Subtyping Motivation

Fundamental tension between static type safety and reusability,

- \( \text{distFromOrigin} = \lambda r: \{ x: \text{Int}, y: \text{Int} \}. \sqrt{r.\text{x} \times r.\text{x}} + (r.\text{y} \times r.\text{y}) \)
- \( \vdash \text{distFromOrigin} : \{ x: \text{Int}, y: \text{Int} \} \rightarrow \text{Real} \)
- \( \text{distFromOrigin} \) accepts records containing \( x \) and \( y \) integer components and no other components
  \[ \vdash (\text{distFromOrigin} \{ x=3, y=4 \}) : \text{Real} \]
  \[ \vdash \{ x=3, y=4, \text{color}=1 \} : \{ x: \text{Int}, y: \text{Int}, \text{color}: \text{Int} \} \]
  \[ (\text{distFromOrigin} \{ x=3, y=4, \text{color}=1 \}) \text{ is not well-typed} \]

Parametric polymorphism doesn’t help,

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

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:
  \[ T ::= \{ \text{l}_1 : T_1, \ldots, \text{l}_n : T_n \} \]
  \[ T_1 \rightarrow T_2 \]
  \[ \text{Bool} \mid \text{Int} \]

Need to formalize subtype substitutability,

- Add a “subsumption” typing rule.

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 \( \text{distFromOrigin} \):

- \( \vdash \text{distFromOrigin} : \{ x: \text{Int}, y: \text{Int} \} \rightarrow \text{Real} \)
- \( \vdash \{ x=3, y=4, \text{color}=1 \} : \{ x: \text{Int}, y: \text{Int}, \text{color}: \text{Int} \} \)
- \( \{ x: \text{Int}, y: \text{Int}, \text{color}: \text{Int} \} \leq \{ x: \text{Int}, y: \text{Int} \} \)
- therefore \( \vdash (\text{distFromOrigin} \{ x=3, y=4, \text{color}=1 \}) : \text{Real} \)

The Base Type System

\[
\begin{align*}
\Gamma \vdash e_1 : T_1 & \quad \ldots \quad \Gamma \vdash e_n : T_n & n \geq 0 \quad \text{(T-Rec)} \\
\Gamma \vdash \{ l_1 = e_1, \ldots, l_n = e_n \} : \{ l_1 : T_1, \ldots, l_n : T_n \} & \quad \text{record type} \\
\Gamma \vdash e, l_m : T_m & n \geq m \geq 0 \quad \text{function type} \\
\Gamma \vdash e : \{ l_1 : T_1, \ldots, l_n : T_n \} & n \geq m \geq 0 \quad \text{base types} \\
\Gamma \vdash e_1 : \text{Bool} & \quad \Gamma \vdash e_2 : T & \quad \Gamma \vdash e_3 : T \quad \text{(T-If)} \\
\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T & \quad \text{if-then-else} \\
\Gamma \vdash \text{false} : \text{Bool} & \quad \text{(T-False)} \\
\Gamma \vdash e : T_1 & \quad \Gamma \vdash \{ x : T_1 \} \vdash e : T_2 \quad \text{(T-Abs)} \\
\Gamma \vdash (\lambda x : T_1. e) : T_1 \rightarrow T_2 & \quad \text{lambda abstraction} \\
\Gamma \vdash e_1 : T_2 & \quad \Gamma \vdash e_2 : T_2 \quad \text{(T-App)} \\
\Gamma \vdash e_1 e_2 : T & \quad \text{application} \\
\Gamma \vdash x : T & \in \Gamma \quad \text{(T-Var)} \\
\end{align*}
\]
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.
  
  \[ T \leq T \quad \text{(S-Refl)} \]

- Subtyping is transitive.
  
  \[ \frac{T_1 \leq T_2 \quad T_2 \leq T_3}{T_1 \leq T_3} \quad \text{(S-Trans)} \]

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.

- $\lambda!:\{p:\{x:\text{Int}, y:\text{Int}\}, q:\{x:\text{Int}, y:\text{Int}\}\}.(\text{if } p.x = q.x \text{ or } p.y = q.y)$

- We should be able to pass $\{p=\{x=3, y=4\}, q=\{x=5, y=4, \text{color}=1\}\}$ to the function.

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

\[ \frac{T_1 \leq T'_1 \quad \ldots \quad T_n \leq T'_n}{\{l_1 : T_1, \ldots, l_n : T_n\} \leq \{l'_1 : T'_1, \ldots, l'_n : T'_n\}} \quad \text{(S-RecDepth)} \]

Order Subtyping for Records

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

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

- $\text{distFromOrigin} = \lambda r:\{x:\text{Int}, y:\text{Int}\}. \text{sqrt}((r.x \ast r.x) + (r.y \ast r.y))$

- We should be able to pass $\{y=8, x=6\}$ to distFromOrigin.

The general rule:

\[ \frac{\{l_1 : T_1, \ldots, l_n : T_n\} \text{ is a permutation of } \{l'_1 : T'_1, \ldots, l'_n : T'_n\}}{\{l_1 : T_1, \ldots, l_n : T_n\} \leq \{l'_1 : T'_1, \ldots, l'_n : T'_n\}} \quad \text{(S-RecPerm)} \]
Example Derivation

Let’s show that \( \{x: \{a:int,b:int\}, y:int\} \leq \{x:a:int\} \).

\[
\frac{\{a:int,b:int\}}{\{a:int\}} \quad \text{(S-RecWidth)}
\]

\[
\frac{\{x:a:int\} \leq \{x:a:int\}}{\{x:a:int\} \leq \{x:a:int\}} \quad \text{(S-RecDepth)}
\]

\[
\frac{\{x:a:int\} \leq \{x:a:int\}}{\{x:a:int\} \leq \{x:a:int\}} \quad \text{(S-Trans)}
\]

Function Subtyping

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

Consider \( g = \lambda f : T_1 \rightarrow T_2. \ldots f(\cdots) \ldots \). 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.

\[
\frac{T_1' \leq T_1 \quad T_2 \leq T_2'}{T_1 \rightarrow T_2 \leq T_1' \rightarrow T_2'} \quad \text{(S-Fun)}
\]

Contravariance Example

\[
\text{test} = \lambda a: \{f:x{:}int,y{:}int,\text{color{:}int}\} \rightarrow \{x{:}int,y{:}int\}, \quad p: \{x{:}int,y{:}int,\text{color{:}int}\}, (a,f,a,p)
\]

It is safe to pass the following function for \( f \).

\[
\text{negate} = \lambda p: \{x{:}int,y{:}int\}. \{x{-}(\neg p,x) \mid y{-}(\neg p,y)\}
\]

- \( \text{test} \{f=\text{negate},p=\{x{-}3,y{-}4,\text{color{-}1}\}\} \rightarrow \{x{-}3,y{-}4\} \)

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

\[
\text{maybeNegate} = \lambda p: \{x{:}int,y{:}int,\text{color{:}int,flag{:}bool}\}, \quad \text{if } p,\text{flag then (negate p) else } p
\]

- \( \text{test} \{f=\text{maybeNegate},p=\{x{-}3,y{-}4,\text{color{-}1}\}\} \rightarrow \text{CRASH} \)
Subsumption

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

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

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.

\[
\text{distFromOrigin} = \lambda r:\{x: \text{Int}, y: \text{Int}\}. \sqrt{((r.x \cdot r.x) + (r.y \cdot r.y))}
\]

Use subsumption to “weaken” the type of the argument.

\[
\vdash \{x: 3, y: 4, \text{color} = \text{Int} \} \quad \text{see above}
\]

Now the regular function application rule applies,

\[
\vdash \text{distFromOrigin} \quad \vdash \{x: 3, y: 4, \text{color} = \text{Int} \} : \{x: \text{Int}, y: \text{Int}\} \quad \vdash \{x: 3, y: 4, \text{color} = \text{Int} \} : \text{Real}
\]

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 \(\lambda\)-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 = \(\{x: \text{Int}, y: \text{Int}, \text{getX}: \text{Pt} \rightarrow \text{Int}, \text{getY}: \text{Pt} \rightarrow \text{Int}\}\)

\(\text{CPt} = \{x: \text{Int}, y: \text{Int}, \text{color}: \text{Int}, \text{getX}: \text{Pt} \rightarrow \text{Int}, \text{getY}: \text{Pt} \rightarrow \text{Int}, \text{getC}: \text{CPt} \rightarrow \text{Int}\}\)
- Note the need for recursive types.

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

\(\text{myCpt}: \text{CPt} = \{x: 3, y: 4, \text{color} = \text{Int}, \text{getX} = \lambda p: \text{Pt}, (p.x), \text{getC} = \lambda p: \text{Pt}, (p.y)\}\)

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

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 set fst(Object new fst) {
    return new Pair(new fst, this.snd); }
}

Informal FJ Evaluation

field access
- new Pair(new A(), new B()).snd → new B()

message send
- new Pair(new A(), new B()).set fst(new B()) →
  [new fst → new B(), this → new Pair(new A(), new B())]
  new Pair(new fst, this.snd) ≡
  new Pair(new B(), new Pair(new A(), new B()).snd) →
  new Pair(new B(), new B())

cast
- ((Pair) new Pair(new A(), new Pair(new A(), new B())).snd).fst →
  ((Pair) new Pair(new A(), new B())).fst →
  new Pair(new A(), new B()).fst →
  new A()
Formal FJ Evaluation

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

Evaluating field access:
\[ e \cdot f \rightarrow e' \cdot f \] 
\[ \text{fields}(C) = \overline{C} \; \tilde{f} \] 
\[ \text{new } C(\overline{v}) \cdot f_i \rightarrow v_i \] 
\[ \text{E-Field} \]
\[ \text{E-Field} \]

A “class table” \( CT \) maps class names to their definitions. These definitions are used to access information about a class’s fields and methods,

\[ \text{fields}(\text{Object}) = \bullet \]

\[ CT(C) = \text{class } C \text{ extends } D \{ \overline{C} \; \tilde{f}; \; \text{K M} \} \]
\[ \text{fields}(D) = \overline{D} \; \tilde{g} \]
\[ \text{fields}(C) = \overline{D} \; \tilde{g}, \; \overline{C} \; \tilde{f} \]

FJ Subtyping

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

\[ C <: C \]
\[ C <: D \]
\[ D <: E \]
\[ CT(C) = \text{class } C \text{ extends } D \{ \ldots \} \]
\[ C <: D \]

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.

Formal FJ Evaluation (cont.)

Evaluating message send (the reduction rule):

\[ \text{mbody}(m, C) = (x, e) \]
\[ \text{new } C(\overline{v}) \cdot m(\overline{u}) \rightarrow [x \mapsto u, \text{this } \mapsto \text{new } C(\overline{v})] e \] 
\[ \text{E-InvkNew} \]

- \( \text{mbody}(m, C) \) returns the formal parameter list and body of class C’s (possibly inherited) method named m

Evaluating casts (the reduction rule):

\[ C <: D \]
\[ (D)(\text{new } C(\overline{v})) \rightarrow \text{new } C(\overline{v}) \] 
\[ \text{E-CastNew} \]

FJ Typechecking

Message send typing:

\[ \Gamma \vdash e_0 : C_0 \]
\[ \text{mtype}(m, C_0) = D \rightarrow C \]
\[ \Gamma \vdash \overline{e} : \overline{C} \]
\[ \Gamma \vdash \overline{c} <: D \]
\[ \Gamma \vdash e_0 \cdot m(\overline{e}) : C \]

Object creation:

\[ \text{fields}(C) = \overline{D} \; \tilde{f} \]
\[ \Gamma \vdash \overline{e} : \overline{C} \]
\[ \Gamma \vdash \overline{c} <: \overline{D} \]
\[ \Gamma \vdash \text{new } C(\overline{e}) : C \]

- “Algorithmic subtyping,” instead of a single subsumption rule.
Method typing:

\[
\begin{align*}
\bar{x} : C, \text{this} : C & \vdash e_0 : D_0 \quad D_0 \ll C_0 \\
CT(C) & = \text{class } C \text{ extends } D \{ \ldots \} \\
\text{override}(m, D, C & \rightarrow C_0) \\
\frac{}{C_0 \ m(C \bar{x}) \ \{ \text{return } e_0 \} \ \text{OK in } C}
\end{align*}
\]

- 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.