsumption** rule:

$$\frac{\Gamma \vdash t : U \quad U <: T}{\Gamma \vdash t : T}$$

- Note, that adding this rule possibly requires revisiting other rules, subtyping is a cross-cutting extension

Subtyping in simplified setting

- Formalism: simply-typed lambda calculus with records, booleans and integers.
- How should subtyping be defined for records?
- That is, when is it safe to use a record of type T , if U is expected?
- Note, we're after structural subtyping here

Subtyping for records

- Order of fields does not matter

S-RecordPermutation

$$\frac{\{l_i : T_i^{i \in 1 \dots n}\} \text{ is a permutation of } \{k_j : U_j^{j \in 1 \dots n}\}}{\{l_i : T_i^{i \in 1 \dots n}\} <: \{k_j : U_j^{j \in 1 \dots n}\}}$$

- Example:

$$\{\text{key} : \text{bool}, \text{value} : \text{int}\} <: \{\text{value} : \text{int}, \text{key} : \text{bool}\}$$

Subtyping for records

- We can always add new fields in the end

$$\text{S-RecordNewFields} \\ \{l_i : T_i^{i \in 1 \dots n+k}\} <: \{l_i : T_i^{i \in 1 \dots n}\}$$

- Example:

$$\{\text{key} : \text{bool}, \text{value} : \text{int}, \text{map} : \text{int} \rightarrow \text{int}\} <: \{\text{key} : \text{bool}, \text{value} : \text{int}\}$$

Subtyping for records

- We can subject the fields to subtyping:

$$\frac{\text{S-RecordElements} \quad \text{for each } i \quad T_i <: U_i}{\{l_i : T_i^{i \in 1 \dots n}\} <: \{l_i : U_i^{i \in 1 \dots n}\}}$$

- Example:

$$\{\text{field1} : \text{bool}, \text{field2} : \{\text{val} : \text{bool}\}\} <: \{\text{field1} : \text{bool}, \text{field2} : \{\}\}$$

- Any rules missing?

General rules for subtyping

- Reflexivity

$$T <: T$$

- Transitivity

$$\frac{T <: U \quad U <: V}{T <: V}$$

Example

- Prove that $\{a : \text{bool}, b : \text{int}, c : \{l : \text{int}\}\} <: \{c : \{\}\}$

Subtyping and functions

- What should the subtyping rule for functions look like?
- That is, what function types would be ok in a context where a function of, say, $T \rightarrow U$ is expected?

Subtyping and functions

- What should the subtyping rule for functions look like?
- That is, what function types would be ok in a context where a function of, say, $T \rightarrow U$ is expected?
- Here, we pass $\{a : A, b : B\}$ into a function of type $\{a : A, b : B\} \rightarrow B$
($\lambda x : \{a : A, b : B\}. x.b$) $\{a = \text{some}A, b = \text{some}B\}$

Subtyping and functions

- What should the subtyping rule for functions look like?
- That is, what function types would be ok in a context where a function of, say, $T \rightarrow U$ is expected?
- Here, we pass $\{a : A, b : B\}$ into a function of type $\{a : A, b : B\} \rightarrow B$
 $(\lambda x : \{a : A, b : B\}. x.b) \{a = \text{some}A, b = \text{some}B\}$
- It is ok, if the function only requires $\{b : B\}$ and is thus of type $\{b : B\} \rightarrow B$
 $(\lambda x : \{b : B\}. x.b) \{a = \text{some}A, b = \text{some}B\}$

Subtyping and functions

- What should the subtyping rule for functions look like?
- That is, what function types would be ok in a context where a function of, say, $T \rightarrow U$ is expected?
- Here, we pass $\{a : A, b : B\}$ into a function of type $\{a : A, b : B\} \rightarrow B$
 $(\lambda x : \{a : A, b : B\}. x.b) \{a = \text{some}A, b = \text{some}B\}$
- It is ok, if the function only requires $\{b : B\}$ and is thus of type $\{b : B\} \rightarrow B$
 $(\lambda x : \{b : B\}. x.b) \{a = \text{some}A, b = \text{some}B\}$
- We expect a type $\{b : B\}$ from this function
 $((\lambda x : \{b : B\}. x) \{b = \text{some}B\}).b$

Subtyping and functions

- What should the subtyping rule for functions look like?
- That is, what function types would be ok in a context where a function of, say, $T \rightarrow U$ is expected?
- Here, we pass $\{a : A, b : B\}$ into a function of type $\{a : A, b : B\} \rightarrow B$
 $(\lambda x : \{a : A, b : B\}. x.b) \{a = \text{some}A, b = \text{some}B\}$
- It is ok, if the function only requires $\{b : B\}$ and is thus of type $\{b : B\} \rightarrow B$
 $(\lambda x : \{b : B\}. x.b) \{a = \text{some}A, b = \text{some}B\}$
- We expect a type $\{b : B\}$ from this function
 $((\lambda x : \{b : B\}. x) \{b = \text{some}B\}).b$
- It is ok if we get $\{a : A, b : B\}$, something more
 $((\lambda x : \{b : B\}. \{b = x.b, a = \text{some}A\}) \{b = \text{some}B\}).b$

Subtyping and functions

$$\frac{T_2 <: T_1 \quad U_2 <: U_1}{T_1 \rightarrow U_2 <: T_2 \rightarrow U_1}$$

- “If variable f has type $T_2 \rightarrow U_1$, is it safe to bind it to a function of type $T_1 \rightarrow U_2$ ”?
- If a call site of f expects f to accept arguments of type T_2 , it surely is fine if f accepts arguments of type T_1 —any object of type T_2 is also of type T_1 .
- If a call site of f expects f to return a value of type U_1 , it surely is fine if f returns a value of type U_2 —any object of type U_2 is also of type U_1 .
- Function subtyping is **covariant** on return types, **contravariant** on parameter types
 - Seen this before?

Supertype of everything

- Often type systems with subtyping have the most general type, which is a supertype of all types

$T <: \textit{top}$

- Where have you seen this before?
- What about *bottom*?

Outline

- 1 Subtype relation
- 2 Subtyping and other extensions**
- 3 About subtyping
- 4 Summary

Subtyping and other extensions

- We noted that subtyping is a cross-cutting extension and potentially affects every other construct
- We must examine all other constructs to see what the interactions are

Ascription

- Reminder. Type rules:

$$\frac{\Gamma \vdash t : T}{\Gamma \vdash t \text{ as } T : T}$$

- Evaluation rules:

$$\frac{t \rightarrow u}{t \text{ as } T \rightarrow u \text{ as } T} \qquad v \text{ as } T \rightarrow v$$

Ascription

- Example

$$\frac{\frac{\frac{\vdots}{\Gamma \vdash t : U}}{\Gamma \vdash t : T} \quad \frac{\vdots}{U <: T}}{\Gamma \vdash t \text{ as } T : T}$$

- Now that we have subtyping, do you see a relation to something familiar?

Casting

- Ascription, as defined so far, is like **up-casting** in C++ or Java
- Up-cast overrides type-checker's reasoning giving a more general type to a term

$$(\lambda x:\text{bool}.\{a = x, b = \text{false}\}) \text{ true as } \{a : \text{bool}\}$$

- How about **down-casting** ?

$$(\lambda x:\text{top}.\{a = \text{false}\}) \{a : \text{bool}\}.a$$

Down-casting

- In down-casting, we override type checker's reasoning **in a potentially unsafe way**.
- We then take the responsibility that the program logic always guarantees safe behavior:

```
class A { ... };  
class B : public A { ... };
```

```
void fill(stack<A*>& s) {  
    for (int i=0; i < 10; ++i) s.push(new B());  
}
```

```
stack<A*> s;  
fill(s);  
B* b = dynamic_cast<B*>(s.pop());
```

- Down-cast says “Don't worry, I know what I'm doing”

Down-casting typing and evaluation rules

- Typing rule

$$\frac{\Gamma \vdash t : U}{\Gamma \vdash t \text{ as } T : T}$$

- Remember the evaluation rules:

$$\frac{t \rightarrow u}{t \text{ as } T \rightarrow u \text{ as } T} \qquad v \text{ as } T \rightarrow v$$

- Is this right? What's the effect on preservation?

Down-casting typing and evaluation rules

- Typing rule

$$\frac{\Gamma \vdash t : U}{\Gamma \vdash t \text{ as } T : T}$$

- Remember the evaluation rules:

$$\frac{t \rightarrow u}{t \text{ as } T \rightarrow u \text{ as } T}$$

$$\frac{\vdash v : T}{v \text{ as } T \rightarrow v}$$

- Note, that this type condition on evaluation relation is a check performed at run-time
- What is the effect on progress?

Aside: Java arrays

- Java has this subtyping rule for arrays:

$$\frac{T <: U}{T [] <: U []}$$

- Why might this be bad?

Aside: Java arrays

- Java has this subtyping rule for arrays:

$$\frac{T <: U}{T [] <: U []}$$

- Why might this be bad?

```
String[] strings = new String[1];
Object[] objects = strings;           // OK by the above rule
objects[0] = new Integer(1);         // OK, but runtime exception
```

Outline

- 1 Subtype relation
- 2 Subtyping and other extensions
- 3 About subtyping**
- 4 Summary

Syntax directed rules?

- We've discussed about *syntax directed rules*
 - Rules that give a type checking algorithm if we just read them from conclusion to premises
 - That is, for each syntactic form of a term there is a corresponding rule
- Is our current system syntax directed?

Typing rules so far

T-Record

$$\frac{\text{for each } i, \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_i = t_i^{i \in 1 \dots n}\} : \{l_i : T_i^{i \in 1 \dots n}\}}$$

T-Projection

$$\frac{\Gamma \vdash t : \{l_i : T_i^{i \in 1 \dots n}\}}{\Gamma \vdash t.l_j : T_j}$$

T-Subsumption

$$\frac{\Gamma \vdash t : U \quad U <: T}{\Gamma \vdash t : T}$$

T-Variable

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

T-Abstraction

$$\frac{\Gamma, x : T \vdash u : U}{\Gamma \vdash \lambda x : T. u : T \rightarrow U}$$

T-Application

$$\frac{\Gamma \vdash t : U \rightarrow T \quad \Gamma \vdash u : U}{\Gamma \vdash t u : T}$$

T-True

$$\vdash \text{true} : \text{bool}$$

T-False

$$\vdash \text{false} : \text{bool}$$

Subtyping rules so far

S-Reflexivity

$$T <: T$$

S-Transitivity

$$\frac{T <: U \quad U <: V}{T <: V}$$

S-RecordPermutation

$$\frac{\{l_i : T_i^{i \in 1 \dots n}\} \text{ is a permutation of } \{k_j : U_j^{j \in 1 \dots n}\}}{\{l_i : T_i^{i \in 1 \dots n}\} <: \{k_j : U_j^{j \in 1 \dots n}\}}$$

S-RecordNewFields

$$\{l_i : T_i^{i \in 1 \dots n+k}\} <: \{l_i : T_i^{i \in 1 \dots n}\}$$

S-RecordElements

$$\frac{\text{for each } i \quad T_i <: U_i}{\{l_i : T_i^{i \in 1 \dots n}\} <: \{l_i : U_i^{i \in 1 \dots n}\}}$$

S-Function

$$\frac{T_1 <: T_2 \quad U_1 <: U_2}{T_2 \rightarrow U_1 <: T_1 \rightarrow U_2}$$

Closer look at subsumption rule

$$\text{T-Subsumption} \\ \frac{\Gamma \vdash t : U \quad U <: T}{\Gamma \vdash t : T}$$

- The term in the conclusion can be anything. It is just a metavariable.
- E.g. which rule should you apply here (T-Abstraction or T-Subsumption)?:

$$\Gamma \vdash (\lambda x : U. t) : ?$$

Transitivity

$$\text{S-Transitivity}$$

$$\frac{T <: U \quad U <: V}{T <: V}$$

- Which subtyping rule to apply here?

$$\{y:\text{int}, x:\text{int}\} <: \{x:\text{int}\}$$

- What kind of an implementation results if one implements the transitivity rule directly?

Transitivity

$$\text{S-Transitivity}$$

$$\frac{T <: U \quad U <: V}{T <: V}$$

- Which subtyping rule to apply here?

$$\{y:\text{int}, x:\text{int}\} <: \{x:\text{int}\}$$

- What kind of an implementation results if one implements the transitivity rule directly?
 - Note, U does not appear in the conclusion
 - Thus, to show $T <: V$, we need to **guess(!)** some U , and then prove $T <: U$ and $U <: V$

Algorithmic subtyping

- Consider the rule of application $t u$, with $\Gamma \vdash t : U \rightarrow V$ and $\Gamma \vdash u : S$.
- Type checker must figure out if $S <: U$
- This is hard with the rules so far
- Redesigning the rules makes it easy
 - We can get rid of the transitivity and reflexivity rules
 - Why did we need transitivity to begin with?

Algorithmic subtyping

- Transitivity was to chain together the three record subtyping rules
- We can replace all three by just one syntax directed rule

$$\text{S-Record} \quad \frac{\{l_i^{i \in 1 \dots n}\} \subseteq \{k_j^{j \in 1 \dots m}\} \quad l_i = k_j \text{ implies } U_i <: T_j}{\{k_j : U_j^{j \in 1 \dots m}\} <: \{l_i : T_i^{i \in 1 \dots n}\}}$$

- Next we'd need to prove:
 - If we can derive $U <: T$ using only the “old” rules for record subtyping, we can derive $U <: T$ using only the “new” rule.
 - ...and also to the other direction
- Also show that $T <: T$ can be derived without S-Reflexivity
 - For other base types (*int* and *bool*) we need special rules $\text{int} <: \text{int}$ and $\text{bool} <: \text{bool}$
- And show that for each derivation of $T <: U$, there is a derivation that does not use S-Transitivity.

Algorithmic subtyping

$$\begin{array}{c}
 \text{S-Record} \\
 \frac{\{l_i^{i \in 1 \dots n}\} \subseteq \{k_j^{j \in 1 \dots m}\} \quad l_i = k_j \text{ implies } U_i <: T_j}{\{k_j : U_j^{i \in 1 \dots j}\} <: \{l_i : T_i^{i \in 1 \dots n}\}} \\
 \\
 \text{S-Function} \\
 \frac{T_1 <: T_2 \quad U_1 <: U_2}{T_2 \rightarrow U_1 <: T_1 \rightarrow U_2}
 \end{array}$$

Theorem (Soundness)

If $T <: U$ can be derived using the algorithmic rules, then $T <: U$ can also be derived using the “old” rules.

Theorem (Completeness)

If $T <: U$ can be derived using the “old” rules, then $T <: U$ can also be derived using the algorithmic rules.

Algorithmic typing

- Similar problems remain with the subsumption rule — it is not syntax directed
- Where is subsumption needed?

$$(\lambda x: \{b : B\}. x.b) \{a = \text{some}A, b = \text{some}B\}$$

- This is the only place where it is essential
- Subsumption can be dropped if the function application rule is modified

T-Application

$$\frac{\Gamma \vdash t : U \rightarrow T \quad \Gamma \vdash u : V \quad V <: U}{\Gamma \vdash t u : T}$$

Algorithmic typing

- Prove these

Theorem (Soundness)

If $\Gamma \vdash t : T$ using the algorithmic rules, then $\Gamma \vdash t : T$ using the “old” rules.

Theorem (Completeness)

If $\Gamma \vdash t : T$ using the old rules, then $\Gamma \vdash t : U$ for some $U \leq T$ using the algorithmic rules.

- Why does completeness have the extra clause on subtypes?

Algorithmic typing

- Prove these

Theorem (Soundness)

If $\Gamma \vdash t : T$ using the algorithmic rules, then $\Gamma \vdash t : T$ using the “old” rules.

Theorem (Completeness)

If $\Gamma \vdash t : T$ using the old rules, then $\Gamma \vdash t : U$ for some $U \leq T$ using the algorithmic rules.

- Why does completeness have the extra clause on subtypes?
- The old rules can freely use subsumption at any point, the algorithmic rules only when it is necessary

Outline

- 1 Subtype relation
- 2 Subtyping and other extensions
- 3 About subtyping
- 4 Summary**

Summarizing our peek to type theory

- We looked at simply typed lambda calculus
- ... defined its operational semantics, and a type system
- ... proved that the evaluator was type safe
 - In particular, we proved **progress** and **preservation**
- A few simple extensions (ascription, pairs)
- A slightly more complicated extension (subtyping)
 - we ignored safety proofs, but discussed the theorems
- Discussed how inference rules correspond to an implementation of a type checker, or how they don't

Look ahead

- **System-F** = polymorphic lambda calculus
 - $\Lambda X.t$ — type abstraction
 - $t [T]$ — type application
 - $\forall X.T$ — universal types
 - Example definition and use of a polymorphic function

$$\Lambda X.\lambda x:X.x : \forall Y.Y \rightarrow Y \qquad (\Lambda X.\lambda x:X.x) [\text{bool}] \text{ false}$$

- **F-bounded polymorphism**

$$\Lambda X<: T.\lambda x:X.u$$

Other stuff that we may or may not cover

- Types of types — kinds
- Typing of a ton of features
 - cells, exceptions, objects, intersection types, ...
- Types in action
 - XStatic, XDuce, $C\omega$, FGJ
 - Ownership types
 - Tracking resource usage
 - ...

Summary of summary

- What you should have gained from the tour of languages and type systems thus far:

Summary of summary

- What you should have gained from the tour of languages and type systems thus far:
- Gain deeper understanding on practical programming languages used today
- Basic understanding of the approach of formal type systems to programming language design, and its significance
- Ability to read and understand research papers on programming language design that discuss type systems
 - E.g. papers from OOPSLA, ECOOP, POPL