Subtyping Motivation

Fundamental tension between static type safety and reusability.

- distFromOrigin = λr:{x:Int, y:Int}. sqrt((r.x * r.x) + (r.y * r.y))
- ⊢ distFromOrigin : {x:Int, y:Int} → Real
- distFromOrigin accepts records containing x and y integer components and no other components
  - ▷ ⊢ (distFromOrigin {x=3, y=4}) : Real
  - ▷ ⊢ {x=3, y=4, color=1} : {x:Int, y:Int, color:Int}
  - ▷ (distFromOrigin {x=3, y=4, color=1}) is not well-typed

Parametric polymorphism doesn't help.

- It would be unsound to give distFromOrigin the type $\alpha \rightarrow$ Real.
- The argument to distFromOrigin must have integer components named x and y.

Subtyping Overview

Introduce a subtyping relation between types.

Informally, if T_1 is a subtype of T_2 (denoted $T_1 \leq T_2$), then T_1 is a "more-specific" type than T_2.

Made concrete by the principle of subtype substitutability: if $T_1 \leq T_2$, then any value of type T_1 can be safely used in a context expecting a value of type T_2

This solves the problem for distFromOrigin:

- ⊢ distFromOrigin : {x:Int, y:Int} → Real
- ⊢ {x=3, y=4, color=1} : {x:Int, y:Int, color:Int}
- {x:Int, y:Int, color:Int} ≤ {x:Int, y:Int}
- therefore ⊢ (distFromOrigin {x=3, y=4, color=1}) : Real

Formalizing Subtyping

Need to define the subtyping relation.

- Typically, each form of type has its own subtyping rule(s).
- Here is the syntax of types we'll discuss:

T	::=	$\{l_1 : T_1, \ldots, l_n : T_n\}$	record type
		$T_1 \rightarrow T_2$	function type
		Bool | Int	base types

Need to formalize subtype substitutability.

- Add a "subsumption" typing rule.

The Base Type System

$$\frac{\Gamma \vdash e_1 : T_1 \quad \cdots \quad \Gamma \vdash e_n : T_n \quad n \geq 0}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : T_1, \ldots, l_n : T_n\}} \text{(T-Rec)}$$

$$\frac{}{\Gamma \vdash \text{true} : \text{Bool}} \text{(T-True)}$$

$$\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \text{(T-If)}$$

$$\frac{\Gamma \vdash e : \{l_1 : T_1, \ldots, l_n : T_n\} \quad n \geq m \geq 0}{\Gamma \vdash e.l_m : T_m} \text{(T-Proj)}$$

$$\frac{}{\Gamma \vdash \text{false} : \text{Bool}} \text{(T-False)}$$

$$\frac{\Gamma \cup \{x : T_1\} \vdash e : T_2}{\Gamma \vdash (\lambda x : T_1 . e) : T_1 \rightarrow T_2} \text{(T-Abs)}$$

$$\frac{\Gamma \vdash e_1 : T_2 \rightarrow T \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1 \; e_2 : T} \text{(T-App)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \text{(T-Var)}$$

Subtyping Judgements

Introduce a new typing judgement of the form $T_1 \leq T_2$.

Define the meaning of the new judgement via a set of inference rules.

T_1 subtypes T_2 if there is a legal derivation tree whose root is $T_1 \leq T_2$.

Preliminaries

- Subtyping is reflexive.

$$\frac{}{T \leq T} \text{(S-Refl)}$$

- Subtyping is transitive.

$$\frac{T_1 \leq T_2 \quad T_2 \leq T_3}{T_1 \leq T_3} \text{(S-Trans)}$$

Width Subtyping for Records

As we've already seen, {x:Int, y:Int, color:Int} $\leq$ {x:Int, y:Int}.

- The type {x:Int, y:Int} now describes records with at least components x and y, both of type Int.
- The type {x:Int, y:Int, color:Int} is more specific, in that it further mandates a color component of type Int.

The general case is known as width subtyping, because we are allowed to use a "wider" record than expected.

$$\frac{n \geq m \geq 0}{\{l_1 : T_1, \ldots, l_n : T_n\} \leq \{l_1 : T_1, \ldots, l_m : T_m\}} \text{(S-RecWidth)}$$

Depth Subtyping for Records

Width subtyping requires the common components to be identically typed.

It is also sound to allow the more-specific record's components to have more-specific types.

The following function determines whether a line (represented by its endpoints) is horizontal or vertical.

- λl:{p:{x:Int,y:Int}, q:{x:Int,y:Int}}.(l.p.x=l.q.x or l.p.y=l.q.y)
- We should be able to pass {p={x=3,y=4}, q={x=5,y=4,color=1}} to the function.

The general case is known as depth subtyping, because we are allowed to use a "deeper" record than expected.

$$\frac{T_1 \leq T_1' \quad \cdots \quad T_n \leq T_n'}{\{l_1 : T_1, \ldots, l_n : T_n\} \leq \{l_1 : T_1', \ldots, l_n : T_n'\}} \text{(S-RecDepth)}$$

Order Subtyping for Records

The only thing you can do to a record — access its components — is insensitive to the order of those components.

Therefore, we should be able to re-order components safely.

- distFromOrigin = λr:{x:Int, y:Int}. sqrt((r.x * r.x) + (r.y * r.y))
- We should be able to pass {y=8,x=6} to distFromOrigin.

The general rule:

$$\frac{\{l_1 : T_1, \ldots, l_n : T_n\} \text{ is a permutation of } \{l_1' : T_1', \ldots, l_n' : T_n'\}}{\{l_1 : T_1, \ldots, l_n : T_n\} \leq \{l_1' : T_1', \ldots, l_n' : T_n'\}} \text{(S-RecPerm)}$$

Example Derivation

Let's show that {x:{a:Int,b:Int},y:Int} $\leq$ {x:{a:Int}}.

$$\dfrac{\dfrac{}{\{x{:}\{a{:}Int,b{:}Int\},y{:}Int\} \leq \{x{:}\{a{:}Int,b{:}Int\}\}}\ (\text{S-RecWidth}) \qquad \dfrac{\dfrac{}{\{a{:}Int,b{:}Int\} \leq \{a{:}Int\}}\ (\text{S-RecWidth})}{\{x{:}\{a{:}Int,b{:}Int\}\} \leq \{x{:}\{a{:}Int\}\}}\ (\text{S-RecDepth})}{\{x{:}\{a{:}Int,b{:}Int,y{:}Int\}\} \leq \{x{:}\{a{:}Int\}\}}\ (\text{S-Trans})$$

Function Subtyping

Since functions are first-class, we must say when it's safe to substitute one function for another.

Consider g = λf:$T_1 \rightarrow T_2$. ...f($\cdots$)... What assumptions can g make about the function f passed to it?

- f can be sent any value of type T_1
- f returns some value of type T_2

Therefore, a function f' can be safely passed to g if:

- f' can be sent any value of some supertype of T_1
- f' returns some value of some subtype of T_2

Function subtyping is contravariant in the argument type and covariant in the result type.

$$\dfrac{T_1' \leq T_1 \quad T_2 \leq T_2'}{T_1 \rightarrow T_2 \leq T_1' \rightarrow T_2'}\ (\text{S-Fun})$$

Contravariance Example

test = λa:{f:{x:Int,y:Int,color:Int} $\rightarrow$ {x:Int,y:Int},
p:{x:Int,y:Int,color:Int}}.(a.f a.p)

It is safe to pass the following function for f.

negate = λp:{x:Int,y:Int}.{x=(-p.x),y=(-p.y)}

- test {f=negate,p={x=3,y=4,color=1} $\longrightarrow$ {x=-3,y=-4}

It is not safe to pass the following function for f.

maybeNegate = λp:{x:Int,y:Int,color:Int,flag:Bool}.
if p.flag then (negate p) else p

- test {f=maybeNegate,p={x=3,y=4,color=1} $\longrightarrow$ CRASH

The Full Subtyping Relation

$$\dfrac{}{T \leq T}\ (\text{S-Refl}) \qquad \dfrac{T_1 \leq T_2 \quad T_2 \leq T_3}{T_1 \leq T_3}\ (\text{S-Trans})$$

$$\dfrac{n \geq m \geq 0}{\{l_1 : T_1, \ldots, l_n : T_n\} \leq \{l_1 : T_1, \ldots, l_m : T_m\}}\ (\text{S-RecWidth})$$

$$\dfrac{T_1 \leq T_1' \quad \cdots \quad T_n \leq T_n'}{\{l_1 : T_1, \ldots, l_n : T_n\} \leq \{l_1 : T_1', \ldots, l_n : T_n'\}}\ (\text{S-RecDepth})$$

$$\dfrac{\{l_1 : T_1, \ldots, l_n : T_n\} \text{ is a permutation of } \{l_1' : T_1', \ldots, l_n' : T_n'\}}{\{l_1 : T_1, \ldots, l_n : T_n\} \leq \{l_1' : T_1', \ldots, l_n' : T_n'\}}\ (\text{S-RecPerm})$$

$$\dfrac{T_1' \leq T_1 \quad T_2 \leq T_2'}{T_1 \rightarrow T_2 \leq T_1' \rightarrow T_2'}\ (\text{S-Fun})$$

Subsumption

Finally, we formalize subtype substitutability with an intuitive subsumption rule:

$$\frac{\Gamma \vdash E : T' \quad T' \leq T}{\Gamma \vdash E : T} \text{(T-Sub)}$$

An expression's type can be "weakened" to a supertype.

This rule is the bridge between the subtyping relation and the expression typing relation.

Subsumption Example

We can now solve the problem in our motivating example.

distFromOrigin = λr:{x:Int, y:Int}. sqrt((r.x * r.x) + (r.y * r.y))

Use subsumption to "weaken" the type of the argument.

$$\frac{\dfrac{\ldots}{\vdash \{x{=}3,\ y{=}4,\ color{=}1\} : \{x{:}Int,\ y{:}Int,\ color{:}Int\}} \quad \dfrac{}{\{x{:}Int,\ y{:}Int,\ color{:}Int\} \leq \{x{:}Int,\ y{:}Int\}}}{\vdash \{x{=}3,\ y{=}4,\ color{=}1\} : \{x{:}Int,\ y{:}Int\}}$$

Now the regular function application rule applies.

$$\frac{\dfrac{\ldots}{\vdash distFromOrigin : \{x{:}Int, y{:}Int\} \rightarrow Real} \quad \dfrac{\text{see above}}{\vdash \{x{=}3,\ y{=}4,\ color{=}1\} : \{x{:}Int,\ y{:}Int\}}}{\vdash (distFromOrigin\ \{x{=}3,\ y{=}4,\ color{=}1\}) : Real}$$

Two Approaches to Object-Oriented Calculi

Encode OO constructs in terms of "standard" language constructs like functions and records.

- allows us to build on existing frameworks, like the λ-calculus
- defines what OO constructs "really" are
- shows how OO constructs interact with other language features
- illustrates how to compile OO constructs

Treat OO constructs as primitives, giving them a direct semantics.

- typically much simpler than the encoding style
- naturally models existing OO languages
- a platform for experimentation with OO language design

The "Encoding" Style: Objects as Records

Pt = {x:Int,y:Int,getX:Pt→Int,getY:Pt→Int}

CPt = {x:Int,y:Int,color:Int,getX:Pt→Int,getY:Pt→Int,getC:CPt→Int}

- Note the need for recursive types.

myPt:Pt = {x=3,y=4,getX=λp:Pt.(p.x), getY=λp:Pt.(p.y)}

myCPt:CPt = {x=3,y=4,color=1,getX=λp:Pt.(p.x),
getY=λp:Pt.(p.y), getInt=λp:CPt.(p.color)}

Need some pretty heavyweight constructs to encode

- classes
- inheritance
- self-application semantics

The "Direct" Style: Featherweight Java

A core calculus for understanding Java's semantics.

- developed by Igarashi, Pierce, and Wadler in 1999.
- significantly simpler than previous formalisms for Java
- each FJ program is (essentially) a legal Java program
- no Greek letters!

Meant to capture the essence of Java, and nothing else.

- contains objects/classes, fields, methods, casting
- omits assignment, interfaces, overloading, super sends, exceptions, access control, base types, ...

Proven sound.

Successfully used to formalize extensions to the base language.

- GJ
- inner classes
- ArchJava

FJ Syntax

CL ::= class C extends D {$\overline{\text{C}}$ $\bar{\text{f}}$; K $\overline{\text{M}}$} classes

K ::= C($\overline{\text{C}}$ $\bar{\text{f}}$) {super($\bar{\text{g}}$); this.$\bar{\text{f}}$=$\bar{\text{f}}$;} constructors

M ::= C m($\overline{\text{C}}$ $\bar{\text{x}}$) {return e;} methods

e ::=		
	x	variable
	e.f	field access
	e.m($\bar{\text{e}}$)	message send
	new C($\bar{\text{e}}$)	object creation
	(C) e	cast

v ::= new C($\bar{\text{v}}$) values

Some FJ Classes

```
class A extends Object { A() { super(); } }

class B extends Object { B() { super(); } }

class Pair extends Object {
  Object fst;
  Object snd;
  Pair(Object fst, Object snd) {
     super(); this.fst=fst; this.snd=snd; }
  Pair setfst(Object newfst) {
     return new Pair(newfst, this.snd); }
}
```


Informal FJ Evaluation

field access

- new Pair(new A(), new B()).snd $\longrightarrow$ new B()

message send

- new Pair(new A(), new B()).setfst(new B()) $\longrightarrow$
  [newfst $\mapsto$ new B(), this $\mapsto$ new Pair(new A(), new B())]
  new Pair(newfst, this.snd) $\equiv$
  new Pair(new B(), new Pair(new A(), new B()).snd) $\longrightarrow$
  new Pair(new B(), new B())

cast

- ((Pair) new Pair(new A(), new Pair(new A(), new B())).snd).fst $\longrightarrow$
  ((Pair) new Pair(new A(), new B())).fst $\longrightarrow$
  new Pair(new A(), new B()).fst $\longrightarrow$
  new A()

Formal FJ Evaluation

A (mostly) standard call-by-value operational semantics.

Evaluating field access:

$$\frac{\mathrm{e} \longrightarrow \mathrm{e}'}{\mathrm{e.f} \longrightarrow \mathrm{e}'.\mathrm{f}}\ \text{(E-Field)} \qquad \frac{\mathrm{fields(C)} = \overline{\mathrm{C}}\ \overline{\mathrm{f}}}{(\mathrm{new\ C}(\overline{\mathrm{v}})).\mathrm{f}_i \longrightarrow \mathrm{v}_i}\ \text{(E-ProjNew)}$$

A "class table" CT maps class names to their definitions. These definitions are used to access information about a class's fields and methods.

$$\mathrm{fields(Object)} = \bullet$$

$$\frac{\begin{array}{c} CT(\mathrm{C}) = \mathrm{class\ C\ extends\ D}\ \{\overline{\mathrm{C}}\ \overline{\mathrm{f}};\ \mathrm{K}\ \overline{\mathrm{M}}\} \\ \mathrm{fields(D)} = \overline{\mathrm{D}}\ \overline{\mathrm{g}} \end{array}}{\mathrm{fields(C)} = \overline{\mathrm{D}}\ \overline{\mathrm{g}},\ \overline{\mathrm{C}}\ \overline{\mathrm{f}}}$$

Formal FJ Evaluation (cont.)

Evaluating message sends (the reduction rule):

$$\frac{\mathrm{mbody(m,C)} = (\overline{\mathrm{x}},\mathrm{e})}{(\mathrm{new\ C}(\overline{\mathrm{v}})).\mathrm{m}(\overline{\mathrm{u}}) \longrightarrow [\overline{\mathrm{x}} \mapsto \overline{\mathrm{u}},\ \mathrm{this} \mapsto \mathrm{new\ C}(\overline{\mathrm{v}})]\mathrm{e}}\ \text{(E-InvkNew)}$$

- mbody(m,C) returns the formal parameter list and body of class C's (possibly inherited) method named m

Evaluating casts (the reduction rule):

$$\frac{\mathrm{C} <: \mathrm{D}}{(\mathrm{D})(\mathrm{new\ C}(\overline{\mathrm{v}})) \longrightarrow \mathrm{new\ C}(\overline{\mathrm{v}})}\ \text{(E-CastNew)}$$

FJ Subtyping

In contrast with the structural subtyping we saw with records and functions, Java (like most OO languages) has by-name (nominal) subtyping.

$$\frac{}{\mathrm{C} <: \mathrm{C}} \qquad \frac{\mathrm{C} <: \mathrm{D} \quad \mathrm{D} <: \mathrm{E}}{\mathrm{C} <: \mathrm{E}} \qquad \frac{CT(\mathrm{C}) = \mathrm{class\ C\ extends\ D}\ \{\ldots\}}{\mathrm{C} <: \mathrm{D}}$$

Structural subtyping is seen as more elegant.

- Types are completely self-describing.
- Subtyping is essentially inferred.
- Easier to manage in a formal setting.

By-name subtyping matches real languages.

- Class names are (sort of) a form of abstract data type.
- Naming provides a simple form of recursion.
- By-name subtyping is natural in the presence of inheritance.
- Class names are tags used for dynamic dispatching.

FJ Typechecking

Message send typing:

$$\frac{\begin{array}{c} \Gamma \vdash \mathrm{e}_0 : \mathrm{C}_0 \\ \mathrm{mtype(m,C_0)} = \overline{\mathrm{D}} \rightarrow \mathrm{C} \\ \Gamma \vdash \overline{\mathrm{e}} : \overline{\mathrm{C}} \quad \Gamma \vdash \overline{\mathrm{C}} <: \overline{\mathrm{D}} \end{array}}{\Gamma \vdash \mathrm{e}_0.\mathrm{m}(\overline{\mathrm{e}}) : \mathrm{C}}$$

Object creation:

$$\frac{\begin{array}{c} \mathrm{fields(C)} = \overline{\mathrm{D}}\ \overline{\mathrm{f}} \\ \Gamma \vdash \overline{\mathrm{e}} : \overline{\mathrm{C}} \quad \Gamma \vdash \overline{\mathrm{C}} <: \overline{\mathrm{D}} \end{array}}{\Gamma \vdash \mathrm{new\ C}(\overline{\mathrm{e}}) : \mathrm{C}}$$

- "Algorithmic subtyping," instead of a single subsumption rule.

FJ Typechecking (cont.)

Method typing:

$$\frac{\overline{\text{x}}\text{:}\overline{\text{C}}\text{,this:C} \vdash \text{e}_0 : \text{D}_0 \quad \text{D}_0 <: \text{C}_0 \quad \mathit{CT}(\text{C}) = \text{class C extends D } \{\ldots\} \quad \text{override(m,D,}\overline{\text{C}} \rightarrow \text{C}_0\text{)}}{\text{C}_0 \text{ m(}\overline{\text{C}}\ \overline{\text{x}}\text{) \{return e}_0\text{;\} OK in C}}$$

- Weird new kind of typing judgement, because methods are not stand-alone entities (and aren't first-class).
- The analogue of the rule for typechecking lambdas.
- The override relation ensures equivariant method overriding.