Subtyping

Thomas Wies
wies@mpi-sb.mpg.de

April 29, 2003

Abstract

Subtyping is a characteristic aspect of type systems for object-oriented languages. It extends a type system with an ordering on types and provides a way to support key features like class-inheritance or information hiding.

This is the extended abstract of a talk on subtyping given at the seminar *Types and Programming Languages*, based on the book of same name by Benjamin C. Pierce.

1 Introduction

Subtyping is a characteristic aspect of type systems for object-oriented languages. It extends a type system with an ordering on types and provides a way to support key features like class-inheritance or information hiding. The idea of subtyping goes back to the 1960s, when the first object oriented languages, such as Simula, appeared. The first formal treatments of subtyping go back to Reynolds [6] and Cardelli [2]. An up to date review on the state of research is given by Bruce [1].

This is the extended abstract of a talk on subtyping given at the seminar *Types and Programming Languages*. The seminar was based on the book of same name by Benjamin C. Pierce [4]. The talk covered the chapters 15 to 18.

This abstract is organized as follows: section 2 motivates the introduction of subtyping and gives a formal definition of the subtype relation. Section 3 discusses algorithmic aspects of subtyping and develops an inference system for a basic typechecker with subtyping. Section 4 shows how subtyping can be combined with some of the language extensions, such as references. That subtyping is not an unproblematic issue with respect to language implementation is pointed out in section 5. In section 6 the developed concepts are used to embed the key features of object-oriented languages in the simply typed lambda calculus with subtyping and references. The last section concludes and gives a short reflection on some of the topics raised in the discussion of the talk.

2 The Subtype Relation

The simply typed lambda calculus is often to restrictive and rejects programs with a clear, well-defined behavior as untypeable. For instance the typing rule for application insists on the fact that in an application term the domain type of the applied function and the argument type coincide exactly. The following example violates this premise and is therefore not well-typed:

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

However, the applied lambda term in this example just expects a record with a field x of type Nat. Any record that provides at least such a field x should be fine as an argument of that term. This lack of flexibility in the type system leads to the notion of *subtyping*.

The intuitive *subset semantics* one has in mind is that whenever S is a subset of T, then any term of type S should also be of type T. In the example the set of records that have at least the fields x and y is a subset of the set of records with at least the field x and hence the above term should be well-typed.

More general we say that whenever it is safe to use a term of type S in a context of type T, then S is called a *subtype* of T, written in relational style: $S <: T$.

2.1 The Rule of Subsumption

Before we define the subtype relation $S <: T$, let us first formalize the notion of subsumption in terms of typing rules to bridge the gap between typing and subtyping:

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

It is astonishing, that subtyping does not interfere with the evaluation of terms, it just works on the level of types. In particular it suffices to introduce one more typing rule.

The rule T-SUB exactly expresses that if $S <: T$ holds, then any term of type S has also type T. The subsumption rule now makes the example from the beginning of this section well-typed, once we have derived that $\{x : Nat, y : Nat\} <: \{x : Nat\}$ holds.

2.2 The Subtype Relation

The inference rules that define the subtype relation can be roughly divided into two groups. One group that describes intrinsic properties of any possible subtype relation, for instance how subtyping behaves on arrow types. And a second group that gives the rules for subtyping on the basic types in the language.

Figure 1 gives an example on how the rules can be used to actually prove a term to be well-typed.

Order Theoretic Rules

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

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

It is quiet natural that the subtype relation is reflexive and transitive, i.e. the subtype relation defines a quasi-ordering on types. As we will see later, the antisymmetry property is broken and we do not obtain a full partial order. Nevertheless, it is possible to define subtyping in a way to gain even lattice properties like existence of joins and meets, see section 7.1 for more details.

Arrow Types

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

At first sight it might be confusing that the rule S-ARROW is contravariant on the domain-types of arrow types and covariant on their result-types. But this can be easily explained via the subset semantics. For a function to be safely used in a super-type context it should at most return values in the range of its super-type, which means covariance on the result-types. On the other hand it must be at least defined on the values of the super-type's domain, which gives us contravariance on the domain-types. The example in figure 1 may give some more intuition.

Top

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

The rule S-TOP introduces a "greatest" type Top and expresses that every type is a subtype of Top. The type Top is not necessarily needed for subtyping, but it is quiet useful in examples and has some natural applications, which will be explored later.

Record Deepening

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

This rule can be used to extend the subtype relation on the types of record fields to the record types itself.

Record Widening

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

The S-RCDWIDTH rule captures the property of record types with respect to the subset semantics that we have already discussed at the beginning of this section. It is safe to add additional fields at the end of record types and thereby obtain a subtype. In particular this implies hat the empty record type $\{\}$ is a subtype of any other record type.

Record Permutation

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

Since projection of record fields does not depend on the order the fields actually appear in the type of the record, it is always save to permute them. The S-RCDPERM rule exactly covers this kind of record subtyping.

The price we pay for this rule is that it makes the subtype relation non anti-symmetric. We only have anti-symmetry modulo application of S-RCDPERM.

2.3 Properties of the Subtype Relation

Subtyping would not be of any use, if it was not type safe. It can be shown that type safety is preserved in the extended system, i.e. the following theorems do hold:

Theorem (Preservation): If $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$

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

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Nat} <: \mathsf{Top}}\ \text{S-TOP}}{\{x : \mathsf{Nat}, y : \mathsf{Nat}\} <: \{x : \mathsf{Top}, y : \mathsf{Top}\}}\ \text{S-RCDDEPTH} \quad \dfrac{}{\{x : \mathsf{Nat}, y : \mathsf{Nat}\} <: \{x : \mathsf{Nat}\}}\ \text{S-RCDWIDTH}}{\{x : \mathsf{Nat}, y : \mathsf{Nat}\} <: \{x : \mathsf{Top}\}}\ \text{S-TRANS}$$

$$\dfrac{\dfrac{\vdots}{\vdash \lambda r : \{x : \mathsf{Top}\}.\, r.x : \{x : \mathsf{Top}\} \to \mathsf{Top}} \quad \dfrac{\{x : \mathsf{Nat}, y : \mathsf{Nat}\} <: \{x : \mathsf{Top}\} \quad \dfrac{}{\mathsf{Top} <: \mathsf{Top}}\ \text{S-REFL}}{\{x : \mathsf{Top}\} \to \mathsf{Top} <: \{x : \mathsf{Nat}, y : \mathsf{Nat}\} \to \mathsf{Top}}\ \text{S-ARROW}}{\vdash \lambda r : \{x : \mathsf{Top}\}.\, r.x : \{x : \mathsf{Nat}, y : \mathsf{Nat}\} \to \mathsf{Top}}\ \text{T-SUB}$$

$$\dfrac{\vdash \lambda r : \{x : \mathsf{Top}\}.\, r.x : \{x : \mathsf{Nat}, y : \mathsf{Nat}\} \to \mathsf{Top} \quad \dfrac{\vdots}{\vdash \{x = 0, y = 1\} : \{x : \mathsf{Nat}, y : \mathsf{Nat}\}}}{\vdash (\lambda r : \{x : \mathsf{Top}\}.\, r.x)\ \ \{x = 0, y = 1\} : \mathsf{Top}}\ \text{T-APP}$$

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

The proofs are quiet similar to those in the pure simply typed lambda calculus[1].

3 Typechecking

In the simply typed lambda calculus we have been in the lucky position that a typechecker could be implemented directly from the inference rules by reading the rules bottom up. In the extended system with subtyping we have to invest some additional work.

In the following we will develop algorithmic subtyping- and typing relations, which are more convenient for typechecking, but sound and complete with respect to the system we have focused on in the last section.

3.1 Subtype Checking

In order to build a typechecker for a type system with subtyping, we first have to build a subtype-checker, which is able to check whether two types are in the subtype relation.

The problem with the subtype relation we have considered so far is, that the inference rules can not be used to implement a subtype-checker directly by reading the rules bottom up. Let us reconsider the transitivity rule to explain this further:

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

The conclusion of this rule mentions the meta-variables S and T. These variables match any possible types. Hence there is no way to decide via the syntactic structure of a type whether to use this rule or some other rule to recursively derive $S <: T$. Even worse, we do not know anything about the type U in the premise of the rule, i.e. we would have to guess the right one. Similar problems do arise for the reflexivity rule.

The solution is to define a new algorithmic subtype relation, written $\mapsto S <: T$. This relation overcomes the above problems, because it will be based on syntax-directed inference rules only.

[1] All detailed proofs of theorems stated in this abstract can be found in [4].

$$\dfrac{\dfrac{\dfrac{}{\mathsf{Nat} <: \mathsf{Top}}\ \text{S-TOP}}{\{\mathsf{x} : \mathsf{Nat}, \mathsf{y} : \mathsf{Nat}\} <: \{\mathsf{x} : \mathsf{Top}, \mathsf{y} : \mathsf{Top}\}}\ \text{S-RCDDEPTH} \quad \dfrac{}{\{\mathsf{x} : \mathsf{Nat}, \mathsf{y} : \mathsf{Nat}\} <: \{\mathsf{x} : \mathsf{Nat}\}}\ \begin{matrix}\text{S-RCDWIDTH}\\ \text{S-TRANS}\end{matrix}}{\{\mathsf{x} : \mathsf{Nat}, \mathsf{y} : \mathsf{Nat}\} <: \{\mathsf{x} : \mathsf{Top}\}}$$

$$\dfrac{\dfrac{\vdots}{\vdash \{\mathsf{x} = 0, \mathsf{y} = 0\} : \{\mathsf{x} : \mathsf{Nat}, \mathsf{y} : \mathsf{Nat}\}} \quad \{\mathsf{x} : \mathsf{Nat}, \mathsf{y} : \mathsf{Nat}\} <: \{\mathsf{x} : \mathsf{Top}\}}{\vdash \{\mathsf{x} = 0, \mathsf{y} = 0\} : \{\mathsf{x} : \mathsf{Top}\}}\ \text{T-SUB}$$

$$\dfrac{\dfrac{\vdots}{\vdash \lambda \mathsf{r} : \{\mathsf{x} : \mathsf{Top}\}.\, \mathsf{r.x} : \{\mathsf{x} : \mathsf{Top}\} \to \mathsf{Top}} \quad \{\mathsf{x} : \mathsf{Nat}, \mathsf{y} : \mathsf{Nat}\} <: \{\mathsf{x} : \mathsf{Top}\}}{\vdash (\lambda \mathsf{r} : \{\mathsf{x} : \mathsf{Top}\}.\, \mathsf{r.x})\ \ \{\mathsf{x} = 0, \mathsf{y} = 1\} : \mathsf{Top}}\ \text{T-APP}$$

Figure 2: Rewritten proof for $\vdash (\lambda \mathsf{r} : \{\mathsf{x} : \mathsf{Top}\}.\, \mathsf{r.x})\ \ \{\mathsf{x} = 0, \mathsf{y} = 1\} : \mathsf{Top}$

The question is, where do we actually use the transitivity and reflexivity rules in a subtype derivation? The first thing to observe is that reflexivity is not needed at all for subtype-checking, we can simply skip that rule. The transitivity rule is only needed for combining derivations involving uses of different record rules, as it is shown in figure 2.

The simple idea is to merge all rules regarding record-subtyping in one single rule and afterwards drop S-TRANS:

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

The new algorithmic subtype relation is now defined by just SA-RCD and the unchanged S-ARROW and S-TOP rules. Of course we have to prove that the two subtype relations in fact do coincide:

Theorem (Soundness and Completeness): $\mathsf{S} <: \mathsf{T}$ iff $\mapsto \mathsf{S} <: \mathsf{T}$.

3.2 Typechecking

For typechecking the same problem arises then for subtype-checking. Reconsidering the rule of subsumption:

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

we find again an unbounded variable t in the conclusion of the rule. Hence the rule applies on all possible terms.

Again the idea of an algorithmic typing relation only defined via syntax-directed inference rules gives us a solution, but how can we achieve that?

Taking a closer look at the derivation trees involving subsumption reveals that we essentially only need subsumption to match the domain types of functions and the types of their arguments. In fact any typing derivation can be rewritten, such that T-SUB is only used to derive the correct type of the argument for applications. Figure 2 shows the rewritten version of the derivation in figure 1.

Hence we just have to enrich the T-APP rule to check for subsumption on the fly:

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

and now we can safely remove the subsumption rule. Of course, again we have to check that the algorithmic typing relation coincides with the old version:

Theorem (Soundness) If $\Gamma \vdash\!\!\rightarrow t : T$, then $\Gamma \vdash t : T$.

Theorem (Completeness) If $\Gamma \vdash t : T$, then $\Gamma \vdash\!\!\rightarrow t : S$ for some $S <: T$.

An important remark on the completeness theorem is that it not only states completeness, but also that each typeable term has a minimal type. This is due to the fact that the algorithmic typing relation turns out to be a function and hence must assign to each term its minimal type.

4 Extensions: References, Casts

In this section we will show how subtyping can be combined with some of the extensions to the simply typed lambda calculus. In particular we will focus on references and ascription. Ascription in combination with subtyping turns out to be a casting operator.

4.1 References

In order to extend subtyping to reference types, we have to check what conditions must hold in order to get $\mathsf{Ref}\ S <: \mathsf{Ref}\ T$. Consider the following example:

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

This term will go wrong, since we try to project an x-field of a referenced record that does only provide a field y. A premise $S <: T$ is needed in order to get safe dereferences. This may not be surprising, but looking at the following example:

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

reveals that we need the premise $T <: S$, too, in order to get safe assignments. Hence the new inference rule we add to the system is invariant in the types S and T:

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

4.2 References refined

Although the inference rule given above is sufficient to handle references and subtyping, we can give a more refined analysis. This will lead to a nice application in section 6.5.

We already have seen that the contravariant premise in the inference rule is needed for assignments, where else the covariant premise is only needed for dereferencing. The idea now is to 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 to modify the typing rules for references accordingly:

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

$$\frac{\Gamma \mid \Sigma \vdash t_1 : \mathsf{Sink}\ T \quad \Gamma \mid \Sigma \vdash t_2 : T}{\Gamma \mid \Sigma \vdash t_1 := t_2 : \mathsf{Unit}}\ \text{(T-ASSIGN)}$$

We now split the previous subtyping rule for references in two new rules. One for Sink and one for Source:

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

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

By forcing Ref T to be a subtype of both Source T and Sink T, we get the right premises on the type T:

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

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

4.3 Up-Casts

The ascription operator t as T was initially used to annotate types that are not necessary for typechecking, but useful for documentation. Here the typing rule for ascription:

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

Combined with subtyping this operator is able to perform type casts.

Up-casts are normally used for information hiding. By assigning a super-type to a term via ascription, some of the information that may be accessible in the context of the ascribed term is hidden. The rule T-ASCRIBE together with the subsumption rule immediately give us up-casts:

$$\{x = 0, y = 1\}\ as\ \{x : Nat\}$$

The term in the example is well-typed and the field y is no longer accessible in the context.

4.4 Down-Casts

Down-casts are casts that ascribe a term with a subtype. In combination with the type Top the canonical application of such casts is a kind of poor men's polymorphism. For instance Java uses this concept to implement container classes like lists or maps using the type Object, which corresponds to Top in our case. Down-casts require an additional typing-rule:

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

Of course, this typing rule will destroy type-safety. We will have to check at runtime, whether the type of the term and its ascribed type actually are in the subtype relation.

5 Coercion Semantics

When we introduced subtyping we stated that it has no effect on the semantics of the simply typed lambda calculus, since evaluation does not change. Nevertheless subtyping may result in performance penalties in a low level language implementation.

For instance it is not clear how one can efficiently perform record-field accesses in the presence of a permutation rule. On a low level implementation records are normally represented as tuples assuming some fixed order on the labels. This representation will not work if we allow field-permutations.

To address such problems the idea is to completely compile away subtyping. This can be done by using the subtyping and typing derivations of terms to produce the additional code in our target language that coerces the right type everywhere subsumption is used. This new semantics is therefore called *coercion semantics*.

Pierce gives an example in his book, where he translates the simply typed lambda calculus with subtyping and records into the pure simply typed lambda calculus with records.

6 Case Study: Featherweight Java

In this section we will use the concepts developed so far to embed the key features of object oriented languages, such as Java, into the simply typed lambda calculus with subtyping and references.

First of all, what are the essential features of imperative objects we want to deal with?

- Multiple representations:

  The operations that one can perform on an object are completely defined via the interface. The concrete implementation is not part of the type of an object, hence we may have multiple representations of the same object that can be used interchangeably.

- Encapsulation and information hiding:

  The internal state of an object is only accessible via its interface and the concrete representation is hidden.

- Inheritance:

  Classes are used as templates for object instantiation. Derived sub-classes can selectively share code with their super-classes.

- Subtyping:

  Objects of subclasses 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 derived sub-classes, which is normally referred to as late-binding.

```
class Counter {
   private int x;

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

   public int get() {return x;}

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

Figure 3: A simple counter class in Java

6.1 Interfaces and Objects

How can we mimic the described features in the simply typed lambda calculus? Let us first try to find a representation for interfaces and objects.

Record types provide all we need to represent interfaces. We just have to introduce a field of functional type for each public method of an object. Consider the simple counter class in figure 3. Here is the definition of the corresponding record type:

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

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

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

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

6.2 Classes

Since we do not want to construct every new instantiated object by hand, we need some representation of classes that encapsulates the method declarations and can be used for object instantiation.

For abbreviation we first define a representation type for the instance variables of the object:

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

The counter class now abstracts on this counter representation:

$$\begin{array}{l}\mathsf{counterClass} =\\ \quad \lambda \mathsf{r} : \mathsf{CounterRep}.\\ \qquad \{\mathsf{get} = \lambda\_ : \mathsf{Unit}.\ !(\mathsf{r.x}),\\ \qquad\ \ \mathsf{inc} = \lambda\_ : \mathsf{Unit}.\ \mathsf{x} := \mathsf{succ}(!(\mathsf{r.x}))\};\\ \triangleright\ \mathsf{counterClass} : \mathsf{CounterRep} \to \mathsf{Counter}\end{array}$$

New objects can be instantiated via an object generator:

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

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

Figure 4: A class inherited from CounterClass

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

6.3 Inheritance and Subtyping

Perhaps the most important feature of classes is inheritance. Figure 4 gives an example of a class derived from our previous Counter class.

In order to declare the ResetCounter class, we need to extend the counter interface:

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

after that we can reuse counterClass within resetCounterClass:

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

Record subtyping provides all we need for subtyping between objects. We have ResetCounter <: Counter, hence any reset-counter can be used safely as just a counter:

```
rc = newResetCounter unit;
▷ rc : ResetCounter

inc3 = λc : Counter. c.inc unit; c.inc unit; c.inc unit;
▷ inc3 : Counter → Unit

(inc3 rc; rc.reset unit; inc3 rc; rc.get unit);
▷ 4 : Nat
```

6.4 Classes with self

In the declaration of methods it is usually possible to access the methods of the same object instance the method is invoked on. This is typically done via a variable self or this. We now want to extend our classes with this self-recursion.

```
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;}
}

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;}

}
```

Figure 5: Example of open recursion in Java

Let us implement a new SetCounter class that provides an additional method to set the counter to a given value. The method for incrementing the counter will then make use of this set method with the help of self. First of all, here is the new interface type that is needed:

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

There are several well-known possibilities to handle recursion. One possible way is to use references. We will use them in the next section. But let us use this time fix-point recursion to introduce self:

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

The recursion works as usual. We first abstract on the variable self of appropriate object type and then apply the fix-point operator to the functional to tie the knot.

6.5 Open Recursion

Figure 5 gives an application of open recursion in Java. The example derives a class BackupCounter from ResetCounter. BackupCounter has the ability to backup the current counter value and restore it if necessary. The method set is redeclared within BackupCounter such that it backups the counter before setting the new value. The open recursion mechanism then will ensure that, although the reset method is declared within SetCounter only, the call of set will be bound to the latter one in

BackupCounter[2] and any invocation of reset will backup the counter first.

First as before the new SetCounter interface:

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

As we already mentioned we will use references now to introduce self. Of course, we could again use fix-point recursion, but in order to make this work here, we would have to protect each access to self with a dummy lambda abstraction. This is due to the eager semantics we use. These additional lambda abstractions would be quite inefficient, because for each method invocation the method table would have to be recomputed.

For the referenced based recursion it suffices to compute the method table only once each time a new object is instantiated. The type of self is now a reference to a method table:

```
setCounterClass =
  λr : SetCounterRep. λself : Ref SetCounter.
      let super = counterClass r in
      {get = super.get,
       inc = super.inc,
       set = λi : Nat. r.x := i,
       reset = λ_ : Unit. (!self).set 1};
▷ setCounterClass : CounterRep → Ref SetCounter → SetCounter
```

For object instantiation we first allocate a dummy method table and use a reference on that to generate the new object. Afterwards we update the reference to the real object:

```
newSetCounter =
  λ_ : Unit. let r = {x = ref 1} in
             let mTbl = ref {get = λ_ : Unit. 0,
                             inc = λ_ : Unit. unit,
                             set = λi : Nat. unit,
                             reset = λ_ : Unit. unit} in
             (mTbl := setCounterClass r mTbl); !mTbl;
▷ newSetCounter : Unit → SetCounter
```

Since all accesses to self within setCounterClass are protected by lambda abstractions, self will not be dereferenced before the update to the object has been performed. Now we try to declare the BackupCounter class:

```
BackupCounter = {get : Unit → Nat, inc : Unit → Unit,
                 set : Nat → Unit, reset : Unit → Unit,
                 restore : Unit → Unit};

BackupCounterRep = {x : Nat, b : Nat};
```

To actually make the methods late-bound, when constructing the super-class' method table within backupCounterClass we pass it the reference to the BackupCounter object:

[2] Remember that in Java all methods are late-bound by default.

```
backupCounterClass =
   λself : Ref BackupCounter.
      λr : BackupCounterRep. let super = setCounterClass r self in
        {get = super.get,
         inc = super.inc,
         set = λi : Nat. r.b =!(r.x); super.set i,
         reset = super.reset,
         restore = λ_ : Unit. r.x :=!(r.b)};
▷ Error :  parameter type mismatch
```

Unfortunately this version does not typecheck, because the rule for subtyping on references requires SetCounter <: BackupCounter which is not satisfied. Now our refined analysis on references from section 4.2 comes into play. Since we do only need read access to an object's own method table we can replace Ref by Source within the declaration of setCounterClass:

```
setCounterClass =
   λr : SetCounterRep.
      λself : Source SetCounter.
        let super = counterClass r in
        {get = super.get,
         inc = super.inc,
         set = λi : Nat. r.x := i,
         reset = λ_ : Unit. (!self).set 1};
▷ setCounterClass : CounterRep → Source SetCounter → SetCounter
```

The weaker premise for subtyping on Source is satisfied, we have:

```
BackupCounter <: SetCounter
```

and the new BackupCounter class now typechecks:

```
BackupCounterClass =
   λself : Source BackupCounter.
      λr : BackupCounterRep. let super = setCounterClass r self in
        {get = super.get,
         inc = super.inc,
         set = λi : Nat. r.b =!(r.x); super.set i,
         reset = super.reset,
         restore = λ_ : Unit. r.x :=!(r.b)};
▷ backupCounterClass : BackupCounterRep →
                       Source BackupCounter → BackupCounter
```

7 Conclusion and Discussion

We have seen how a type system can be extended formally with subtyping and how we can typecheck such a system. We used this framework to embed essential features of object-oriented languages in the simply typed lambda calculus. We have also seen that subtyping may cause penalties in the performance on the low level language implementation and mentioned an alternative coercion semantics to overcome this problem.

Now some of the topics not covered by the previous sections, but raised in the seminar discussion will be presented roughly.

7.1 Joins and Meets

We already mentioned that the subtype relation defines a quasi ordering on types, but our system has even more lattice theoretic properties that might be of interest.

In the discussed framework we only introduced a type Top, which is a supertype of all other types. Dually one can of course introduce a smallest type Bot. Doing this leads to a system in which we can show existence of joins and meets between any two subtypes.

But even if we do not introduce Bot we can still show that at least joins always exist and that if two types have any common subtype they will have a greatest one, too. These joins and meets are effectively computable.

7.2 Multiple Inheritance

In the discussion of class-inheritance we only considered inheritance from a single super-class. Of course, we can also try to encode multiple inheritance. In order to accomplish this we need to extend our system with intersection-types [3].

7.3 Alternative Views

Although subtyping is perhaps the more popular view on type systems of object-oriented languages, it is not the only possible view. An alternative formalism is based on row-variable polymorphism, developed by Wand [7] and others. This concept is the basis of the object oriented features in the OCaml language [5].

References

[1] Kim B. Bruce. *Foundations of Object-Oriented Languages: Types and Semantics*. MIT Press, 2002.

[2] Luca Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Summary in *Semantics of Data Types*, Kahn, MacQueen, and Plotkin, eds., Springer-Verlag LNCS 173, 1984.

[3] Adriana B. Compagnoni and Benjamin C. Pierce. Intersection types and multiple inheritance. 6(5):469–501, October 1996. Preliminary version available as University of Edinburgh technical report ECS-LFCS-93-275 and Catholic University Nijmegen computer science technical report 93-18, Aug. 1993, under the title "Multiple Inheritance via Intersection Types".

[4] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[5] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. Summary in , 1997.

[6] John C. Reynolds. Using category theory to design implicit conversions and generic operators. In N. D. Jones, editor, *Proceedings of the Aarhus Workshop on Semantics-Directed Compiler Generation*, number 94, January 1980.

[7] Wand and Mitchell. Complete type inference for simple objects. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, Ithaca, NY, June 1987.