Subtyping

Thomas Wies
wies@mpi-sb.mpg.de

Seminar: Types and Programming Languages, WS 02/03

Pierce, ch. 15-18

Overview

① The subtype relation

② Typechecking

③ Extensions: references, casts

④ Case study: featherweight Java

FRAMEWORK

Language used in this talk:

simply typed lambda-calculus + records:

Example:

$\mathsf{r} = (\lambda \mathsf{r} : \{\mathsf{x} : \mathsf{Nat}, \mathsf{y} : \mathsf{Nat}\}.\, \mathsf{r})\ \ \{\mathsf{x} = 1, \mathsf{y} = 2\};$

$\triangleright\, \mathsf{r} : \{\mathsf{x} : \mathsf{Nat}, \mathsf{y} : \mathsf{Nat}\}$

$\mathsf{r.x};$

$\triangleright\, 1 : \mathsf{Nat}$

in the context of imperative objects:

simply typed lambda-calculus + records + references

Problem:

Simply typed lambda-calculus is often to restrictive.

Example: $(\lambda r : \{x : \mathsf{Nat}\}.\ r.x)\ \ \{x = 0, y = 1\}$ is not well-typed.

Intuition:

- *Subset semantics:* whenever S is a subset of T, then any term of type S should also be of type T.
- More general: whenever it is safe to use a term of type S in a context of type T, then S is a subtype of T, written $S <: T$.

In the example: $\{x : \mathsf{Nat}, y : \mathsf{Nat}\} <: \{x : \mathsf{Nat}\}$

WHAT IS NEEDED?

- We have to extend our typing rules:

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

- We have to formalize what $\mathsf{S} <: \mathsf{T}$ means.

Notice: Evaluation is not effected by the introduction of subtyping.

THE SUBTYPE RELATION

Top:

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

Reflexivity:

$S <: S \quad$ (S-REFL)

Transitivity:

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

Arrow-Types:

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

THE SUBTYPE RELATION (2)

Record deepening:

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

Record widening:

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

Record permutation:

$$\frac{\{k_i : S_i{}^{i \in 1..n}\} \text{ is a permutation of } \{l_i : T_i{}^{i \in 1..n}\}}{\{k_i : S_i{}^{i \in 1..n}\} <: \{l_i : T_i{}^{i \in 1..n}\}} \quad \text{(S-RCDPERM)}$$

Example: A Subtype Derivation

Derivation of:

$$\vdash (\lambda \mathsf{r} : \{\mathsf{x} : \mathsf{Top}\}.\, \mathsf{r.x}) \ \{\mathsf{x} = 0, \mathsf{y} = 1\} : \mathsf{Top}$$

TYPE SAFETY

Type safety is preserved in presence of subtyping:

Theorem (Preservation): If $\Gamma \vdash \mathtt{t} : \mathsf{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathsf{T}$

Theorem (Progress): If $\mathtt{t}$ is a closed, well-typed term, then either $\mathtt{t}$ is a value or $\mathtt{t} \longrightarrow \mathtt{t}'$.

TYPECHECKING

- Algorithmic subtyping
- Algorithmic typing

A Typechecker with Subtyping

How to implement a subtypechecker checking $\mathsf{S} <: \mathsf{T}$ for two types S and T?

Problem:

$\mathsf{S} <: \mathsf{S}$ (S-Refl)

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

S and T match any types.

➜ Rules can be applied in any situation.

➜ The subtype relation considered so far can not be used to implement a subtypechecker directly.

Idea: Introduce an algorithmic subtype relation $\mapsto \mathsf{S} <: \mathsf{T}$, s.t. $\mapsto \mathsf{S} <: \mathsf{T}$ iff $\mathsf{S} <: \mathsf{T}$

Observations:

- Reflexivity is not needed for typechecking.
  - ➜ drop S-REF
- Transitivity is only needed for record-types.
  - ➜ merge record-rules in one single rule
  - ➜ drop S-TRANS

New rule for record-subtyping:

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

- S-TOP and S-ARROW do not change.

Algorithmic Typing

For typechecking we have a similar problem:

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{S} \qquad \mathsf{S} <: \mathsf{T}}{\Gamma \vdash \mathsf{t} : \mathsf{T}} \ (\text{T-Sub})$$

t matches any term, hence the rule T-Sub fires on any term.

➜ We need an algorithmic typing relation $\Gamma \mapsto \mathsf{t} : \mathsf{T}$

ALGORITHMIC TYPING (2)

Observation:

- T-SUB is only needed to match the argument- and domain-types in application terms.

  ➜ merge T-SUB into T-APP

New rule for applications:

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

Theorem: Algorithmic typing is sound and complete.

① if $\Gamma \mapsto t : T$, then $\Gamma \vdash t : T$.

② if $\Gamma \vdash t : T$, then $\Gamma \mapsto t : S$ for some $S <: T$.

- References
- Up- and down-casts

Subtyping and References

What conditions must hold in order to get $\mathsf{Ref\ S} <: \mathsf{Ref\ T}$?

Example:

$(\lambda \mathsf{r} : \mathsf{Ref}\ \{\mathsf{x} : \mathsf{Nat}, \mathsf{y} : \mathsf{Nat}\}.\ !\mathsf{r.x})\ \ (\mathsf{ref}\ \{\mathsf{y} = 0\})$ will go wrong.

➜ We need $\mathsf{S} <: \mathsf{T}$ in order to get safe dereferences.

$(\lambda \mathsf{r} : \mathsf{Ref}\ \{\mathsf{x} : \mathsf{Nat}, \mathsf{y} : \mathsf{Nat}\}.\ \mathsf{r} := \{\mathsf{x} = 1\};\ !\mathsf{r.y})\ \ (\mathsf{ref}\ \{\mathsf{x} = 0, \mathsf{y} = 1\})$
will go wrong, too.

➜ We also need $\mathsf{T} <: \mathsf{S}$ in order to get safe assignments.

Simple inference rule:

$$\frac{\mathsf{S} <: \mathsf{T} \quad \mathsf{T} <: \mathsf{S}}{\mathsf{Ref\ S} <: \mathsf{Ref\ T}} \ \text{(S-Ref)}$$

References refined

Decompose $\mathsf{Ref\ T}$ in two new types

- $\mathsf{Source\ T}$: capability to read from a reference cell
- $\mathsf{Sink\ T}$: capability to write into a reference cell

and modify the typing rules for references accordingly:

$$\frac{\Gamma \mid \Sigma \vdash \ \mathsf{t} : \mathsf{Source\ T}}{\Gamma \mid \Sigma \vdash \mathsf{!t} : \mathsf{T}} \quad \text{(T-Deref)}$$

$$\frac{\Gamma \mid \Sigma \vdash \ \mathsf{t}_1 : \mathsf{Sink\ T} \qquad \Gamma \mid \Sigma \vdash \ \mathsf{t}_2 : \mathsf{T}}{\Gamma \mid \Sigma \vdash \ \mathsf{t}_1 := \mathsf{t}_2 : \mathsf{Unit}} \quad \text{(T-Assign)}$$

REFERENCES REFINED (2)

Now, subtyping for references is easy:

$$\frac{\mathsf{S} <: \mathsf{T}}{\mathsf{Source}\ \mathsf{S} <: \mathsf{Source}\ \mathsf{T}} \quad \text{(S-SOURCE)}$$

$$\frac{\mathsf{T} <: \mathsf{S}}{\mathsf{Sink}\ \mathsf{S} <: \mathsf{Sink}\ \mathsf{T}} \quad \text{(S-SINK)}$$

Ref is just a subtype of both Source and Sink:

$$\mathsf{Ref}\ \mathsf{T} <: \mathsf{Source}\ \mathsf{T} \quad \text{(S-REFSOURCE)}$$

$$\mathsf{Ref}\ \mathsf{T} <: \mathsf{Sink}\ \mathsf{T} \quad \text{(S-REFSINK)}$$

Ascription as a Casting Operator

Idea: use ascription operator $\mathsf{t}\ \mathsf{as}\ \mathsf{T}$ to perform type casts.

$$\frac{\Gamma \vdash \mathsf{t} : \mathsf{T}}{\Gamma \vdash \mathsf{t}\ \mathsf{as}\ \mathsf{T} : \mathsf{T}} \quad \text{(T-Ascribe)}$$

Up-casts

Application: information-hiding.

Ascription + subsumption immediately gives us Up-casts.

Example: $\{x = 0, y = 1\}\ \mathsf{as}\ \{x : \mathsf{Nat}\}$ is well-typed and y is hidden in the context of the ascribed term.

Notice: up-casts do not require ascription, they can be performed using lambda-terms, too.

Ascription as a Casting Operator (2)

Down-casts

Application: down-casts + Top provide simple form of polymorphism.

Example: container classes in Java.

Down-casts require an additional typing-rule:

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

Problem: down-casts may be unsound.

Solution: Add dynamic type tests to the evaluation-rules for ascription.

COERCION SEMANTICS

Problem: subtyping may result in performance penalties on the low-level language implementation.

Example:

How to perform efficiently record-field accesses in the presence of a permutation rule?

Idea: *coercion semantics:* Use the type- and subtype-derivation trees to generate additional code for type conversions.

CASE STUDY: FEATHERWEIGHT JAVA

- Interfaces
- Inheritance
- Subtyping
- self and open recursion

IMPERATIVE OBJECTS

What are the essential features of imperative objects?

- Multiple representations
  - same interface, but different implementations
- Encapsulation and information hiding
  - internal state only accessible via interface
  - concrete representation hidden
- Inheritance
  - classes are used as templates for object instantiation
  - derived sub-classes can selectively share code with their super-classes

FEATURES OF IMPERATIVE OBJECTS (CONT'D.)

- Subtyping
  - objects of sub-classes can be used in any super-class context
- self and open recursion
  - methods are allowed to invoke other methods of the same object via self or this
  - in particular: super-classes may invoke methods declared in sub-classes (late-binding).

A Simple Java Example

Simple implementation of a counter in Java:

```
class Counter {
    private int x;

    public Counter() {super(); x=1;}

    public int get () { return x;}

    public void inc () { x++;}
}
```

Question: How can we mimic this within the simply typed lambda-calculus with subtyping?

INTERFACES

The interfaces can be described by using record types:

- a label with functional type for each public method
- a label for each public instance variable with appropriate type

In the example:

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

OBJECTS

A counter object can be implemented now by allocating the instance variable and constructing the method table:

$$
\begin{aligned}
&\mathsf{c} = \mathsf{let\ x} = \mathsf{ref\ 1\ in}\\
&\qquad\quad \{\mathsf{get} = \lambda\_ : \mathsf{Unit.}\,!\mathsf{x},\\
&\qquad\quad\ \ \mathsf{inc} = \lambda\_ : \mathsf{Unit.}\,\mathsf{x} := \mathsf{succ}(!\mathsf{x})\};\\
&\triangleright\ \mathsf{c} : \mathsf{Counter}
\end{aligned}
$$

$$
\begin{aligned}
&(\mathsf{c.inc\ unit};\mathsf{c.inc\ unit};\mathsf{c.get\ unit});\\
&\triangleright\ 3 : \mathsf{Nat}
\end{aligned}
$$

A Simple Class

Define a representation type for the instance variables:

$\mathsf{CounterRep} = \{\mathsf{x} : \mathsf{Ref\ Nat}\};$

The counter class now abstracts over the counter representation:

$$
\begin{aligned}
&\mathsf{counterClass} = \\
&\quad \lambda \mathsf{r} : \mathsf{CounterRep}. \\
&\qquad\quad \{\mathsf{get} = \lambda\_ : \mathsf{Unit}.\, !(\mathsf{r.x}), \\
&\qquad\quad\ \ \mathsf{inc} = \lambda\_ : \mathsf{Unit}.\, \mathsf{x} := \mathsf{succ}(!(\mathsf{r.x}))\};
\end{aligned}
$$

$\triangleright\ \mathsf{counterClass} : \mathsf{CounterRep} \to \mathsf{Counter}$

New objects can be instantiated via an object generator:

$$
\begin{aligned}
&\mathsf{newCounter} = \\
&\quad \lambda\_ : \mathsf{Unit}.\, \mathsf{let\ r} = \{\mathsf{x} = \mathsf{ref}\ 1\}\ \mathsf{in} \\
&\qquad\qquad\qquad \mathsf{counterClass\ r};
\end{aligned}
$$

$\triangleright\ \mathsf{newCounter} : \mathsf{Unit} \to \mathsf{Counter}$

INHERITANCE IN JAVA

Example of an inherited class in Java:

```
class ResetCounter extends Counter {
    public ResetCounter() {super();}

    public void reset () { x  = 1;}
}
```

Inheritance

First we need to extend the counter interface:

$$\mathsf{ResetCounter} = \{\mathsf{get} : \mathsf{Unit} \to \mathsf{Nat}, \mathsf{inc} : \mathsf{Unit} \to \mathsf{Unit},$$
$$\mathsf{reset} : \mathsf{Unit} \to \mathsf{Unit}\};$$

Then we can reuse counterClass within resetCounterClass:

$$\begin{aligned}
&\mathsf{resetCounterClass} = \\
&\quad \lambda \mathsf{r} : \mathsf{CounterRep}. \\
&\qquad \mathsf{let\ super} = \mathsf{counterClass\ r\ in} \\
&\qquad\ \ \{\mathsf{get} = \mathsf{super.get}, \\
&\qquad\quad \mathsf{inc} = \mathsf{super.inc}, \\
&\qquad\quad \mathsf{reset} = \lambda_ : \mathsf{Unit}.\, \mathsf{r.x} := 1\};
\end{aligned}$$

$\triangleright$ $\mathsf{resetCounterClass} : \mathsf{CounterRep} \to \mathsf{ResetCounter}$

Subtyping

Record-subtyping provides all we need for subtyping between objects:

$\mathsf{ResetCounter} <: \mathsf{Counter}$

Hence any reset-counter can be used safely as a counter:

$\mathsf{rc} = \mathsf{newResetCounter\ unit};$

$\triangleright\ \mathsf{rc} : \mathsf{ResetCounter}$

$\mathsf{inc3} = \lambda \mathsf{c} : \mathsf{Counter}.\ \mathsf{c.inc\ unit};\ \mathsf{c.inc\ unit};\ \mathsf{c.inc\ unit};$

$\triangleright\ \mathsf{inc3} : \mathsf{Counter} \to \mathsf{Unit}$

$(\mathsf{inc3\ rc};\ \mathsf{rc.reset\ unit};\ \mathsf{inc3\ rc};\ \mathsf{rc.get\ unit});$

$\triangleright\ 4 : \mathsf{Nat}$

Classes with self

Let us implement a new SetCounter class that provides a method to set the counter to a given amount:

$$\begin{aligned}\mathsf{SetCounter} = \{&\mathsf{get} : \mathsf{Unit} \to \mathsf{Nat}, \mathsf{set} : \mathsf{Nat} \to \mathsf{Unit},\\ &\mathsf{inc} : \mathsf{Unit} \to \mathsf{Unit}\};\end{aligned}$$

We use fixpoint recursion to introduce self:

$$\begin{array}{l}\mathsf{setCounterClass} =\\ \quad \lambda \mathsf{r} : \mathsf{CounterRep}.\\ \quad\quad \mathsf{fix}\\ \quad\quad\quad (\lambda \mathsf{self} : \mathsf{SetCounter}.\\ \quad\quad\quad\quad \{\mathsf{get} = \lambda\_ : \mathsf{Unit}.\, !(\mathsf{r.x}),\\ \quad\quad\quad\quad\;\; \mathsf{set} = \lambda \mathsf{i} : \mathsf{Nat}.\, \mathsf{r.x} := \mathsf{i},\\ \quad\quad\quad\quad\;\; \mathsf{inc} = \lambda\_ : \mathsf{Unit}.\, \mathsf{self.set}\ (\mathsf{succ}\ (\mathsf{self.get}\ \mathsf{unit}))\});\\ \triangleright\ \mathsf{setCounterClass} : \mathsf{CounterRep} \to \mathsf{SetCounter}\end{array}$$

Open Recursion in Java

Example of open recursion in Java:

(in Java all methods are late-bound)

```
class SetCounter extends Counter {
    public SetCounter() {super();}

    // set will be bound to a sub−class' method later
    public void reset () { this.set 1}

    public void set(int i ) { x = i ;}
}
```

OPEN RECURSION IN JAVA (2)

```
class BackupCounter extends SetCounter {

    private int b;

    // bind super−class declaration of set to this one
    public void set(int i ) { b = x ; super.set i}

    public void restore () { x = b;}

}
```

Open Recursion

There are several possibilities to implement open recursion:

- We can use fixpoint recursion again.
- We can use references.

We will use references here, which is the more efficient solution.

Open Recursion via References

New SetCounter interface:

$$\mathsf{SetCounter} = \{\mathsf{get} : \mathsf{Unit} \to \mathsf{Nat}, \mathsf{inc} : \mathsf{Unit} \to \mathsf{Unit},$$
$$\mathsf{set} : \mathsf{Nat} \to \mathsf{Unit}, \mathsf{reset} : \mathsf{Unit} \to \mathsf{Unit}\};$$

self is now a reference to a method table:

$$
\begin{array}{l}
\mathsf{setCounterClass} = \\
\quad \lambda \mathsf{r} : \mathsf{SetCounterRep}.\ \lambda \mathsf{self} : \mathsf{Ref\ SetCounter}. \\
\qquad\quad \mathsf{let\ super} = \mathsf{counterClass\ r\ in} \\
\qquad\quad \{\mathsf{get} = \mathsf{super.get}, \\
\qquad\quad\ \ \mathsf{inc} = \mathsf{super.inc}, \\
\qquad\quad\ \ \mathsf{set} = \lambda \mathsf{i} : \mathsf{Nat}.\ \mathsf{r.x} := \mathsf{i}, \\
\qquad\quad\ \ \mathsf{reset} = \lambda\_ : \mathsf{Unit}.\ (!\mathsf{self}).\mathsf{set}\ 1\}; \\
\triangleright\ \mathsf{setCounterClass} : \mathsf{CounterRep} \to \mathsf{Ref\ SetCounter} \to \mathsf{SetCounter}
\end{array}
$$

Object Generation for Open Recursion

For object generation a dummy object must be allocated first:

$$
\begin{aligned}
&\mathsf{newSetCounter} = \\
&\quad \lambda\_ : \mathsf{Unit}.\ \mathsf{let}\ \mathsf{r} = \{\mathsf{x} = \mathsf{ref}\ 1\}\ \mathsf{in} \\
&\qquad\qquad \mathsf{let}\ \mathsf{mTbl} = \mathsf{ref}\ \{\mathsf{get} = \lambda\_ : \mathsf{Unit}.\ 0, \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{inc} = \lambda\_ : \mathsf{Unit}.\ \mathsf{unit}, \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{set} = \lambda \mathsf{i} : \mathsf{Nat}.\ \mathsf{unit}, \\
&\qquad\qquad\qquad\qquad\qquad \mathsf{reset} = \lambda\_ : \mathsf{Unit}.\ \mathsf{unit}\}\ \mathsf{in} \\
&\qquad\qquad (\mathsf{mTbl} := \mathsf{setCounterClass}\ \mathsf{r}\ \mathsf{mTbl});\ \ !\mathsf{mTbl};
\end{aligned}
$$

$\triangleright$ $\mathsf{newSetCounter} : \mathsf{Unit} \to \mathsf{SetCounter}$

BackupCounter Types

The required BackupCounter interface, and representation:

$$\begin{aligned}\mathsf{BackupCounter} = \{&\mathsf{get} : \mathsf{Unit} \to \mathsf{Nat}, \mathsf{inc} : \mathsf{Unit} \to \mathsf{Unit},\\ &\mathsf{set} : \mathsf{Nat} \to \mathsf{Unit}, \mathsf{reset} : \mathsf{Unit} \to \mathsf{Unit},\\ &\mathsf{restore} : \mathsf{Unit} \to \mathsf{Unit}\};\end{aligned}$$

$$\mathsf{BackupCounterRep} = \{\mathsf{x} : \mathsf{Nat}, \mathsf{b} : \mathsf{Nat}\};$$

First Attempt - Fails!

Problem: S-REF does not allow the required subtyping:

$$
\begin{aligned}
&\mathsf{BackupCounterClass} = \\
&\quad \lambda \mathsf{self} : \mathsf{Ref\ BackupCounter}. \\
&\qquad \lambda \mathsf{r} : \mathsf{BackupCounterRep}.\ \mathsf{let\ super} = \mathsf{setCounterClass\ r\ self\ in} \\
&\qquad\quad \{\mathsf{get} = \mathsf{super.get}, \\
&\qquad\quad\ \ \mathsf{inc} = \mathsf{super.inc}, \\
&\qquad\quad\ \ \mathsf{set} = \lambda \mathsf{i} : \mathsf{Nat}.\ \mathsf{r.b} =!(\mathsf{r.x});\ \mathsf{super.set\ i}, \\
&\qquad\quad\ \ \mathsf{reset} = \mathsf{super.reset}, \\
&\qquad\quad\ \ \mathsf{restore} = \lambda\_ : \mathsf{Unit}.\ \mathsf{r.x} :=!(\mathsf{r.b})\}; \\
&\triangleright\ \mathsf{Error:}\ \ \mathsf{parameter\ type\ mismatch}
\end{aligned}
$$

Refined Version

Solution:

In setCounterClass only read-access to the method table is needed.

➜ Use Source SetCounter instead of Ref SetCounter:

$$
\begin{array}{l}
\mathsf{setCounterClass} = \\
\quad \lambda \mathsf{r} : \mathsf{SetCounterRep}. \\
\qquad \lambda \mathsf{self} : \mathsf{Source\ SetCounter}. \\
\qquad\quad \mathsf{let\ super} = \mathsf{counterClass\ r\ in} \\
\qquad\quad \{\mathsf{get} = \mathsf{super.get}, \\
\qquad\quad\ \ \mathsf{inc} = \mathsf{super.inc}, \\
\qquad\quad\ \ \mathsf{set} = \lambda \mathsf{i} : \mathsf{Nat}.\ \mathsf{r.x} := \mathsf{i}, \\
\qquad\quad\ \ \mathsf{reset} = \lambda\_ : \mathsf{Unit}.\ (!\mathsf{self}).\mathsf{set}\ 1\};
\end{array}
$$

$\triangleright$ $\mathsf{setCounterClass} : \mathsf{CounterRep} \to \mathsf{Source\ SetCounter} \to \mathsf{SetCounter}$

Refined Version (2)

Now the backup-counter class typechecks:

$$
\begin{array}{l}
\mathsf{BackupCounterClass} = \\
\quad \lambda\mathsf{self} : \mathsf{Source\ BackupCounter}. \\
\qquad \lambda\mathsf{r} : \mathsf{BackupCounterRep}.\ \mathsf{let\ super} = \mathsf{setCounterClass\ r\ self\ in} \\
\qquad\quad \{\mathsf{get} = \mathsf{super.get}, \\
\qquad\quad\ \ \mathsf{inc} = \mathsf{super.inc}, \\
\qquad\quad\ \ \mathsf{set} = \lambda\mathsf{i} : \mathsf{Nat}.\ \mathsf{r.b} =!(\mathsf{r.x});\ \mathsf{super.set\ i}, \\
\qquad\quad\ \ \mathsf{reset} = \mathsf{super.reset}, \\
\qquad\quad\ \ \mathsf{restore} = \lambda\_ : \mathsf{Unit}.\ \mathsf{r.x} :=!(\mathsf{r.b})\}; \\
\triangleright\ \mathsf{backupCounterClass} : \mathsf{BackupCounterRep} \to \\
\qquad\qquad\qquad \mathsf{Source\ BackupCounter} \to \mathsf{BackupCounter}
\end{array}
$$

Conclusion and Outlook

- Typing can be extended to respect subset-relations on types.
- Object-oriented language features can be expressed in the simply typed lambda-calculus.
- Subtyping introduces efficiency problems.
  Possible solution: *coercion semantics*

Aspects not considered in this talk:

- Additional extensions: variants, lists, ...
- Additional types: the `bottom` type, joins and meets, ...
- Method-sharing between objects of the same class.
  ➜ Bounded quantification *cf. Pierce, ch. 27*