Subtyping

COS 441
Princeton University
Fall 2004

Inclusive vs. Coercive Relationships

- Inclusive
  - Every cat is a feline
  - Every dog is a canine
  - Every feline is a mammal
- Coercive/isomorphism
  - Integers can be converted into floating point numbers
  - Booleans can be converted into interegers
  - Mammals with a tail can be converted to a mammals without a tail (ouch!)

Subtype Relation

Read $\tau_1 <: \tau_2$ as τ_1 is a subtype of τ_2 or τ_2 is a supertype of τ_1

Subtype relation is reflexive and transitive

$$\frac{}{\tau_1 <: \tau_2}\ \textbf{refl} \qquad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}\ \textbf{trans}$$

We say $(\tau_1 = \tau_2)$ iff $(\tau_1 <: \tau_2)$ and $(\tau_2 <: \tau_1)$

Implicit vs Explicit

Typing rules for subtyping can be rendered in either implicit or explicit form

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash e : \tau} \qquad \frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash (\tau)\, e : \tau}$$

cast

Simplification for Type-Safety

- Inclusive/Coercive distinction independent of Implicit/Explicit distinction
- Harper associates inclusive with implicit typing and coercive with explicit typing because it simplifies the type safety proof
  - You can have a inclusive semantics with explicit type casts
  - You can have a coercive semantics with implicit typing

Dynamic Semantics

- For inclusive system primitives must operate equally well for all subtypes of a give type for which the primitive is defined
- For coercive systems dynamic semantics simply must cast/convert the value appropriately

Varieties of Systems

- Implicit, Inclusive – Described by Harper
- Explicit, Coercive – Described by Harper
- Implicit, Coercive – Non-deterministic insertion of coercions
- Explicit, Inclusive – Type casts are no-ops in the operational semantics

Subtype Relation (cont.)

Given

$$\frac{}{\texttt{bool} <: \texttt{int}}\ \textbf{b2i} \qquad \frac{}{\texttt{int} <: \texttt{float}}\ \textbf{i2f}$$

via transitivity we can conclude
(`bool` <: `float`)

Subtyping of Functions

weight:	`mammal → float`
numberOfTeeth:	`mammal → int`
numberOfWiskers:	`feline → int`
pitchOfMeow:	`feline → float`
printInfo:	`(mammal * (mammal → float))→ unit`
printFelineInfo:	`(feline * (feline → float))→ unit`

Subtyping Quiz

`mammal → int`	<:	`mammal → float`
`feline → int`	<:	`feline → float`
`mammal → float`	<:	`feline → float`
`mammal → int`	<:	`feline → int`
`mammal → int`	<:	`feline → float`

Co/Contra Variance

argument is contravariant

return type is covariant

$$\frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

both are covariant

$$\frac{\tau_1 <: \tau_1' \quad \tau_2 <: \tau_2'}{(\tau_1 * \tau_2) <: (\tau_1' * \tau_2')}$$

Width vs Depth Subtyping

Consider the n-tuple (τ_1 * … * τ_n)

Width Subtyping

(`int * int * float`) <: (`int * int`)

$$\frac{m > n}{(\tau_1 * \ldots * \tau_m) <: (\tau_1 * \ldots * \tau_n)}\ \textbf{width}$$

Depth Subtyping

(`int * int`) <: (`float * float`)

$$\frac{\tau_1 <: \tau'_1 \ \ldots\ \tau_n <: \tau'_n}{(\tau_1 * \ldots * \tau_n) <: (\tau'_1 * \ldots * \tau'_n)}\ \textbf{depth}$$

Width and Depth for Records

Similar rule for records $\{l_1 : \tau_1,\ldots,l_n:\tau_n\}$
- Width considers any subset of labels since order of labels doesn't matter.

Implementing this efficiently can be tricky but doable

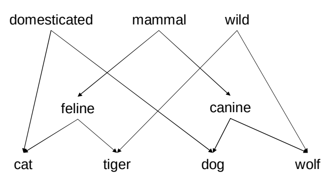
Subtyping and Mutability

Mutability destroys the ability to subtype
- τ ref = {get:unit $\rightarrow \tau$, set:$\tau \rightarrow$ unit}
- τ' ref = {get:unit $\rightarrow \tau'$, set:$\tau' \rightarrow$ unit}

Assume τ <: τ' from that we conclude
- unit $\rightarrow \tau$ <: unit $\rightarrow \tau'$ and
- $\tau' \rightarrow$ unit <: $\tau \rightarrow$ unit

Subtyping Defines Preorder/DAG

Subtyping relation can form any DAG



Typechecking With Subtyping

With explicit typing every expression has a unique type so we can use type synthesis to compute the type of an expression

Under implicit typing an expression may have many different types, which one should we choose?

e.g. CalicoCat : `mammal`,

CalicoCat : `feline`, and CalicoCat : `cat`

Which Type to Use?

Consider weight: `mammal` $\rightarrow$ `float`
countWiskers: `feline` $\rightarrow$ `int`

let val c = CalicoCat
in (weight c,countWiskers c)
end

What type should we use for c?

Which Type to Use?

Consider weight: `mammal` $\rightarrow$ `float`
countWiskers: `feline` $\rightarrow$ `int`

let val c: `mammal` = CalicoCat
in (weight c,countWiskers c)
end

What type should we use for c?

Which Type to Use?

Consider weight: `mammal` → `float`
countWiskers: `feline` → `int`

let val c: `feline` = CalicoCat
in (weight c,countWiskers c)
end

How do we know this is the “best” solution?

Which Type to Use?

Consider weight: `mammal` → `float`
countWiskers: `feline` → `int`

let val c: `cat` = CalicoCat
in (weight c,countWiskers c)
end

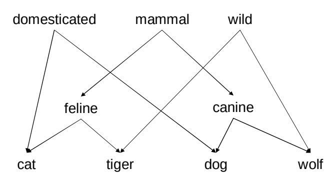
Choose the most specific type.

Principal Types

- Principal type is the “most specific” type. It is the least type in a given pre-order defined by the subtype relation
- Lack of principal types makes type synthesis impossible with *implicit* subtyping unless programmer annotates code
- Not at as big a problem for *explicit* subtyping rules

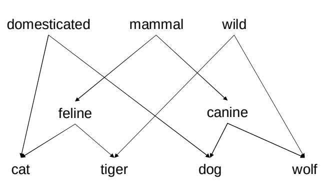
Subtyping Defines Preorder/DAG

Q: What is the least element “principal type” for “mammal”?



Subtyping Defines Preorder/DAG

A: “mammal” has no principal type in the subtyping relation defined below



Implementing Subtyping

- For inclusive based semantics maybe hard to implement or impose restrictions on representations of values
- Coercive based semantics give more freedom on choosing appropriate representation of values
- Can use “type-directed translation” to convert inclusive system to coercive system

Subtyping with Coercions

Define a new relation $\tau_1 <: \tau_2 \rightsquigarrow v$

Where v is a function of type $(\tau_1 \to \tau_2)$

$$\frac{}{\tau <: \tau \rightsquigarrow \mathtt{id}_\tau} \qquad \frac{\rho <: \sigma \rightsquigarrow v \quad \sigma <: \tau \rightsquigarrow v'}{\rho <: \tau \rightsquigarrow v;v'}$$

$$\frac{}{\mathtt{int} <: \mathtt{float} \rightsquigarrow \mathtt{to\_float}} \qquad \frac{\tau_1 <: \sigma_1 \rightsquigarrow v_1 \quad \sigma_2 <: \tau_2 \rightsquigarrow v_2}{\sigma_1 \to \sigma_2 <: \tau_1 \to \tau_2 \rightsquigarrow v_1 \to v_2}$$

Subtyping with Coercions (cont)

1. Primitive conversion: `to_float`.
2. Identity: $\mathtt{id}_\tau = \mathtt{fn}\, x{:}\tau\, \mathtt{in}\, x$.
3. Composition: $v;v' = \mathtt{fn}\, x{:}\tau\, \mathtt{in}\, v'(v(x))$.
4. Functions: $v_1 \to v_2 =$ $\mathtt{fn}\, f{:}\sigma_1 \to \sigma_2\, \mathtt{in}\, \mathtt{fn}\, x{:}\tau_1\, \mathtt{in}\, v_2(f(v_1(x)))$.

$$\frac{\Gamma \vdash e : \sigma \rightsquigarrow e' \quad \sigma <: \tau \rightsquigarrow v}{\Gamma \vdash e : \tau \rightsquigarrow v(e')}$$

Implementing Record Subtyping

Implementing subtyping on tuples is easy since address index “does the right thing”

((1, 2, 3) : (int * int * int)).2

((1, 2, 3) : (int * int)).2

Selecting the field label with records is more challenging

({a=1,b=2,c=3} : {a:int,b:int,c:int}).c

({a=1,b=2,c=3} : {a:int,c:int}).c

Approaches to Record Subtyping

Represent record as a “hash-table” keyed by label name

Convert record to tuple when coercing create new tuple that represents different record with appropriate fields

Two level approach represent record as “view” and value. Dynamically coerce views.

(Java interfaces are implemented this way, but you can statically compute all the views in Java)

By Name vs Structural Subtyping

Harper adopts a structural view of subtyping. Things are subtypes if they are some how isomorphic.

Java adopts a “by name” view. Things are subtypes if they are structurally compatible and the user declared them as subtypes.

Java approach leads to simpler type-checking and implementation but is arguably less modular than a pure structural approach

Summary

Coercive vs Inclusive
- Operational view of what subtyping means

Implicit vs Explicit
- How type system represents subtyping

Systems can support all possible combinations

Need to think things through to avoid bugs
- Tuples/records have both width and depth subtyping
- Functions are contravariant in argument type
- References are invariant