Subtyping

"*Type checking object-oriented languages is difficult*"
Kim Bruce, Foundations of Object Oriented Languages, MIT Press, 2002.

April 4, 2019

Introduction

- Since mid 80s numerous efforts to provide rigorous foundations for OOPL
- Two alternatives:
  - Encode objects in functional languages
    - Pioneering work of Cardelli in 1984
    - Use functions, records, recursion and subtyping
    - [Bruce, 2002] Foundations of Object Oriented Languages
  - Propose foundational calculi (like lambda calculi) for OOP
    - [Abadi y Cardelli, 1996] A Theory of Objects
    - [Castagna, 1997] Object-Oriented Programming: A Unified Foundation

What is a Type Error in an OOPL?

- In addition to standard errors such as:
  - methods receive the right number and type of parameters
  - assignments respect type of variables
- There is a new kind of error:
  - Invocation of inexistant methods
- We'll see details later

Next Topics

- Types in OOPL
- Subtyping and inheritance ("Width-subtyping")
- Invariant type systems
- Drawbacks of invariant type systems
- Subtyping and inheritance ("Depth-subtyping")
- Drawbacks (eg. binary methods)

Example

```
class Point {
  int x,y;
  public int getX() { ... }
  public int getY() { ... }
  public int dist(Point aPoint) { ... }
}
```

Type of `new Point()`?

Nominal	Structural

the name `Point`	the record type
	`PointType = {`
	`  getx: Unit -> Int;`
	`  gety: Unit -> Int;`
	`  dist: PointType -> Int`
	`}`

Types for OOPL

- Notion of class and type is separate
  - Class: inherently related to implementation (eg. instance variables, source code of methods, etc.)
  - Type of an object: its public interface
    - names of its methods
    - type of the arguments of each method and type of result

  Specification of an object (type) $\neq$ Implementation (class)
- This separation benefits modular development: various classes can implement the same type
- The type of an object is sometimes known as its interface type

Subtyping Judgement

- If the class is `C`, then we write `CType` for its type
- If `C` is a subclass of `D`, how is `CType` and `DType` related?

Subsumption

$$\sigma <: \tau$$

- Read, "In every context where one expects an expression of type τ, one may use an expression of type σ in its place without causing a run-time error"
- In particular, if `D` is a subclass of `C`, then one expects:

$$\texttt{DType} <: \texttt{CType}$$

  - But more general situations are also captured
- What is the relation between $\Gamma \triangleright M : \sigma$ and $\sigma <: \tau$?

Substitution Principle

$$\sigma <: \tau$$

- Read, "In every context where one expects an expression of type τ, one may use an expression of type σ in its place without causing a run-time error"
- Reading is reflected in the theory with a new rule called Subsumption:

$$\frac{\Gamma \triangleright M : \sigma \quad \sigma <: \tau}{\Gamma \triangleright M : \tau}\ (\text{T-Subs})$$

- We next recall the type system of the Lambda Calculus with records

Typing for LC with Records and Subtyping ($\lambda^{\rightarrow}_{<:}$)

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

$$\frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd \lambda x : \sigma . M : \sigma \to \tau}\ \text{(T-Abs)} \quad \frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M\, N : \tau}\ \text{(T-App)}$$

$$\frac{\Gamma \rhd M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \rhd \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}}\ \text{(T-Rcd)}$$

$$\frac{\Gamma \rhd M : \{l_i : \sigma_i\ ^{i \in 1..n}\} \quad j \in 1..n}{\Gamma \rhd M.l_j : \sigma_j}\ \text{(T-Proj)}$$

$$\frac{\Gamma \rhd M : \sigma \quad \sigma <: \tau}{\Gamma \rhd M : \tau}\ \text{(T-Subs)}$$

Subtyping as a Preorder

$$\frac{}{\sigma <: \sigma}\ \text{(S-Refl)} \qquad \frac{\sigma <: \tau \quad \tau <: \rho}{\sigma <: \rho}\ \text{(S-Trans)}$$

Note:

- No antisymmetry

Subtyping for Base Types

- For base types we assume that we have been informed of how they are related; for example

$$\begin{array}{rcl} Nat & <: & Float \\ Int & <: & Float \\ Bool & <: & Nat \end{array}$$

Digression: Nominal Typing à la Java

- Java (nominal subtyping):
  - Associates a symbol #C to each class C
  - New subtyping axioms::

    $$\#C <: \#D$$

    if `class C extends D` appears in our program
- Our approach (structural subtyping):
  - Associate a record type CType to each class C
  - Determine if CType<:DType on the basis of the structure of the records
  - We next take a look at subtyping for record types

Reading: Is Structural Subtyping Useful? An Empirical Study. Donna Malayeri, Jonathan Aldrich, ESOP 2009.

Width Subtyping for Record Types

```
{name: String, age:Int} <: {name:String}
```

The general case is:

$$\frac{}{\{l_i : \sigma_i | i \in 1..n+k\} <: \{l_i : \sigma_i | i \in 1..n\}} \text{ (S-RcdWidth)}$$

Note:

- $\sigma <: \{\}$, for all record types σ
- Is there any record type τ s.t. $\tau <: \sigma$, for all record types σ?

Another Example

```java
class Point {
  int x,y;
  public int getX() { ... }
  public int getY() { ... }
  public int dist(Point aPoint) { ... }
}
```

```java
class ColorPoint extends Point {
  String col;
  public String getCol() { ... }
}
```

Note that `ColorPointType<:PointType` where

```
PointType = {
  getx: Unit -> Int;
  gety: Unit -> Int;
  dist: PointType -> Int;
}
```

```
ColorPointType = {
  getx: Unit -> Int;
  gety: Unit -> Int;
  getCol: Unit -> Int;
  dist: PointType -> Int;
}
```

Limitations of Width Subtyping – Shallow Cloning

- Cloning: operation for copying an object
- Shallow cloning (in contrast with deep cloning):
  - Copy values of instance variables and same set of instance methods
  - If instance variables refer to other objects, then only copy the references themselves (and not the objects referred to)

Example – Cloning

```
class Object {
 public ?? clone() {
    ↪  ... }
}
```

```
class Cell extends
    ↪ Object {
 public ?? clone() {
    ↪  ... }
}
```

- What is the type of `clone`?
  - In `Object`: must return value of type `ObjectType`
  - In `Cell`: must return value of type `CellType`
  - In invariant type systems, `clone` must have type `Object`, even if the methods returns a value of type`CellType`!

```
ObjectType = {
  clone: Unit->ObjectType;
}
```

```
CellType = {
  clone: Unit->ObjectType;
}
```

- Programmer forced to insert type cast to "correct" the type system

Example - Shallow Cloning

```
ObjectType = {
  clone: Unit->ObjectType;
}
```

```
CellType = {
  clone: Unit->ObjectType;
}
```

- If `m` is a method of the class `Cell` and `o` is an instance variable of type `CellType`, the expression

  `(o clone()) m()`

  generates a type error
- The programmer must insert type cast

  `[CellType](o clone()) m()`

Type Casts

- Type casts are a means to help the type system out
- Two kinds of typecast
  - "up cast": `[CType]e` where `e` has type `DType` and `D` is a subclass of `C`
  - "down cast": `[DType]e` where `e` has type `CType` and `D` is a subclass of `C`
- In contrast to down casts, up casts are rarely used

Note: Need to resort to type casts are evidence of the limitations of a type system

Can we avoid casts? Depth Subtyping

```
class Object {
 public ?? clone() { ... }
}
```

```
class Cell extends Object {
 public ?? clone() { ... }
}
```

It would be desirable to have `CellType<:ObjectType`, where

```
ObjectType = {
  clone: Unit->ObjectType;
}
```

```
CellType = {
  clone: Unit->CellType;
}
```

Depth Subtyping for Record Types

```
{a: Student , b:Int} <: {a:Person}
```

The rule is

$$\frac{\sigma_i <: \tau_i \quad i \in I = \{1..n\}}{\{l_i : \sigma_i\}_{i \in I} <: \{l_i : \tau_i\}_{i \in I}} \text{(S-RcdDepth)}$$

Examples

$$\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} \\ <: \\ \{x : \{a : Nat\}, y : \{\}\}$$

$$\dfrac{\dfrac{}{\{a : Nat, b : Nat\} <: \{a : Nat\}}\ (\text{S-RcdWidth}) \qquad \dfrac{}{\{m : Nat\} <: \{\}}\ (\text{S-RcdWidth})}{\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{\}\}}\ (\text{S-RcdDepth})$$

Examples

$$\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} \\ <: \\ \{x : \{a : Nat\}, y : \{m : Nat\}\}$$

$$\dfrac{\dfrac{}{a : Nat, b : Nat <: a : Nat}\ (\text{S-RcdWidth}) \qquad \dfrac{}{m : Nat <: \{m : Nat\}}\ (\text{S-Refl})}{\{x : \{a : Nat, b : Nat\}, y : \{m : Nat\}\} <: \{x : \{a : Nat\}, y : \{m : Nat\}\}}\ (\text{S-RcdDepth})$$

Permutations of Fields

- The order of fields in a record should be irrelevant

$$\frac{\{k_j : \sigma_j | j \in 1..n\} \text{ permutation of } \{l_i : \tau_i | i \in 1..n\}}{\{k_j : \sigma_j | j \in 1..n\} <: \{l_i : \tau_i | i \in 1..n\}} \text{ (S-RcdPerm)}$$

Note:

- (S-RcdPerm) may be used in combination with (S-RcdWidth) y (S-Trans) to ignore fields in any part of a record type

Combining Width, Depth and Permutation

$$\frac{\{l_i|\ i \in 1..n\} \subseteq \{k_j|\ j \in 1..m\} \qquad k_j = l_i \Rightarrow \sigma_j <: \tau_i}{\{k_j : \sigma_j|\ j \in 1..m\} <: \{l_i : \tau_i|\ i \in 1..n\}} \text{ (S-Rcd)}$$

Subtyping

Up to now we have considered:

- Base types
- Records

We now consider

- Functions
- Lists
- Arrays
- References

Subtyping for Function Types

$$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'} \text{(S-Func)}$$

- Note: $<:$ reverses the type of the domain but not that of the range
- We say that the functional type constructor is contravariant in its first argument and variant in its second.

For example:

$$\mathit{Unit} \to \mathit{CellType} <: \mathit{Unit} \to \mathit{ObjectType}$$

Subtyping for Function Types

$$\frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \rightarrow \tau <: \sigma' \rightarrow \tau'} \text{(S-Func)}$$

If a context/program P expects an expression f of type $\sigma' \rightarrow \tau'$ it may receive one of type $\sigma \rightarrow \tau$ if the indicated conditions hold

- f is applied to arguments of type σ'
- These are coerced to arguments of type σ
- f, whose real type is $\sigma \rightarrow \tau$, is applied
- Finally, the result is coerced to τ', the type that P is expecting

For example:

$$\mathit{Unit} \rightarrow \mathit{CellType} <: \mathit{Unit} \rightarrow \mathit{ObjectType}$$

The type *Top*

Similar to `Object` class in Smalltalk

$$\frac{}{\sigma <: Top} \text{ (S-Top)}$$

- Is there a type σ s.t. $\sigma \to \sigma <: \sigma$?
- Note that $Top \times Top <: Top$
- What happens with $Top \to Top$? $Top \to Top <: Top$

Subtyping Collections

List σ Is it covariant? What about contravariant?

$$\frac{\sigma <: \tau}{\mathit{List}\ \sigma <: \mathit{List}\ \tau}$$

It is covariant (in most languages)

Subtyping References

Covariant? Imagine the rule:

$$\frac{\sigma <: \tau}{\mathit{Ref}\ \sigma <: \mathit{Ref}\ \tau}$$

What happens?

Ref is not Covariant

```
let r = ref aStudent    (*  r:Ref Student *)
in
  r := aPerson;
  (!r).aStudentId       (* Runtime exception *)
```

$$\frac{Student <: Person}{Ref\ \ Student <: Ref\ \ Person}$$

Ref is not Contravariant

Contravariant? Imagine this rule:

$$\frac{\sigma <: \tau}{\mathit{Ref}\,\tau <: \mathit{Ref}\,\sigma}$$

Again, what happens?

Ref is not Contravariant

```
let r = ref aPerson   (* Ref Person *)
in (!r).studentId   (* Runtime exception *)
```

$$\frac{Student <: Person}{Ref\ \ Person <: Ref\ \ Student}$$

Ref is Invariant

$$\frac{\sigma <: \tau \quad \tau <: \sigma}{\mathit{Ref}\,\sigma <: \mathit{Ref}\,\tau}$$

"Only references of equivalent types may be compared."

Covariant Subtyping for Arrays in Java

- The following code passes the type checker but generates a run-time error!
- `Exception in thread "main" java.lang.ArrayStoreException: prueba.A`
  `at prueba.Main.main(arreglo.java:11)`

```java
package prueba;
import java.io.*;

class A { int j; };
class B extends A { int i; };
class Main {
 public static void main(String argv[]) throws
     ↪ IOException {
 B[] b = new B[5];
 A[] a = b;
 a[1] = new A();
}};
```

Refining the Ref Type Constructor

- Reynolds in Forsythe (1988) separated references in two kinds:
- *Source* σ read
- *Sink* σ write
- We still have $\mathit{Ref}\,\sigma$ for read/write

$$\frac{\Gamma|\Sigma \rhd M : \mathit{Source}\ \sigma}{\Gamma|\Sigma \rhd\ !M : \sigma} \qquad \frac{\Gamma|\Sigma \rhd M : \mathit{Sink}\ \sigma \quad \Gamma|\Sigma \rhd N : \sigma}{\Gamma|\Sigma \rhd M := N : \mathit{Unit}}$$

Example of use of Source

$$\frac{\sigma <: \tau}{Source\,\sigma <: Source\,\tau}\ (SSource) \qquad \frac{Student <: Person}{Source\ Student <: Source\ Person}$$

!r may be seen as float even though r is source int (due to t-sub)

```
let r = ref aStudent (* r:Source Student *)
in
!r (* Source Student <: Source Person *)
end :: Person
```

"If one expects to read from a ref to T, then one may expect a ref to a lower, less informative, type"'

Example of use of Sink

$$\frac{\tau <: \sigma}{\mathit{Sink}\ \sigma <: \mathit{Sink}\ \tau}\ (\mathit{SSink}) \qquad \frac{\mathit{Student} <: \mathit{Person}}{\mathit{Sink\ Person} <: \mathit{Sink\ Student}}$$

```
let r = ref aPerson (* r:Sink Person *)
in
  r := aStudent; (* Sink Person <: Sink Student *)
  !r
```

- `r := aStudent` holds since r is Sink Person and (due to t-sub) it can be seen as Sink Student.
- "If one expects to write to a ref T, then may expect a ref of a higher, less informative type"

Relating Sink and Source with Ref

Every context in which we expect Source (or Sink), can receive Ref instead:

$$\frac{}{\mathit{Ref}\ \tau <: \mathit{Source}\ \tau}\ \text{(S-RefSource)} \qquad \frac{}{\mathit{Ref}\ \tau <: \mathit{Sink}\ \tau}\ \text{(S-RefSink)}$$

Exercise

Let σ be a type. Which of these are related by $<:$?

- *Ref* σ
- *Ref Ref* σ
- *Sink* σ
- *Source* σ
- *Ref Sink* σ
- *Source Ref* σ
- *Source Source* σ
- *Source Sink* σ

Introduction

Types for OOPL

Formalizing Subtyping

Subtyping Function, Lists, Arrays and References

Algorithmic Subtyping

Typing Rules as Algorithmic Specification

- All typing rules except for subtyping are syntax directed.
- It is simple to implement a type checker for syntax directed rules

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

$$\frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd \lambda x : \sigma . M : \sigma \to \tau}\ (\text{T-Abs}) \quad \frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M\, N : \tau}\ (\text{T-App})$$

$$\frac{\Gamma \rhd M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \rhd \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}}\ (\text{T-Rcd})$$

$$\frac{\Gamma \rhd M : \{l_i : \sigma_i{}^{i \in 1..n}\} \quad j \in 1..n}{\Gamma \rhd M.l_j : \sigma_j}\ (\text{T-Proj})$$

Subsumption

- Subsumption is not syntax directed.
- Not obvious how to implement type-checking when this rule is present

$$\frac{x : \sigma \in \Gamma}{\Gamma \rhd x : \sigma}\ \text{(T-Var)} \qquad \frac{\Gamma \rhd M : \sigma \quad \sigma <: \tau}{\Gamma \rhd M : \tau}\ \text{(T-Subs)}$$

$$\frac{\Gamma, x : \sigma \rhd M : \tau}{\Gamma \rhd \lambda x : \sigma . M : \sigma \to \tau}\ \text{(T-Abs)} \quad \frac{\Gamma \rhd M : \sigma \to \tau \quad \Gamma \rhd N : \sigma}{\Gamma \rhd M\,N : \tau}\ \text{(T-App)}$$

$$\frac{\Gamma \rhd M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \rhd \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}}\ \text{(T-Rcd)}$$

$$\frac{\Gamma \rhd M : \{l_i : \sigma_i\ {}^{i \in 1..n}\} \quad j \in 1..n}{\Gamma \rhd M.l_j : \sigma_j}\ \text{(T-Proj)}$$

"Hard-wiring" Subsumption

- A quick look at the rules determines that the only place that one needs subtyping is the argument of a function type
- Thus we propose the following variant: $\lambda^{\rightarrow}_{<:,alg}$

$$\frac{x : \sigma \in \Gamma}{\Gamma \mapsto x : \sigma}\ \text{(T-Var)} \qquad \frac{\Gamma, x : \sigma \mapsto M : \tau}{\Gamma \mapsto \lambda x : \sigma.M : \sigma \to \tau}\ \text{(T-Abs)}$$

$$\frac{\Gamma \mapsto M : \sigma \to \tau \quad \Gamma \mapsto N : \rho \quad \rho <: \sigma}{\Gamma \mapsto M\,N : \tau}\ \text{(T-App)}$$

$$\frac{\Gamma \mapsto M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \mapsto \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}}\ \text{(T-Rcd)}$$

$$\frac{\Gamma \mapsto M : \{l_i : \sigma_i\ ^{i \in 1..n}\} \quad j \in 1..n}{\Gamma \mapsto M.l_j : \sigma_j}\ \text{(T-Proj)}$$

Syntax-Directed Variant

- Before addressing type-checking a question
- What is the relation between $\lambda^{\rightarrow}_{<:,alg}$ and $\lambda^{\rightarrow}_{<:}$?

Proposition:

1. $\Gamma \mapsto M : \sigma$ implies $\Gamma \rhd M : \sigma$
2. $\Gamma \rhd M : \sigma$ implies there exists τ such that $\Gamma \mapsto M : \tau$ with $\tau <: \sigma$

Towards Implementing Type-Checking

- It remains to be seen how to implement checking for $\sigma <: \tau$

$$\frac{x : \sigma \in \Gamma}{\Gamma \mapsto x : \sigma}\ \text{(T-Var)} \qquad \frac{\Gamma, x : \sigma \mapsto M : \tau}{\Gamma \mapsto \lambda x : \sigma . M : \sigma \to \tau}\ \text{(T-Abs)}$$

$$\frac{\Gamma \mapsto M : \sigma \to \tau \quad \Gamma \mapsto N : \rho \quad \rho <: \sigma}{\Gamma \mapsto M\, N : \tau}\ \text{(T-App)}$$

$$\frac{\Gamma \mapsto M_i : \sigma_i \quad \forall i \in I = \{1..n\}}{\Gamma \mapsto \{l_i = M_i\}_{i \in I} : \{l_i : \sigma_i\}_{i \in I}}\ \text{(T-Rcd)}$$

$$\frac{\Gamma \mapsto M : \{l_i : \sigma_i\ ^{i \in 1..n}\} \quad j \in 1..n}{\Gamma \mapsto M.l_j : \sigma_j}\ \text{(T-Proj)}$$

Subtyping Rules – Review

$$\frac{}{\sigma <: \sigma}\ \text{(S-Refl)} \qquad \frac{}{\sigma <: \mathit{Top}}\ \text{(S-Top)}$$

$$\frac{}{\mathit{Nat} <: \mathit{Float}}\ \text{(S-NatFloat)} \frac{}{\mathit{Int} <: \mathit{Float}}\ \text{(S-IntFloat)} \frac{}{\mathit{Bool} <: \mathit{Nat}}\ \text{(S-BoolNat)}$$

$$\frac{\sigma <: \tau \quad \tau <: \rho}{\sigma <: \rho}\ \text{(S-Trans)} \qquad \frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'}\ \text{(S-Func)}$$

$$\frac{\{l_i \mid i \in 1..n\} \subseteq \{k_j \mid j \in 1..m\} \qquad k_j = l_i \Rightarrow \sigma_j <: \tau_i}{\{k_j : \sigma_j \mid j \in 1..m\} <: \{l_i : \tau_i \mid i \in 1..n\}}\ \text{(S-Rcd)}$$

- Not syntax-directed...
- The problem: (S-Refl) and (S-Trans)

Dropping (S-Refl) and (S-Trans)

- Note that one can prove $\sigma <: \sigma$
- We do of course have to include reflexivity for base types:
  - $Nat <: Nat$
  - $Bool <: Bool$
  - $Float <: Float$

Dropping (S-Trans)

- One may prove transitivity
- We must assume though, that we have transitivity of base types:
  - We have:
    - $Nat <: Float$
    - $Int <: Float$
    - $Bool <: Nat$
  - We add:
    - $Bool <: Float$

The Algorithm for Subtype-Checking (ignoring the axioms for Nat, Bool, Float)

```
let rec subtype (S,T) =
  match S,T with
   | _,Top -> true
   | (S1→ S2),(T1→ T2) ->
       subtype (T1,S1) && subtype (S2,T2)
   | {kj:Sj, j∈1..m},{li:Ti, i∈1..n} ->
       ({li, i∈1..n} ⊆ {kj, j∈1..m}) &&
            (∀i.∃j.kj = li) && subtype (Sj,Ti)
   | _ -> false
```

Reading

- A Theory of Objects, Martín Abadi, Luca Cardelli, Monographs in Computer Science, Springer-Verlag, 1996.
- Foundations of Object Oriented Languages, Kim Bruce, MIT Press, 2002.
- Some Challenging Typing Issues in Object-Oriented Languages, Kim Bruce. Electronic Notes in Theoretical Computer Science 82, no. 8 (2003). (see author's webpage).
- On binary methods, Kim Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Objects Group, Gary T. Leavens, and Benjamin Pierce. Theory and Practice of Object Systems, 1(1995).
- Types and Programming Languages, Benjamin C. Pierce, The MIT Press, 2002.