

Subtyping (cont)

Lecture 15

CS 565

4/3/08

Formalization of Subtyping

- Inversion of the subtype relation:
 - If $\sigma \prec: \tau_1 \rightarrow \tau_2$ then σ has the form $\sigma_1 \rightarrow \sigma_2$, and $\tau_1 \prec: \sigma_1$ and $\sigma_2 \prec: \tau_2$
 - If $\sigma \prec: \{l_i: \tau_i \mid i \in 1 \dots n\}$, then σ has the form $\{k_j: \sigma_j \mid j \in 1 \dots m\}$ with at least the labels $\{l_i \mid i \in 1 \dots n\} \subseteq \{k_j \mid j \in 1 \dots m\}$ and with $\sigma_j \prec: \tau_i$ for each common label $l_i = k_j$
 - This lemma says that a subtype of an arrow type must be an arrow type that is contravariant in the argument and covariant in the result, and
 - a subtype of a record type must be a record type that either has more fields (width) in some order (permute) and that the types of common fields obey the subtype relation (depth)
- Proof of preservation and progress follow as before using induction on typing derivations.

Casts and Ascription

- Ascription ($t \text{ as } \tau$) is a form of checked documentation:
 - it allows the programmer to record an assertion that the subterm of a complex expression has some specific type.
 - Ascription in the presence of subtyping raises interesting issues:
 - Casting

Casts

- Upcasts: a term is ascribed a supertype of the type that would normally be assigned.
 - Benign since it effectively “hides” some parts of a value so they cannot be used in some surrounding context.
- Downcasts: assign a type to a term that would not be assigned by the typechecker.
 - No specific relation between the term expected and the term desired.
 - List objects in Java expect elements belonging to `Object`.
 - When an element of type τ is stored, it is promoted to `Object`.
 - When an element is extracted, it must be downcast to a type suitable for the context in which it is to be used.
 - Requires runtime checks
 - Complicates implementations since type information must be presented at runtime to validate the cast.

Subtyping References

- Try covariance:

$$\frac{\sigma <: \tau}{\sigma \text{ ref } <: \tau \text{ ref}}$$

- Assume $\sigma <: \tau$
 - The following holds:
 - $x : \tau, y : \sigma \text{ ref}, f : \sigma \rightarrow \text{int} \vdash y := x ; f (! y)$
 - Unsound: f is called on a τ but is defined only on σ
- If we want covariance of references we can recover type safety with a runtime check for each $y := x$ assignment
 - The actual type of x matches the actual type of y
 - This is not a particularly good design.
 - Note: Java has covariant arrays

Subtyping References

Another approach (contravariance):

$$\frac{\sigma <: \tau}{\tau \text{ ref} <: \sigma \text{ ref}}$$

- Assume $\sigma <: \tau$
- The following holds:
 - $x : \tau, y : \tau \text{ ref}, f : \sigma \rightarrow \text{int} \vdash y := x; f (!y)$
 - Unsound: f is called on a τ but is defined only on σ
- references can be used in two ways:
 - for reading, context expects a value of type σ but the reference yields a value of type τ then we need $\tau <: \sigma$.
 - for writing, the new value provided by the context will have type σ . If the actual type of the reference is $\text{ref } \tau$, then this value may be read and used as a τ . This will only be safe if $\sigma <: \tau$.

Solution

- References should be typed invariantly:
 - No subtyping for references (unless there are runtime checks)
 - Arrays (which are implemented using updates) should be typed invariantly.
 - Similarly, mutable records should also be invariant.

Refinements

- A value of type $\text{ref } \tau$ can be used in two different ways: as a source for reading and a sink for writing:
 - Split $\text{ref } \tau$ into three parts:
 - $\text{source } \tau$: reference cell with "read" capability
 - $\text{sink } \tau$: reference cell with "write" capability
 - $\text{ref } \tau$: cell with both capabilities

Modified Typing Rules

$$\Gamma, \Sigma \vdash e_1 : \text{source } \tau_1$$

$$\Gamma, \Sigma \vdash !e_1 : \tau_1$$
$$\Gamma, \Sigma \vdash e_1 : \text{sink } \tau \quad \Gamma, \Sigma \vdash e_2 : \tau$$

$$\Gamma, \Sigma \vdash e_1 := e_2 : \text{Unit}$$

Subtyping Rules

$$\frac{\sigma <: \tau}{\text{source } \sigma <: \text{source } \tau}$$

covariant on reads

$$\frac{\tau <: \sigma}{\text{sink } \sigma <: \text{sink } \tau}$$

contravariant on writes

$\text{ref } \tau <: \text{source } \tau$

refs can be "downgraded"

$\text{ref } \tau <: \text{sink } \tau$

to source or sink

Coercion Semantics

- We have used a subset interpretation to reason about subtyping about base types.
 - Thus, $\text{int} \prec \text{real}$, but this is not particularly practical since integers and reals typically have different internal representations.
 - $4.5 + 6$ is well-typed under this interpretation
 - Performance penalties
 - Tagging datatypes
 - permutation rule on records

Coercions

- Adopt a different semantics:
 - compile-away subtyping by replacing it with runtime coercions:
 - If an Int is promoted to Real by the typechecker, at runtime we convert its representation from a machine integer to a machine float.
 - Express coercions using a conversion rule on types.

Type Coercions

$$\llbracket \text{Top} \rrbracket \Rightarrow \text{Unit}$$

$$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket \Rightarrow \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket$$

$$\llbracket \{l_i : \tau_i\} \rrbracket \Rightarrow \{l_i : \llbracket \tau_i \rrbracket\}$$

Subsumption Coercion

$$\left[\frac{}{\tau <: \tau} \right] \Rightarrow \lambda x: [\tau].x$$

$$\left[\frac{}{\tau <: \text{Top}} \right] \Rightarrow \lambda x: [\tau].\text{unit}$$

$$\left[\frac{C1::\tau_1 <: \tau_2 \quad C2::\tau_2 <: \tau_3}{\tau_1 <: \tau_3} \right] \Rightarrow \lambda x: [\tau_1]. [C2] ([C1]x)$$

$$\left[\frac{C1::\tau_3 <: \tau_1 \quad C2::\tau_2 <: \tau_4}{\tau_1 \rightarrow \tau_2 <: \tau_3 \rightarrow \tau_4} \right] \Rightarrow$$
$$\lambda f: [\tau_1 \rightarrow \tau_2]. \lambda x: [\tau_3]. [C2](f([C1]x))$$

Typing Statement Coercion

$$\left[\frac{x:\tau \in \Gamma}{\Gamma \vdash x:\tau} \right] \Rightarrow X$$

$$\left[\frac{D::\Gamma, x:\tau_1 \vdash t:\tau_2}{\Gamma \vdash \lambda x:\tau_1. t:\tau_1 \rightarrow \tau_2} \right] \Rightarrow \lambda x:[\tau_1]. [D]$$

$$\left[\frac{D1::\Gamma \vdash t1:\tau_1 \rightarrow \tau_2 \quad D2::\Gamma \vdash t2:\tau_1}{\Gamma \vdash t1 t2:\tau_2} \right] \Rightarrow [D1] [D2]$$

$$\left[\frac{D::\Gamma \vdash t:\tau_1 \quad C::\tau_1 <:\tau_2}{\Gamma \vdash t:\tau_2} \right] \Rightarrow [C] [D]$$

Coherence

- The place where coercions are injected into type derivations is relevant:
 - There may be several possible derivations that inject coercions at different places, leading to different behaviors.
- Other issues:
 - Can we build arbitrary subtype relations just because we can write conversion functions?
 - Is $\text{real} <: \text{int}$ simply because the “floor” function is a conversion?
 - What is the conversion from “ $\text{real} \rightarrow \text{int}$ ” to “ $\text{int} \rightarrow \text{int}$ ”?
- A system of conversion functions is coherent if whenever we have $\tau <: \tau' <: \sigma$:
 - $\llbracket \tau <: \tau \rrbracket = \lambda x.x$
 - $\llbracket \tau_1 <: \tau_2 \rrbracket = \llbracket \tau_3 <: \tau_2 \rrbracket \circ \llbracket \tau_1 <: \tau_3 \rrbracket$ -- coercions compositional

Example

- Say we want the following subtyping relations:
 - $\text{int} <: \text{real} \Rightarrow \lambda x:\text{int}. \text{toieee } x$
 - $\text{real} <: \text{int} \Rightarrow \lambda x:\text{real}. \text{floor } x$
- For this system to be coherent we need:
 - $\llbracket \text{int} <: \text{real} \rrbracket \circ \llbracket \text{real} <: \text{int} \rrbracket = \lambda x.x$
 - $\llbracket \text{real} <: \text{int} \rrbracket \circ \llbracket \text{int} <: \text{real} \rrbracket = \lambda x.x$
- This means that $\forall x: \text{real}, \text{toieee}(\text{floor } x) = x$
 - This is not true

Example

- Consider the type derivation of “printreal 1”:

$$\frac{1 : \text{int} \quad \text{int} <: \text{real}}{\text{printreal} : \text{real} \rightarrow \text{unit} \quad 1 : \text{real}}$$
$$\text{printreal } 1 : \text{unit}$$

we convert 1 to real: $\llbracket \text{int} <: \text{real} \rrbracket 1$

- This is also a valid derivation:

$$\frac{\text{printreal} : \text{real} \rightarrow \text{unit} \quad \text{real} \rightarrow \text{unit} <: \text{int} \rightarrow \text{unit}}{\text{printreal} : \text{int} \rightarrow \text{unit} \quad 1 : \text{int}}$$
$$\text{printreal } 1 : \text{unit}$$

we convert printreal: $\llbracket \text{real} \rightarrow \text{unit} <: \text{int} \rightarrow \text{unit} \rrbracket \text{printreal}$

- Which one is right?