

# Formal Systems and their Applications

Course nr: H04H8A

Frank Piessens

[Frank.Piessens@cs.kuleuven.be](mailto:Frank.Piessens@cs.kuleuven.be)

Acknowledgment: these slides are based in part on slides from Benjamin Pierce

# Course Overview

- Introduction
- Formal models of programming languages
- Simple type systems
- Subtyping
  - Subtyping for variants of the lambda calculus
  - Algorithmic subtyping
  - Modeling OO languages
- Conclusion

# Subtyping

# Motivation

- Consider the term:

```
(λr:{x:Nat}. r.x) {x=0,y=1}
```

- This term is not well-typed in the systems we studied up to now.
- But clearly, it behaves well
- The subtype relation will try to capture the principle of safe substitution:
  - Sometimes terms of type S can always safely be substituted for terms of type T

# Formalization

- To formalize this, we introduce the subtyping relation:  $S <: T$ 
  - This relation will be defined inductively and we will need to think about an appropriate definition for each form of type
- And we use the subtyping relation in the following additional typing rule:
  - The “Rule of subsumption”:

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \quad (\text{T-SUB})$$

# The subtype relation

- General rules:

$$S <: S \quad (\text{S-REFL})$$

$$\frac{S <: U \quad U <: T}{S <: T} \quad (\text{S-TRANS})$$

# Rules for records

- Width subtyping: adding fields creates subtypes

$$\{l_i : T_i \mid i \in 1..n+k\} <: \{l_i : T_i \mid i \in 1..n\} \quad (\text{S-RCDWIDTH})$$

- Permutation: permuting fields does not matter

$$\frac{\{k_j : S_j \mid j \in 1..n\} \text{ is a permutation of } \{l_i : T_i \mid i \in 1..n\}}{\{k_j : S_j \mid j \in 1..n\} <: \{l_i : T_i \mid i \in 1..n\}} \quad (\text{S-RCDPERM})$$

- Depth subtyping: subtyping fields creates subtypes

$$\frac{\text{for each } i \quad S_i <: T_i}{\{l_i : S_i \mid i \in 1..n\} <: \{l_i : T_i \mid i \in 1..n\}} \quad (\text{S-RCDDEPTH})$$

# Rules for functions and Top

- Arrow types:
  - Contravariant in argument, covariant in result

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \quad (\text{S-ARROW})$$

- Top type:

$$S <: \text{Top} \quad (\text{S-TOP})$$

# Preservation and Progress

- Preservation and Progress theorems continue to hold
  - But proofs are slightly more elaborate
  - We will go over the Preservation proof in some detail

# Preservation

## (with proof for the T-APP case)

- Inversion of the subtype relation:

*Lemma: If  $U <: T_1 \rightarrow T_2$ , then  $U$  has the form  $U_1 \rightarrow U_2$ , with  $T_1 <: U_1$  and  $U_2 <: T_2$ . (Proof: by induction on subtyping derivations.)*

- Inversion of the typing relation:

*Lemma: If  $\Gamma \vdash \lambda x:S_1. s_2 : T_1 \rightarrow T_2$ , then  $T_1 <: S_1$  and  $\Gamma, x:S_1 \vdash s_2 : T_2$ .*

- Preservation:

*Theorem: If  $\Gamma \vdash t : T$  and  $t \longrightarrow t'$ , then  $\Gamma \vdash t' : T$ .*

# Ascription and Casting

- With subtypes, ascription can describe up-casts:

$$\frac{\Gamma \vdash t_1 : T}{\Gamma \vdash t_1 \text{ as } T : T} \quad (\text{T-ASCRIIBE})$$

$$v_1 \text{ as } T \longrightarrow v_1 \quad (\text{E-ASCRIIBE})$$

- But note that “down-casts” are something else!

$$\frac{\Gamma \vdash t_1 : S}{\Gamma \vdash t_1 \text{ as } T : T} \quad (\text{T-CAST})$$

$$\frac{\vdash v_1 : T}{v_1 \text{ as } T \longrightarrow v_1} \quad (\text{E-CAST})$$

# Subtyping makes variants nicer! (no ascription needed anymore)

$$\langle l_j : T_j \rangle_{i \in 1..n} <: \langle l_j : T_j \rangle_{i \in 1..n+k} \quad (\text{S-VARIANTWIDTH})$$

$$\frac{\text{for each } i \quad S_i <: T_i}{\langle l_j : S_j \rangle_{i \in 1..n} <: \langle l_j : T_j \rangle_{i \in 1..n}} \quad (\text{S-VARIANTDEPTH})$$

$$\frac{\langle k_j : S_j \rangle_{j \in 1..n} \text{ is a permutation of } \langle l_j : T_j \rangle_{i \in 1..n}}{\langle k_j : S_j \rangle_{j \in 1..n} <: \langle l_j : T_j \rangle_{i \in 1..n}} \quad (\text{S-VARIANTPERM})$$

$$\frac{\Gamma \vdash t_1 : T_1}{\Gamma \vdash \langle l_1 = t_1 \rangle : \langle l_1 : T_1 \rangle} \quad (\text{T-VARIANT})$$

# Subtyping and mutable data structures

- Typing rule for references:

$$\frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Ref } S_1 <: \text{Ref } T_1} \quad (\text{S-REF})$$

- Should be similar for arrays:

$$\frac{S_1 <: T_1 \quad T_1 <: S_1}{\text{Array } S_1 <: \text{Array } T_1} \quad (\text{S-ARRAY})$$

- So why does Java have a more relaxed rule?

# Algorithmic Subtyping

# Motivation

- The typing rules with subtyping do not lead directly to a type checking algorithm:
  - For a given context  $\Gamma$  and term  $t$ , several rules might be applicable to derive the type of  $t$  under  $\Gamma$
  - If the last rule was T-SUB, this actually does not bring us any closer to an answer
- Similar observations hold for the subtyping relation
  - In particular S-TRANS is a bad rule: it requires us to “guess” an intermediate type

# Goals of this section

- Give alternative definitions of the subtype and typing relations:
  - That are algorithmically well-behaved (syntax-directed)
  - That are semantically “the same” as the original relations
    - We will spell out precisely what “the same” means

# The subtype relation

$$S <: S \quad (\text{S-REFL})$$

$$\frac{S <: U \quad U <: T}{S <: T} \quad (\text{S-TRANS})$$

$$\{l_i : T_i \mid i \in 1..n+k\} <: \{l_i : T_i \mid i \in 1..n\} \quad (\text{S-RCDWIDTH})$$

$$\frac{\text{for each } i \quad S_i <: T_i}{\{l_i : S_i \mid i \in 1..n\} <: \{l_i : T_i \mid i \in 1..n\}} \quad (\text{S-RCDDEPTH})$$

$$\frac{\{k_j : S_j \mid j \in 1..n\} \text{ is a permutation of } \{l_i : T_i \mid i \in 1..n\}}{\{k_j : S_j \mid j \in 1..n\} <: \{l_i : T_i \mid i \in 1..n\}} \quad (\text{S-RCDPERM})$$

$$\frac{T_1 <: S_1 \quad S_2 <: T_2}{S_1 \rightarrow S_2 <: T_1 \rightarrow T_2} \quad (\text{S-ARROW})$$

$$S <: \text{Top} \quad (\text{S-TOP})$$

# Issues with the subtype relation

- The 3 RCD rules overlap and need S-TRANS
  - Solution: one (more complex) rule instead of three:

$$\frac{\{l_i^{i \in 1..n}\} \subseteq \{k_j^{j \in 1..m}\} \quad k_j = l_i \text{ implies } S_j <: T_i}{\{k_j : S_j^{j \in 1..m}\} <: \{l_i : T_i^{i \in 1..n}\}} \quad (\text{S-RCD})$$

- The rules S-REFL and S-TRANS overlaps with every other rule. Solution: remove them (!)

**Lemma:**  $S <: S$  can be derived for every type  $S$  without using S-REFL.

**Lemma:** If  $S <: T$  can be derived, then it can be derived without using S-TRANS.

# Algorithmic subtype relation

$\vdash S <: \text{Top}$  (SA-TOP)

$$\frac{\vdash T_1 <: S_1 \quad \vdash S_2 <: T_2}{\vdash S_1 \rightarrow S_2 <: T_1 \rightarrow T_2}$$
 (SA-ARROW)

$$\frac{\{l_i^{i \in 1..n}\} \subseteq \{k_j^{j \in 1..m}\} \quad \text{for each } k_j = l_i, \vdash S_j <: T_i}{\vdash \{k_j : S_j^{j \in 1..m}\} <: \{l_i : T_i^{i \in 1..n}\}}$$
 (SA-RCD)

**Theorem:**  $S <: T$  iff  $\vdash S <: T$ .

# Implementing the new relation

$subtype(S, T) =$

if  $T = \text{Top}$ , then *true*

else if  $S = S_1 \rightarrow S_2$  and  $T = T_1 \rightarrow T_2$

then  $subtype(T_1, S_1) \wedge subtype(S_2, T_2)$

else if  $S = \{k_j : S_j^{j \in 1..m}\}$  and  $T = \{l_i : T_i^{i \in 1..n}\}$

then  $\{l_i^{i \in 1..n}\} \subseteq \{k_j^{j \in 1..m}\}$

$\wedge$  for all  $i \in 1..n$  there is some  $j \in 1..m$  with  $k_j = l_i$

and  $subtype(S_j, T_i)$

else *false*.

# The typing relation

- For typing, the only new rule to consider is:

$$\frac{\Gamma \vdash t : S \quad S <: T}{\Gamma \vdash t : T} \quad (\text{T-SUB})$$

- This rule is “bad” because it overlaps with every other rule
- Let’s investigate where we **really** need the rule
  - So that we can limit its applicability

# The rule of subsumption

- Conjecture: We only need the rule of subsumption:

- For typing applications:

$$(\lambda r:\{x:\text{Nat}\}. r.x) \{x=0, y=1\}$$

- To weaken the type of a term at the end of a derivation

- We will try to convince ourselves by showing that for all other rules, applications of subsumption can be “pushed under them”

# Example: The T-ABS rule

$$\begin{array}{c}
 \vdots \\
 \hline
 \Gamma, x:S_1 \vdash s_2 : S_2 \\
 \hline
 \Gamma, x:S_1 \vdash s_2 : T_2 \quad (T\text{-SUB}) \\
 \hline
 \Gamma, x:S_1 \vdash s_2 : T_2 \\
 \hline
 \Gamma \vdash \lambda x:S_1 . s_2 : S_1 \rightarrow T_2 \quad (T\text{-ABS})
 \end{array}$$

becomes

$$\begin{array}{c}
 \vdots \\
 \hline
 \Gamma, x:S_1 \vdash s_2 : S_2 \\
 \hline
 \Gamma \vdash \lambda x:S_1 . s_2 : S_1 \rightarrow S_2 \quad (T\text{-ABS}) \\
 \hline
 \Gamma \vdash \lambda x:S_1 . s_2 : S_1 \rightarrow S_2 \\
 \hline
 \Gamma \vdash \lambda x:S_1 . s_2 : S_1 \rightarrow T_2 \quad (T\text{-SUB})
 \end{array}
 \quad
 \begin{array}{c}
 \vdots \\
 \hline
 S_1 <: S_1 \quad (S\text{-REFL}) \\
 \hline
 S_1 \rightarrow S_2 <: S_1 \rightarrow T_2 \\
 \hline
 S_2 <: T_2 \\
 \hline
 S_1 \rightarrow S_2 <: S_1 \rightarrow T_2 \quad (S\text{-ARROW}) \\
 \hline
 \Gamma \vdash \lambda x:S_1 . s_2 : S_1 \rightarrow T_2
 \end{array}$$

# Example: The T-SUB rule

$$\frac{\frac{\vdots}{\Gamma \vdash s : S} \quad \frac{\vdots}{S <: U}}{\Gamma \vdash s : U} \text{ (T-SUB)} \quad \frac{\vdots}{U <: T} \text{ (T-SUB)}$$

$$\Gamma \vdash s : T$$

becomes

$$\frac{\frac{\vdots}{\Gamma \vdash s : S} \quad \frac{\frac{\vdots}{S <: U} \quad \frac{\vdots}{U <: T}}{S <: T} \text{ (S-TRANS)}}{\Gamma \vdash s : T} \text{ (T-SUB)}$$

# But: T-APP on the left ...

$$\begin{array}{c}
 \vdots \\
 \hline
 \Gamma \vdash s_1 : S_{11} \rightarrow S_{12}
 \end{array}
 \quad
 \begin{array}{c}
 \vdots \\
 \hline
 T_{11} <: S_{11}
 \end{array}
 \quad
 \begin{array}{c}
 \vdots \\
 \hline
 S_{12} <: T_{12}
 \end{array}
 \quad
 \begin{array}{c}
 \vdots \\
 \hline
 \Gamma \vdash s_2 : T_{11}
 \end{array}$$

$$\begin{array}{c}
 \hline
 \Gamma \vdash s_1 : T_{11} \rightarrow T_{12}
 \end{array}
 \quad
 \begin{array}{c}
 \hline
 \Gamma \vdash s_2 : T_{11}
 \end{array}$$

$$\hline
 \Gamma \vdash s_1 s_2 : T_{12}$$

(T-APP)

becomes

$$\begin{array}{c}
 \vdots \\
 \hline
 \Gamma \vdash s_1 : S_{11} \rightarrow S_{12}
 \end{array}
 \quad
 \begin{array}{c}
 \vdots \\
 \hline
 \Gamma \vdash s_2 : T_{11}
 \end{array}
 \quad
 \begin{array}{c}
 \vdots \\
 \hline
 T_{11} <: S_{11}
 \end{array}
 \quad
 \begin{array}{c}
 \vdots \\
 \hline
 \Gamma \vdash s_2 : T_{11}
 \end{array}$$

$$\begin{array}{c}
 \hline
 \Gamma \vdash s_1 s_2 : S_{12}
 \end{array}
 \quad
 \begin{array}{c}
 \hline
 \Gamma \vdash s_2 : T_{11}
 \end{array}
 \quad
 \begin{array}{c}
 \hline
 S_{12} <: T_{12}
 \end{array}$$

$$\hline
 \Gamma \vdash s_1 s_2 : T_{12}$$

(T-SUB)

# But: T-APP on the right...

$$\frac{\frac{\Gamma \vdash s_1 : T_{11} \rightarrow T_{12}}{\Gamma \vdash s_1 : T_{11} \rightarrow T_{12}} \quad \frac{\frac{\Gamma \vdash s_2 : T_2 \quad T_2 <: T_{11}}{\Gamma \vdash s_2 : T_{11}}}{\Gamma \vdash s_1 s_2 : T_{12}}}{\Gamma \vdash s_1 s_2 : T_{12}} \text{ (T-APP)}$$

becomes

$$\frac{\frac{\frac{\Gamma \vdash s_1 : T_{11} \rightarrow T_{12}}{\Gamma \vdash s_1 : T_{11} \rightarrow T_{12}} \quad \frac{\frac{T_2 <: T_{11} \quad T_{12} <: T_{12}}{T_{11} \rightarrow T_{12} <: T_2 \rightarrow T_{12}}}{\Gamma \vdash s_1 : T_2 \rightarrow T_{12}}}{\Gamma \vdash s_1 : T_2 \rightarrow T_{12}} \text{ (T-SUB)} \quad \frac{\Gamma \vdash s_2 : T_2}{\Gamma \vdash s_2 : T_2}}{\Gamma \vdash s_1 s_2 : T_{12}} \text{ (T-APP)}$$

# Summary

- T-SUB is really needed:
  - Once, in combination with T-APP
  - Once at the root of the derivation tree
- This leads to the following solutions:
  - We make a more complex T-APP, incorporating T-SUB
  - We don't care for subsumptions at the root: the new relation will compute a minimal (most precise) type

# The algorithmic typing rules

$$\frac{x:T \in \Gamma}{\Gamma \vdash x : T} \quad (\text{TA-VAR})$$

$$\frac{\Gamma, x:T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x:T_1. t_2 : T_1 \rightarrow T_2} \quad (\text{TA-ABS})$$

$$\frac{\Gamma \vdash t_1 : T_1 \quad T_1 = T_{11} \rightarrow T_{12} \quad \Gamma \vdash t_2 : T_2 \quad \vdash T_2 <: T_{11}}{\Gamma \vdash t_1 t_2 : T_{12}} \quad (\text{TA-APP})$$

$$\frac{\text{for each } i \quad \Gamma \vdash t_i : T_i}{\Gamma \vdash \{l_1=t_1 \dots l_n=t_n\} : \{l_1:T_1 \dots l_n:T_n\}} \quad (\text{TA-RCD})$$

$$\frac{\Gamma \vdash t_1 : R_1 \quad R_1 = \{l_1:T_1 \dots l_n:T_n\}}{\Gamma \vdash t_1.l_i : T_i} \quad (\text{TA-PROJ})$$

# Correctness

- Soundness of algorithmic typing:

**Theorem:** If  $\Gamma \triangleright t : T$ , then  $\Gamma \vdash t : T$ .

- Completeness of algorithmic typing:

**Theorem [Minimal Typing]:** If  $\Gamma \vdash t : T$ , then  $\Gamma \triangleright t : S$  for some  $S <: T$ .

# Minimal typing and conditionals

- What is the minimal type of:

```
if true then {x=true,y=false} else {x=true,z=true}
```

- In general, the minimal type of

```
if t1 then t2 else t3
```

is the least common supertype (the *join*) of the minimal types of t<sub>1</sub> and t<sub>2</sub>

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T_2 \quad \Gamma \vdash t_3 : T_3}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T_2 \vee T_3} \quad (\text{T-IF})$$

# Do joins necessarily exist?

- They do in this system:

**Theorem:** For every pair of types  $S$  and  $T$ , there is a type  $J$  such that

1.  $S <: J$
2.  $T <: J$
3. If  $K$  is a type such that  $S <: K$  and  $T <: K$ , then  $J <: K$ .

I.e.,  $J$  is the smallest type that is a supertype of both  $S$  and  $T$ .

- A dual notion is the meet of two types, the largest type that is a subtype of both types
  - Computing joins needs computation of meets

# Do meets necessarily exist?

- No, for instance  $\{\}$  and  $\text{Top} \rightarrow \text{Top}$  have no meet
- but a slightly weaker property holds:

**Theorem:** For every pair of types  $S$  and  $T$ , if there is any type  $N$  such that  $N <: S$  and  $N <: T$ , then there is a type  $M$  such that

1.  $M <: S$

2.  $M <: T$

3. If  $O$  is a type such that  $O <: S$  and  $O <: T$ , then  $O <: M$ .

I.e.,  $M$  (when it exists) is the largest type that is a subtype of both  $S$  and  $T$ .

# Wrap up

- We introduced subtyping with as primary motivation the desire to type more programs
  - In particular “polymorphic” functions
- Subtyping seriously complicates type checking
  - There are alternative approaches to polymorphism that behave better
- But subtyping is an essential feature of Object Oriented programming languages

# Course Overview

- Introduction
- Formal models of programming languages
- Simple type systems
- Subtyping
  - Subtyping for variants of the lambda calculus
  - Algorithmic subtyping
  - Modeling OO languages
    - Encoding OO in the lambda calculus
    - Direct modeling: Featherweight Java
- Conclusion

# Encoding Object Orientation in the Lambda Calculus

# Key Characteristics of OO languages

- Dynamic dispatch / multiple representations
- Encapsulation
- Subtyping
- Inheritance
- Open recursion -- late binding of self/this

# Dynamic Dispatch

```
class A {  
    int x = 0;  
    int m() { x = x+1; return x; }  
    int n() { x = x-1; return x; }  
}  
  
class B extends A {  
    int m() { x = x+5; return x; }  
}  
  
class C extends A {  
    int m() { x = x-10; return x; }  
}
```

# Encapsulation

```
class A {  
    protected int x = 0;  
    int m() { x = x+1; return x; }  
    int n() { x = x-1; return x; }  
}  
  
class B extends A {  
    int m() { x = x+5; return x; }  
}  
  
class C extends A {  
    int m() { x = x-10; return x; }  
}
```

# Subtyping

- Objects are subtyped as “records of methods”

```
// ... class A and subclasses B and C as above...

class D {
    int p (A myA) { return myA.m(); }
}

...

D d = new D();
int z = d.p (new B());
int w = d.p (new C());
```

# Inheritance

- A code reuse mechanism, typically class-based
  - Classes can be *instantiated* to give objects
  - Classes can be *extended* to give new classes

```
class A {
    protected int x = 0;
    int m() { x = x+1; return x; }
    int n() { x = x-1; return x; }
}

class B extends A {
    int o() { x = x*10; return x; }
}
```

# Open recursion / late bound self

- Method calls on pseudo variable this/self are also dynamically dispatched

```
class E {  
    protected int x = 0;  
    int m() { x = x+1; return x; }  
    int n() { x = x-1; return this.m(); }  
}  
  
class F extends E {  
    int m() { x = x+100; return x; }  
}
```

# Open recursion / super calls

- Access to overridden methods is provided via super calls:

```
class E {
    protected int x = 0;
    int m() { x = x+1; return x; }
    int n() { x = x-1; return this.m(); }
}

class G extends E {
    int m() { x = x+100; return super.m(); }
}
```

# Encoding in the lambda calculus

- We develop a few successive refinements of an encoding of objects in the lambda calculus
- To see that:
  - This is feasible to some extent
  - This gets ugly / hard for advanced OO features

# Encapsulated State

```
class Counter {  
    protected int x = 1;           // Hidden state  
    int get() { return x; }  
    void inc() { x++; }  
}
```

```
void inc3(Counter c) {  
    c.inc(); c.inc(); c.inc();  
}
```

```
Counter c = new Counter();  
inc3(c);  
inc3(c);  
c.get();
```

# Encoding

```
c = let x = ref 1 in
      {get = λ_:Unit. !x,
       inc = λ_:Unit. x:=succ(!x)};
```

$\implies c : \text{Counter}$

where

$\text{Counter} = \{\text{get}:\text{Unit} \rightarrow \text{Nat}, \text{inc}:\text{Unit} \rightarrow \text{Unit}\}$

```
inc3 = λc:Counter. (c.inc unit; c.inc unit; c.inc unit);
```

$\implies \text{inc3} : \text{Counter} \rightarrow \text{Unit}$

```
(inc3 c; inc3 c; c.get unit);
```

$\implies 7$

# Object creation

```
newCounter =  
  λ_:Unit. let x = ref 1 in  
            {get = λ_:Unit. !x,  
              inc = λ_:Unit. x:=succ(!x)};  
⇒ newCounter : Unit → Counter
```

Or better: (grouping several instance variables in one record)

```
newCounter =  
  λ_:Unit. let r = {x=ref 1} in  
            {get = λ_:Unit. !(r.x),  
              inc = λ_:Unit. r.x:=succ(!(r.x))};
```

The local variable `r` has type `CounterRep = {x: Ref Nat}`

# Subtyping and inheritance

```
class Counter {  
    protected int x = 1;  
    int get() { return x; }  
    void inc() { x++; }  
}
```

```
class ResetCounter extends Counter {  
    void reset() { x = 1; }  
}
```

```
ResetCounter rc = new ResetCounter();  
inc3(rc);  
rc.reset();  
inc3(rc);  
rc.get();
```

# Encoding subtyping

- Based on record subtyping:

```
ResetCounter = {get:Unit→Nat,  
                inc:Unit→Unit,  
                reset:Unit→Unit};
```

```
newResetCounter =
```

```
  λ_:Unit. let r = {x = ref 1} in  
    {get    = λ_:Unit. !(r.x),  
     inc    = λ_:Unit. r.x:=succ(!(r.x)),  
     reset  = λ_:Unit. r.x:=1};
```

```
⇒ newResetCounter : Unit → ResetCounter
```

# Encoding inheritance

- To support inheritance, we need to decouple the methods from allocation of the fields
  - A class will define the methods

```
counterClass =  
  λr:CounterRep.  
    {get = λ_:Unit. !(r.x),  
     inc = λ_:Unit. r.x:=succ(!(r.x))};  
⇒ counterClass : CounterRep → Counter
```

- Object construction will allocate the fields

```
newCounter =  
  λ_:Unit. let r = {x=ref 1} in  
    counterClass r;  
⇒ newCounter : Unit → Counter
```

# Now we can support inheritance and super calls

```
resetCounterClass =  
  λr:CounterRep.  
    let super = counterClass r in  
      {get    = super.get,  
       inc    = super.inc,  
       reset = λ_:Unit. r.x:=1};
```

⇒ resetCounterClass : CounterRep → ResetCounter

```
newResetCounter =  
  λ_:Unit. let r = {x=ref 1} in resetCounterClass r;
```

⇒ newResetCounter : Unit → ResetCounter

# Overriding and adding instance vars

```
class Counter {  
    protected int x = 1;  
    int get() { return x; }  
    void inc() { x++; }  
}
```

```
class ResetCounter extends Counter {  
    void reset() { x = 1; }  
}
```

```
class BackupCounter extends ResetCounter {  
    protected int b = 1;  
    void backup() { b = x; }  
    void reset() { x = b; }  
}
```

# Overriding and adding instance vars

```
BackupCounter = {get:Unit→Nat, inc:Unit→Unit,  
                 reset:Unit→Unit, backup: Unit→Unit};  
BackupCounterRep = {x: Ref Nat, b: Ref Nat};  
  
backupCounterClass =  
  λr:BackupCounterRep.  
    let super = resetCounterClass r in  
      {get      = super.get,  
       inc      = super.inc,  
       reset    = λ_:Unit. r.x:=!(r.b),  
       backup   = λ_:Unit. r.b:=!(r.x)};
```

⇒

```
backupCounterClass : BackupCounterRep → BackupCounter
```

# Open recursion through self/this

- In Java, we can write:

```
class SetCounter {  
    protected int x = 0;  
    int get () { return x; }  
    void set (int i) { x = i; }  
    void inc () { this.set( this.get() + 1 ); }  
}
```

- And if a subclass overrides get() or set() but not inc(), the new methods will be called
- Modeling this is possible in the lambda calculus, but is tricky and ugly

# Wrap up

- We can explain many of the features of OO languages by encoding them in the lambda calculus
  - Multiple representations
  - Encapsulation
  - Subtyping
  - Inheritance
- But as we try to tackle advanced features, the encoding gets quite complicated

# Directly modeling Object Orientation: Featherweight Java

# A simple Java Model

- Instead of going through the lambda calculus, we may arrive at a simpler model if we directly model Java
- The model we present is a great simplification. It only models:
  - Core OO features
  - Type checking for them
- It does not model:
  - Assignment, concurrency, class loading, exceptions, interfaces, ...

# Important deltas with respect to our previous encoding

- Open recursion through self
- Classes are also types
- Support for casting
- Support for recursive types
- Nominal types instead of structural types
  - Programmer explicitly declares subtype relations, and the type checker only verifies

# Example Program

```
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); }  
}
```



# Conventions

- Always include superclass in class definition
- Always write out constructor (no defaults)
- Always explicitly call `super()` in constructor
- Always explicitly name receiver, also if it is “this”
- Method bodies consist of a single return expression
- Constructors:
  - Take same number (and type) of params as fields of the class
  - Assign these params to these local fields (using super calls for fields from the superclass)
  - Do nothing else

# Representing program state

- These conventions make it possible to model the program state of a Java program as just a source program
  - Objects are modeled as constructor invocations
    - `new C(v1, ..., vn)`
  - These will be the only values in FJ

# Example Evaluation

```
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);  
  }  
}
```

```
new Pair(new A(), new B()).setfst (new B());
```

Evaluates to

```
new Pair(new B(), new B())
```

# Syntax

$t ::=$

$x$

$t.f$

$t.m(\bar{t})$

$\text{new } C(\bar{t})$

$(C) t$

*terms*

*variable*

*field access*

*method invocation*

*object creation*

*cast*

$v ::=$

$\text{new } C(\bar{v})$

*values*

*object creation*

# Syntax (ctd)

$K ::=$  *constructor declarations*  
 $C(\bar{C} \bar{f}) \{ \text{super}(\bar{f}); \text{this}.\bar{f}=\bar{f}; \}$

$M ::=$  *method declarations*  
 $C m(\bar{C} \bar{x}) \{ \text{return } t; \}$

$CL ::=$  *class declarations*  
 $\text{class } C \text{ extends } C \{ \bar{C} \bar{f}; K \bar{M} \}$

# The subtype relation

$C <: C$

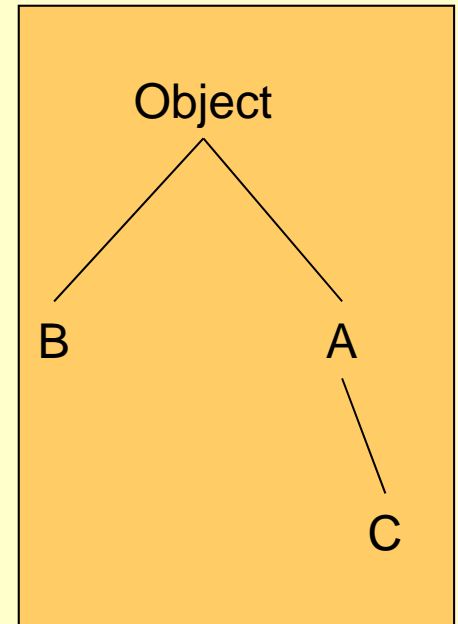
$$\frac{C <: D \quad D <: E}{C <: E}$$
$$\frac{\text{class } C \text{ extends } D \{ \dots \}}{C <: D}$$

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

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

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

```
A <: Object  
B <: Object  
C <: A
```



# Looking up the fields of a class

$$\text{fields}(\text{Object}) = \emptyset$$
$$CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \ \bar{f}; \ K \ \bar{M} \}$$
$$\text{fields}(D) = \bar{D} \ \bar{g}$$

---

$$\text{fields}(C) = \bar{D} \ \bar{g}, \bar{C} \ \bar{f}$$

```
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);  
    }  
}
```

$$\text{fields}(\text{Pair}) = (\text{Object fst}, \text{Object snd})$$

# Looking up method bodies

$$\frac{CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \ \bar{f}; \ K \ \bar{M} \}}{B \ m \ (\bar{B} \ \bar{x}) \ \{ \text{return } t; \} \in \bar{M}}$$
$$mbody(m, C) = (\bar{x}, t)$$
$$\frac{CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \ \bar{f}; \ K \ \bar{M} \}}{m \text{ is not defined in } \bar{M}}$$
$$mbody(m, C) = mbody(m, D)$$

```
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);
    }
}
```

```
mbody (setfst, Pair) =
(newfst, new Pair(newfst,this.snd))
```

# Operational semantics: computation

$$\frac{fields(C) = \bar{C} \ \bar{f}}{(new \ C(\bar{v})) . f_i \longrightarrow v_i} \quad (\text{E-PROJNEW})$$

$$\frac{mbody(m, C) = (\bar{x}, t_0)}{(new \ C(\bar{v})) . m(\bar{u}) \longrightarrow [\bar{x} \mapsto \bar{u}, \text{this} \mapsto new \ C(\bar{v})] t_0} \quad (\text{E-INVKNEW})$$

$$\frac{C <: D}{(D) (new \ C(\bar{v})) \longrightarrow new \ C(\bar{v})} \quad (\text{E-CASTNEW})$$

# Operational semantics: congruence

$$\frac{t_0 \longrightarrow t'_0}{t_0.f \longrightarrow t'_0.f} \quad (\text{E-FIELD})$$

$$\frac{t_0 \longrightarrow t'_0}{t_0.m(\bar{t}) \longrightarrow t'_0.m(\bar{t})} \quad (\text{E-INVK-RECV})$$

$$\frac{t_i \longrightarrow t'_i}{v_0.m(\bar{v}, t_i, \bar{t}) \longrightarrow v_0.m(\bar{v}, t'_i, \bar{t})} \quad (\text{E-INVK-ARG})$$

$$\frac{t_i \longrightarrow t'_i}{\text{new } C(\bar{v}, t_i, \bar{t}) \longrightarrow \text{new } C(\bar{v}, t'_i, \bar{t})} \quad (\text{E-NEW-ARG})$$

$$\frac{t_0 \longrightarrow t'_0}{(C)t_0 \longrightarrow (C)t'_0} \quad (\text{E-CAST})$$

# Operational semantics: example

*fields* (Pair) = (Object fst, Object snd)

**new Pair(new A(), new B()).snd → [E-ProjNew]  
new B()**

*mbody* (setfst, Pair) =  
(newfst, new Pair(newfst, this.snd))

**new Pair(new A(), new B()).setfst(new B()) → [E-InvkNew]  
new Pair(new B(), new Pair(new A(), new B()).snd) → [E-New-Arg] [E-ProjNew]  
new Pair(new B(), new B())**

# FJ Typing

# Looking up method types

$$CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \ \bar{f}; \ K \ \bar{M} \}$$
$$B \ m \ (\bar{B} \ \bar{x}) \ \{ \text{return } t; \} \in \bar{M}$$

---

$$mtype(m, C) = \bar{B} \rightarrow B$$
$$CT(C) = \text{class } C \text{ extends } D \{ \bar{C} \ \bar{f}; \ K \ \bar{M} \}$$
$$m \text{ is not defined in } \bar{M}$$

---

$$mtype(m, C) = mtype(m, D)$$

```
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);
  }
}
```

$$mtype(\text{setfst}, \text{Pair}) =$$
$$(\text{Object}) \rightarrow \text{Pair}$$

# Typing rules for terms (1)

$$\frac{x:C \in \Gamma}{\Gamma \vdash x : C} \quad (\text{T-VAR})$$

$$\frac{\Gamma \vdash t_0 : C_0 \quad \text{fields}(C_0) = \bar{C} \ \bar{f}}{\Gamma \vdash t_0.f_i : C_i} \quad (\text{T-FIELD})$$

$$\frac{\Gamma \vdash t_0 : C_0 \quad \text{mtype}(m, C_0) = \bar{D} \rightarrow C \quad \Gamma \vdash \bar{t} : \bar{C} \quad \bar{C} <: \bar{D}}{\Gamma \vdash t_0.m(\bar{t}) : C} \quad (\text{T-INVK})$$

$$\frac{\text{fields}(C) = \bar{D} \ \bar{f} \quad \Gamma \vdash \bar{t} : \bar{C} \quad \bar{C} <: \bar{D}}{\Gamma \vdash \text{new } C(\bar{t}) : C} \quad (\text{T-NEW})$$

*fields* (Pair) = (Object fst, Object snd)

**this:Pair |-- this.snd : Object**

**newfst:Object, this:Pair |-- new Pair(newfst,this.snd) : Pair**

# Typing rules for casts

$$\frac{\Gamma \vdash t_0 : D \quad D <: C}{\Gamma \vdash (C)t_0 : C} \quad (\text{T-UCAST})$$

$$\frac{\Gamma \vdash t_0 : D \quad C <: D \quad C \neq D}{\Gamma \vdash (C)t_0 : C} \quad (\text{T-DCAST})$$

# Typing rules for methods and classes

$$\frac{\bar{x} : \bar{C}, \text{this} : C \vdash t_0 : E_0 \quad E_0 <: C_0 \quad CT(C) = \text{class } C \text{ extends } D \{ \dots \} \quad \text{override}(m, D, \bar{C} \rightarrow C_0)}{C_0 \text{ m } (\bar{C} \ \bar{x}) \{ \text{return } t_0; \} \text{ OK in } C}$$

$$\frac{K = C(\bar{D} \ \bar{g}, \bar{C} \ \bar{f}) \{ \text{super}(\bar{g}); \text{this}.\bar{f} = \bar{f}; \} \quad \text{fields}(D) = \bar{D} \ \bar{g} \quad \bar{M} \text{ OK in } C}{\text{class } C \text{ extends } D \{ \bar{C} \ \bar{f}; K \ \bar{M} \} \text{ OK}}$$

$$\frac{mtype(m, D) = \bar{D} \rightarrow D_0 \text{ implies } \bar{C} = \bar{D} \text{ and } C_0 = D_0}{\text{override}(m, D, \bar{C} \rightarrow C_0)}$$

# Typing rules for methods: example

$$\frac{\bar{x} : \bar{C}, \text{this} : C \vdash t_0 : E_0 \quad E_0 <: C_0 \quad CT(C) = \text{class } C \text{ extends } D \{ \dots \} \quad \text{override}(m, D, \bar{C} \rightarrow C_0)}{C_0 \ m \ (\bar{C} \ \bar{x}) \ \{\text{return } t_0;\} \ \text{OK in } C}$$

```
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);
  }
}
```

```
newfst:Object, this:Pair |--
new Pair(newfst,this.snd) : Pair
```

# Properties

# Progress

- Well-typed programs get stuck!

```
(A)new Object()
```

- Intuition: a well-typed program either:
  - Is a value
  - Can make a step
  - Or is stuck at a failing cast

# Formalizing Progress

- Evaluation Contexts
  - Capture the notion of “next subterm to be reduced”

$E ::=$

$[]$

$E.f$

$E.m(\bar{t})$

$v.m(\bar{v}, E, \bar{t})$

$\text{new } C(\bar{v}, E, \bar{t})$

$(C)E$

*evaluation contexts*

*hole*

*field access*

*method invocation (receive)*

*method invocation (arg)*

*object creation (arg)*

*cast*

# Formalizing Progress

*Theorem* [Progress]: Suppose  $t$  is a closed, well-typed normal form. Then either (1)  $t$  is a value, or (2)  $t \longrightarrow t'$  for some  $t'$ , or (3) for some evaluation context  $E$ , we can express  $t$  as  $t = E[(C) (\text{new } D(\bar{v}))]$ , with  $D \not\prec C$ .

- Proof is a straightforward induction on typing derivations

# Preservation

- Preservation does not hold either!

$$(A) \underline{(\text{Object})\text{new } B()} \longrightarrow (A)\text{new } B()$$

- Addressed by adding a new rule:

$$\frac{\Gamma \vdash t_0 : D \quad C \not\prec: D \quad D \not\prec: C}{\Gamma \vdash (C)t_0 : C} \quad \text{(T-SCAST)}$$

- Then, the following theorem holds:

*Theorem* [Preservation]: If  $\Gamma \vdash t : C$  and  $t \longrightarrow t'$ , then  $\Gamma \vdash t' : C'$  for some  $C' \prec: C$ .

# Wrap up

- The techniques we have used to study the safety of extensions of the lambda-calculus also apply for models of Java
- Similar to what we have seen for the lambda-calculus, there are various extensions/variants of FJ that add language features
  - Middleweight Java, ClassicJava, GFJ, ...